EMAN ABDULLAH ALOMAR, Stevens Institute of Technology, USA

Large Language Models (LLMs), such as ChatGPT, have become widely popular for various software engineering tasks, including programming, testing, code review, and program comprehension. However, their impact on improving software quality in educational settings remains uncertain. This paper explores our experience teaching the use of Programming Mistake Detector (PMD) to foster a culture of bug fixing and leverage LLM to improve software quality in the classroom. This paper discusses the results of an experiment involving 155 submissions that carried out a code review activity of 1,658 rules. Our quantitative and qualitative analysis reveals that a set of PMD quality issues influences the acceptance or rejection of the issues, and design-related categories that take longer to resolve. Although students acknowledge the potential of using ChatGPT during code review, some skepticism persists. Further, constructing prompts for ChatGPT that possess clarity, complexity, and context nurtures vital learning outcomes, such as enhanced critical thinking, and among the 1,658 issues analyzed, 93% of students indicated that ChatGPT did not identify any additional issues beyond those detected by PMD. Conversations between students and ChatGPT encompass five categories, including ChatGPT's use of affirmation phrases like 'certainly' regarding bug fixing decisions, and apology phrases such as 'apologize' when resolving challenges. Through this experiment, we demonstrate that code review can become an integral part of the educational computing curriculum. We envision our findings to enable educators to support students with effective code review strategies, increasing awareness of LLMs, and promoting software quality in education.

$\label{eq:concepts: Concepts: Applied computing $$ \rightarrow Education; $$ of tware and its engineering $$ \rightarrow Software maintenance tools. $$$

Additional Key Words and Phrases: large language models, education, bugfix, static analysis, code quality

ACM Reference Format:

1 INTRODUCTION

Linting is a code inspection practice in which developers use static analysis to identify bad coding patterns, known as issues. These issues can hinder the quality of the code, making it harder to understand and more prone to errors. Since their inception, linters have been introduced early to students to help them conceptualize the avoidance of poor programming practices [33]. Yet, the linters output is in the form of warnings with no recommended fix. Due to the non-actionable nature of these warnings [57], and the lack of their comprehension [56], many developers end up considering them as false positives [22].

However, the rise of Large Language Models (LLMs), such as the Generative Pre-trained Transformer (the core model behind ChatGPT), has gained popularity for its ability to generate responses and design solutions for a variety of input problems, including education. This technology has sparked a debate about its potential benefits and drawbacks for

Author's address: Eman Abdullah AlOmar, ealomar@stevens.edu, Stevens Institute of Technology, Hoboken, New Jersey, USA.

© 2025 Association for Computing Machinery.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

student learning [38]. With diverse applications in source code, including code quality [4, 52], programming [10], bug management [21], and program comprehension [31], it presents an opportunity to bridge the gap between developers and the proper adoption of static analysis. In this context, little is known about ChatGPT's ability to help students effectively comprehend and address static analysis outcomes, as part of learning code inspection.

Our paper reflects on the experience of integrating ChatGPT in the code linting process, to support students with their task of debugging and improving the quality of existing systems. The Programming Mistake Detector $(PMD)^1$ is a state-of-the-art static analysis plugin that identifies potential code problems, such as code violations and bad practices. It helps developers maintain code quality by providing a range of pre-defined rules that can be customized according to project needs. In this study, students are instructed to utilize PMD to examine an open-source system (code that is not theirs) and pinpoint potential problems in various categories such as documentation, design, and security. For each identified issue, students are required to discuss whether it is worth fixing and, if so, propose a fix through code modification. Students may use ChatGPT as a helper in deciding whether to address an issue and determining the appropriate corrective measure. This involves utilizing the language model's explanatory capabilities alongside its code generation and debugging functions. These experiments aim to fulfill educational objectives related to the improvement of code debugging skills of students, which are typically acquired in industry [28]. In addition, it introduces students to poor coding practices and methods for their refactoring. Furthermore, it trains students not only to use ChatGPT but also to review its recommendations and reason over their validity, with respect to analyzed code, before making a decision of whether and how it should be fixed. This latter objective is particularly significant, as it gradually elevates students' inherent assumption of ChatGPT's ability to hold the ground truth. In fact, students discover how the language model's code recommendations might not resolve the issues identified by the PMD tool.

This study builds on recent work in the field by offering more details on the approach and design [4]. It also addresses three new research questions and expands the study to include a total of 155 students and 1,658 PMD rulesets. This paper advances the adoption of static analysis warnings and leverages LLMs to enhance software quality in educational settings by (i) designing a practical assignment aimed at improving software systems' quality and (ii) detailing the experience with the PMD tool and ChatGPT within a Software Quality Assurance course taken by 155 undergraduate and graduate students. As part of the contributions of this paper, we provide the assignment description and the tool documentation for educators to adopt and extend².

The remainder of this paper is organized as follows: Section 2 reviews the existing studies related to the usage of ChatGPT in software engineering tasks. Section 3 outlines our experimental setup in terms of data analysis and research questions. Section 4 discusses our findings, while the reflection is discussed in Section 5. Section 6 captures any threats to the validity of our work, before concluding with Sections 7 and 8.

2 RELATED WORK

Recent studies utilized ChatGPT for different software engineering tasks. Due to the importance of the topic, several domains for the use of ChatGPT have been proposed, including code quality [51, 52], programming [10, 29, 45, 46], testing [41], bug management [21, 44, 53], security [24], and program comprehension [31]. In this section, we are only interested in research on the use of ChatGPT to solve software engineering tasks, summarized in Table 1.

White *et al.* [51, 52] presented prompt design techniques for software engineering using patterns to solve common problems when using LLMs. The study provides a catalog of patterns for software engineering (*e.g.*, change request

¹https://pmd.sourceforge.io

²https://refactorings.github.io/education/

Manuscript submitted to ACM

Study	Year	Context	Focus	Dataset	Evaluation
White et al. [51, 52]	2023	Code quality	Prompt design patterns	N/A	N/A
Biswas [10]	2023	Programming	Program assistant	N/A	N/A
Siddiq et al. [41]	2023	Testing	Unit tests generation	354 classes	Codex/CodeGen/ExpertEval/Evosuite
Sun et al. [45]	2023	Programming	Code summarization	14,918 code samples	SOTA(NCS/CodeBERT/CodeT5)
Haque & Li [21]	2023	Bug management	Debugging & bug fixing	N/A	N/A
Sobania et al. [44]	2023	Bug management	Bug fixing	40 code samples	Codex/CoCoNut/APR
Tian et al. [46]	2023	Programming	Code generation/summarization/repair	4,345 code samples	Codex/Refactory
Liu et al. [29]	2023	Programming	Code generation	HumanEval [12]	19 LLMs
Khoury et al. [24]	2023	Security	Code generation	21 classes	N/A
Xia & Zhang [53]	2023	Bug management	Bug fixing	80 code samples	CodexRepair/selfAPR/RewardRepair
					Reorder/TBar/CURE/COCONut
Ma et al. [31]	2023	Program comprehension	Code semantics	2,327 code samples	Tree-Sitter/CodeBERT/Slither
					ExpertEval/Frama-C/Patches scripts
This work		Education	Debugging & bug fixing	1,658 code samples	ExpertEval

Table 1. Related work on ChatGPT to solve software engineering tasks.

simulation, code clustering, principled code, etc) that classifies patterns according to the types of problems they solve. Additionally, the authors explored several prompt patterns developed to improve code quality, refactoring, requirements elicitation, and software design. The authors concluded that the depth of LLMs capabilities is not widely appreciated. Also, human involvement and expertise are needed to effectively leverage LLMs for software automation. Building on these findings, the current research explores how ChatGPT specifically influences bug fixing skills among students. In another study, Biswas [10] discussed an overview of ChatGPT as a language model developed by OpenAI that offers a wide range of computer programming capabilities. These capabilities include code completion, correction, prediction, error fixing, optimization, document generation, chatbot development, text-to-code generation, and technical query answering. The author highlighted the ability of ChatGPT to provide explanations and guidance to users and concluded that it is a powerful tool for the programming community. This study aims to analyze how ChatGPT can supplement collaborative efforts and facilitate a deeper understanding of programming concepts. On the other hand, Siddiq et al. [41] explored the effectiveness of LLMs in generating unit tests using two benchmarks: HumanEval and Evosuite. The authors evaluated the models based on compilation rates, test correctness, test coverage, and test smells. Their findings show that Codex model achieved above 80% coverage for the HumanEval dataset. Still, no model had more than 2% coverage for the EvoSuite SF110 benchmark, and the generated tests also contained test smells such as Duplicated Asserts and Empty Tests. The current research investigates bad testing practices mainly chosen by students and how ChatGPT contributes to the resolutions. Sun et al. [45] utilized ChatGPT to automate code summarization by using Python dataset called CSN-Python and comparing it with several state-of-the-art code summarization models. Their approach first explored an appropriate prompt to guide ChatGPT in generating in-distribution comments and then used such a prompt to ask ChatGPT to generate comments for all code snippets in the CSN-Python test set. The authors also adopted widely-used metrics to assess the quality of the generated comments. The findings show that ChatGPT performs worse than the state-of-the-art models. This research expands on those discussions by explicitly evaluating the implications of ChatGPT on student critical thinking in programming and code summarization tasks. Haque and Li [21] explored the capabilities of ChatGPT as a debugging tool and the best practices for integrating it into the software development workflow. Their findings show that ChatGPT is a useful tool for debugging but should be used cautiously in software development. The authors concluded that ChatGPT's ability to learn from previous debugging sessions and provide natural language suggestions can be valuable for improving code quality. However, it has several limitations, including limited domain knowledge, context awareness, and integration with development tools. To address these challenges, the present study examines the effectiveness of ChatGPT in providing real-time feedback during coding Manuscript submitted to ACM

assignments when integrating it with static analysis tools, thereby enhancing the learning experience. In another study, Sobania et al. [44] automatically analyzed bug fixing performance of ChatGPT. They evaluated ChatGPT on a standard bug fixing benchmark, QuixBugs, and compared its performance with Codex, CoCoNut, and APR. Compared to these approaches, ChatGPT fixes 31 out of 40 bugs, indicating that ChatGPT's bug fixing performance is competitive with the common deep learning approaches. However, the success rate can be further increased. This research builds on that foundation by exploring how LLMs can be integrated with other tools to improve bug fixing performance. To understand if ChatGPT is the ultimate programming assistant, Tian et al. [46] empirically experimented by emphasizing three code-related tasks: code generation, code summarization, and program repair. For code generation, ChatGPT struggles to generalize to new and unseen problems (*i.e.*, potential data leakage problem [55]) due to the uncertainty of whether or not ChatGPT has already learned the solution references in a given dataset. For program repair, ChatGPT achieves competitive results compared to Refactory. For code summarization, ChatGPT may not be able to explain the intention of the given code samples accurately. This study seeks to expand on that by assessing how LLMs can facilitate student-ChatGPT interactions during coding assignments. Liu et al. [29] answered whether the code generated by ChatGPT is correct. To do so, the authors combined LLM-based and mutation-based input generation to evaluate the accuracy of LLM-generated code. Their main results reveal that prior popular code synthesis evaluation results do not accurately reflect the performance of LLMs for code synthesis. This research builds on that foundation by exploring how students perceive the code generated by ChatGPT, potentially increasing student engagement and program comprehension. In a security-related context, Khoury et al. [24] investigated how secure the code generated by ChatGPT was by mainly asking ChatGPT to generate computer programs to evaluate the security of the resulting code. Their results suggested that ChatGPT is aware of potential vulnerabilities but can generate code that lacks robustness to certain attacks. Building on these findings, this study explores whether ChatGPT can improve code quality and security in programming tasks by providing students with reliable and secure coding solutions. Xia and Zhang [53] proposed ChatRepair to automate a conversation-driven automated program repair approach. The tool feeds the LLM with information on relevant test failures and then learns from failures and successes of earlier patching attempts of the same bug. Their tool achieved the new state-of-the-art in program repair performance. This study contributes to that conversation by evaluating the adaptability of LLMs in providing varied levels of support based on individual student needs. Ma et al. [31] performed a study evaluating ChatGPT capabilities and limitations in software engineering from three aspects: 1) syntax understanding, 2) static behavior understanding, and 3) dynamic behavior understanding. The authors focused on ChatGPT's ability to comprehend code syntax and semantic structures, including abstract syntax trees, control flow graphs, and call graphs, and assessed ChatGPT's performance on cross-language tasks involving C, Java, Python, and Solidity. They concluded that ChatGPT possesses capabilities akin to an abstract syntax tree parser. However, ChatGPT is susceptible to hallucination when interpreting code semantic structures and fabricating nonexistent facts, which underscores the need to explore methods for verifying the correctness of ChatGPT outputs to ensure its dependability in software engineering tasks. This study seeks to expand on that by investigating if ChatGPT is susceptible to hallucination when it comes to coding semantic structures, especially by analyzing potentially inappropriate results of the model, which can raise the student's awareness of its limitations.

To summarize, since LLMs, such as ChatGPT, have gained widespread popularity and usage in various software engineering tasks, their effectiveness in improving software quality in the classroom remains uncertain. Our work investigates how ChatGPT can support students in improving code quality and reflects on how students use it. This paper discusses the way in which we conducted a study in an educational setting using open-source projects with many





issues to further understand student learning capabilities in the discovery and correction of bugs and whether ChatGPT eventually would be a code companion or a code replacement.

3 STUDY DESIGN

3.1 Goal & Research Questions

We formulate the main objective of our study based on the Goal Question Metric template [49], as follows:

Analyze the use of PMD for the purpose of cultivating the culture of bug fix and demonstrating a practical case study of leveraging Large Language Models (LLMs) with respect to support educating software quality from the point view of educators in the context of undergraduate and graduate students in Computer Science and Software Engineering who analyze Java-based software projects.

According to our goal, our aim is to answer the following Research Questions (RQs):

• RQ₁. What PMD-related problems are typically selected by students?

<u>Motivation</u>: This RQ explores what type of quality issues students are addressing using the help of the model. It explores whether students naturally gravitate toward certain kinds of problems (*e.g.*, performance, or design-related issues). The findings can shed some light on the feasibility of this learning activity in educating students to perform effective code reviews and quality control.

Measurement: We report the types of issues that are being addressed, clustered by PMD ruleset categories.

• RQ₂. What type of issues typically takes longer to be fixed?

<u>Motivation</u>: This RQ investigates which PMD ruleset takes a longer time to be fixed by students, despite the use of ChatGPT. The finding raises educators' awareness of types of issues that are difficult for students to understand and address. Understanding which issues take longer to fix allows educators to pinpoint where students struggle the most (*e.g.*, whether due to the complexity of the issues, unfamiliarity with certain coding practices, or inadequacies in their debugging skills).

Measurement: We examine the resolution time taken by students to fix each ruleset category.

• RQ₃. To what extent was ChatGPT successful in addressing the students' debugging needs?

Eman Abdullah AlOmar

C • 60	FIV	D generateu	a list of t								
a 🏼 Pac		violatio	on								
- 28	Gan**				TT Violations Durniew X						
> 8	9 s	New	>			ALC 1.4	alc. 1. c			f	
> 2	4 J	Go Into			ciement	+ violations	+ violations/kLOC	 violations/Method 	Project		
> =	A F				NullAssignment	2	1.4	0.05	Gantt		
> 6	5 ¢	Open in New Window			CognitiveComplexity	11	7.6	0.25	Gantt		
> 2	5.6	Open Type Hierarchy	F4		Child in the life way of the		0.2	0.20	Gantt		
	8 A -	Show In	Alt+Shift+W >		Field/eclarationsonouidbeAtstantUtCk		1.4	0.05	Gantt		
	E E	Show in Local Terminal	>		Francis Claud anoth		0.7	0.02	Gantt		
	E) E	-			Short/ariable	122	84.1	2.77	Gantt		
	1 C 🗈	Copy	Ctrl+C		Emote CatchBlock	1	0.7	0.02	Gantt		
	5 1 Ba	Copy Qualified Name			CyclomaticComplexity	14	9.6	0.32	Gantt		
2	hwa m	Paste	Ctrl+V		AvoidLiteralsInIfCondition	27	18.6	0.61	Gantt		
2	HW 👻	Delete	Delete		UnnecessaryConversionTemporary	2	1.4	0.05	Gantt		
2 2	w 1 🐣		oune		CommentSize	17	11.7	0.39	Gantt		
2 🖉	Ho	Build Path	>		UnusedPrivateField	2	1.4	0.05	Gantt		
	Uni	Source	Alt+Shift+S >		AddEmptyString	5	3.4	0.11	Gantt		
22	abi	Reference	Ab. Child. To		LocalVariableCouldBeFinal	116	79.9	2.64	Gantt		
2.5	abi	Relactor	ART SHITT /		GodClass	1	0.7	0.02	Gantt		
2.5	abi Est	Import			RedundantFieldInitializer	6	4.1	0.14	Gantt		
1.5		Export			NPathComplexity	6	4.1	0.14	Gantt		
1.5		caporen			AccessorMethodGeneration	18	12.4	0.41	Gantt		
1.5	e	Refresh	F5		CollapsibleIfStatements	1	0.7	0.02	Gantt		
	ab7	7 Close Project 8 Close Unrelated Projects			UnusedAssignment	11	7.6	0.25	Gantt		
	abl				SingularField	1	0.7	0.02	Gantt		
1 44			_		AvoidPrintStackTrace	3	2.1	0.07	Gantt		
					UseProperClassLoader		1.4	0.05	Gantt		
lonor	cold	acted PMD too			UsecollectionisEmpty		0.7	0.02	Gantt		
siopei	2010	ected FIND tot	л <u>(</u>		LooraCounting		4.1	0.14	Gantt		
					TooMan Mathada		0.7	0.02	Gantt		
5 😂	Nut "				AvoidInstantiationObjectsIoLoops	14	9.6	0.32	Gantt		
> 🍅	Refz	Restore from Local History			CommentRequired	33	22.7	0.75	Gantt		
> 😽	refa 👓	PMD	>	Generate Reports	ConfusingTernary	6	4.1	0.14	Gantt		
> 🔂	Reff	Team	>	🛏 Clear Violation R	MissingSerialVersionUID	1	0.7	0.02	Gantt		
> 😂	rihn	Compare With	,	en Find Suspect Cut	AvoidDuplicateLiterals	1	0.7	0.02	Gantt		
> 😂 !	taxC	Castinua		and Class Vielations	UnusedFormalParameter	4	2.8	0.09	Gantt		
> 😂 '	Test	Configure	,	Clear violations	SwitchStmtsShouldHaveDefault	8	5.5	0.18	Gantt		
		Properties	Alt+Enter	Check Code	MutableStaticState	1	0.7	0.02	Gantt		
					SimplifiedTernary	2	1.4	0.05	Gantt		
		-			ControlStatementBraces	12	8.3	0.27	Gantt	2	
	4				CouplingBetweenObjects	1	0.7	0.02	Gantt	2	
		4			PrimitiveWrapperInstantiation	2	1.4	0.05	Gantt	DMD seems	a al a sa al
					Lau OfDernater	40	27.6	0.91	Gantt	PMD scanne	ea and
		Develop	er selecte	ed the	ExcessiveMethodLength	5	3.4	0.11	Gantt		
		Develope	5 3010010		 osestringsurreirorstningAppenas 	2	1.4	0.05	Gantt	reviewed	d code
		violation to be fixed		ixed	LongVariable	2	1.4	0.05	Gantt	101101101	

Fig. 2. PMD workflow.

<u>Motivation</u>: This RQ explores the ability of the language model to correctly explain the rationale of errors, and to provide potential code fixes when requested. The findings can inform educators about how ChatGPT can better support students with code improvement and whether LLMs can serve as a reliable educational tool and complement static analysis tools.

<u>Measurement</u>: We examine students' feedback to extract the necessary ratios of the language model's success in addressing their queries.

• RQ4. What strategies students use to formulate prompts that facilitate effective communication with ChatGPT?

<u>Motivation</u>: This RQ analyzes students' approaches to engineer prompts to enhance effective interaction with ChatGPT. Understanding the prompt strategies students use provides insights into how well they can articulate problems and interact with AI tools. This also sheds light on whether students need guidance in prompt engineering.

<u>Measurement:</u> We examine students' prompt engineering approach to extract the necessary aspects of the prompts.

• RQ₅. Does ChatGPT detect any other quality issues that were not detected by PMD?

<u>Motivation</u>: This RQ investigates the ability of the language model to pinpoint other quality issues that were not detected by PMD. This RQ sheds light on the effectiveness and complementarity of AI-driven and traditional static analysis approaches.

Measurement: We report students' responses to analyze quality issues that were captured by ChatGPT.

• RQ6. What are the response patterns of ChatGPT to student requests?

<u>Motivation</u>: This RQ analyzes the ChatGPT patterns and whether they help in providing insights into ChatGPT's performance and effectiveness in addressing student inquiries within an educational context. This RQ sheds light Manuscript submitted to ACM

on the refinement and enhancement of ChatGPT's capabilities to better meet the diverse needs of students and educators.

Measurement: We report ChatGPT's response patterns to analyze quality issues.

As part of this paper's contributions, we provide the assignment description, dataset, and tool documentation for educators to adopt and extend.

3.2 Course Overview

Software Quality Assurance course, available at the undergraduate and graduate levels, features two weekly sessions of 1 hour and 15 minutes each. This course delves into the principles of software quality and maintenance, introducing challenges associated with different aspects of software evolution, and providing tools to address these issues. It also explores various software analysis and testing concepts, along with the practical tools commonly used as industry standards. Students undertake several practical assignments, focusing on software quality metrics, code refactoring, bug reporting, unit and mutation testing, and managing technical debt. Course deliverables include five individual homework assignments, a scholarly article reading and presentation, and a long-term group project.

3.3 PMD

Programming Mistake Detector (PMD), an open-source static code analysis tool, reviews source code written in up to 16 different programming languages to identify frequent programming concerns, such as unsafe threading, over-complicated classes, and violations of naming conventions. These issues are categorized into eight groups: 'Best Practices', 'Code Style', 'Design', 'Documentation', 'Error Prone', 'Multithreading', 'Performance', and 'Security'. Each problem is detected using specific detection rules. PMD utilized a set of rules to evaluate the quality of the code according to the focus areas. In total, 33 configurations of the rule set were used for Java projects. The rulesets can also be customized according to the user's preference. In addition, violations of rules are measured on a priority scale of 1 to 5, with 1 being the most severe and 5 the least. In addition to the 8 categories mentioned above, PMD categorizes the rules according to five priority levels (from P1 'Change absolutely required' to P5 'Change highly optional') [27]. The tool has become popular, as it can be integrated with modern CI/CD servers. As shown in Figure 2, PMD can be run as an Integrated Development Environment (IDE) plugin, where the current project is scanned and a set of issues and violations are reported. As an illustrative example, Figure 2, shows a step-by-step scenario to automatically detect the code violation instance. Once the user opens the Java file and selects the PMD tool, it scans and analyzes the code. Once the scan is completed, the PMD tool window displays a list of identified code violations. The user can then click on a specific violation rule (*e.g.*, ExcessiveMethodLength), and apply the necessary fix.

The use of static analysis tools has been extensively studied (*e.g.*, [30, 37, 39]). The choice of PMD is motivated by different factors: its widespread use in the Java community and its maturity (*i.e.*, it has been available since 2002 [47]), and works with Java source code to find coding style problems, which is not the case with FindBugs and JLint, which work with byte code and focus on finding programming errors and neglecting programming style issues [37]. CheckStyle focuses more on readability problems compared to PMD, which tends to highlight suspicious situations in the source code [36].





Years of Experience	Programming Experi- ence (%)	Java Experience (%)	Industry Experience (%)
none	0 (0%)	0 (0%)	20 (19.6%)
< 1	4 (3.9%)	42 (41.2%)	40 (39.2%)
1-3	58 (56.9%)	51 (50.0%)	33 (32.4%)
4-6	36 (35.3%)	8 (7.8%)	7 (6.9%)
7-10	3 (2.9%)	1 (1.0%)	1 (1.0%)
> 10	1 (1.0%)	0 (0%)	1 (1.0%)

Table 2. Participant professional development experience in years.

3.4 Teaching Context and Participants

The study involves one assignment in the software quality assurance course. In this study, the course was taught in two private institutions located in the northeastern United States: Stevens Institute of Technology (Stevens) and Rochester Institute of Technology (RIT). 60 students were enrolled in the course for Stevens and 95 at RIT. Although both institutions share similar geographic locations and private statuses, the instructors collaborated closely to ensure consistency in the course content. Both instructors used the same slides and materials and met regularly to coordinate the assignment. This study was approved by the Institutional Review Board (IRB). All participants provided their informed consent before participating in the study. Before taking on the assignment, students had been introduced to various aspects of coding and design quality: (1) understanding code quality by learning concepts and metrics for assessing software quality, (2) identifying code smells to recognize poor programming practices that violate design principles, (3) mastering code refactoring with strategies to enhance software quality, and (4) managing bugs with techniques for identifying and resolving software issues. The assignment accounted for 7.5% of the final grade and was due two weeks after these four sessions.

In total, 155 students completed the assignment. This diverse set of 155 submissions allowed us to assess how ChatGPT suggestions varied between different levels of programming proficiency. This helps highlight the tool's adaptability, or lack thereof when faced with different student needs. The programming experience of the respondents ranged from 1 to more than 10 years, their Java experience ranged from 1 to 10 years, and their industry/coop experience ranged from 0 to more than 10 years. Table 2 summarizes students' experience in programming, Java experience, and industry experience.

3.5 Assignment Content and Format

Assignment Design. Initially, students were instructed to analyze a particular version of Java software that had been approved by the instructor, ensuring it is not only popular but also compiles correctly, as PMD necessitates this. The reason for allowing students to select a project is to let them choose one that is comfortable and that fits their interests. For those students who preferred not to find their own project, a pre-compiled list by the instructor was provided (refer to Table 3). These projects were chosen because they are known to contain diverse software defects. Following this, the students were instructed to set up and run PMD to assess the production code of their selected project. They could choose to use either the standalone version of the tool or the Eclipse plugin, which helps them become familiar with the coding environment while reducing the setup complexity. When PMD was executed, students needed to identify at least 10 warnings, ensuring that each applicable category was represented by at least one warning. This method guarantees exposure to a wide range of issues, from design to multithreading, and documentation. It also enriches the learning process, as students must develop various solutions rather than repeatedly using the same one. However, they had the freedom to decide which specific instances to address. This inherently requires them to review several warnings <u>Manuscript submitted to ACM</u>

in all categories, increasing opportunities for incidental learning. In addition, allowing students to choose the code fragments to examine helps to build their confidence in evaluating warnings.

Finally, the students utilize ChatGPT to resolve PMD issues, guided by an online questionnaire. For designing our survey, we adhered to the methodological guidelines by Kitchenham and Pfleeger [25]. The survey is split into two parts. The initial part consists of demographic questions about the participants. The subsequent part explores their experiences with using ChatGPT to address PMD issues and their perceptions of ChatGPT's usefulness. Following the recommendations by Kitchenham and Pfleeger [25], our survey comprises 6 open-ended questions and 16 multiple-choice questions, with an optional "Other" choice, enabling participants to express any additional thoughts not covered in the list. As observed in Table 9, each survey question was developed to focus on particular aspects of the research questions.

Pilot Study. We conducted a preliminary study involving two students to improve the experimental tools and clarify the instructions of the experiment. After this pilot study, we iteratively refined both the protocol and the assignment questions. The pilot study participants were undergraduate Software Engineering students. One key insight we gained was the importance of effective, prompt engineering when using ChatGPT, which led us to adjust the survey questions for a more detailed exploration of the topic. Consequently, we restructured the assignment and opted not to include data from the pilot study in our analysis. In summary, the students followed these steps:

- (1) Install the PMD.
- (2) Run PMD on a project of students' choice and select 10 issues of different types.
- (3) Use ChatGPT to analyze the issues and decide on whether to fix them and what is the appropriate code change action.
- (4) Report the findings for each issue: (a) the source code, (b) the type of issue, (c) how long it took to check it / fix it, and (d) the code snippet.
- (5) Add to the report a concise comment on the experience with ChatGPT (optional).

The evaluation of the artifacts of the submissions was based on 1) the assessment of the students' ability to understand the type of issues (concept understanding); and 2) the assessment of whether students have provided acceptable fixes or proper justification in both cases or accepted or rejected PMD's recommendation (program analysis and evolution). Students' subjective perception of the code was excluded from the evaluation process, as it could have introduced bias into the experiment. Specifically, students might not provide genuine feedback on the quality of the code due to the potential pressure of being graded or their desire to align their answers with what they believe instructors expect. This could lead them to fill out surveys arbitrarily rather than based on an honest assessment of the code. Furthermore, feedback was anonymous and was not mandatory to increase the magnitude of PMD and ChatGPT usage experience. Although feedback was optional, all students completed it. Figure 1 shows an overview of the setup and execution of our experiment.

3.6 ChatGPT Usage

Similar to PMD training, the students were equipped with essential background knowledge to utilize ChatGPT (version 3.5). Our focus on students' engagement with the language model led us to clearly instruct them to report all tool usage via a specified survey link. Figure 3 illustrates a PMD issue encountered during the analysis of the GanttProject³. This specific issue, labeled as ExcessiveMethodLength, was found in the paintCalendar201d(Graphics g) method of ³https://github.com/bardsoftware/ganttproject

the GanttGraphicArea.java file. PMD evaluated this as an 'Urgent' violation due to a CognitiveComplexity of 50, a CyclomaticComplexity of 39, and an NPathComplexity of 1140, each exceeding the default acceptable limits. ChatGPT recommends refactoring by breaking the extensive method into smaller, manageable parts and offering a suggested code fix. Students can implement this fix if they agree with it. As depicted in the figure, adopting the proposed solution successfully eliminated CognitiveComplexity and NPathComplexity issues, and reduced CyclomaticComplexity to 12.

3.7 Data Analysis

We analyze the responses to the open-ended question to create a comprehensive high-level list of themes by adopting a thematic analysis approach based on guidelines provided by Cruzes et al. [15]. Thematic analysis is one of the most widely used methods in the software engineering literature [2, 3, 42]. This is a technique for identifying and recording patterns (or "themes") within a collection of descriptive labels, which we call "codes". For each response, we proceeded with the analysis using the following steps: i) Initial reading of the survey responses; ii) Generating initial codes (*i.e.*, labels) for each response; *iii*) Translating codes into themes, sub-themes, and higher-order themes; *iv*) Reviewing the themes to find opportunities for merging; v) Defining and naming the final themes, and creating a model of higher-order themes and their underlying evidence. The above-mentioned steps were performed independently by two annotators. An annotator conducted the labeling of the students' comments independently of the other annotator who was responsible for reviewing the currently drafted themes. By the end of each iteration, the annotators met and refined the themes to reach a consensus. For example, the annotators showed strong agreement when coding responses related to the 'automation' theme for the following response: "The advantages of ChatGPT include its affordability, capacity for automating tedious processes". In contrast, the annotators disagreed on how to categorize certain responses between 'functionality' and 'usefulness' in the following response, "Overall I had a positive experience with ChatGPT as it was able to detect the errors that PMD reported. However, for the last 5 code segments I tested, it started generating code that produced compile errors and did not always generate code in Java on the first few attempts". To resolve this disagreement, the annotators discussed the difference between 'functionality' and 'usefulness' in this context. Functionality refers to how well the tool performs its intended tasks. In this case, the response mentions specific technical aspects of ChatGPT's performance. Usefulness refers to the overall value or benefit that the tool provides to the user, regardless of its technical performance. After review, they agreed that the response could be coded under 'functionality'. It is important to note that the approach is not a single-step process. As the codes were analyzed, some of the first cycle codes were subsumed by other codes, relabeled, or dropped altogether. As the two annotators progressed in translation to themes, there was some rearrangement, refinement, and reclassification of the data into different or new codes. We used the thematic analysis technique to address RQ₃.

4 RESULTS

4.1 What PMD-related problems are typically selected by students?

In Tables 4 and 5, we illustrate PMD rules that are perceived by students. It is worth noting the diversity of these warnings/violations, *i.e.*, they spread from warnings regarding style, code practice, and documentation, to warnings dealing with design and performance.

Project	# commits	# contributors	Domain
Ant	14,887	64	Java builder
GanttProject	4,361	38	Project management
Hutool	4,074	191	Code design
JCommander	1,009	64	Command line parsing
JFreeChart	4,218	24	Data visualization
JHotDraw	804	3	Data visualization
Log4J	12,211	137	Logging
Nutch	3,293	46	Web crawler
Rihno	4,119	80	Script builder
RxJava	6,004	289	Java VM
Xerces	6,463	5	XML parser

Table 3. The list of open-source projects used in the assignment [4].

Upon analyzing students' assignment solutions, we cluster the issues according to the PMD ruleset categories listed in the PMD official documentation⁴, namely, 'Best Practices', 'Code Style', 'Design', 'Documentation', 'Error Prone', 'Multithreading', 'Performance', and 'Security'. These categories were captured at different levels of granularity (*e.g.*, package, class, method, and attributes).

In the rest of this subsection, we provide a more in-depth analysis of these categories and the associated PMD rulesets. These categories were defined in our previous work [1], and we include them in this paper to make it self-contained.

Category #1: Best Practices. This category pertains to the guidelines that uphold widely recognized best practices, essential for a comprehensive evaluation of software quality. It helps students identify code that can violate fundamental design and coding principles or highlight areas that are potentially inefficient or difficult to maintain. Examples of the rules perceived by students as true positives include LooseCoupling, SwitchStmtsShouldHaveDefault, and JUnitAssertionsShouldIncludeMessage.

Category #2: Code Style. This category refers to the rules that enforce a specific coding style. Most prominently, brace and naming rules consist of good coding practices regarding code blocks and naming conventions. This can be illustrated briefly by rules IfStmtsMustUseBraces and ClassNamingConventions, which shows that it should be followed by braces even if it is followed only by a single instruction and class names should be in upper camel case naming conventions to improve the naming quality in the code and reflect the actual purpose of the parameters and variables.

Category #3: Design. This category pertains to the guidelines that aid in identifying design problems. Students documented design rules incorporating best practices related to the general structure of the code. By way of illustration, AvoidDeeplyNestedIfStmts rule indicates avoiding deeply nested if statements and GodClass shows the violation of single responsibility principles that increases the complexity of the code (*e.g.*, CyclomaticComplexity).

Category #4: Documentation. This category pertains to rules associated with code documentation. Documentation involves explaining code modifications in plain language, which is essential as it discloses the developer's reasoning for their coding choices. CommentRequired rule is a good illustration of this group and shows that students seem to pay attention to the quality of the code comments.

Category #5: Error Prone. This category pertains to the guidelines used to identify constructs that are either broken, extremely confusing, or prone to run time errors. A crucial element in preventing such problems is ensuring readability. If the students refactor the code to be easily read and understood, there is less chance for misunderstandings $\frac{1}{4}$ https://pmd.sourceforge.io

and coding mistakes, so students spend less time comprehending code. This is exemplified by error-prone-related rules such as AvoidDuplicateLiterals and AssignmentInOperand.

Category #6: Multithreading. This category encompasses rules that identify problems when managing multiple execution threads. For example, PMD recommends AvoidUsingVolatile to be avoided as the keyword 'volatile' is used to fine-tune a Java application and requires the expertise of the Java Memory Model.

Category #7: Performance. TThis classification pertains to guidelines that identify inefficient code. Since performance plays a vital role, students are encouraged to follow best coding standard practices such as AvoidArrayLoops and AvoidInstantiatingObjectsInLoops rules to avoid degrading the performance of the code.

For our study, we analyzed a total of 1,658 rules of students' selected issues violating 195 distinct PMD rules. The students' selection of PMD rulesets was crucial in determining whether ChatGPT could reliably identify stylistic and best practice violations in the code. Clearly depicted by Tables 4 and 5, there are various violations chosen by the students to be fixed by ChatGPT. Although our results are not intended to be generalized, as they require further experiments with larger sample sizes, our experience shows the success of ChatGPT in triggering students' critical thinking about the problems outlined in the issues. PMD has attracted them successfully to explore the various types of problems, so students can utilize ChatGPT to code the needed fix for the problem. Moreover, it achieves another goal of our maintenance class, as it trains students to comprehend and act on code that they do not own.

Looking at the PMD rules, Figure 4 depicts the percentages of PMD rules, clustered by category. As can be seen, the most common PMD ruleset category concerns 'Code Style', representing 32.3% of the issues. This observation is in line with the findings of previous studies that describe that most code reviewers look for style conformance when evaluating code quality [43]. The next most common categories are 'Best Practices', 'Design', and 'Error Prone', representing 21.1%, 17.4%, and 12.3% of the issues, respectively. This might indicate that students have different perspectives on whether developers follow the best practices, improve the architecture design of code, or make code less susceptible to errors. The categories 'Documentation', 'Performance', and 'Multithreading' had the least number of issues, with a ratio of 8.5%, 6.8%, and 1.7%, respectively.

Summary for RQ1: Among the 1,658 analyzed issues, the most common PMD ruleset category concerns 'Code Style', representing 32.3% of the issues.

4.2 What type of issues typically takes longer to be fixed?

Figure 5 shows that issues belonging to the 'Design', 'Error Prone', and 'Best Practices' categories ($\mu = 7.19$, $\mu = 3.81$, and $\mu = 3.51$, respectively) tend to be more time-consuming for students to address. Notably, 'Design' issues show a notable increase compared to other categories. Upon closer examination, we observed a trend: submissions where students could not receive support with addressing some type of 'Design' issues, specifically GodClass problems. God Classes are characterized by their excessive size and the accumulation of numerous responsibilities as they control and monitor multiple other classes. Their presence poses challenges for program maintenance and understanding [8]. We speculate that design issues take significantly longer to be resolved due to substantial modifications that must be made for God classes. In fact, fixing a God class requires its decomposition into multiple classes, which requires the separation of multiple tangled concerns while maintaining their coupling to other classes. The needed fix translates into a series of refactorings that go beyond changing one or a few instructions, such as when fixing other types of issues. When

Category	Rule	Category	Rule
	AbstractClassWithoutAbstractMethod		AbstractClassWithoutAnyMethod
	AccessorMethodGeneration	1	AvoidCatchingGenericException
	ArrayIsStoredDirectly	1	AvoidDeeplyNestedIfStmts
	AvoidPrintStackTrace	1	AvoidThrowingNullPointerException
	AvoidReassigningParameters		AvoidThrowingRawExceptionTypes
	AvoidReassigningCatchVariables	1	AvoidUncheckedExceptionsInSignatures
	AvoidReassigningLoopVariables AvoidStringBufferField		ClassWithOnlyPrivateConstructorsShouldBeFinal
			CognitiveComplexity
	AvoidUsingHardCodedIP	1	CollapsibleIfStatements
	ConstantsInInterface	1	CouplingBetweenObjects
	DefaultLabelNotLastInSwitchStmt	1	CyclomaticComplexity
	ForLoopCanBeForEach	1	DataClass
	ForLoopVariableCount	1	ExceptionAsFlowControl
_	GuardLogStatement	1	ExcessiveImports
ectices	JUnit4TestShouldUseBeforeAnnotation	5	ExcessiveMethodLength
st Pra	JUnitAssertionsShouldIncludeMessage	Desi	ExcessiveParameterList
B	JUnitTestContainsTooManyAsserts	1	ExcessivePublicCount
	JUnitTestsShouldIncludeAssert	1	FinalFieldCouldBeStatic
	LiteralsFirstInComparisons	1	FormalParameterNamingConventions
	LooseCoupling	1	GodClass
	MethodReturnsInternalArray	1	ImmutableField
	MissingOverride	1	LawOfDemeter
	OneDeclarationPerLine		LogicInversion
	PositionLiteralsFirstInCaseInsensitiveComparisons		ModifiedCyclomaticComplexity
	PositionLiteralsFirstInComparisons		MutableStaticState
	PreserveStackTrace		NPathComplexity
	PrimitiveWrapperInstantiation		SignatureDeclareThrowsException
	ReplaceEnumerationWithIterator		SimplifiedTernary
	ReplaceHashtableWithMap	1	SimplifyBooleanReturns
	ReplaceVectorWithList	1	SimplifyBooleanExpressions
	SwitchStmtsShouldHaveDefault	1	SingularField
	SystemPrintln	1	TooManyMethods
	UnusedAssignment	1	UseUtilityClass
	UnusedFormalParameter		CommentRequired
	UnusedImports	ation	CommentSize
	UnusedLocalVariable	ment	UncommentedEmptyConstructor
	UnusedPrivateField	Docu	UncommentedEmptyMethodBody
	UnusedPrivateMethod		AddEmptyString
	UseAssertSameInsteadOfAssertTrue	1	AvoidFileStream
	UseCollectionIsEmpty		AvoidInstantiatingObjectsInLoops
	UseTryWithResources	8	AvoidUsingShortType
	UseVarargs		BooleanInstantiation
	AvoidSynchronizedAtMethodLevel	Perfo	ConsecutiveAppendsShouldReuse
	AvoidUsingVolatile	-	ConsecutiveLiteralAppends
ing	DoNotUseThreads		InefficientEmptyStringCheck
hread	DoubleCheckedLocking	1	InsufficientStringBufferDeclaration
fultit	NonThreadSafeSingleton	1	IntegerInstantiation
MM L	UnsynchronizedStaticFormatter	1	LongInstantiation
	UseConcurrentHashMap	1	OptimizableToArrayCall
	UseNotifyAllInsteadOfNotify	1	RedundantFieldInitializer

Table 4. PMD rules that were chosen by students, broken down by category.

Category	Rule	Category	Rule
	AbstractNaming	Style	VariableNamingConventions
	AtLeastOneConstructor	Code	WhileLoopsMustUseBraces
	AvoidFinalLocalVariable		StringInstantiation
	AvoidPrefixingMethodParameters	ance	TooFewBranchesForASwitchStatement
	AvoidUsingNativeCode	rforn	UseArrayListInsteadOfVector
	BooleanGetMethodName	Pe	UseIndexOfChar
	CallSuperInConstructor		UseStringBufferForStringAppends
	ClassNamingConventions		AssignmentInOperand
	CommentDefaultAccessModifier		AssignmentToNonFinalStatic
	ConfusingTernary		AvoidCatchingThrowable
	ControlStatementBraces		AvoidDuplicateLiterals
	DefaultPackage		AvoidFieldNameMatchingMethodName
	EmptyControlStatement		AvoidFieldNameMatchingTypeName
	EmptyMethodInAbstractClassShouldBeAbstract		AvoidInstanceOfChecksInCatchClause
Style	FieldDeclarationsShouldBeAtStartOfClass	9	AvoidLiteralsInIfCondition
ode	FieldNamingConventions	Pron	AvoidUsingOctalValues
U U	FinalParameterInAbstractMethod	Error	CloseResource
	ForLoopsMustUseBraces		CompareObjectsWithEquals
	FormalParameterNamingConventions		ConstructorCallsOverridableMethod
	IdenticalCatchBranches		DataFlowAnomalyAnalysis
	IfElseStmtsMustUseBraces		EmptyCatchBlock
	IfStmtsMustUseBraces		EmptyIfStmt
	LinguisticNaming		EmptyStatementBlock
	LocalVariableCouldBeFinal		EmptyWhileStmt
	LocalVariableNamingConventions		FinalizeDoesNotCallSuperFinalize
	LongVariable		FinalizeShouldBeProtected
	MethodArgumentCouldBeFinal		ImplicitSwitchFallThrough
	MethodNamingConventions		MissingSerialVersionUID
	NoPackage		NonStaticInitializer
	OnlyOneReturn		NullAssignment
	PrematureDeclaration		OptimizableToArrayCall
	ShortClassName		ReturnEmptyArrayRatherThanNull
	ShortMethodName		ReturnEmptyCollectionRatherThanNull
	ShortVariable		SimpleDateFormatNeedsLocale
	UnnecessaryConstructor	1	SuspiciousEqualsMethodName
	UnnecessaryFullyQualifiedName		UnnecessaryConversionTemporary
	UnnecessaryImport	1	UseEqualsToCompareStrings
	UnnecessaryLocalBeforeReturn	1	UseLocaleWithCaseConversions
	UnnecessaryModifier	1	UseProperClassLoader
	UnnecessaryReturn	1	UselessOperationOnImmutable
	UnnecessarySemicolon	1	
	UseDiamondOperator	1	
	UseShortArrayInitializer	1	
	UselessParentheses		
	UselessQualifiedThis	1	

Table 5. PMD rules that were chosen by students, broken down by category (Cont'd).



Fig. 4. What PMD-related category have you chosen? [4]



Fig. 5. Boxplots of time taken to fix issues, clustered by PMD ruleset categories.

analyzing students' prompts for fixing for God classes, we noticed that ChatGPT was unable to propose a fix, mostly because the input windows of the model cannot receive the needed set of instructions, which represent the God class, and all its dependencies. providing a snapshot of the God class code would not be sufficient for the model to carry out a class extraction. The language model was only able to provide examples of extract class refactorings, for relatively smaller-scale classes. For instance, Figure 6 exhibits ChatGPT response to a student query asking to address a God Class issue, for a given input class. In general, ChatGPT has input limitations when grasping the various contexts in large code segments. Due to this limitation, there have been instances where ChatGPT makes suggestions based on misunderstandings or false assumptions about the code. Through this experiment, students can experience how ChatGPT performance varies based on the length of the input. Increasing the size of the input can eventually lead to catastrophic forgetting [20].

ChatGPT response:

I apologize, but I cannot modify the code as I don't have access to the entire codebase or know what the code is supposed to do. However, I can provide an example of how to address the PMD CommentSize issue based on the code snippet you provided.

Fig. 6. ChatGPT response to a student query to refactor a God Class.

On the other hand, ChatGPT provided more positive answers to student questions related to addressing issues when they are located in a continuous set of instructions (single location), such as CyclomaticComplexity. ChatGPT was able to propose the extract method refactoring to break down the methods with a high score of cyclomatic complexity.

Further, our previous work shows that design-related problems generally demand more time for resolution, indicating a more profound challenge for students [1]. These antipatterns indicate inadequate design or architectural choices, necessitating that students first grasp the intended design to identify the root causes of the antipatterns. Unlike simpler issues, this involves more than just following a few instructions; it requires analyzing the methods and classes along with their architecture. Although empirical research shows that these antipatterns markedly increase error proneness [23] and impede program understanding [9], there is no agreed-upon method for identifying [34] and remedying them [26]. Consequently, students may struggle with the subjective nature of these problems and take longer to decide on a refactoring approach.

Similar to how design patterns are widely used in modeling classes [6, 14, 40], it is important that students also learn about antipatterns. When students are educated on design patterns, they are tasked with pinpointing the essential features of common design frameworks that enhance reusability, all while maintaining Object-Oriented design principles. Yet, existing large and complex systems are known to exhibit the existence of antipatterns [35]. Consequently, students need to identify symptoms of bad design and programming practices, which is essentially problem-based learning. This educational approach uses complex practical issues (*e.g.*, antipatterns) to teach concepts (*e.g.*, design principles) [17]. A key success factor of this approach is its ability to motivate students to design various solutions to address these issues. Antipatterns stimulate students' design thinking as they work through their resolution. Since each antipattern offers several refactoring opportunities, students must justify their selections. This justification can be facilitated by evaluating design quality using structural metrics, both before and after the refactoring process. Design evaluation is another aspect that students need to develop. Unlike correcting errors, where testing code for correctness is straightforward, validating a design change's effectiveness necessitates actively measuring its impact on quality. The presence of various possible solutions for a single antipattern can serve as a valuable educational tool, promoting cooperative learning tasks. This approach enables students to compare their solutions by examining the effects of their refactorings on antipattern resolutions and overall software quality [1].

Summary for RQ_2 : 'Design' PMD ruleset category takes longer to resolve. Unlike other PMD category, this requires going beyond one or few instructions, and ChatGPT has a limited understanding of the broader context and the entirety of the codebase.

4.3 To what extent was ChatGPT successful in addressing the students' debugging needs?

Figure 7 illustrates students' experiences with ChatGPT across four key areas: (1) effectiveness in resolving problems, (2) the risk of ChatGPT causing code regression, (3) usage frequency for debugging, and (4) the introduction of additional errors as identified by PMD. When we asked students "Was ChatGPT able to fix the buggy code?", 85.9% of the students affirmed ChatGPT's efficacy in problem-solving, while 14.1% disagreed. Overall, students expressed satisfaction with ChatGPT's assistance. Regarding code regression, 79.6% of students did not encounter regression, while 20.4% did. In terms of usage frequency, 40% used ChatGPT once, and 21.3% used it twice. The remaining students (38.7%) used ChatGPT between three or more times. Concerning the potential for additional violations detected by PMD, 86.4% of the students reported no such errors in ChatGPT , but some indicated otherwise.

The success of using any language model heavily depends on students' ability to properly prompt it. Therefore, one of this work's outcomes is sensibilizing students to the power of prompt engineering, in the context of debugging. As depicted in Figure 8, an analysis of the initial prompts from students indicates that 27.7% believe that simply pasting the erroneous code is enough for ChatGPT to reveal the prompt's intent. Meanwhile, 68.6% of students explained that they supplement the buggy code fragment with its error description to give more context. Additionally, 20.6% stated that they paste the faulty code along with a textual guide on how to repair it. A small number of students indicated that they provide both buggy and corrected code pairs, requesting ChatGPT to perform similarly with the problematic code provided. Besides these activities, 2.9% of the students mentioned in the "other" option: "asking ChatGPT that the PMD violation still remained when it could not fix the code", "asking ChatGPT how to fix the error in general before copy pasting the offending code", "prompted ChatGPT with the actual line that was causing the error and asked it if there was something specific in the code segment that could be optimized", "asked ChatGPT to provide an example of how a similar situation could be fixed", and "ChatGPT says this is a false positive and is descriptive of what the variable is representing and I agree with it". Figure 8 shows how the students' prompts are not uniform, where the majority of students argue over how to extract the necessary action from the models, while others overestimate the capabilities of the model, as outlined in previous studies that have proven that ChatGPT is susceptible to hallucination when it comes to coding semantic structures [31, 52]. So, experiencing the potentially inappropriate results of the model would raise the students' awareness of its limitations. In addition, it helps students refine their prompts, as shown in Figure 7 where most students have used more than one prompt per issue (60%).

Furthermore, 68.6% of the students' prompts were zero-shot, *i.e.*, students rely on the generative ability of the model to either understand an issue, or to propose its corresponding code fix. Zero-shot learning challenges the model to make a decision over presumably unseen data [54]. The model relies on approximating the input with previously trained code. For instance, as shown in Figure 9, the student asks the model to reduce the cyclomatic complexity of an input source code. The same prompt can be augmented by adding a label to the unseen data, *i.e.*, one-shot [18]. For example, in Figure 9, mentioning how the reduction can be performed can guide the model towards the decision to take (method extraction). Other students opted for a few-shot learning by entering pairs of buggy and fixed code fragments and asked ChatGPT to propose similar fixes for an alternative buggy code. Few-shot learning is a response when dealing with complex tasks to steer the model towards better decision making by allowing in-context learning from provided examples [50]. In Figure 9, the input shows examples of code changes that address the complexity of a given method.

Diverse prompts used in our activities have encouraged the students to consider strategies to query the model to counter its inherent limitations. This has encouraged a critical evaluation of ChatGPT's responses instead of accepting them as the ground truth. For example, 20.4% of students observed regression in their code after implementing ChatGPT's



detected by PMD?

Fig. 7. Student's attitude toward utilizing ChatGPT for bug resolution.

generated code, with issues ranging from compiler errors to failed unit tests. Such difficulties could stem from bugs in ChatGPT's code outputs or improper integration by students into their codebases, and sometimes even inaccurate queries from students. Although it would be interesting to dive into this analysis more in-depth, our primary aim was to raise students' awareness of the error-prone nature of querying language models, emphasizing the importance of verifying results prior to implementation. We also aimed to alert students to the existence of multiple querying techniques, which are known in data science curricula, without providing any prompt engineering training.

In Table 6, we report the main thoughts, comments, and suggestions about the overall impression of the usefulness, usability, functionality, and recommendation of ChatGPT, according to the labeling performed. The table also presents samples of the students' comments to illustrate their impressions of each theme.

Usefulness. Overall, the respondents identified five key benefits of ChatGPT: automation, quality, scope, awareness, and experience. Students highlighted ChatGPT's efficiency in fixing bugs automatically, its cost-effectiveness, its capability to automate tedious debugging, and its ability to generate accurate and comprehensive responses. About



Fig. 8. How did you use ChatGPT to fix buggy code?

85.9% of the students noted that ChatGPT is beneficial as it provides both explanations of the problem and solutions. Furthermore, 23.22% remarked that it is an excellent tool for learning, particularly for beginners in software quality and debugging. Some students (20.64%) noted that ChatGPT was quick in analyzing issues within contiguous instructions, but struggled with larger problems such as GodClass and DataClass. Finally, 4.5% of students stated that identifying issues enhances *readability*, thereby improving the overall quality of the code.

Usability. According to student feedback, important aspects of usability highlighted were documentation, ease of use, explainability, and the user interface. Approximately 28.38% of students mentioned that the tool is easy to use, features a friendly interface, and clarifies how each bug fix was generated and how to prevent similar issues later. Additional remarks noted that the use of these large language models requires careful adjustment of the model output due to a lack of originality and possible biases.

Functionality. Based on feedback from the students regarding the tool's capabilities, 16.77% of the responses praise ChatGPT's features for correction, detection, and debugging, expressing satisfaction with multiple aspects of its functionality. They noted that this feature aids in understanding poor programming practices in practical situations. Furthermore, the students mentioned that it allows them to apply various methods to solve problems.

Recommendation. Based on students' feedback, we have derived suggestions to improve LLM. Notably, 3.8% of the comments included specific recommendations to enhance the tool's functionality. These recommendations pertain to context, promptness, accuracy, precision, and verification. Students acknowledged the transformative potential of ChatGPT in the programming workflow due to its capacity to generate diverse solutions and adapt to a developer's method. However, others pointed out how ChatGPT can eventually hamper developers' creativity. They foresee ChatGPT revolutionizing coding processes in the future. Certain students noted ChatGPT's efficiency in addressing language-specific coding errors when well explained, alongside solving PMD default issues. However, querying becomes

Zero-shot learning:

You are a Java expert. How do I fix this section of code that has a cyclomatic complexity of 21:

"' Original code "

One-shot learning:

You are a Java expert. Can you fix this section of code that has a cyclomatic complexity of 21 by breaking down the code into smaller methods: "' Original code "

Few-shot learning:

You are a Java expert. I provide you with a pair of fixed and buggy code fragments, and I request that you apply similar fixes to a newly given buggy code.

An example of buggy code that has a cyclomatic complexity of 21: *"* Original buggy code "

An example of fixed: "" Original fixed code ""

Here is the newly buggy code, please propose similar fixes for this buggy code: "' *Newly buggy code* "'

Fig. 9. Different types of learning settings used by students when prompting ChatGPT in the context of debugging.

challenging when errors are distributed across multiple locations. Such limitations could be addressed with formal training in prompt engineering. Thus, careful use and verification from other sources are essential. The reliability of ChatGPT's responses depends heavily on the quality of the prompts and the training data, potentially affecting the response precision and its applicability to specific tasks. Students highlighted ChatGPT's utility in numerous scenarios, but noted its difficulty in resolving issues without context. For example, ChatGPT might find the code comprehensive and easy to read, but when students mention the problem with the code, it recognizes the issue. The more detailed the description, the more specific and accurate the solution made by ChatGPT. However, some issues require investigating various coupled files that may or may not be present as part of the input. Therefore, even after inputting a proper description of the buggy code, there are no guarantees on the model's ability to debug. In addition to the detailed description of the issue, providing the most relevant code fragments is key to avoid dragging ChatGPT attention to unnecessary details or bug-free code fragments.

Theme	Sub-theme	Example (Excerpts from a related student's comment)
	Automation	"The advantages of ChatGPT include its affordability, capacity for automating tedious processes, and ability to deliver precise and thorough responses for customer service departments."
	Quality	"PMD is a useful tool that can help identify potential issues in code and provide suggestions for improvement. It can also help enforce coding standards and best practices, leading to cleaner , more maintainable code . In my experience, both ChatGPT and PMD have been valuable tools. ChatGPT has been able to quickly provide helpful responses to various questions, while PMD has been able to catch potential issues that may have been missed during manual code reviews. However, it's important to note that these tools should be used as aids and not as replacements for human analysis and judgment. Overall, I believe that the use of AI language models and code analysis tools like PMD can greatly benefit developers and lead to better code quality and efficiency "
Usefulness	Scope	"ChatGPT was very informative about giving context and definitions of errors, and was even able to provide definitions and recommendations specific to the PMD plugin. Negatives included when code errors were too vague or covered code segments that were too large for ChatGPT to understand. I conclude that ChatGPT is useful for refactoring small segments of code like fixing errors in smaller methods, nested if statements, and generally cleaning up the code to run faster, but not for larger issues like god classes ."
	Awareness	"ChatGPT was extremely useful. It had all of my answers I needed, and even was able to answer other questions for me. When it fixed the code, it also gave explanations to each change it made to the code. Overall, ChatGPT was an extremely useful and easy to understand."
	Experience	"The refactored code given by ChatGPT was bug free most of the time but at few times there were bugs in it. Overall the chatgpt is very nice application specially for those who are begineers in software development and programming."
	Documentation	"My experience with ChatGPT was largely positive. The tool was able to quickly identify a range of issues in the codebase, including code style issues, best practices, performance issues, and design problems. The reports generated by ChatGPT were very informative and easy to understand, providing clear explanations of each issue and suggesting steps to fix them."
Usability	Ease of use	"It was quite easy to fix errors using ChatGPT. The trick is to find the right prompt so that it understands what it needs to do quickly rather than going around explaining the issue without providing the fix. However, since ChatGPT only had limited code visibility (the code snippet provided by the user), it generated a few fixes which would break the code (ex change variable names, use packages without importing, etc.). We can't say that the code fix provided by ChatGPT was entirely wrong though. Overall, it is easy and quick to use ChatGPT to fix most PMD Errors, with a little bit of oversight."
	Explanability	"when you're working on a small scale project it's extremely useful. Can also explain basic concepts you might have forgotten very well."
	User interface	"Chatgpt has a user-friendly interface. It provides precise and thorough responses for customer service teams. However, there are also potential disadvantages to using these models, such as the need to carefully adjust the model's output. It lacks originality and might give biased output."
	Correction	"ChatCPT would sometimes misunderstand what error I was trying to indicate, or would be confused by the fact that I was giving it a code excerpt rather than the full code, which was too large to provide to it. Its corrections sometimes also misunderstood the purpose of an excerpt and would slightly alter the logic, but this was expected."
Functionality	Detection	"Overall I had a positive experience with ChatGPT as it was able to detect the errors that PMD reported . However for the last 5 code segments I tested, it started generating code that produced compile errors and it didn't always generate the code in java on the first few attempts."
	Debugging	"ChatGPT is usually unable to identify the problems the pmd extension provides. However when it does it is able to go into great detail about the problems, and either fix the problems entirely or give guidance as to how to fix them. I enjoy using chatgpt for debugging and other things and I believe it is a great academic learning tool."
	Context	"ChatGPT was definitely helpful to fix errors. However, many errors required a lot more context than what chatgpt accepts and that leads to incorrect fix or sometimes makes the problem worse. There is also the problem with it dropping context time and again. I feel chatgpt is best considered as a supplement to the existing options like stackoverflow but cannot be blindly trusted. That being said it is still a great upgrade from stackoverflow and may be in future be more accurate. Overall a positive experience."
	Promptness	"It was very useful, there was a bit of a learning curve as it took a few attempts to understand what exactly to prompt it to give you the information you want to know. I had used it before so thats why I understood how to problem solve in that way but chat GPT its self doesn't really suggest other ways to prompt if it doesn't get enough to give an answer."
Recommendation	Accuracy	"GPT is not primarily designed for debugging but can assist in identifying syntax errors and offering debugging strategies. It can also suggest alternative solutions and offer insights into the program flow and potential issues. However, its suggestions may not always be accurate, and it may not have a deep understanding of complex code structures, making it less effective for identifying all types of errors."
	Resolution	"While ChatGPT was able to identify issues in the code, it did not always provide suggestions for how to fix them. This meant that developers still needed to have a good understanding of coding best practices in order to address the issues identified by ChatGPT."
	Verification	"ChatGPT is perfect as a code-companion but not as a replacement . The reason for this is because ChatGPT is unable to find context of the applications of code. This makes it more prone to missing logical bugs like the fall-through of a switch-case statement. Having a human-in-the-loop who is able to understand the applications and has a good idea of the overall structure of the project is much more desirable ."

Table 6. Student's insight about the usefulness, usability and functionality of ChatGPT [4].

Summary for RQ₃: Overall, this assignment helps cultivate analytical and critical thinking skills as students engage in the debugging process. In addition, it teaches students to be skeptical towards the use of ChatGPT by shedding light on the limitations of ChatGPT in identifying and solving problems. It highlights the importance of incorporating other traditional static analysis tools and techniques to improve the accuracy and efficiency of their Manuscript submitted ACM of ACM value of the involvement of a human-in-the-loop, capable of comprehending code, can be highly valuable and desirable.

4.4 What strategies students use to formulate prompts that facilitate effective communication with ChatGPT?

Upon analyzing students' strategies in engineering the prompts, we observe that they focus on three main aspects that we refer to as the 3 *Cs*: *Clarity, Complexity*, and *Context*. In the following, we present samples of student prompts to illustrate these three aspects.

 Clarity. Students ensure clarity in their question or request to ChatGPT, as ambiguous prompts may lead to misunderstandings or irrelevant responses.

"Provide me necessary steps to fix this cyclomatic complexity for this code keeping the functionality same. code: [...]".

• *Complexity.* Students consider the complexity of the prompt and whether it can be adequately addressed within the constraints of a single response.

"Hello, I have a piece of Java code that has been flagged for a PMD violation. I need help identifying whether the violation is a true or false positive and, if true, how to fix it. Here are the details: Code Snippet: char[] charSource = null; PMD Issue Type: Performance Violation Reported: RedundantFieldInitializer Question: The type of issue Is this a true or false positive PMD violation? If it's a true positive, what are the necessary steps to fix it?".

- Context. Students provide contextual details when asking ChatGPT to generate accurate responses.
 - "I'm using PMD to detect code violations in a Java project. It has found a violation of UnusedLocalVariable in the following code block [...]. Can you fix this and remove the unused local variable while retaining the same functionality?".

We believe that these 3 *Cs* not only enhance students' communication skills and interaction with ChatGPT, but also foster key learning outcomes that involve, but are limited to, critical thinking, problem-solving, and creativity. Considering the *Complexity* and *Context* of the prompts allows students to think critically about the coding issues. Furthermore, crafting the prompts with *Clarity* challenges students to think about the desired outcome of their interaction with ChatGPT. This can also foster creativity as this requires students to think creatively about the questions they need to ask ChatGPT and the responses they anticipate.

Summary for RQ₄: Crafting prompts to ChatGPT with Clarity, Complexity, and Context fosters essential learning outcomes, including improved critical thinking, problem-solving abilities, and creativity.

4.5 Does ChatGPT detect any other quality issues that were not detected by PMD?

Figure 7e illustrates the percentages of ChatGPT's quality issues that were not detected by PMD. As can be seen, 93% of the students indicated that ChatGPT did not detect other issues that were not detected by PMD. In Table 7, we show the quality issues identified by ChatGPT that may have been overlooked by PMD. These issues are related to improving the documentation (*e.g.*, AddComments), readability (*e.g.*, UseConsistentIndentation), and quality in general (*e.g.*, ReduceCouplingBetweenPackages). It is worth noting that although ChatGPT identified additional issues not detected by PMD, some of these issues are considered PMD rules. However, PMD did not flag them as violations for the specific code fragment that students asked about. We present samples of ChatGPT responses to illustrate these quality issues.

" It is good practice to always use curly braces for the if and else statements, even if there is only one line of code to be executed. This can help to avoid errors when more code is added later."

"The code is using catch-all exception handling, which can be problematic. It is generally better to catch only the specific exceptions that you expect to be thrown and to handle them accordingly."

"Test names should be descriptive and follow a consistent naming convention."

"It's a good practice to use the @Override annotation when overriding a method from a parent class or interface. This can help catch errors at compile time if the method signature changes in the future. So you could add the @Override annotation to the clone method."

Furthermore, we report in Table 8 PMD rules that were flagged by ChatGPT as false positive. Clearly, most of the violated rules are accepted by ChatGPT as true positives, and a few of these violated rules (*i.e.*, 12 rules) are rejected and perceived by ChatGPT as false positives. These rules mainly belong to 'Code Style' category.

While analyzing students' submissions, we realized some disagreement cases between quality issues detected by PMD and ChatGPT. We believe these cases can be attributed to the following factors:

Detection methodology. PMD focuses on the syntactic and structural aspects of the code. It analyzes code to identify potential quality issues based on predefined rulesets. On the other hand, ChatGPT relies on natural language processing techniques to understand user's prompts, and generate responses accordingly which may involve potential quality improvement.

Scope of analysis. PMD scans the entire code by examining each file for violations of predefined rulesets to detect issues related to coding standards, and code smells. ChatGPT's analysis is restricted by the user's prompts. It may not have visibility into the entire code and may only limited in addressing issues that are explicitly mentioned in the prompts.

Semantic understanding. PMD detects issues based on its understanding of programming language syntax and semantics. It has a limited ability to identify quality issues that require contextual knowledge and retain expertise. However, ChatGPT is limited to the textual prompts provided by the users. ChatGPT understanding of code semantics maybe less accurate compared to PMD.

False positives and false negatives. Disagreement cases between PMD and ChatGPT may arise due to the differences in the detection thresholds. Some issues detected by PMD may be considered to be false positives if these are not applicable in the given context. Conversely, ChatGPT may be unsuccessful in detecting quality issues that require semantic analysis or domain-specific knowledge, which might lead to false negatives.

Context sensivitity. While PMD's analysis may consider factors such as code complexity and patterns to determine the severity of the issues, ChatGPT's responses may lack context sensitivity. This is due to the fact that the responses are based on the provided prompts with a limited understanding of the broader codebase.

Human validation. Disagrement cases between PMD and ChatGPT highlight the importance of human validation, intervention, and assessment. Students need to review and validate the suggestions provided by both tools to confirm their accuracy in the given context.

The observation from this RQ can provide insights into the complementary nature of LLMs and static analysis tools. PMD is capable of identifying common coding issues based on predefined rules. On the other hand, ChatGPT leverages natural language understanding and machine learning to analyze the code, which could potentially capture issues that might not be detected by PMD's rule-based approach. This finding can have implications for software development practices, as it can provide students with more comprehensive insights into code quality. Furthermore, it is worth noting that ChatGPT has limitations that it can effectively identify, especially the limited understanding of the overall codebase and its restricted input size. For example, for the UnusedMethods rule flagged by ChatGPT, the accuracy of

Rule
UnusedImports
UnusedParameters
UnusedMethod
UnusedVariable
UnnecessaryModifier
AddOverrideAnnotation
AddJavadoc
NullAssignment
ExplicitlySetDefaultValues
UseDescriptiveVariableNames
MakeFieldsPrivate
ReduceCouplingBetweenPackages
UseDependencyInjectionInsteadOfStaticMethodCalls
AddBraces
RemoveUnnecessaryComments
UseConsistentIndentation
AddProperSpacingAroundOperatorsAndAfterKeywords
SimplifyVariableDeclaration
AddComments
SimplifyVariableDeclaration
UseCurlyBracesForIfElseStatements

Table 7. ChatGPT rules that were suggested to students that were not captured by PMD.

this detection can vary based on the size and complexity of the codebase. In large codebases where ChatGPT's input size may be insufficient to capture the entire context, its ability to accurately identify UnusedMethods could be compromised. Therefore, students need to verify the validity of ChatGPT's output before taking action based on its recommendations.

-

Table 8. PMD rules that were flagged by ChatGPT as false positives.

l	Rule
L	awOfDemeter
F	PrematureDeclaration
L	ongVariable
ŀ	AvoidThrowingRawExceptionTypes
L	iteralsFirstInComparisons
0	CommentSize
ι	JselessParentheses
E	BooleanGetMethodName
0	ShortVariable
A	AbstractNaming
٢	NullAssignment
ŀ	AvoidInstantiatingObjectsInLoops

Summary for RQ_5 : Out of the 1,658 analyzed issues, 93% of students reported that ChatGPT did not uncover any additional issues beyond those detected by PMD.

4.6 What are the response patterns of ChatGPT to student requests?

As illustrated by Figure 10, we analyze the response patterns of ChatGPT to student requests in an educational context. ChatGPT's apologies (*i.e.*, acknowledging errors) and affirmations (*i.e.*, maintaining assurance or agreement) play a role in the debugging and bug fixing process. Apologies can be issued if the process introduces errors or unexpected outcomes. Affirmations can be used to acknowledge successful resolution. Upon analyzing ChatGPT's responses, we cluster the responses into five categories. Next, we provide an analysis of these categories.

Apology. ChatGPT's responses utilize the following keywords 'My apologies' and 'I apologize' when acknowledging errors about its recommendation. These expressions of apology indicate recognition of errors and a commitment to addressing them. ChatGPT conveys a sentiment of regret when introducing bugs in the code, misunderstanding the content of the code, missing update functionality, and introducing bad practices. ChatGPT response exemplifies this category, as shown in Figure 10. It is evident that ChatGPT apologizes for the inability to fix lengthy and complex code. This demonstrates ChatGPT's awareness of the importance of maintaining code quality.

Affirmation. ChatGPT's responses utilize the following keywords 'certainly', 'absolutely', 'of course' as confirmation of the student's prompt about the fixes. These affirmations signify a strong level of agreement or assurance in response to the student's inquiry or suggestion. ChatGPT maintains the same level of confidence or agreement throughout its recommendations. When suggesting fixes, ChatGPT applies refactoring to improve quality, provides examples of how to fix the code, complies with established guidelines or principles for refactoring, applies best coding practices, such as modularization and abstraction, to enhance code maintainability and readability, and presents supplementary approaches to refactoring, providing developers with alternative strategies or techniques to consider. Figure 10 presents an example of the ChatGPT responses that exemplify this category. As shown, ChatGPT adheres to the best practice of improving the code style.

Compilation errors. ChatGPT's responses contain syntactical errors, which can lead to confusion, misinterpretation, difficulty in comprehending, or inaccuracies. This is apparent from Figure 10 that compilation errors were introduced after the fixes suggested by ChatGPT.

Provision of guidelines. ChatGPT may offer guidelines, tips, and instructions to help students address the PMD issue. This pattern involves providing step-by-step instructions to assist students in effectively navigating their problems. This is demonstrated in Figure 10, when ChatGPT provides instructions on how to reduce coupling between objects.

Disagreement cases. In some instances, ChatGPT's response may contradict the violation rules flagged by PMD. This pattern showcases potential discrepancies between ChatGPT's generated content and PMD, prompting students to further investigate the underlying causes and potential bug resolution. As depicted by Figure 10, the disagreement instance between ChatGPT and PMD, as ChatGPT indicates there is no code violation in the student's request.

Summary for RQ_6 : The response patterns of ChatGPT to student requests in an educational context, included a variety of interactions such as affirmations, apologies, compilation errors, provision of guidelines, and instances where ChatGPT's feedback differed from that of PMD, offering valuable insights into how LLMs can assist in the learning process.

5 REFLECTION

This section provides the lessons learned from both the educator's and the student's perspectives.

Lesson #1: *Develop complementary assignments.* ChatGPT identified the PMD violations and attempted to correct them. In certain instances, the issue disappeared entirely from the PMD analyzer's detection. However, in other cases, despite applying a correction, the PMD analyzer continued to flag the issue. ChatGPT revises the code upon request and sometimes enhances its output. However, with repeated prompts, it typically produce the same response, with changes occurring only after numerous attempts. Consequently, students can use ChatGPT as a complementary tool in conjunction with standard static analysis tools to enhance the efficiency of their software development workflow.

Actionable Result. Design assignments that require students to apply LLMs alongside static analysis tools. This helps encourage students to tackle real-world coding problems while considering multiple resources.

Recommendation. Develop complementary assignments in which LLMs suggestions are used for problem-solving, and require students to improve them using static analysis tools. This can promote critical thinking and problem solving. **Lesson #2:** *Limited understanding of the broader context of the codebase.* While ChatGPT was informative about giving context and definitions of errors and could provide specific recommendations for PMD violations, it has

Apology pattern:

I apologize, but I cannot modify the code as I don't have access to the entire codebase or know what the code is supposed to do.

Affirmation pattern:

Certainly! To avoid the naming conflict, you can rename the field or the method. Here, I'll rename the field for clarity:

" Modified code "

Now, the field is named 'fixed', and the getter and setter methods are using the standard naming convention without any conflict.

Compilation errors:

You are correct, my apologies for the confusion. 'env' is indeed a field of type 'Annotation-ProcessorEnvironment' and not of 'Messager'. To fix the "Law of Demeter" violation in this case, you can use the Law of Demeter principle to reduce the coupling.

Provision of guideline:

To address the PMD error of "Possible God Class (WMC=118, ATFD=32, TCC=6.085%)", you can consider refactoring the code by breaking down the BaseFunction class into smaller and more focused classes. This can be done by identifying the functionality that is specific to the class and separating it into smaller, more manageable classes.

Here are some steps you can follow to refactor the BaeFunction class:

1. Identify the methods and properties of the BaseFunction class that are tightly related to its core functionality. 2. Separate there methods and properties into a new class that has a specific responsibility.

3. Update the BaseFunction class to use the new class as a dependency, rather than implementing the functionality itself.

4. Repeat steps 1-3 until the BaseFunction class is reduced to a smaller and more focused class.

By refactoring the code in this way, you can reduce the complexity of the BaseFunction class and make it easier to maintain and extend over time.

Disagreement cases pattern:

Since there is no violation of the Law of Demeter in the code you provided, there is no need to fix it.

Manuscript submitted to ACM

Fig. 10. Example of patterns in ChatGPT response.

a limited understanding of the broader context and the entire codebase, which could lead to missing dependencies and codependencies. This limitation can cause instances where ChatGPT offers advice based on misunderstandings or incorrect assumptions about the code. The study's assignment reveals this limitation in a practical manner, aiding students in comprehending how the model operates, rather than viewing it as a black box that autogenerates acceptable answers for any given query. During this experiment, not only could students not reach an acceptable answer for some queries, they also experienced how suggested code changes can be even problematic, as seen in Figure 7. Students experienced various negative side-effects of the autogenerated code. For instance, in certain scenarios, code provided by ChatGPT triggered compiler errors. Additionally, some students reported that the suggested refactoring did push some tests to fail, despite the fact that refactoring is supposed to preserve the system's internal behavior. Moreover, ChatGPT proposed a solution for the 'Best Practices' category; however, the solution increased the cyclomatic complexity of the method, likely due to insufficient context and lack of clarity in the code snippets. This finding is consistent with a previous study [13] that discovered that poor warning messages and the absence of suggested fixes are frequent complaints among industry software developers utilizing program analyzers.

Actionable Result. Encourage students to explore the context of a codebase through deeper exploration before addressing code problems.

Recommendation. Introduce group activities that ask students to understand the system structure before fixing bugs or modifying code. This strategy promotes contextual understanding in coding.

Lesson #3: *Verify ChatGPT's responses with human expertise.* Although ChatGPT can assist with some debugging tasks, it should not be the sole tool used for debugging. ChatGPT is perfect as a code companion, but not a replacement. This is because ChatGPT lacks the ability to interpret the context of code applications. Therefore, including a human who comprehends the applications is more advantageous. Consequently, skilled developers and specific debugging tools are still crucial for comprehensive and efficient debugging.

Actionable Result. Foster group work where students collaborate to evaluate the suggestions provided by LLMs, and compare them with expert insights.

Recommendation. Create peer review activities where ChatGPT-generated solutions are verified and discussed with instructors or experienced developers to advocate for teamwork and reflective learning.

Lesson #4: Automate the debugging and bug fixing process. ChatGPT is an effective tool for programmers seeking help. However, providing clear and detailed questions is essential to get the most accurate and helpful responses. It has helped students debug and clarify what each code segment did and explained how it performed each bug fix. Thus, the process is significantly shortened. Yet, in a few cases, it required some experimentation before coming up with a set of questions to accomplish the task. Additionally, it suggested changes that caused compiler errors, *e.g.*, renaming a variable in a field declaration or fixing a different issue in the code instead of the one given to analyze. Furthermore, ChatGPT tends to view PMD-related stylistic or best practice issues as not problematic and is too light-handed in explaining the potential issues with stylistic/best practice problems in the source code. For example, it may overlook issues like inconsistent indentation, unclear variable names, or improper use of comments. In practice, this can lead to the student developing bad coding habits such as neglecting code readability and failing to follow established coding conventions, which ultimately reduce the productivity of the individual and the team to which they belong. In addition, some of the flaws were observed with inexplicable fixes or suggestions that can further contribute to confusion and poor coding practices.

Actionable Result. Implement hands-on class activities that require students to integrate automated static analysis tools into the development workflow.

Recommendation. Assign class projects where students use automated tools for debugging, reflect on the experience, and then integrate ChatGPT to assist in these projects to improve troubleshooting skills.

Lesson #5: Improve ChatGPT's accuracy and effectiveness. ChatGPT proved helpful in addressing certain violations of PMD rules and offered good recommendations for resolutions. Nevertheless, in some instances, it is challenging to definitively identify PMD rule violations, even with the provided code context. For certain violations, it responded with 'may', suggesting that it met specific conditions considering only the given code snippet, not the surrounding code. We believe that with targeted training for the specific ruleset, ChatGPT could efficiently help reduce PMD rule violations. However, solely depending on ChatGPT is not advisable; developers need to possess a degree of knowledge and comprehension to make the right corrections, as it sometimes results in code that leads to compilation errors.

Actionable Result. Allow students to engage in iterative experimentation with ChatGPT's outputs and assess its performance independently.

Recommendation. Encourage students to develop their own methods to verify and refine ChatGPT-generated code and to set their own goals to improve accuracy and effectiveness.

Lesson #6: *Handle complex code and errors.* The effectiveness of ChatGPT is closely related to the quality of its training data set. Based on feedback from students, ChatGPT offers valuable, high-level advice on bugs. For example, it excels at addressing less urgent issues such as code styling, inserting comments, and enhancing the logic of if statements. However, there are instances where it misinterprets the error the student intended to highlight or becomes confused because the input consisted of a code snippet instead of the entire code, which was too lengthy to include. Its suggestions occasionally misinterpret the purpose of a section and subtly change the logic. Therefore, the more complex the given code input, the more likely the code output will not work properly.

Actionable Result. Create hands-on activities in which students work with complex code, making adjustments based on the ChatGPT recommendation.

Recommendation. Develop lab sessions where students handle complex code issues, leveraging both ChatGPT's insights and their manual debugging skills. Students should reflect on what went right/wrong to reinforce learning.

Lesson #7: Examine ChatGPT's tone impact on bug resolution decisions. The manner in which ChatGPT expresses confidence or offers apologies could potentially influence students' decisions about bug resolution. Future research can explore whether the affirmation or apologetic tone of ChatGPT's responses correlates with students' acceptance of its recommendations. Specifically, understanding how students perceive and respond to ChatGPT's confident assertions or apologies about the recommendation could shed light on the effectiveness and trustworthiness of the model's suggestions. For example, does a confident assertion from ChatGPT lead students to more readily accept its recommendations, even if they introduce potential issues? In contrast, do ChatGPT apologies mitigate concerns about such issues and encourage students to overlook potential flaws in its suggestions?

Actionable Result. Engage students in group discussions about the tone of ChatGPT's responses and its effect on their decision-making processes.

Recommendation. Facilitate discussions where students evaluate ChatGPT's conversation style influences their interpretation and understanding of the solutions. This interactive activity can reveal the psychological and decision-making aspects of human-AI interaction.

Lesson #8: *Explore alternative approaches to utilizing ChatGPT for programming assistance.* Upon reviewing a variety of prompts written by students to tackle PMD issues, it became evident that some students requested ChatGPT to function as a distinct AI assistant dedicated to supporting programmers during development. Although we have not evaluated the effects of this approach, we suspect that instructing ChatGPT to "act as" a separate AI for programming Manuscript submitted to ACM

assistance will probably result in more direct code responses. This strategy could potentially offer an alternative for students who are dissatisfied with ChatGPT's performance when approached in a straightforward manner.

Actionable Result. Motivate students to try various prompting techniques by asking ChatGPT to impersonate a dedicated AI assistant for programming. This approach may enable students to receive more customized and effective coding guidance.

Recommendation. Develop a guide that showcases various strategies for framing prompts, such as directing ChatGPT to function as an AI specifically for programming tasks. This resource could aid students in crafting successful prompts.

6 THREATS TO VALIDITY

In this section, we describe potential threats to the validity of our research method and the actions we took to mitigate them.

External Validity. Regarding the generalizability and applicability of our findings, our investigation includes 155 submissions. While we gathered valuable insights and conducted a precise analysis, the findings might not accurately reflect the broader student population utilizing static analysis tools and ChatGPT to resolve issues. Furthermore, our analysis focused on mature open-source Java projects with varying sizes, contributors, and commit counts. However, we cannot claim that our findings can be generalized to projects written in other programming languages or from different ecosystems. More research on additional projects is required to address this limitation. Given that ChatGPT's training involves a large corpus of source code, sourced from platforms like GitHub and StackOverflow, there is a possibility of data leakage, i.e., the proposed fixes were previously seen in the training set, so the fixes were previously memorized. However, the projects used actually contain some of these warnings in their current versions, which means that ChatGPT may have experienced these fixes in other projects, and its current decisions are based on inference. In addition, the performance of ChatGPT was not uniform across categories and underperformed for design-level issues. Finally, our study is solely focused on ChatGPT, and it is recommended that future research extend to other general-purpose chatbots applicable to software development, such as the recently released Google Bard [7, 48], which may have limitations that ChatGPT does not possess, including access to up-to-date content. Also, we recommend that students use the free version of ChatGPT (currently 3.5), but we did not control who uses the premium service, which features GPT-4 and GPT4o.

Internal and Construct Validity. To assess our interpretation of open-ended comments regarding the tool's use, we did not extensively discuss every comment, as many are open to multiple readings and additional interviews are required for clarification. To avoid personal bias during manual analysis, two annotators independently reviewed each stage until agreement was reached. Our selection of PMD as a static analysis tool could result in bias because the identification of poor programming practices and code smells is inherently subjective [1, 5, 11, 16, 19, 32]. In addition, students may have had a different experience if another tool was selected in this assignment. We opted for PMD as it is one of the popular state-of-the-art tools, but we intend to explore other static analysis tools in the future to determine whether they achieve similar levels of satisfaction. Moreover, any training provided to students could introduce bias. To address this, the PMD and ChatGPT trainings were conducted independently. During ChatGPT training, students were exposed to prompts related to software quality concepts they could relate to, though none were directly aimed at addressing PMD-raised issues. Another threat relates to the lack of specific performance metrics for ChatGPT in different coding categories. In our study, we encourage students to think critically about the evaluation process, rather than relying solely on metrics as a black-box indicator. However, the absence of comparison metrics may hinder Manuscript submitted to ACM

the ability to comprehensively assess the effectiveness of ChatGPT. Future work should aim to include measurable performance indicators and a more critical approach to provide a more robust evaluation of its capabilities in various coding contexts.

7 CONCLUSION

Understanding the practice of reviewing code to improve quality is of paramount importance for education. Although modern code review is widely adopted in open-source and industry projects, the relationship between using LLMs such as ChatGPT and the way students perceive it during code analysis remains unexplored. Our main contribution underscores how the integration of PMD with ChatGPT can engage students to improve the quality of the software. This synergy enhances automated static code analysis and AI-powered code understanding, offering a more comprehensive approach to code quality and improvement. Using the strengths of both tools, our work demonstrates how AI can enhance traditional software development practices, leading to efficient and effective outcomes. In this study, we conducted a quantitative and qualitative study to explore the effectiveness of PMD and GPT in familiarizing students with improving the quality of the source code by i) detecting code issues and antipatterns and ii) implementing fixes for their correction. The paper develops a culture of reviewing and patching unknown code. Our results reveal several types of static analysis tools that students should pay more attention to during code review; reviewing design-related changes takes longer to complete compared to other changes; students rated some aspects of ChatGPT positively while also providing valuable ideas for future model improvement; engineering prompts with clarity, complexity, and context foster learning outcomes; 93% of analyzed issues shows that ChatGPT did not highlight any additional issues beyond those detected by PMD; and the interaction between students and ChatGPT includes several patterns such as affirmation, apologies, and compilation errors.

8 FUTURE WORK

We foresee many promising avenues for future work. For example, we plan to explore the impact of LLM tone on user experience by conducting controlled experiments to explore how ChatGPT tone affects user comprehension and engagement. By varying the tone in different coding scenarios, researchers could measure the impact on students' learning outcomes, confidence in code correctness, and problem-solving efficiency. We will use surveys and performance metrics to gather empirical evidence, making the study more rigorous. Furthermore, to address the issues with ChatGPT's handling of complex code or errors, future studies could design benchmark datasets that involve difficult coding problems, including those with security vulnerabilities, edge cases, and performance inefficiencies. Researchers could systematically analyze ChatGPT performance in these cases, documenting specific types of errors (e.g., incomplete solutions) and providing metrics such as error rates, code robustness, and time-to-resolution compared to human programmers. Furthermore, future research could involve a comparative study to evaluate the performance of ChatGPT in various coding categories, such as object-oriented programming, functional programming, and security-sensitive code. Measurement of specific metrics, such as code correctness, time to completion, and frequency of security vulnerabilities, can provide a more quantitative assessment of the strengths and weaknesses of ChatGPT in different contexts. In addition, we plan to use other static analysis tools to complement and validate our current study. This will provide the software engineering community with a more comprehensive view of using static analysis tools to engage students with software quality improvement from the educator and student perspectives. Additionally, we plan to conduct a long-term study to track students using ChatGPT over multiple semesters, comparing their learning outcomes (e.g., grades, code quality, and problem-solving abilities) with those of students not using ChatGPT. By incorporating Manuscript submitted to ACM

Table 9. The summary of survey questions.

Question ID	Survey Question	Research Question (RQ)
	I. Background	
Question 1	How many years of general programming experience do you have?	
Question 2	How many years of Java programming experience do you have?	
Question 3	What is your academic program?	
Question 4	Please select the years of software engineering industry experience that you have.	
Question 5	What is your academic major?	
Question 6	How often have you generally fixed bugs?	
	II. PMD	
Question 7	Which project do you select?	RQ ₁
Question 8	What PMD-related category have you chosen?	$RQ_1 \& RQ_2$
Question 9	What PMD rule have you chosen? Please write the exact PMD rule name as listed in PMD documentation?	$RQ_1 \& RQ_5$
	III. ChatGPT	
Question 10	Did you try ChatGPT to fix the buggy code?	RQ ₃
Question 11	How did you use ChatGPT to fix buggy code?	RQ ₃
Question 12	If you selected 'Other' in the previous question, please specify how did you use ChatGPT to fix the buggy code.	RQ ₃
Question 13	Write the exact ChatGPT prompt you have decided to use.	RQ_4
Question 14	Was ChatGPT able to fix the buggy code?	RQ ₃
Question 15	Did the fix contain any other bugs/errors detected by PMD?	RQ3 & RQ5
Question 16	Did one of the proposed fixes generate compiler errors?	RQ3 & RQ6
Question 17	Did ChatGPT detect any other bugs/errors that were not detected by PMD? If yes, copy-paste the ChatGPT message.	RQ ₅
Question 18	How many times have you interacted with ChatGPT before finding (or not) a fix to the buggy code?	RQ ₃
Question 19	How long it took you to find (or not) a fix using ChatGPT? Please specify the approximate time in minutes.	RQ ₂
Question 20	Upload a screenshot of the fixed source code generated by ChatGPT.	RQ ₆
	VI. Final thoughts	
Question 21	Was ChatGPT helpful/useful in fixing the errors?	RQ ₃
Question 22	Add a concise comment about your experience with ChatGPT.	RQ3 & RQ6

assessments at different intervals, the study would provide empirical data on the tool's long-term impact on student learning, highlighting both benefits and potential drawbacks precisely. Finally, we plan to incorporate both quantitative metrics and a critical analysis to provide a more comprehensive evaluation of the effectiveness of ChatGPT.

9 ACKNOWLEDGMENTS

Declaration of generative AI and AI-assisted technologies in the writing process. During the preparation of this work, the author used the ChatGPT web interface and the Overleaf Wrietfull tool to improve the language, paraphrased descriptions, and readability of the manuscript. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

10 APPENDIX

The survey questions are enumerated in Table 9.

REFERENCES

- Eman Abdullah AlOmar, Salma Abdullah AlOmar, and Mohamed Wiem Mkaouer. 2023. On the use of static analysis to engage students with software quality improvement: An experience with pmd. (2023).
- [2] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2021. Refactoring practices in the context of modern code review: An industrial case study at Xerox. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 348–357.
- [3] Eman Abdullah AlOmar, Moataz Chouchen, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack. In Proceedings of the 19th International Conference on Mining Software Repositories. 1–13.
- [4] Eman Abdullah AlOmar and Mohamed Wiem Mkaouer. 2024. Cultivating Software Quality Improvement in the Classroom: An Experience with ChatGPT. In 2024 36th International Conference on Software Engineering Education and Training (CSEE&T). IEEE, 1–10.

- [5] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.
- [6] Owen Astrachan, Garrett Mitchener, Geoffrey Berry, and Landon Cox. 1998. Design Patterns: An Essential Component of CS Curricula. In Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education (Atlanta, Georgia, USA) (SIGCSE '98). Association for Computing Machinery, New York, NY, USA, 153–160. https://doi.org/10.1145/273133.273182
- [7] Try Bard. 2023. an AI experiment by Google.
- [8] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software 107 (2015), 1–14.
- [9] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? an empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.
- [10] Som Biswas. 2023. Role of ChatGPT in Computer Programming.: ChatGPT in Computer Programming. Mesopotamian Journal of Computer Science 2023 (2023), 8–16.
- [11] Sérgio Bryton, Fernando Brito e Abreu, and Miguel Monteiro. 2010. Reducing subjectivity in code smells detection: Experimenting with the long method. In 2010 Seventh International Conference on the Quality of Information and Communications Technology. IEEE, 337–342.
- [12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021).
- [13] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In Proceedings of the 31st IEEE/ACM international conference on automated software engineering. 332–343.
- [14] Henrik Bærbak Christensen. 2004. Frameworks: Putting design patterns into perspective. In Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education. 142–145.
- [15] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In 2011 international symposium on empirical software engineering and measurement. IEEE, 275–284.
- [16] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: are we there yet?. In 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner). IEEE, 612–621.
- [17] Barbara J Duch, Susan E Groh, and Deborah E Allen. 2001. The power of problem-based learning: a practical" how to" for teaching undergraduate courses in any discipline. Stylus Publishing, LLC.
- [18] Li Fei-Fei, Robert Fergus, and Pietro Perona. 2006. One-shot learning of object categories. IEEE transactions on pattern analysis and machine intelligence 28, 4 (2006), 594–611.
- [19] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. J. Object Technol. 11, 2 (2012), 5–1.
- [20] Robert M French. 1999. Catastrophic forgetting in connectionist networks. Trends in cognitive sciences 3, 4 (1999), 128-135.
- [21] Md Asraful Haque and Shuai Li. 2023. The Potential Use of ChatGPT for Debugging and Bug Fixing. EAI Endorsed Transactions on AI and Robotics 2, 1 (2023), e4-e4.
- [22] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. 2019. How do developers act on static analysis alerts? an empirical study of coverity usage. In 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 323–333.
- [23] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An exploratory study of the impact of code smells on software changeproneness. In 2009 16th Working Conference on Reverse Engineering. IEEE, 75–84.
- [24] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt?. In 2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, 2445–2451.
- [25] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. In Guide to advanced empirical software engineering. Springer, 63-92.
- [26] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. Journal of Systems and Software 167 (2020), 110610.
- [27] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. Journal of Systems and Software 198 (2023), 111575.
- [28] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a framework for teaching debugging. In Proceedings of the Twenty-First Australasian Computing Education Conference. 79–86.
- [29] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems 36 (2024).
- [30] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. 2018. Mining fix patterns for findbugs violations. IEEE Transactions on Software Engineering 47, 1 (2018), 165–188.
- [31] Wei Ma, Shanqqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. arXiv preprint arXiv:2305.12138 (2023).
- [32] Mika V Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Software Engineering 11, 3 (2006), 395–431.

- [33] Susan A Mengel and Vinay Yerramilli. 1999. A case study of the static analysis of the quality of novice student programs. In The proceedings of the thirtieth SIGCSE technical symposium on Computer science education. 78–82.
- [34] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. Journal of Software Engineering Research and Development 5, 1 (2017), 1–28.
- [35] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.
- [36] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews?. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 161–170.
- [37] R Plosch, Harald Gruber, A Hentschel, Gustav Pomberger, and Stefan Schiffer. 2008. On the relation between external software quality and static code analysis. In 2008 32nd Annual IEEE Software Engineering Workshop. IEEE, 169–174.
- [38] Md Mostafizer Rahman and Yutaka Watanobe. 2023. ChatGPT for education and research: Opportunities, threats, and strategies. Applied Sciences 13, 9 (2023), 5783.
- [39] Simone Romano, Fiorella Zampetti, Maria Teresa Baldassarre, Massimiliano Di Penta, and Giuseppe Scanniello. 2022. Do Static Analysis Tools Affect Software Quality when Using Test-driven Development?. In ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 80–91.
- [40] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H Paterson. 2010. An introduction to program comprehension for computer science educators. Proceedings of the 2010 ITiCSE working group reports (2010), 65–86.
- [41] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. arXiv preprint arXiv:2305.00418 (2023).
- [42] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Seattle, WA, USA) (FSE 2016). ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305
- [43] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, 101–105.
- [44] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In 2023 IEEE/ACM International Workshop on Automated Program Repair (APR). IEEE, 23–30.
- [45] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? arXiv preprint arXiv:2305.12865 (2023).
- [46] 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).
- [47] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in apache open source projects. *Empirical Software Engineering* 25, 6 (2020), 5137–5192.
- [48] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. In Proceedings of the 21st International Conference on Mining Software Repositories. 571–583.
- [49] Rini Van Solingen, Vic Basili, Gianluigi Caldiera, and H Dieter Rombach. 2002. Goal question metric (gqm) approach. Encyclopedia of software engineering (2002).
- [50] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. ACM computing surveys (csur) 53, 3 (2020), 1–34.
- [51] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382 (2023).
- [52] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2024. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. (2024), 71–108.
- [53] 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).
- [54] Yongqin Xian, Bernt Schiele, and Zeynep Akata. 2017. Zero-shot learning-the good, the bad and the ugly. In Proceedings of the IEEE conference on computer vision and pattern recognition. 4582–4591.
- [55] Chenyang Yang, Rachel A Brower-Sinning, Grace Lewis, and Christian Kästner. 2022. Data leakage in notebooks: Static detection and better processes. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 1–12.
- [56] Xueqi Yang, Zhe Yu, Junjie Wang, and Tim Menzies. 2021. Understanding static code warnings: An incremental AI approach. Expert Systems with Applications 167 (2021), 114134.
- [57] Rahul Yedida, Hong Jin Kang, Huy Tu, Xueqi Yang, David Lo, and Tim Menzies. 2023. How to find actionable static analysis warnings: A case study with FindBugs. IEEE Transactions on Software Engineering (2023).