Automatic Unit Test Generation for Programming Assignments Using Large Language Models

Kaisheng Zheng, Yuanyang Shen, Yida Tao*
Department of Computer Science and Engineering
Southern University of Science and Technology, Shenzhen, China
{12110722, 12112217}@mail.sustech.edu.cn, taoyd@sustech.edu.cn

***Abstract*—Programming knowledge is a crucial aspect of computer science education, and unit testing is commonly employed to automatically assess programming assignments. Instructors and teaching assistants typically invest considerable efforts in writing unit tests, which may still be vulnerable to human oversight and mistakes. In this work, we explored the feasibility of using Large Language Models (LLMs) to automate the assessment of programming assignments. In particular, we proposed two approaches: the plain approach that uses GPT-4o-mini in a vanilla setting, and the augmented approach that integrates additional strategies such as tailored prompts with syntax and semantic constraints, and a feedback mechanism with information on test-effectiveness metrics.**

We evaluate the two approaches on six real-world programming assignments from an introductory-level programming course at our university. Compared to the plain approach, the augmented approach improves the usability and effectiveness of the generated unit tests, reducing 85% compilation errors while enhancing the statement coverage and mutation scores by 1.7x and 2.1x, respectively. In addition, the augmented approach also complements human-written tests by covering additional program behaviors. In a case study of 1296 students' submissions that pass human-written tests, the augmented approach successfully detected new bugs in 13% submissions, with an accuracy of 27%. These results not only demonstrate the potentials of LLMs in generating useful unit tests for programming assignments, but also highlight the strategies that can effectively enhance LLMs' capabilities to augment human-written tests, offering practical benefits for both educators and students.

***Index Terms*—Unit test generation, programming assignments, Large language models.**

I. Introduction

Coding is a fundamental aspect of computer science education, with students typically developing coding skills through hands-on programming assignments. However, grading programming assignments presents significant challenges, as it requires instructors to compile each assignment submission, execute it, and manually verify program behaviors. Fortunately, the formal structure of programming assignments allows for automatic assessments, much like a software product can be automatically tested in industrial settings. Research has also suggested that modern classrooms are rapidly shifting from manual efforts to automated assessments for programming assignments [1]–[3]. At our university, an Online Judge (OJ) platform was implemented to facilitate the automated assessment of programming assignments. Given unit tests written by the teaching team, the OJ platform can automatically compile assignment submissions, execute the unit tests, and provide testing results as instant feedback to students.

Nonetheless, writing unit tests manually is both time-consuming and error-prone. In our experience, it often takes a teaching team days to create a complete test suite for a single assignment. Since assignment setters and test creators are deliberately assigned to different individuals to enable cross validation, the unit tests sometimes are not consistent with the assignment requirements. In addition, the quality of the unit tests varies depending on the individual test creators: some may focus on testing the basic requirements, while others may pay extra attention to edge cases, resulting in huge confusions among students. Hence, to enhance students' learning experience and instructors' working efficiency, we are looking for a more scalable approach to producing consistent and high-quality unit tests for programming assignments.

Large Language Models (LLMs) are computational models trained with deep learning algorithms and vast amount of data. Recently, LLMs have achieved remarkable breakthroughs in a wide range of fields such as natural language processing and content generation. In the field of computer science and software engineering, tools like GitHub Copilot [4] and Cursor [5], which are empowered by LLMs, have demonstrated great potentials in boosting developers' coding and debugging productivity. Meanwhile, educators are also actively seeking opportunities of incorporating LLMs and related AI technologies into the classrooms [6]–[9]. Nevertheless, how LLMs can be applied in unit test generation for programming assignments is rarely explored in the literature, and our study aims to bridge this gap.

In this study, we first investigate how bare LLMs perform in generating unit tests when supplied with only the minimum necessary information as prompts. We refer to this as the *plain approach*, which resembles how the teaching teams would normally interact with the LLMs. Based on the result analysis, we further proposed an *augmented approach*, which extends the plain approach with novice strategies to improve the test generation performance. Specifically, our study addresses the following research questions:

*Yida Tao is the corresponding author.

- **RQ1**: How does the plain LLM approach perform in generating unit tests for programming assignments, compared to human-written tests?
- **RQ2**: Which strategies can be adopted to improve the test generation performance of the plain approach?
- **RQ3**: What are the potential benefits of the proposed augmented approach for both instructors and students?

For evaluation, we used GPT-4 as the LLM in the plain approach, and applied it on six unique assignments from a popular programming course at our university. The plain approach does not yield satisfactory results in the experiments. Nearly half of its generated tests are unusable for containing syntax and logic errors. The remaining half of the generated tests have lower statement coverage and mutation scores compared to the human-written tests. In other words, regular usages of existing LLMs, as in the plain approach, may not be able to produce unit tests that are sufficiently usable and effective for practical adoption in the classrooms.

To address the limitations of the plain approach, we proposed an augmented approach that adopts more sophisticated strategies for unit test generation. Specifically, the augmented approach combines various source of information to produce useful prompts, which instruct the LLM to generate syntactically correct tests with certain degree of semantic complexity. The augmented approach also integrates a feedback mechanism, which leverages coverage metrics and mutation testing to keep enhancing the LLM's performance in subsequent iterations. In addition, the augmented approach supports configurable strategies for rejecting the generated unit tests, adding another tier of quality control to LLM-generated output.

We implemented a prototype tool, `AugGPT`, for the augmented approach. On the same evaluation dataset, `AugGPT` improves the usability of generated tests by reducing 85% compilation errors and 28% misinterpretation errors compared to the plain approach. `AugGPT` also improves the effectiveness of the generated tests by achieving 1.7x the statement coverage and 2.1x the mutation score compared to the tests generated by the plain approach, indicating the viability of the proposed strategies.

Our experiments also demonstrate the strong potential of `AugGPT` for use in educational settings. For instance, `AugGPT` achieves a level of test coverage on par with the human-written tests. Combining `AugGPT`-generated tests with human-written tests further enhances the test coverage and mutation scores compared to using the human-written tests alone. Moreover, in a case study on 1296 real-world assignment submissions, `AugGPT` was able to detect real bugs that slipped through human-written tests in 169 (13%) submissions.

In summary, `AugGPT` delivered promising results in assisting manual testing-writing efforts, by improving the overall test effectiveness and identifying real problems in students' programming assignments that the human-written tests failed to detect. We envision that instructors and teaching assistants will benefit from `AugGPT` to create high-quality test cases for programming assignments more efficiently. Meanwhile, students can also leverage `AugGPT` to diagnose their programming assignments, which in turn consolidate their programming knowledge and practical debugging skills.

II. Related Work

A. Unit Test Generation

Unit testing is a software testing technique where the smallest testable units of a software system, such as methods and classes, are tested in isolation to validate their individual behaviors. In the life cycle of a software product, unit testing is the earliest testing efforts, laying the groundwork for subsequent integration testing and user acceptance testing before release. In addition, the quantity of unit tests typically accounts for the vast majority of all the tests (70%-80%) for a software product [10]. Hence, significant human efforts are dedicated to writing unit tests due to their critical role in software development.

Nevertheless, manually writing unit tests is costly, and such tests are often incomplete or even flawed due to human errors. To better support unit testing with reduced cost and enhanced test quality, researchers and practitioners have proposed methodologies for automatic unit test generation. Two of the most classic approaches along this line of work include *random testing* and *search-based testing*. Random testing aims to cover a wide range of program behaviors by randomly generated inputs. `Randoop` [11] is a specific tool that automates random testing for Java programs. It randomly generates method invocation sequences, and use the execution results of the invocation sequences to generate assertions in the unit tests. Search-based testing aims to find a test suite that best meets the testing objectives by defining fitness functions that capture the patterns of testing goals and employing a search-based optimization algorithm. `Evosuite` [12] is one of the well-known search-based testing tools for Java. It defines a fitness function to evaluate the effectiveness of the generated tests in terms of branch coverage, and automatically evolves the generated tests using genetic algorithms. While traditional approaches have shown promising test generation results, several challenges remain, including low readability and usability of the generated unit tests, which limit their practical use [13] [14].

With the rapid progress in LLMs and their notable success in code generation tasks, researchers have also begun to explore the use of LLMs for unit test generation. A recent survey [14] reported over 20 top-conference papers published on this subject in the year 2023 alone, and identifies two major types of approaches adopted in these studies. The first type of approach pre-trains or fine-tunes LLMs for unit test generation, with the aim of embedding domain knowledge in LLMs to improve their test generation performance [15] [16]. The second type of approach focuses on prompt engineering, which explores how different prompt designs, input context and feedback information enhance LLMs' ability to generate better unit tests [17] [18]. The most widely studied LLMs include ChatGPT [19] and Codex [20], both of which demonstrate

TABLE I: Size of the input code (i.e., the standard answer).

Assignment	# Classes	# Methods	# Tokens
Calendar	2	10	868
Treasure	6	27	1282
Course	3	41	2031
Canvas	8	35	2631
Shop	4	24	1543
Chess	11	33	18781

promising results with respect to the correctness and coverage of generated tests.

Nevertheless, most studies of LLM-based unit test generation are evaluated on public dataset of programming problems and defects collected from open source repositories. Few studies use LLMs to generate unit tests for students' programming assignments, which differ in many ways from the open-source data. Our work aims to bridge this gap.

B. LLMs for CS Education

Building on their promising results in text comprehension and code generation, LLMs also demonstrate significant potential in advancing computer science education. The most common applications of LLMs in the education landscape is *AI tutoring*, where students interact with LLMs to acquire personalized feedback and assistance on their coursework. In particular, researchers have proposed LLM-based AI tutoring for learning activities such as program comprehension [21], software testing [22], writing UML diagrams [8], and coding or debugging [6] [7] [9] [23]. Online education platforms like Khan Academy also integrate LLM-powered AI tutor to provide instant feedback tailored to the individual's learning pace [24]. Most of these studies have utilized ChatGPT due to its conversational nature and interactivity, which align well with the valued attributes of an effective tutor.

While AI-tutoring is often considered beneficial primarily from the students' perspectives, LLMs are also being applied to assist course instructors in their teaching activities. For instance, Hua Leong Fwa leveraged the advanced NLP capabilities of ChatGPT to analyze the errors and misconceptions in students' code solutions, which would otherwise require substantial manual effort [25]. Sarsa et al. used OpenAI Codex to generate programming exercises with sample solutions, explanations and test cases, most of which were sensible and ready for use [26]. Marcin Jukiewicz explored roles of ChatGPT in assessing programming assignments, and reported a strong agreement between grades generated by ChatGPT and human instructors [27]. In addition to assignment creation and evaluation, writing unit tests for programming assignments is also a crucial part of the workload for instructors. In this study, we explore how LLMs can assist with this task.

III. The Plain Approach

In this section, we introduce the basic settings of the plain approach and evaluate its generated test cases on real programming assignments.

A. Settings

To address RQ1, we employ existing LLMs in their standard configuration, without any additional modifications or enhancements. In other words, the plain approach should reflect how most users would typically use LLMs in their daily tasks.

In the context of this study, the task for the LLM is to generate a sufficient number of unit tests for the given programming assignment. After discussing with our teaching teams, which consist of instructors and teaching assistants, we proposed the following prompt and interaction pattern to simulate their typical workflows:

1) System prompt: "*You are a helpful assistant.*"
2) User: "*This is a piece of the student's Java program code, please generate a set of unit tests based on it.* `<CODE>`"
3) LLM: `<answer>`
4) User: "*Please refine it and provide a new set of tests.*"
5) LLM: `<answer>`
6) Repeat 4-5 until the accumulated number of generated unit tests reaches N.

In this study, we used $N = 48$ after trials to balance the system runtime and the quality of the generated tests. We used `GPT-4o-mini` as the LLM model through its APIs [28]. For simplicity, we refer to this plain approach as **GPT** for the remainder of the paper.

B. Dataset

To evaluate the plain approach, we used assignments from an introductory-level programming course at our university, which introduces foundational knowledge of computer programming using Java as the primary programming language. Specifically, we selected six assignments from the Fall 2022, Fall 2023 and Spring 2024 semesters as described below.

- *Calendar*: a basic calendar application that allows users to manage events and track time.
- *Treasure*: a simple game in which multiple players search on the same map for hidden treasures.
- *Course*: a credit-based course selection program for students.
- *Canvas*: a program that enables users to place various shapes on a canvas, and calculate their areas and overlaps.
- *Shop*: an online shopping application that allows customers to purchase various products from an online store.
- *Chess*: A chess game implemented as a console program.

These programming assignments were released near the end of each semester and therefore covered nearly all aspects of the curriculum, including data types, control structures, method invocations, and object-oriented concepts. These assignments are also highly diverse, each possessing unique characteristics. Using them together allows for a more comprehensive evaluation of our approach.

For each assignment, the teaching team prepared a "standard answer", which is a Java program that correctly fulfills the assignment requirements. The standard answer was released to students after the assignment deadline. In this study, we

used the standard answers as the input <code> for the LLM. Table I shows the size of the input source code in terms of the number of classes, methods, and tokens. For each assignment, the teaching team also wrote a set of test cases to evaluate students' submissions. We refer to these manually-written test cases as the **Human** approach in our evaluation.

C. Usability of the Generated Tests

The plain approach generates 48 test cases for each assignment, resulting in $48 \times 6 = 288$ test cases in total. First, we evaluate the usability of these generated tests. If we assumed that the standard answer is correct, tests that failed on the standard answer are therefore problematic and unusable. Hence, we manually inspected the generated tests that failed on the standard answers and identified three types of failures: compilation errors, misinterpretation, and logic errors. Figure 1 shows the percentage of each type of the failure.

One type of failure is due to the misinterpretation of the source program. Basically, GPT failed to understand the assignment requirement given only the standard answer. For example, in *Treasure*, GPT misinterprets the coordinate system design of the game map; in *Course*, GPT incorrectly asserts that only the student who bids higher points will be enrolled, even though the required enrollment workflow ensures that two students can both be enrolled without bidding if the course capacity is at least two.

Another type of failure is compilation errors. For instance, the tests generated by GPT may illegally access the private fields in the source program. Finally, GPT makes mistakes in common-sense knowledge and mathematical calculations, such as the example below. Since this type of errors is related to LLMs' inherent logic and independent from assignment requirements, we refer to them as logic errors.

```
@Test
void testAddDaysCrossingMonth() {
    // Logic error
    // The answer should be 2024-03-02.

    MyDate date = new MyDate(2024, 2, 28);
    date.addDays(3);
    assertEquals("2024-03-01",date.toString());
}
```

According to Figure 1, 49% of the tests generated by the plain approach failed and cannot be used directly. In other words, if the teaching team used the LLM in its basic form as in the plain approach, nearly half of the generated tests would be unusable for assessing students' programming assignments.

D. Effectiveness of the Generated Tests

As shown in Figure 1, 51% of the generated tests passed. For these remaining tests, we leverage the following two metrics to evaluate their effectiveness:

- Statement Coverage: statement coverage measures the percentage of the statements that are executed by the test cases. A higher statement coverage typically indicates a stronger test suite. We used the `JaCoCo` tool [29] to compute this metric.

TABLE II: The average logical lines of code per test case.

Assignment	Human-written	GPT-generated	`AugGPT`-generated
Calendar	12.60	4.14	10.08
Treasure	22.08	5.13	13.88
Course	19.30	3.94	15.35
Canvas	10.80	4.46	12.75
Shop	21.52	3.48	12.38
Chess	20.48	9.58	17.60

- Mutation Score: mutation testing is a software testing technique, which generates "mutants" by altering the target source code with small and deliberate modifications. A mutant is considered "killed" if the existing test suite fails on it, indicating that the test suite is effective in catching errors. The mutation score is calculated as the percentage of killed mutants among all mutants. Hence, a higher mutation score indicates a more effective test suite, as it suggests that the tests are better at detecting potential defects in the code [30]. We used the `Pitest` tool [31] to calculate the mutation score.

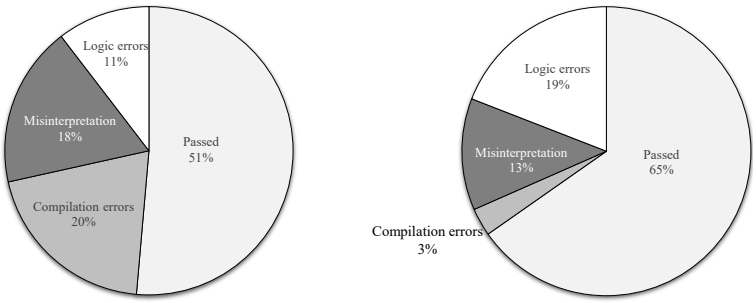
As shown in Table III, human-written tests outperform GPT-generated tests on both the statement coverage and mutation score for all of the assignments. One possible explanation is that GPT tends to generate shorter test cases with fewer statements compared to human-written tests, as shown in Table II. However, a short program is inherently constrained by its complexity. Therefore, the generated test cases may not be able to cover sufficient code logic compared to human-written tests that are carefully crafted by the teaching team.

> **RQ1**: Nearly half of the tests generated by the plain approach contain errors and cannot be used directly. The passing tests also exhibit lower statement coverage and mutation scores compared to human-written tests. In other words, using existing LLMs in their unmodified form, as in the plain approach, may not yield the desired outcomes for generating usable and effective tests for programming assignments.

E. Insights

Based on the previous analysis, we have gained the following insights into the underlying causes of the plain approach's deficiencies.

1) **Insufficient task description**: the plain approach leverages only the source program to generate tests. However, as discussed in Section III-C, GPT struggled to "reverse engineer" the assignment requirements from the source program, which resulted in misinterpretation errors in the generated tests.
2) **Lack of syntactic guidelines**: with few syntactic guidelines in the prompt, the plain approach sometimes generates tests with compilation errors. For example, GPT-generated tests may access private members illegally. In human-written tests, private members are tested through



(a) The plain approach (b) The augmented approach

Fig. 1: The usability of test cases generated by different approaches.

Java reflection, which is unlikely to be adopted by GPT if not stated explicitly in the prompt.

3) **Absence of semantic constraints**: the prompt used in the plain approach does not set any constraints on the semantics of the generated tests. Consequently, the plain approach tends to generate simple test cases with fewer statements and shorter method invocation chains, which limits the semantic depth of the generated tests.
4) **Uninformative feedback**: even though the plain approach has a feedback loop, it does not provide any new information to GPT in the feedback. Hence, newly generated tests do not demonstrate improved quality; in some cases, their quality has even declined. For instance, we observed that GPT has a tendency to generate degraded tests with shorter code and simpler logic in later iterations if no new information is further provided.
5) **Lack of quality control**: there is no rejection option in the plain approach. In other words, all the tests generated at each iteration will be accepted regardless of their quality. Therefore, the final test suite may contain lots of undesired redundancies or even erroneous tests.

Although GPT's performance was not quite satisfactory on its own, adding GPT-generated tests to human-written tests indeed resulted in marginal improvement on both the statement coverage and mutation score, as shown by the **GPT+Human** row in Table III. This indicates that GPT may complement human-written tests and potentially improve the overall test effectiveness. Motivated by these observations, we proposed an approach to augment GPT in the test generation task, which is presented in the next section.

TABLE III: Evaluation results of the test cases generated by different approaches (GPT+Human and `AugGPT`+Human are direct merge of two test set).

Assignment	Approach	Statement Coverage	Mutation Score	Number of Tests
Calendar	Human	1.00	0.86	10
	GPT	0.89	0.48	48
	`AugGPT`	1.00	0.78	48
	GPT+Human	1.00	0.87	58
	`AugGPT`+Human	1.00	0.88	58
Treasure	Human	0.98	0.92	12
	GPT	0.86	0.75	48
	`AugGPT`	0.99	0.91	48
	GPT+Human	0.98	0.94	60
	`AugGPT`+Human	0.99	0.95	60
Course	Human	0.88	0.85	10
	GPT	0.81	0.65	48
	`AugGPT`	0.88	0.76	48
	GPT+Human	0.91	0.89	58
	`AugGPT`+Human	0.92	0.92	58
Canvas	Human	0.79	0.74	10
	GPT	0.68	0.47	48
	`AugGPT`	0.82	0.68	48
	GPT+Human	0.81	0.75	58
	`AugGPT`+Human	0.87	0.79	58
Shop	Human	0.89	0.90	26
	GPT	0.80	0.59	48
	`AugGPT`	0.89	0.69	48
	GPT+Human	0.90	0.90	74
	`AugGPT`+Human	0.93	0.92	74
Chess	Human	0.63	0.47	23
	GPT	0.10	0.04	48
	`AugGPT`	0.46	0.24	48
	GPT+Human	0.65	0.48	71
	`AugGPT`+Human	0.71	0.54	71

IV. The Augmented Approach

Based on the findings of the plain approach, we proposed an augmented approach, `AugGPT`, to generate unit tests for programming assignments. Compared to the plain approach, `AugGPT` adopts advanced prompt strategies and a more sophisticated workflow. We introduce the design and implementation of `AugGPT` in this section.

A. Design

Motivated by the insights described in Section III-E, we propose a new workflow for `AugGPT`, as illustrated in Figure 2. Basically, `AugGPT` leverages assignment description for GPT to generate an initial set of test cases, which are evaluated by test quality metrics. The evaluation results, together with complementary information, are crafted as feedback to GPT for the next iteration of test generation. The process stops when the test quality meets the expectation. Next, we elaborate on the design concepts of each part of the workflow.

1) Assignment Description as the Context: Motivated by insight 1, we use the textual description of assignment re-

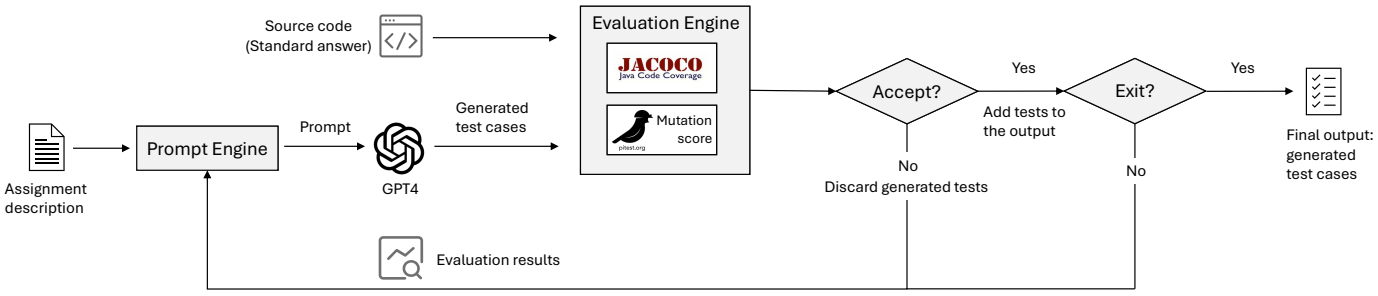


Fig. 2: The workflow of the augmented approach `AugGPT`.

quirements in the initial prompt. The description, identical to the one released to students, includes all the information needed to solve the problem. We assume that GPT can benefit from this "first-hand" information to understand assignment requirements directly, hence reducing the misinterpretation errors of the plain approach.

2) Syntax and Semantic Constraints in the Prompt: Motivated by insights 2 and 3, we add 11 constraints in the prompt to ensure the syntactic correctness and semantic depth of GPT-generated tests, as shown in Figure 3. Specifically,

- Constraints 1-4: ensure that the generated tests are syntactically correct without compilation errors.
- Constraints 6-9: guide GPT to generate a sufficient number of semantically correct tests with a certain level of complexity. Constraint 9 also ensures the readability of the generated tests.
- Constraints 10-11: define the output format.

Other than the normal prompt shown in Figure 3, the system prompt is also rewritten to better prepare GPT for the specific task. For example, the model is asked to understand and utilize the assignment document along with the feedback information, which is described next.

3) Test Quality as Feedback: Motivated by insight 4, a feedback strategy is adopted to improve GPT's test generation performance. Specifically, the tests generated in the previous iteration are executed on the code of the standard answer, which produces the following information that is provided as the feedback for the next iteration:

- The statement coverage score, together with the code block that is not covered by the generated tests.
- The mutation score, together with the mutants that the generated tests failed to kill.
- The compilation error log.

In addition to the above information, the feedback prompt also contains the the same syntactic and semantic constraints in Figure 3, with one additional constraint (Constraint 5) that instructs GPT to focus on testing the uncovered statements.

4) Rejection Strategy: Motivated by insight 5, we applied a rejection strategy to filter out undesired tests. Basically, the newly generated tests will be rejected if neither the statement coverage nor the mutation score is improved compared to the previous iteration, indicating that the newly generated tests do not really contribute to the test effectiveness. The rejected tests will not be included in the final output of `AugGPT`.

We maintain a *rejection counter* to track the number of rejections, which increments by one for each rejection and is reset to 0 if the tests are accepted. However, if the rejection counter reaches a threshold, `AugGPT` will accept the tests anyway. This rule sets an upper bound of rejections to prevent infinite loops, in case `AugGPT` is having trouble finding better tests. In general, this strategy guides GPT to maximize test effectiveness while minimizing test redundancy.

5) Exit Strategy: While the results of generated tests are given to GPT to generate new tests, hence forming a feedback loop, we need an exit strategy for when the generated tests finally meet the expectation. Currently, `AugGPT` terminates the process if one of the following conditions are met:

- The quantity of generated tests meets the expectation.
- The number of iterations exceeds a threshold.
- The test quality metrics decline or improve by a minimum threshold for N consecutive iterations.

The first and second conditions set an upper limit on the running time of `AugGPT`, while the third aims to interrupt the system as soon as it exhibits signs of performance degradation.

B. Implementation

We implement `AugGPT` as a prototype tool, which consists of the major components illustrated in Figure 2. Specifically, we used the `PDFBox` package [32] to parse assignment descriptions, which are typically in PDF format, to textual input. For LLM implementations, `AugGPT` still uses the GPT-4o-mini model of ChatGPT as it demonstrates certain potentials in the plain approach. For the communication with the LLM, we leveraged the ChatGPT Java SDK [33] to support the usage of OpenAI API. For the calculation of coverage and mutation scores, `AugGPT` also uses the same version of `JaCoCo` and `Pitest` as the plain approach.

In addition to calculating test quality metrics, the *Evaluation Engine* in `AugGPT` also preprocesses the generated tests to automate the compilation and execution processes. For instance, `AugGPT` adds random suffixes to the names of the generated tests in case they have duplicate names that result in compilation errors. `AugGPT` also annotates the failed tests with the `@Disabled` annotation, so that they will not stop the other tests from executing by throwing runtime errors.

```
1. Use Junit5, especially org.junit.jupiter.api.Assertions. Do NOT use @BeforeEach or global
    variables. Do NOT use List.of(). Try to test through calling public methods.
2. Do NOT use ANY method NOT PRESENTED in the DOCUMENT, which will cause compilation error. Try
     to test without using it.
3. PAY ATTENTION ON THE CORRECTNESS with the help of the document and general knowledge.
4. Do NOT use getter/setter, instead, use REFLECTION to access PRIVATE fields if getter/setter
    are not provided in the document. Pay attention to whether the data type is object or
    primitive and whether the method/field is defined in its superclass or class.
5. Try to test the PARTLY COVERED and UNCOVERED lines provided in the coverage information.
6. MUST generate at least 4 tests.
7. Each test MUST contain COMPLEX logic with COMPOSITE method call chains, with at least 8
    lines each, covering a wide range of methods, classes and cases.
8. Focus on the COMPLEX LOGIC TEST. The LOGIC should be completely based on the description of
    the document. The test should FOCUS on the COVERAGE INFORMATION to cover more cases.
9. Use detailed comments to derive the test logic step by step and write down the process
    before the corresponding instruction. The assertion result should be derived from the
    document.
10. You MUST put all tests in a single class, with @Test public void.
11. ONLY show me the implementation, NO extra words. Wrap the code with a markdown code block.
```

Fig. 3: Syntax and semantic constraints used in the prompt of AugGPT.

The exit strategy is implemented as the configurations of AugGPT, allowing users to set thresholds for test quantity, quality, and loop iterations. For the purpose of tool evaluation, we use the first exit strategy, which is for AugGPT to generate the same number of test cases as the plain approach in order to make a fair comparison. For the rejection strategy, we set the rejection counter threshold to 4.

Finally, AugGPT outputs all the generated tests. Users can configure AugGPT to include also the failed tests with the @Disabled annotation. The reason is that, in addition to the failure reasons described in Figure 1, failed tests may also indicate real bugs in the source programs (i.e., the standard answers) that are used as the input to the *Evaluation Engine*.

RQ2(a): To improve the plain approach, the augmented approach 1) uses assignment description to reduce misinterpretation errors; 2) adds syntax instructions to reduce compilation errors; 3) adds semantic constraints to ensure the semantic complexity of the generated tests; 4) adopts informative feedback to improve the usability and effectiveness of the generated tests, and 5) adopts rejection and exit strategies to further safeguard the test quality.

V. Evaluation

In this section, we present the evaluation results of AugGPT on the same dataset used for the plain approach. We also report a case study that applies AugGPT to over a thousand of students' assignment submissions for automatic assessment.

A. Usability of the Generated Tests

For a fair comparison, AugGPT was configured to generate the same number of test cases ($N = 48$) for each assignment used to evaluate the plain approach. Similar to Section III-C, we analyzed the failure reasons for these generated tests. Figure 1 shows the overall distribution while Table IV reports the detailed results for individual failure types. Compared to the plain approach, AugGPT significantly reduced the compilation error rate by 85%. In particular, the compilation errors in the tests generated for assignments *Course* and *Chess* have dropped to zero (Table IV). These results indicate that the syntax constraints we added to the prompt could be effective in improving the syntax correctness of the tests generated by the LLM.

Compared to the plain approach, AugGPT also reduced the misinterpretation error rate by 28%, indicating that the strategy of incorporating assignment description may assist the LLM in better understanding the assignment intent. Nevertheless, as shown in Table IV, this improvement is more profound for *Chess*, with nearly half of the misinterpretation errors eliminated. In contrast, other assignments show only marginal improvements in this regard. Thus, we speculate that the impact of assignment descriptions may be more significant for complex assignments such as *Chess*, which involve intricate rules and require strict adherence in the test cases.

Interestingly, the logic errors in the tests generated by AugGPT increased from 11% to 19% compared to those generated by the plain approach. In addition, this trend is consistent for four out of the six assignment subjects, as shown in Table IV. One possible reason is that, solving commonsense and mathematical problems has been a well-known challenge for LLMs [34] [35], and AugGPT's incorporation of various strategies may further amplify the problem.

B. Effectiveness of the Generated Tests

According to Table III, when generating the same number of test cases, AugGPT outperforms GPT in the both the statement coverage and mutation scores for all of the assignments. On average, the tests generated by AugGPT achieve 1.7x the statement coverage and 2.1x the mutation score compared to those generated by GPT, indicating that AugGPT covered a wider range of source code and had a higher likelihood of

detecting errors. Table III also shows that the assignments *Chess* and *Canvas* exhibit the most notable improvement. For instance, the tests generated by AugGPT for *Chess* yield 4.6x greater coverage and 6.0x higher mutation scores than those generated by GPT. Interestingly, *Chess* and *Canvas* have the largest number of tokens among all assignments (Table I), possibly indicating that the effectiveness of AugGPT is more evident for larger and more complicated programs.

Compared to the Human approach, AugGPT has equal or higher statement coverage except for *Chess*. In general, the average statement coverage of test cases generated by AugGPT (0.84) is slightly lower than that of the human-written tests (0.86). Our further inspection shows that human-written tests tend to be superior to AugGPT-generated tests in covering sophisticated usage scenarios, especially for complicated assignments such as *Chess*. Also, the Human approach has less redundancy compared to AugGPT, since it achieved a better result with fewer test cases.

Meanwhile, the size of the tests generated by AugGPT is generally larger than that of the plain approach and closer to those written by humans, as shown in Table II. This observation indicates the potential usefulness of the semantic constraints added to AugGPT, which may guide the LLM to generate tests with greater semantic depth.

Finally, combining human-written tests and the tests generated by AugGPT yields the best test effectiveness of all the settings, as indicated by the **AugGPT+Human** rows in Table III. In particular, the combined results of AugGPT+Human exceed those achieved by AugGPT alone or Human alone in all cases. For example, the statement coverage of *Canvas* for Human and AugGPT is 0.79 and 0.82, respectively. For AugGPT+Human, the coverage metric is 0.87, higher than each of the individual results. This observation indicates that human-written tests and AugGPT-generated tests complement each other, without one completely dominating the other.

RQ2(b): Compared to the plain approach, AugGPT reduced the compilation error rate by 85% and the misinterpretation error rate by 28% in the generated tests, while enhancing the statement coverage and mutation scores by 1.7x and 2.1x, respectively. The tests generated by AugGPT can also cover program behaviors that are not covered by human-written tests, indicated by the increased test-effectiveness metrics on the combination of the generated and human-written tests.

C. Case Study

To further evaluate the practicality of the augmented approach, we conducted a case study that applies AugGPT to assess real-world programming assignment submissions from students. Specifically, we collected 1296 student submissions that passed all human-written tests on the six assignments and then executed the $48 \times 6 = 288$ test cases generated by AugGPT on these submissions.

As shown in Table V, AugGPT reported errors on 636 submissions. Through manual inspection, we found that 169 (26.6%) of these submissions are indeed buggy. Specifically, we observed two types of bugs, *functional bugs* and *missing-requirement bugs*, which are elaborated next.

TABLE IV: The number of test failures for each approach.

Assignment	Approach	Compilation errors	Logic errors	Misinterpretation
Calendar	GPT	8	4	9
	AugGPT	1	9	9
Treasure	GPT	7	7	3
	AugGPT	1	14	2
Course	GPT	9	2	1
	AugGPT	0	5	2
Canvas	GPT	0	15	12
	AugGPT	0	14	10
Shop	GPT	16	0	2
	AugGPT	7	0	0
Chess	GPT	18	2	25
	AugGPT	0	13	14

TABLE V: Errors detected by AugGPT-generated tests on real-world programming assignment submissions.

	Total submissions	Submissions w/ reported errors	Submissions w/ real bugs
Calendar	191	189	46
Treasure	164	27	27
Course	180	120	18
Canvas	198	42	36
Shop	295	3	3
Chess	268	255	39
Total	1296	636	169

Functional bugs refer to errors that produce incorrect results, indicating that the functionality of the submitted assignment is not working as required. Figure 4a shows an example of buggy submissions for the assignment *Calendar*. The student's implementation on the function addDays incorrectly outputs 2023-02-29 as the result of adding one day to 2023-02-28, while the expected output should be 2023-03-01. This bug was detected by the tests generated by AugGPT, but passed the human-written tests that overlooked such edge cases. The functional bugs accounted for approximately 75% of the real bugs detected by AugGPT.

Missing-requirement bugs refer to issues in the assignments where necessary elements or features are not implemented as required (e.g., missing methods or fields). Figure 4b shows a submission of *Calendar*, in which the fields day, month, year required in the assignment description were not defined. This bug escaped human-written tests but was identified by AugGPT's test through reflection. The missing-requirement bugs accounted for 25% of the real bugs.

On the other hand, among the 636 submissions on which AugGPT reported errors, 467 (73.4%) are "false alarms" that do not correspond to real bugs in assignment submissions. As shown in Table V, most of the "false alarms" occur in the assignments *Calendar*, *Course* and *Chess*. We manually in-

```
// current fields: month=2, day=28, year=2023
// test input: days = 1
// expected result: 2023-03-01
// actual result: 2023-02-29
public void addDays(int days){
    int[] mon
        ={31,29,31,30,31,30,31,31,30,31,30,31};
    day+=days;
    while(day>mon[month-1]){
        if(isRunnian(year)) mon[1]=29;
        else mon[1]=28;
        day-=mon[month-1];
        month++;
        if(month>12){
            year++;
            month\%=12;
        }
    }
}
```

(a) A functional bug from *Calendar*

```
public class MyDate {
    // not following assignment requirement:
    // missing fields ``day'', ``month'', ``year''

    LocalDate date;
    String fmt = ``yyyy-MM-dd'';
    DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern(fmt);

    public MyDate(int y, int m, int d){
        date = LocalDate.of(y,m,d);
    }

    public void addDays(int d){
        date = date.plusDays(d);
    }

    // ...
}
```

(b) A missing-requirement bug from *Calendar*

```
// source program
public class ChessboardPoint {
    private int x,y;
    public ChessboardPoint(int x, int y) {
        this.x = x; this.y = y;
    }
    // ...
}

@Test
public void test(){
    // Setup a chessboard and get a Pawn at (1, 1)
    // ...
    List<ChessboardPoint> movs = pawn.canMoveTo();
    // Failed because ChessboardPoint did not
        override the ``equals'' method, hence the
        .contains() cannot compare correctly.
    assertTrue(movs.contains(
                new ChessboardPoint(2, 1)));
    assertTrue(movs.contains(
                new ChessboardPoint(3, 1)));
}
```

(c) An implicit assumption example from *Chess*

```
// source program
public class CourseManager {
    private boolean ifOpen;

    public void setIfOpen(boolean ifOpen) {
        this.ifOpen = ifOpen;
    }
// ...
}


@Test
public void test() throws Exception {
    // ...
    // Test fails because the parameter list of
    // setIfOpen should be ``Boolean'',
    // rather than ``boolean''
    Method method = CourseManager.class
            .getDeclaredMethod(``setIfOpen'',
                               Boolean.class);
    //...
}
```

(d) An overrigid requirement example from *Course*

Fig. 4: Examples of the real bugs and false alarms reported by `AugGPT`-generated tests.

spected these assignments and found that a primary reason for false alarms is GPT's tendency to impose implicit assumptions that are not explicitly required in the assignment description.

Figure 4c shows an example from *Chess*. `AugGPT` invokes the `contains` method to assert the expected moves for a `ChessboardPoint` object (e.g., a pawn), which implicitly assumes that this class should override the `equals` method. However, overriding the `equals` method is not required in the assignment description. Hence, the test failure does not indicate real bugs in the assignment submission. For *Chess*, 203 out of the 255 (80%) test failures were due to this implicit assumption.

In addition to making implicit assumptions, GPT may also adhere too rigidly to the assignment descriptions. Figure 4d shows an `AugGPT`-generated test for the assignment *Course*, which asserts that the `setIfOpen` method in the `CourseManager` class must accept a `Boolean` parameter. However, using the boxed type `Boolean` is mentioned in the assignment description but is not strictly required. As a result, many students used the primitive `boolean` type in their submissions, which we considered fine but failed the tests generated by `AugGPT`. For *Course*, 99 out of the 120 (82.5%) test failures were due to this reason.

In general, `AugGPT` demonstrates practical effectiveness in generating unit tests for programming assignments, successfully detecting real bugs in 169 submissions that passed human-written tests, with an accuracy of 26.6%. On the other hand, 73.4% of the errors reported by `AugGPT`-generated tests turned out to be false alarms, largely due to the implicit assumptions and rigid requirements imposed by GPT during the test generation process.

RQ3: When executed on 1296 students' submissions of the subject assignments, the tests generated by AugGPT were able to detect real bugs in 169 submissions that are either incorrect or incomplete with respect to assignment requirements. More importantly, these bugs were originally not detected by the human-written tests from the teaching team. This result demonstrates the value of AugGPT in assisting instructors to create more effective unit tests for assignment evaluation.

VI. Discussions

A. Classroom Integration

In the future, we plan to adopt the AugGPT tool in our programming courses. Instructors and teaching assistants may use the tool to generate the JUnit test suite for programming assignments from scratch, or augment existing human-written tests for better quality. In either case, the teaching team could save time and effort spent on writing and refining unit tests.

In addition to educators, students may also benefit from this approach. For instance, students may use the unit tests generated by AugGPT to identify bugs in their programs, which potentially promotes active learning and enhances students' problem-solving skills.

B. Considerations for Users

As discussed in Section V, there are also potential pitfalls associated with the use of AugGPT. One major concern is the false-alarm test failures, where correct assignment implementation might be incorrectly flagged as erroneous. To mitigate this issue, we plan to integrate human guidance into the feedback loop to better adapt the system to different contexts of assignments. For example, users may explicitly tell the LLM to avoid testing certain cases that are not required in the assignment.

C. Threats to Validity

1) Internal Validity: In this study, the internal validity may be threatened by the hyper-parameters of the model, such as "top_p" and "temperature", which are provided by the OpenAI API [28] to regulate the behavior of the LLM. To maintain the fairness of our comparative analysis, these parameters were held constant across both the plain and the augmented approach, and therefore alleviated the threat.

Another threat to internal validity is the assumption that the standard program is flawless when assessing our system's performance. If the standard program contains errors, it might compromise the perceived quality of the generated tests. To mitigate this threat, we utilized the assignments and standard answers that are already published and reviewed by both the instructors and students. Future studies might assess how varying source programs affects test generation robustness.

In addition, due to the randomness of GPT's output, the quality of the generated unit tests may vary in different launches, which may require further investigation.

2) Construct Validity: To evaluate the usefulness of the proposed approaches, we mainly focus on the compilation errors and runtime errors of the generated unit tests. However, other metrics, such as code readability, are not measured systematically in this study, which may miss certain code patterns related to usefulness and user experience in real-world contexts, thereby threatening the construct validity. We used only statement coverage and mutation score to evaluate the effectiveness of the generated tests, which might be biased. Additional metrics, such as branch coverage and test redundancy, might be further integrated to reduce the bias.

3) External Validity: We used GPT-4o mini as the LLM implementation in our proposed approaches, since GPT-4 is one of the most popular large language models. However, the conclusions of our study may not fully generalize to other LLMs like Llama-3 [36]. Second, as our approaches were designed and evaluated specifically for Java programs, the experiment results may not generalize to assignments written in other programming languages. Finally, our evaluation dataset involves assignments from introductory-level programming courses. However, how our approach performs for complex assignments from advanced programming courses, such as Data Structure and Algorithm Analysis, requires further research.

VII. Conclusions

In this work, we investigate how LLMs can be used to generate high-quality unit tests for programming assignments. Using GPT-4 as the LLM implementation, we observed that LLMs in vanilla settings may not be a good choice for this task, often producing tests that are unusable or less effective than human-written tests. Instead, more sophisticated strategies should be adopted to reduce the syntax errors of LLM-generated tests and increase test effectiveness.

We proposed an approach that integrates multiple quality-control strategies on top of the vanilla process and implemented a prototype tool AugGPT. On a large dataset of real-world programming assignments and students' submissions, AugGPT not only significantly enhanced the usability of generated tests, but also augmented human-written tests in terms of statement coverage and mutation scores. In addition, AugGPT identified new bugs in 169 students' submissions that the human-written tests failed to detect, further demonstrating the practical benefits of our proposed approach. In the future, we plan to adopt this approach in programming courses to improve the work efficiency of instructors and the learning experience of students.

To support reproducibility and further research, we have made the experimental data and the implementation of AugGPT publicly available at Zenodo [37].

Acknowledgment

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] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Computer Science Education*, vol. 15, no. 2, pp. 83–102, 2005, doi: 10.1080/08993400500150747. [Online]. Available: https://doi.org/10.1080/08993400500150747

[2] S. Combéfis, "Automated code assessment for education: review, classification and perspectives on techniques and tools," *Software*, vol. 1, no. 1, pp. 3–30, 2022.

[3] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon, "On automated grading of programming assignments in an academic institution," *Computers & Education*, vol. 41, no. 2, pp. 121–131, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360131503000307

[4] "Github copilot," https://github.com/features/copilot, 2024, accessed: 2024-09-04.

[5] "Cursor: The ai code editor," https://www.cursor.com/, 2024, accessed: 2024-09-04.

[6] E. Frankford, C. Sauerwein, P. Bassner, S. Krusche, and R. Breu, "Ai-tutoring in software engineering education," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ser. ICSE-SEET '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 309–319. [Online]. Available: https://doi.org/10.1145/3639474.3640061

[7] N. Kiesler, D. Lohr, and H. Keuning, "Exploring the potential of large language models to generate formative programming feedback," 2023. [Online]. Available: https://arxiv.org/abs/2309.00029

[8] Y. Xue, H. Chen, G. R. Bai, R. Tairas, and Y. Huang, "Does chatgpt help with introductory programming?an experiment of students using chatgpt in cs1," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ser. ICSE-SEET '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 331–341. [Online]. Available: https://doi.org/10.1145/3639474.3640076

[9] J. Zhang, J. P. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen, "Pydex: Repairing bugs in introductory python assignments using llms," vol. 8, no. OOPSLA1, apr 2024. [Online]. Available: https://doi.org/10.1145/3649850

[10] H. Wright, T. D. Winters, and T. Manshreck, *Software Engineering at Google*, 2020.

[11] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 2007, Conference Proceedings, pp. 815–816, export Date: 06 April 2024; Cited By: 384. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-42149184208&doi=10.1145%2f1297846.1297902&partnerID=40&md5=3004f733bf41c3d93a76bff884491c09

[12] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2011, Conference Proceedings, pp. 416–419, export Date: 06 April 2024; Cited By: 719. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-80053202643&doi=10.1145%2f2025113.2025179&partnerID=40&md5=8e0d6bb9574c9de9d57928530832bf3b

[13] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 263–272.

[14] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, pp. 1–27, 2024. [Online]. Available: https://dx.doi.org/10.1109/tse.2024.3368208

[15] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *arXiv preprint arXiv:2302.10352*, 2023.

[16] S. Hashtroudi, J. Shin, H. Hemmati, and S. Wang, "Automated test case generation using code models and domain adaptation," *arXiv preprint arXiv:2308.08033*, 2023.

[17] V. Li and N. Doiron, "Prompting code interpreter to write better unit tests on quixbugs functions," *arXiv preprint arXiv:2310.00483*, 2023.

[18] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3660783

[19] OpenAI, "Chatgpt," https://chat.openai.com/, 2024, large language model.

[20] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[21] T. Lehtinen, C. Koutcheme, and A. Hellas, "Let's ask ai about their programs: Exploring chatgpt's answers to program comprehension questions," ser. ICSE-SEET '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 221–232. [Online]. Available: https://doi.org/10.1145/3639474.3640058

[22] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: Promises & perils," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Apr. 2023. [Online]. Available: http://dx.doi.org/10.1109/ICSTW58534.2023.00078

[23] J. Heo, H. Jeong, D. Choi, and E. Lee, "Referent: Transformer-based feedback generation using assignment information for programming course," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2023, pp. 101–106.

[24] N. Kshetri, "The economics of generative artificial intelligence in the academic industry," *Computer*, vol. 56, no. 08, pp. 77–83, aug 2023.

[25] H. L. Fwa, "Experience report: Identifying common misconceptions and errors of novice programmers with chatgpt," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ser. ICSE-SEET '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 233–241. [Online]. Available: https://doi.org/10.1145/3639474.3640059

[26] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic generation of programming exercises and code explanations using large language models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ser. ICER '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 27–43. [Online]. Available: https://doi.org/10.1145/3501385.3543957

[27] M. Jukiewicz, "The future of grading programming assignments in education: The role of chatgpt in automating the assessment and feedback process," *Thinking Skills and Creativity*, vol. 52, p. 101522, 2024.

[28] OpenAI, "Openai api reference," Website, 2024, https://platform.openai.com/docs/api-reference/chat/create.

[29] "Jacoco," 2024. [Online]. Available: https://github.com/jacoco/jacoco

[30] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[31] H. Coles, "Pitest," 2024. [Online]. Available: https://github.com/hcoles/pitest

[32] A. S. Foundation, "Apache pdfbox," 2024. [Online]. Available: https://github.com/apache/pdfbox

[33] "Chatgpt java sdk," 2023. [Online]. Available: https://github.com/Grt1228/chatgpt-java

[34] N. Bian, X. Han, L. Sun, H. Lin, Y. Lu, B. He, S. Jiang, and B. Dong, "Chatgpt is a knowledgeable but inexperienced solver: An investigation of commonsense problem in large language models," *arXiv preprint arXiv:2303.16421*, 2023.

[35] V. Cheng and Z. Yu, "Analyzing ChatGPT's mathematical deficiencies: Insights and contributions," in *Proceedings of the 35th Conference on Computational Linguistics and Speech Processing (ROCLING 2023)*, J.-L. Wu and M.-H. Su, Eds. Taipei City, Taiwan: The Association for Computational Linguistics and Chinese Language Processing (ACLCLP), Oct. 2023, pp. 188–193. [Online]. Available: https://aclanthology.org/2023.rocling-1.22

[36] Meta, "Llama 3," Website, 2024, https://www.llama.com/.

[37] "Replication package," 2024, https://doi.org/10.5281/zenodo.13858438.