A Study of Large Language Models in Detecting Python Code Violations

Hekar A. Mohammed Salih and Qusay I. Sarhan†

Department of Computer Science, College of Science, University of Duhok,
Duhok, Kurdistan Region – F.R. Iraq

***Abstract*—Adhering to good coding practices is critical for enhancing a software's readability, maintainability, and reliability. Common static code analysis tools for Python, such as Pylint and Flake8, are widely used to enforce code quality by detecting coding violations without executing the code. Yet, they often fail to handle deeper semantic understanding and contextual reasoning. This study investigates the effectiveness of large language models (LLMs) compared to traditional static code analysis tools in detecting Python coding violations. Six state-of-the-art LLMs: ChatGPT, Gemini, Claude Sonnet, DeepSeek, Kimi, and Qwen were evaluated against Pylint and Flake8 tools. To do so, a curated dataset of 75 Python code snippets, annotated with 27 common code violations, was used. In addition, three common prompting strategies: Structural, chain-of-thought, and role-based, were used to instruct the selected LLMs. The experimental results reveal that Claude Sonnet achieved the highest F1-Score (0.81), outperforming Flake8 (0.79) and demonstrating strong precision (0.99) and recall (0.69). However, LLMs showed differences in performance, with Qwen and DeepSeek underperforming relative to others. Moreover, LLMs that identified documentation and design violations (such as type hints and nested method structures) performed better than stylistic consistency and complex semantic reasoning. The results were heavily influenced by the prompting approach, with structural prompts yielding the most balanced performance in the majority of cases. This research contributes to the practical empirical work on employing LLMs for code quality assurance while also demonstrating their potential role as complementary static code analysis tools for Python, with methodologies that may extend to other languages.**

***Index Terms*—Large language models, Software metrics, Software quality, Static code analysis.**

I. Introduction

Adherence to good coding remains critical in contemporary software engineering, affecting a given software project's reliability, maintainability, and security (Moratis, et al., 2024; Wadhwa, et al., 2024). One aspect of poor coding is termed "code smells", which can sharply reduce quality and may lead to costly errors that are expensive to fix later in the software development lifecycle (AlOmar and Mkaouer, 2024). To solve this problem, static code analysis tools have emerged as essential components within quality assurance workflows for software products (Guo, et al., 2023b). Tools for static code analysis, such as Pylint, Flake8, PMD, FindBugs, and SonarQube, are designed to analyze source code without execution; they flag violations, such as coding defects, vulnerabilities, and style inconsistencies against pre-defined rules (Mohajer, et al., 2023). While such tools often generate high false positives (FPs), which can be overwhelming for developers and require manual checks, their refinement could help detect early issues, reducing maintenance costs (Li, et al., 2023a).

In parallel, large language models (LLMs), such as ChatGPT, have a deep impact on software engineering and they are significantly transform diverse aspects of its domains (Ságodi, Siket and Ferenc, 2024; Zhang, et al., 2024). LLMs are trained on datasets of both natural language and source code; therefore, they demonstrate remarkable capabilities in natural language interpretation as well as in comprehending source code. Consequently, they can perform tasks, such as code generation, code review, and program repair (Ságodi, Siket and Ferenc, 2024; Zhang, et al., 2024). Their strength lies in their capability for code understanding and detecting subtle code patterns related to bugs or vulnerabilities. They also provide help with automated code improvements using natural language commands (Purba, et al., 2023; Ma, et al., 2024). However, understanding code behaviors is often difficult for LLMs. Since LLMs are prone to hallucinations that can lead to functionally incorrect or vulnerable code (Jesse, et al., 2023; Pearce, et al., 2023; Haindl and Weinberger, 2024). This study aims to quantitatively and qualitatively evaluate the performance of LLMs against established static code analysis tools in identifying and rectifying Python coding violations, including violations of Python Enhancement Proposal 8 (PEP-8)[1]. Our aims are to investigate the capabilities and limitations of various LLMs in detecting and explaining Python coding violations, benchmarking

ARO-The Scientific Journal of Koya University
Vol. XIII, No.2 (2025), Article ID: ARO.12395. 11 pages
DOI: 10.14500/aro.12395
Received: 01 July 2025; Accepted: 14 September 2025
Regular research paper; Published: 10 October 2025
†Corresponding author's e-mail: qusay.sarhan@uod.ac

1 https://peps.python.org/pep-0008/

their performance against traditional static code analysis tools, and exploring the influence of prompting strategies, code characteristics, and inter-LLM consistency on their effectiveness.

Python was selected over other programming languages for this research as it is currently ranked as the number one programming language according to the TIOBE Index[2]. Furthermore, it is the most widely used language in artificial intelligence/machine learning development due to its simple syntax, extensive libraries (such as TensorFlow, PyTorch, and scikit-learn), and strong community support (Raschka, Patterson and Nolet, 2020). Moreover, Python has become a dominant language for the Internet of Things (IoT), where it is used for programming microcontrollers (such as Raspberry Pi and MicroPython-based devices), integrating sensors, and building rapid IoT prototypes (Kedia, Kumari and Mundra, 2023). Its high-level abstractions, cross-platform compatibility, and many libraries (e.g., paho-mqtt and adafruit-io) make Python especially suitable for connecting devices, managing data, and enabling communication in IoT environments. As IoT systems increasingly rely on intelligent processing and edge computing, Python's role in bridging AI capabilities and IoT infrastructures becomes crucial for innovation and static code analysis applicability.

This study contributes to code analysis by evaluating how effectively and consistently LLMs detect coding violations in comparison to traditional static code analysis tools, as follows:

1. Comparative effectiveness analysis: Evaluates how well LLMs detect coding violations and benchmarks their performance against traditional static code analysis tools.
2. Taxonomy of violations: Develops a detailed taxonomy that classifies coding violations, showing where LLMs perform better or worse, thereby highlighting model strengths and limitations.
3. Cross-model consistency: Analyzes six state-of-the-art LLMs to identify consistent model-specific patterns across violation detection.
4. Batch versus Individual code analysis: Investigates the impact of granularity by comparing LLMs' performance on batch code snippets versus individual snippet analysis, thereby guiding effective use cases.
5. Dataset construction: Builds and releases a new, specialized dataset focused on evaluating Python coding violations, providing a benchmarking standard for benchmarking present and future models.

Collectively, this study advances the understanding of LLMs' applicability in Python code quality assessment, while offering a methodology that may be extended to other programming languages in future studies.

The remainder of this study is structured as follows: Section II reviews related work. Section III details the research methodology that have been used in this study. Section IV presents and discusses the experimental results of this study. Section V highlights the main limitations of using LLMs for static code analysis. Finally, Section VI concludes the study and suggests directions and future research.

II. Related Works

This section presents the most related works to this study. LLMs are being evaluated for their capability to detect software vulnerabilities in (Noever, 2023; Purba, et al., 2023). This includes evaluating their performance against traditional tools, such as Snyk and Fortify, or with pre-trained models for vulnerability detection. Specific approaches involve using LLMs for static taint analysis to infer APIs' taint specifications and classify vulnerable paths as True or FPs through contextual analysis, as in (Li, Dutta and Naik, 2025). Beyond mere detection, LLMs are also explored for broader software vulnerability analysis tasks, including vulnerability assessment (e.g., qualitative severity ratings), vulnerability location, and vulnerability description, providing more contextual information that can improve LLMs' vulnerability assessment capabilities (Yin, Ni and Wang, 2024). LLMs are used to identify and address general code quality problems (Souma, et al., 2023). This covers both error detection and correction (such as syntax and type errors). By producing and ranking code fixes to increase acceptance rates and reduce developer workload, LLMs assist in addressing code quality issues with tools, such as CodeQL and SonarQube, in the context of static code analysis (Wadhwa, et al., 2024).

In addition, LLMs are used to refactor and fix bugs found by static code analysis tools, especially those related to design, error proneness, best practices, and code style (AlOmar and Mkaouer, 2024). Although LLMs are good at detecting issues in single locations, they might not be able to grasp the larger code context for intricate design problems (Guo, et al., 2023b; Li, et al., 2023a). To determine whether potential bugs found by static code analysis are real bugs or FPs, LLMs might be consulted, particularly in circumstances that are unclear (Li, et al., 2023b). To lower the rate of FPs, tools, such as SkipAnalyzer, combine automatic bug repair, false-positive alerts, and LLM-based static bug detection (Mohajer, et al., 2023). Assessing LLMs' ability to perform program analysis tasks, such as abstract syntax tree generation, Expression Matching, control flow graph generation, call graph generation, data dependency analysis, taint analysis, and pointer analysis is a crucial component of integrating them with static code analysis (Ma, et al., 2024). For LLMs to be used correctly and sensibly in code-related activities, it is essential to appreciate both their strengths and weaknesses in understanding program syntax and statically approximating program behavior. Even if LLMs are promising for applications, such as type inference, more complex domains, such as CG analysis, may still be better served by traditional techniques (Venkatesh, et al., 2024). To summarize, the tasks assigned to LLMs start with a fundamental understanding of the code, along with static behavior approximation, followed by higher-order tasks, such as detecting and resolving numerous types of violations,

2 https://www.tiobe.com/tiobe-index/

and critically, refining the output of traditional static code analysis tools.

While prior research works have explored the capabilities of LLMs in software vulnerability detection, taint analysis, and broader vulnerability assessment, our study extends the research by focusing specifically on Python code violations, a critical yet underexplored aspect of static code analysis. Unlike previous works that primarily assess LLMs against security-focused tools, such as Snyk and Fortify, our study provides a comprehensive comparison with traditional static code analysis tools (i.e., Pylint and Flake8), emphasizing coding quality. In addition, while existing studies highlight LLMs' potential in error correction and false-positive reduction, our study introduces a novel taxonomy of 27 Python-specific coding violations and systematically evaluates six state-of-the-art LLMs across multiple prompting strategies. This granular approach not only identifies LLMs' strengths in detecting different violations but also reveals their limitations in semantic reasoning and stylistic consistency, a contribution that refines the understanding of LLMs' role in code quality assurance. Furthermore, our findings on batch versus individual code analysis and prompt sensitivity offer practical insights for integrating LLMs into developer workflows, advancing beyond the theoretical aspects proposed in earlier studies.

III. Research Methodology

This study's methodology consists of eight steps (shown in Fig. 1): identifying research questions (RQs), preparing the code snippet dataset, manually labeling the data, selecting LLMs and static code analysis tools, designing prompts for the LLMs, defining evaluation metrics, conducting experiments and analyzing results to assess LLMs' ability to detect Python coding violations. Each step is detailed in the following subsections:

A. RQs

Several RQs have been identified and answered in this study. Each RQ addresses a particular aspect of the topic, as follows:

- RQ1 (overall performance comparison): How effective are LLMs at identifying coding violations compared to traditional static code analysis tools?
- RQ2 (violation-type detection strengths and weaknesses): What types of coding violations are LLMs better at detecting compared to static code analysis tools, and which types do they struggle with?
- RQ3 (model consistency): How consistent are different LLMs in identifying the same coding violations?
- RQ4 (prompting strategies): Do different prompting strategies affect an LLM's model's ability to identify coding violations?
- RQ5 (Batch vs. individual code analysis): Does analyzing code snippets in batches, as opposed to individually, lead to different outcomes in the identification of coding violations?

B. Code Snippets Dataset Preparation

This study curated 75 Python code snippets extracted from LeetCode[3] website, across three difficulty levels (25 Easy, 25 Medium, and 25 Hard). Each snippet represents a realistic programming task rather than artificially constructed examples. The snippets were manually annotated by the authors for quality. The annotation followed the PEP-8 standards and included 27 common violation categories (Table I). Examples include stylistic violations (e.g., missing whitespace), documentation issues (e.g., absent function docstrings), and structural/design flaws (e.g., nested methods). Each violation was labeled independently by both authors, and disagreements were resolved through discussion, ensuring consistent ground truth labels. The full dataset and annotation guidelines are made publicly available in a GitHub repository[4].

Fig. 2 illustrates a sample code snippet from our dataset that demonstrates several violations of coding, their details are presented in Table II.

3 https://leetcode.com/
4 https://github.com/CodingWithLLMs/Online-CodeBase

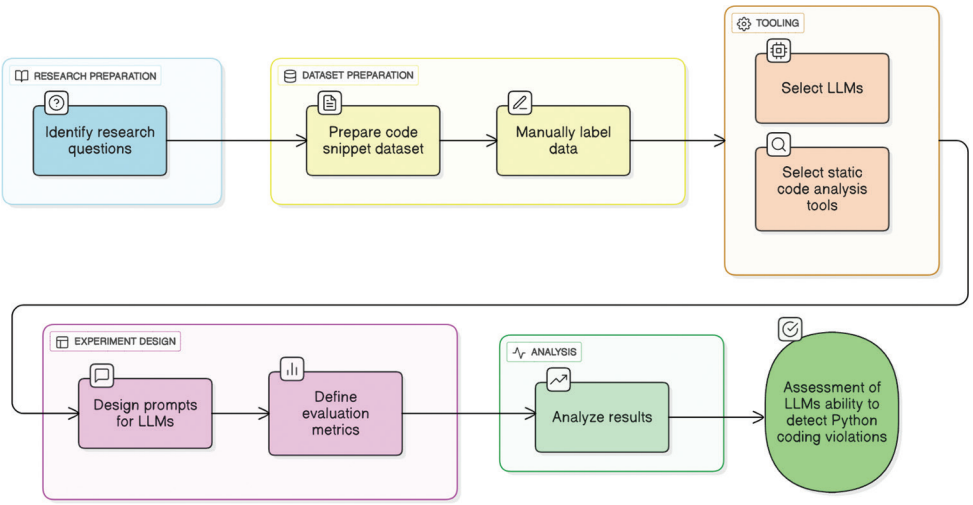

Fig. 1. The methodology of this study.

 http://dx.doi.org/10.14500/aro.12395

TABLE I
VIOLATIONS THAT LLMS DETECT OR MISS

#	Violations	Types	Claude	Gemini	Qwen	ChatGPT	Kimi	DeepSeek
1.	Too-many-local-variables-(>15)	Complexity/modularity	×	×	×	×	×	×
2.	Function-return-not-implemented	Design/completeness	×	✓	×	✓	×	×
3.	Class-inside-method	Design/modularity	×	×	×	×	×	×
4.	Method-inside-method-(no-single-responsibility)	Design/modularity	✓	×	×	×	×	×
5.	Add-meaningful-comments-to-complex-logic	Documentation/clarity	✓	✓	×	×	×	×
6.	Function-type-hints-Not-define	Documentation/clarity	✓	×	✓	×	×	×
7.	Class-doc-string-Not-written	Documentation/stylistic	✓	✓	×	×	×	×
8.	Function-doc-string-Not-written	Documentation/stylistic	✓	✓	✓	×	✓	✓
9.	Expected-blank-line-after-class	Layout/formatting	✓	✓	×	✓	×	×
10.	Expected-blank-line	Layout/formatting	✓	✓	×	×	×	×
11.	Expected-blank-line-before-a-nested-definition	Layout/formatting	✓	✓	×	×	×	×
12.	Expected-a-blank-line-after-method	Layout/formatting	✓	✓	×	×	×	×
13.	Line-too-long-(>-79-characters)	Layout/formatting	✓	✓	×	×	✓	✓
14.	Missing-Space-around-operator	Layout/formatting	✓	✓	✓	✓	✓	✓
15.	Missing-whitespace-after-(comma)	Layout/formatting	✓	✓	✓	✓	×	×
16.	Missing-whitespace-after-(colon)	Layout/formatting	✓	✓	✓	×	×	×
17.	Unused-variable	Maintainability/efficiency	×	×	×	×	×	×
18.	Function-name-not-snake-case	Naming	×	✓	✓	✓	✓	×
19.	Variables-naming-conventions	Naming	✓	×	×	×	×	×
20.	Class-name-should-be-snake-case	Naming	×	×	×	×	×	×
21.	Consider-using-enumerate-instead-of-iterating	Readability	×	×	×	×	×	×
22.	Unnecessary-“elif”-after-“return”	Readability/refactoring	×	×	×	×	×	×
23.	Unnecessary-“else”-after-“return”	Readability/refactoring	×	×	×	×	×	×
24.	Redefining-built-in-“next”	Reliability/clarity	×	×	×	×	×	×
25.	Functions-input-not-validated-early	Robustness/security	✓	✓	×	×	×	×
26.	Undefined-variable	Runtime error/logic	×	✓	×	×	×	×
27.	Import-should-be-outside-top-level	Structural/layout	✓	✓	×	✓	×	✓

TABLE II
CODING VIOLATIONS IDENTIFIED

#	Violation	Frequency
1.	Function-name-not-snake-case	2
2.	Missing-whitespace-after-comma	8
3.	Function-type-hints-Not-written	2
4.	Function-doc-string-Not-written	2
5.	Missing-Space-around-operators	3
6.	Expected-blank-line-before-a-nested-definition	1
7.	Add-meaningful-comments-to-logic	1
8.	Method-inside-method-(no-single-responsibility)	1
9.	Functions-input-not-Validate-early	1
Total violations		21

```
def combinationSum(candidates,target):
    result=[]
    def backtrack(start,path,remaining):
        if remaining==0:
            result.append(path)
            return
        for i in range(start,len(candidates)):
            if candidates[i]>remaining:
                continue
            backtrack(i,path+[candidates[i]],remaining-candidates[i])
    backtrack(0,[],target)
    return result
```

Fig. 2. Sample Python code with different violations.

C. LLMs Selection

To assess the capabilities and limitations of LLMs in detecting different coding violations while ensuring robust and reliable results, we selected six state-of-the-art models: DeepSeek-V3[5] (DeepSeek) from DeepSeek, ChatGPT 4-Turbo[6] (ChatGPT) from OpenAI, Claude 3.7-Sonnet[7] (Claude) from Anthropic, Gemini 2.5-pro-preview-03-25c[8] (Gemini) from Google, Kimi[9] from Moonshot AI and Qwen 2.5-Max[10] (Qwen) from Alibaba. These models were chosen to ensure a comprehensive evaluation of present LLM performance in code-related tasks.

D. Static Code Analysis Tools Selection

To ensure rigorous evaluation of code quality, we employed two industry-standard Python static code analysis tools, Pylint (Pylint-Dev., 2024) and Flake8 (Flake8-Dev, 2025). These tools complement each other: Flake8 ensures strict PEP-8 compliance, while Pylint covers broader software engineering and coding best practices, together providing a strong selection for evaluating both style and code quality.

5 https://www.deepseek.com/
6 https://chatgpt.com/
7 https://claude.ai/
8 https://gemini.google.com/
9 https://www.kimi.com/
10 https://chat.qwen.ai/

E. Prompt Design

We employed three common prompting strategies to evaluate LLMs' performance:

- Structural prompt (P1): Example of structural prompt: *"Analyze the following Python code for any PEP-8 violations or potential issues"*.
- Chain of thought (CoT) prompt (P2): Example of CoT prompt: *"Analyze the provided Python code for PEP-8 violations, potential issues, or areas of improvement. Use a Chain of Thought (CoT) approach to systematically evaluate the code"*.
- Role-based prompt (P3): Example of role-based prompt: *"As a software engineer, you are tasked with analyzing the following Python code for any PEP-8 violations or potential issues"*.

The three prompt types structural, CoT, and role-based, were selected due to their proven ability to enhance LLM performance by providing essential guidance. Structural prompts boost clarity and specificity by incorporating detailed task descriptions, explicit output formats (such as JSON), and crucial code context, leading to more precise and relevant outputs (Liu, Yang and Liao, 2024). CoT prompts enable LLMs to tackle complex problems by encouraging step-by-step reasoning and problem decomposition, which in turn improves accuracy and the robustness of responses (Li, et al., 2023a). Finally, role-based prompts assign a specific persona to the LLM (e.g., "vulnerability detection system"), establishing a clear task context that helps the model focus and significantly improves its performance in generating desired outputs (Hajipour, et al., 2024).

F. Evaluation Metrics

To address the majority of the RQs, we employed a set of established evaluation metrics, including true positive (TP), false negative (FN), FP, precision, recall, and F1-Score (Omar and Shiaeles, 2023; Ignatyev, et al., 2024). For specific cases requiring additional scrutiny, supplementary metrics, such as false discovery rate (FDR) (Li, Dutta and Naik, 2025), consistency rate (CR), average CR (Carandang, et al., 2025), prompt agreement (PA) (Mousavi, Alghisi and Riccardi, 2025), and lines of code were utilized.

- To answer RQ1 (overall performance comparison): After obtaining the results from the static code analysis tools and LLMs (processing 25 code snippets at a time in JSON format), we manually labeled each prediction as TP, FP, or FN based on ground truths. The correctness of labeling was manually validated by the authors. We then calculated precision, recall, F1-Score, and FDR for the entire dataset, as defined in equations (1), (2), (3), and (4).

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2}$$

$$\text{F1} - \text{Score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

$$\text{FDR} = \frac{\text{FP}}{\text{FP} + \text{TP}} \tag{4}$$

Where:

- TP: Correctly predicted positive case.
- FP: Incorrectly predicted positive when the actual is negative.
- FN: Incorrectly predicted negative when the actual is positive.
- Precision: How well the model avoids FPs.
- Recall: How well the model finds all actual positives.
- F1-Score: Harmonic mean of precision and recall.
- FDR: Proportion of FPs, among all predicted positives.

- To answer RQ2 (violation-type detection strengths and weaknesses): The results were filtered by classification (FN and TP) and grouped by type and subtype to assess the LLM's strengths and weaknesses in detecting violation types.
- To answer RQ3 (model consistency): The obtained results were filtered to find cases with both FN and TP for each LLM, enabling consistency assessment. Average CR was computed across three prompts (P1, P2, and P3) to evaluate performance stability, and PA was calculated to identify the LLM with the highest agreement rate, as defined in equations (5) and (6).

$$\text{AverageCR} = \frac{1}{n} \sum_{i=1}^{n} \text{Recall}\, i \tag{5}$$

$$\text{PA} = \frac{\#\ \text{Prompts with} \geq 1\ \text{TP}}{\text{Total Prompts}} \tag{6}$$

Where:

- N: Total number of prompts.
- PA: Measures LLM consistency by evaluating responses to varied (perturbed) versions of the same prompt.

- To answer RQ4 (prompting strategies): To improve LLM output consistency, we used three prompting strategies: structured, CoT, and role-based, and evaluated them using precision, recall, and F1-Score to assess their impact on the LLMs' performance.
- To answer RQ5 (Batch vs. Individual Analysis): We selected 15 code snippets (five per difficulty level) and ran an evaluation using only the (P1) prompt to check if individual snippet assessments differ across all LLMs. We then manually annotated model predictions as TP, FP, or FN by comparing them to ground truth labels. Using this, we calculated precision, recall, and F1-Score for the (P1) dataset and compared these results with those from (RQ1) to identify any discrepancies or consistencies.

IV. Results and Discussion

In this section, the experimental results of this study are presented and discussed. Each RQ is summarized with a short title and discussed in its respective subsection based on the study's findings.

http://dx.doi.org/10.14500/aro.12395

A. Overall Performance Comparison (RQ1)

The obtained results for this RQ, based on three sessions per LLM using 25 code snippets each with P1, are shown in Fig. 3. Claude achieves the highest F1-Score (0.81), demonstrating a strong balance between precision (0.99) and recall (0.69), closely followed by Flake8 with F1-Score (0.79), which maintains perfect Precision (1.00) but slightly lower recall (0.66). Gemini also performs well with F1-score (0.78), with near-perfect precision (0.99), and moderate recall (0.64). In contrast, Kimi with F1-Score (0.56), DeepSeek with F1-Score (0.46), and Qwen with F1-Score (0.36) exhibit declining performance due to lower recall values of 0.39, 0.34, and 0.23, respectively, despite relatively high Precision. Notably, Pylint and ChatGPT show the weakest performance, with F1-Scores of 0.35 and 0.13, respectively, primarily due to their extremely low recall of 0.21 and 0.07, despite perfect or near-perfect Precision. These results reveal that while some LLMs, such as Claude, can outperform traditional static code analysis tools.

Another aspect is important for the overall performance of LLMs, which is FDR (Table III), which compares the FDR of the LLMs across three prompts (P1, P2, and P3), revealing significant variations in their reliability. Kimi emerges as the top performer with a perfect FDR of 0.00 across all prompts, demonstrating exceptional precision. Claude and Gemini also perform remarkably well, both maintaining an average FDR of just 0.01, with Claude achieving a flawless (0.00) with the third prompt. ChatGPT shows strong overall performance with an average FDR of 0.06, though its rate rises to 0.17 with the third prompt, indicating some prompt sensitivity. DeepSeek exhibits consistently higher FDRs between 0.26 and 0.35, averaging 0.29, suggesting a tendency for more FP. Most notably, Qwen displays extreme variability, with FDRs ranging from 0.00 with the second prompt to 0.92 with the third prompt, resulting in an average of 0.36, this dramatic fluctuation highlights potential instability in certain contexts. These findings suggest that while models, such as Kimi, Claude, and Gemini, are highly reliable for precision-critical tasks, others like Qwen, may require additional safeguards or carefully engineered prompts to mitigate false discoveries. The results underscore the importance of both model selection and prompt design when deploying LLMs in applications with high performance variability, emphasizing the importance of ensuring consistency across different prompt types.

TABLE III
LLMs' FDR Across Three Prompts

#	LLMs	FDR			Average FDR
		Round-1-(P1)	Round-2-(P2)	Round-3-(P3)	
1.	Qwen	0.17	0.00	0.92	0.36
2.	DeepSeek	0.26	0.35	0.27	0.29
3.	ChatGPT	0.01	0.01	0.17	0.06
4.	Gemini	0.01	0.00	0.02	0.01
5.	Claude	0.01	0.02	0.00	0.01
6.	Kimi	0.00	0.00	0.00	0.00

FDR: False discovery rate

B. Violation-Type Detection Strengths and Weaknesses (RQ2)

Here, the obtained data from three prompts (P1, P2, and P3) are used to evaluate how well static code analysis tools and LLMs identify 27 coding violations in code snippets, distinguishing correct from incorrect identifications, as shown in Fig. 4. The results demonstrate significant variability in performance across the evaluated models. Claude and Gemini emerge as the most effective models, each correctly identifying 16 violations while making only 11 incorrect identifications. In contrast, traditional static code analysis tools, such as Flake8, show moderate performance with 15 correct identifications and 12 incorrect ones, while Pylint performed only 8 correct detections against 19 incorrect ones. The remaining LLMs exhibit progressively weaker performance; Qwen and ChatGPT both performed 6 correct and 21 incorrect detections, demonstrating particularly low accuracy. Notably, Kimi and DeepSeek perform the poorest, with only 4 correct identifications, alongside 23 incorrect ones. These findings suggest that while some LLMs, such as Claude, can outperform traditional static code analysis tools in identifying coding violations, many present LLMs still struggle with accuracy in this domain. The results highlight the need for further refinement of LLMs for static code analysis tasks, particularly in reducing FP while maintaining detection rates.

In addition, the capability of LLMs to detect violations that are missed by static code analysis tools is shown in Table IV, which reveals that LLMs are generally more

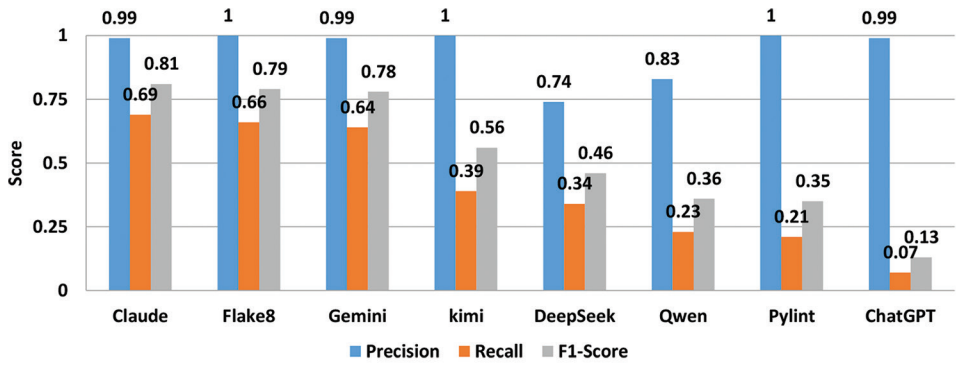

Fig. 3. Performance comparison of large language models versus static code analysis tools.

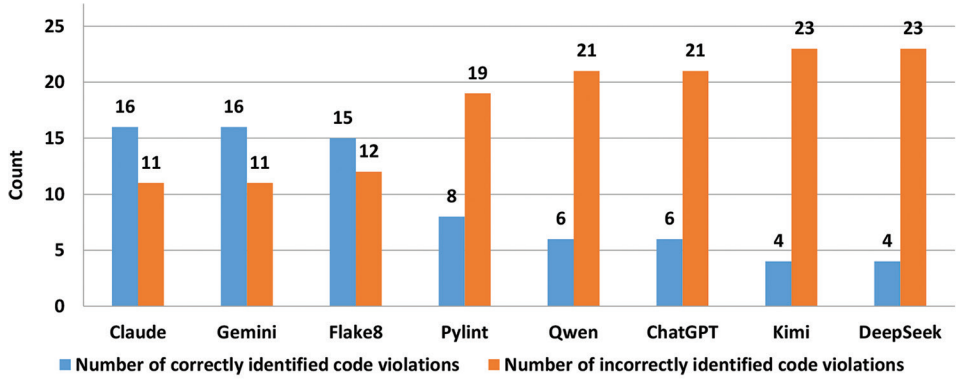

Fig. 4. Detected violations by static code analysis tools versus large language models.

TABLE IV
LLMs' Detection of Violations Missed by Static Code Analysis Tools

#	LLMs	Violations	Types
1.	ChatGPT	Import-should-be-outside-top-level	Structural/layout
		Function-return-not-implemented	Design/completeness
		Class-attributes-should-be-spaced	Layout/formatting
2.	Claude	Add-comments-to-complex-logic	Documentation/clarity
		Type-hints-are-not-defined	Documentation/clarity
		Method-inside-method-(no-single-responsibility)	Design/modularity
		Function's-input-not-validated-early	Robustness/security
		Variables-naming-conventions	Naming
3.	Kimi	Use-consistent-string-quotes	Layout/formatting
4.	Gemini	Add-comments-to-complex-logic	Documentation/clarity
		Function's-input-not-validated-early	Robustness/security
		Type-hints-are-not-defined	Documentation/clarity
5.	Qwen	Type-hints-are-not-define	Documentation/clarity
		Avoid-hardcoding-values	Naming
6.	DeepSeek	NON	

effective in identifying coding violations than traditional static code analysis tools. These violations are categorized into Structural/Layout, Naming, and Robustness/Security. ChatGPT is particularly effective in identifying stylistic violations, while Claude is strong in detecting documentation gaps and design flaws. Gemini and Qwen primarily highlight documentation-related issues, whereas Kimi focuses on stylistic consistency. DeepSeek, on the other hand, failed to identify any additional violations detected by static code analysis tools, indicating lower performance. Despite their variability in accuracy, LLMs show particular strengths in identifying nuanced or context-dependent violations, suggesting they could complement static code analysis tools by addressing gaps in detecting documentation and design-related issues.

Collectively, Table I highlights the coding violations that LLMs correctly detect (marked with a ✓ sign) and miss (marked with a × sign), illustrating their limitations. ChatGPT missed critical violations, such as undefined variables and complex methods, while Claude failed to detect modularity violations. Qwen showed the worst performance, confirming its lower accuracy. Overall, LLMs struggle with consistency and context-sensitive violations, making them unreliable as standalone linters. Hybrid approaches that combine static code analysis tools for syntax checks with LLMs for higher-level guidance are recommended to address these gaps.

To further illustrate the limitations of LLMs, the results in Table I confirm that LLMs struggle with consistency and context-sensitive violations, limiting their reliability as standalone tools. A closer look in Fig. 5 and Table V shows how Gemini exemplifies these weaknesses. In the combination Sum function, Gemini detected some stylistic issues (e.g., missing whitespace and missing docstrings) but failed to flag deeper problems, such as method nesting, missing input validation, and lack of comments. It also underreported the frequency of stylistic errors. Taken together, the aggregate evidence from Table I and this qualitative case confirms a consistent trend: LLMs handle surface-level violations reasonably well but frequently miss structural and semantic violations, reinforcing the need for hybrid approaches that combine static code analysis tools with LLM-based reasoning.

Ground truth violations identified:

- Function-name-not-in-snake_case
- Missing-whitespace-after-comma (8 instances)
- Missing-type-hints (2 instances)
- Missing-function-docstring (2 instances)
- Missing-class-docstring
- Missing-space-around-operator (3 instances)
- Missing-blank-line-before-nested-definition
- Missing-meaningful-comments
- Method-defined-inside-method
- Function-input-not-validated-early

C. Model Consistency (RQ3)

LLMs can produce unreliable and inconsistent responses due to stochasticity, output unpredictability, and hallucination, where they generate non-sensical or inaccurate answers. Prompt engineering was selected as a solution to mitigate stochasticity and hallucinations, particularly given the limitations of free web-based versions without API control. Three tailored prompts (P1, P2, and P3) were applied iteratively to batches of 25 code snippets. Only clear TP and FN results were collected and summarized in Fig. 6. This approach allowed for a structured evaluation of model performance under constrained experimental conditions. The results reveal that Claude and Gemini

achieved the highest average detection rate, with 5 violations each. Claude demonstrated consistent performance across prompts 4, 7, and 5, while Gemini showed a notable peak in P2, detecting 8 violations. ChatGPT follows with a moderate average 4 violations, while DeepSeek, Qwen, and Kimi exhibit significantly lower detection rates averaging 1, 1, and 0 violations, respectively. The study indicates that while some LLMs can identify violations with contextual or semantic complexity, others struggle with simple violations, highlighting the need for model-specific refinement.

To further evaluate the consistency of LLMs, two additional metrics, average CR and PA, were employed. The results presented in Fig. 7 reveal pronounced disparities in LLMs' reliability. Claude achieved the highest average CR (0.59) and PA (0.41). However, its CR variability (0.70 in P1 vs. 0.50 in P2) suggests lingering stochasticity. Gemini followed with moderate consistency CR (0.48) and PA (0.32), though its CR drops sharply in P3 (0.24), indicating prompt sensitivity. In contrast, ChatGPT demonstrated critically low consistency CR (0.06) and PA (0.16), mirroring its middling detection rates and underscoring its unreliability for static code analysis. The remaining models (Qwen, Kimi, and DeepSeek) exhibited inconsistent CR trends (e.g., DeepSeek's P3 surged to 0.36 vs. 0.06 in P2) and negligible PA (≤0.13), corroborating their poor violation detection noted earlier. These findings confirm that LLMs, including top models, such as Claude and Gemini, face challenges related to randomness and hallucinations, emphasizing the importance of consistency when applying LLMs for static code analysis.

TABLE V
GEMINI OUTPUT VERSUS GROUND TRUTH

#	Violation	Gemini result	Evaluation
1	Missing-whitespace-after-comma	Detected 2	TP (2), FN (6)
2	Missing-space-around-operator	Detected 1	TP (1), FN (2)
3	Missing-function-docstring	Detected 2	TP (2)
4	Function-name-not-snake_case	Missed	FN
5	Missing-type-hints	Missed	FN (2)
6	Missing-class-docstring	Missed	FN
7	Function-input-not-validated-early	Missed	FN
8	Missing-blank-line-before-nested-definition	Missed	FN
9	Add-meaningful-comments	Missed	FN
10	Method-inside-method	Missed	FN

TP: True positive, FN: False negative

```
def combinationSum(candidates,target):
    result=[]
    def backtrack(start,path,remaining):
        if remaining==0:
            result.append(path)
            return
        for i in range(start,len(candidates)):
            if candidates[i]>remaining:
                continue
            backtrack(i,path+[candidates[i]],remaining-candidates[i])
    backtrack(0,[],target)
    return result
```

Fig. 5. Code Snippet with ground-truth violations.

D. Prompting Strategies (RQ4)

The design of prompts significantly influences the accuracy and overall performance of LLMs (Guo, et al., 2023a). In this study, we evaluate three prompt designs (P1, P2, and P3), with Fig. 8 showing their comparison across six LLMs. The results reveal substantial variations in performance. ChatGPT demonstrated high precision (0.99) across P1 and P2 but suffered from extremely low recall (0.07) and F1-Scores (0.13), indicating a trade-off between accuracy and coverage. Gemini performed notably well with P1, achieving an F1-Score of 0.78, but its efficacy declined with P2 (0.72) and P3 (0.39), suggesting that P1 may be more effective for this model. Claude showed strong performance across all prompt types, particularly in P1 with F1-Score of 0.81 and in P3 with F1-Score of 0.78, highlighting its adaptability. In contrast, DeepSeek and Qwen exhibited inconsistent results. Qwen showed particularly poor performance in P3 with F1-Score of 0.01. Kimi maintained perfect Precision (1.0) across all prompts but struggled with recall and F1-Scores, indicating a potential limitation in response completeness. Overall, the results indicate that prompt design has a major effect on LLM performance, with P1 typically producing the most balanced and reliable outcomes across models.

E. Batch versus Individual Code Analysis (RQ5)

Analyzing code snippets in batches versus individually can yield different results in detecting coding violations by LLMs (Yin, Ni and Wang, 2024). The impact depends on the batching strategy and task complexity. Fig. 9 shows LLM's performance when analyzing individual

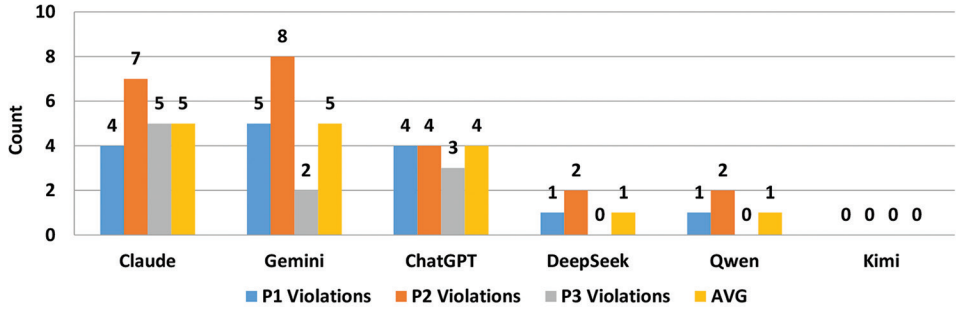

Fig. 6. Frequency of coding violations with simultaneous true positive and true negative.

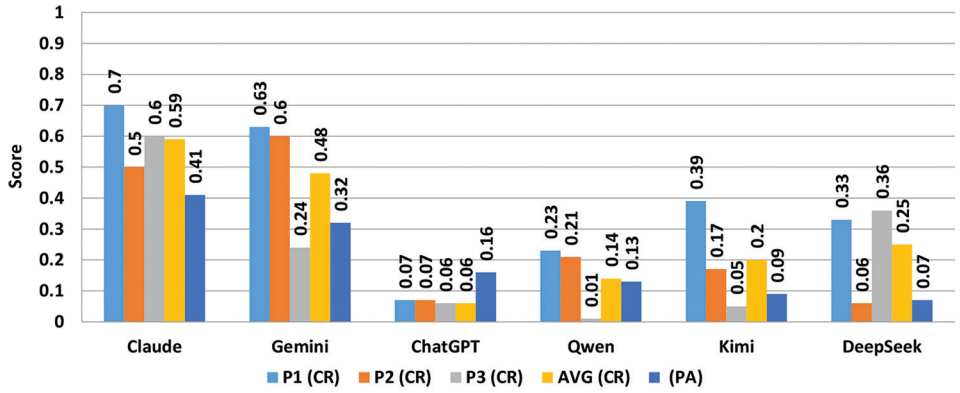

Fig. 7. Average consistency rate and prompt agreement across evaluated large language models.

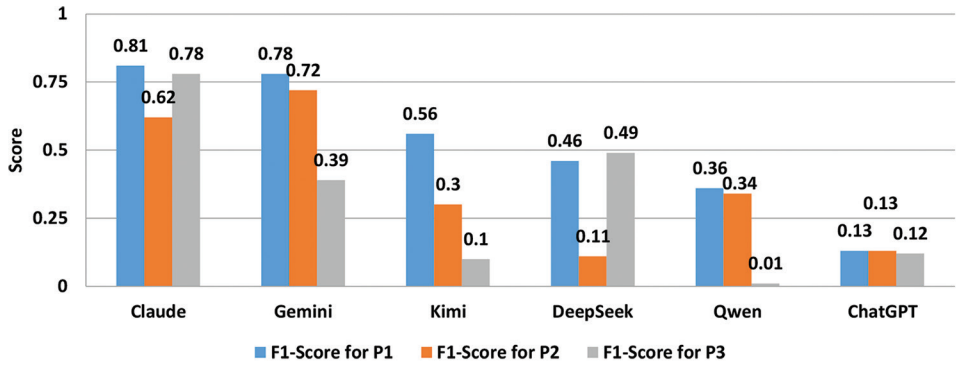

Fig. 8. Comparing prompt design efficiency in large language models.

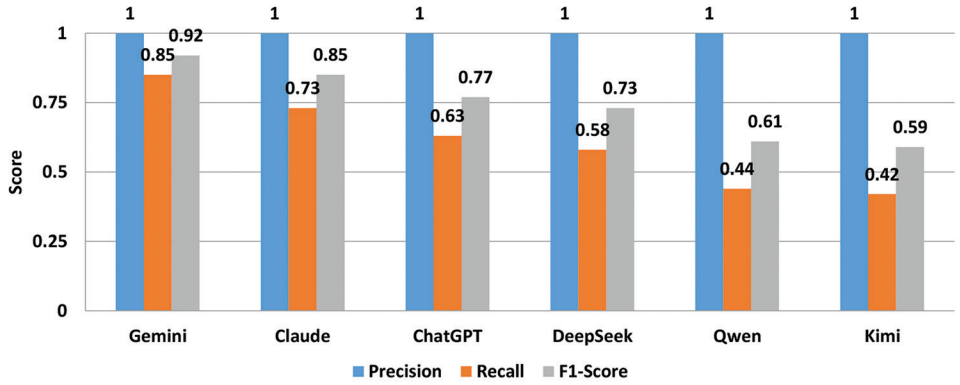

Fig. 9. Overall performance of large language models (Individual code).

code snippets. Notably, all models achieved perfect Precision (1.0), indicating that their identified violations were consistently correct. However, recall scores varied significantly, reflecting differences in the LLM's ability to detect all existing violations. Gemini demonstrated the highest recall (0.85) and F1-score (0.92), suggesting robust overall performance, while Claude with a recall (0.73) and F1-score (0.85) and ChatGPT with a recall (0.63) and F1-score (0.77) exhibited moderate effectiveness. In contrast, DeepSeek, Qwen, and Kimi showed markedly lower recall of 0.58, 0.44, and 0.42, respectively, and F1-Scores, underscoring their limitations in comprehensive violation detection. The findings suggest that while all models excel at avoiding precision, their utility for thorough static code analysis hinges on improving recall, particularly for lower-performing models.

Furthermore, Table VI presents a comparative analysis of the F1-Score performance of various LLMs when evaluating batch versus individual code input. The results demonstrate that all LLMs exhibit improved performance when analyzing code snippets individually rather than in batches. ChatGPT shows the most significant improvement, increasing its F1-score from 0.13 in batch mode to 0.77 in individual mode, yielding an improvement of +0.64. DeepSeek and Qwen also demonstrate notable gains of +0.27 and +0.25, respectively. Gemini shows a moderate increase of +0.14, while Claude and Kimi display minimal improvements of +0.04 and +0.03, respectively. These findings suggest that processing

http://dx.doi.org/10.14500/aro.12395

TABLE VI
F1-Score Performance Comparison (Batch versus Individual code)

#	LLMs	F1-score (batch)	F1-score (individual)	Overall improvement
1.	ChatGPT	0.13	0.77	+0.64
2.	DeepSeek	0.46	0.73	+0.27
3.	Qwen	0.36	0.61	+0.25
4.	Gemini	0.78	0.92	+0.14
5.	Claude	0.81	0.85	+0.04
6.	Kimi	0.56	0.59	+0.03

code snippets individually rather than in batches positively influences the effectiveness of LLMs in identifying coding violations.

To sum up, our study provides Python-specific empirical evidence that certain LLMs can effectively complement static code analysis tools. Future research should investigate whether these findings generalize to other programming languages and domains.

V. Limitations and Challenges

Implementing LLMs to detect coding violations faces several key challenges. Their limited context window restricts the understanding of large or multi-file codebases, hindering recognition of project-wide dependencies and design issues (Ignatyev, et al., 2024). Although they are effective in syntax analysis, LLMs struggle with deep semantic comprehension and intricate logic; they may misinterpret issues or offer suggested solutions that inadvertently increase complexity (Souma, et al., 2023). An important limitation of relying solely on LLMs is the risk of FN, where genuine violations or vulnerabilities are missed. Unlike deterministic static code analysis tools that guarantee detection of specific rule-based violations, LLMs provide probabilistic outputs without formal correctness guarantees. In security-sensitive contexts, such omissions could leave critical flaws undetected, creating a false sense of assurance for developers. For example, failing to detect invalidated user input or hardcoded credentials could result in exploitable vulnerabilities despite the model reporting clean code. Hybrid approaches, where LLMs provide contextual reasoning and static code tools enforce deterministic checks, may offer a more reliable pathway toward safe adoption in industrial settings.

LLMs frequently rely on well-crafted user prompts and tend to prioritize syntax over functional accuracy (Liu, Yang and Liao, 2024). Due to inherent biases and limitations in training data, LLMs could memorize patterns rather than fully internalize best practices. Without careful consideration, it is difficult for developers to trust their suggestions because of their ambiguous reasoning. Although LLMs show promise for code violation detection, their deployment in industrial settings faces challenges beyond accuracy. Cloud-based APIs introduce latency and cannot guarantee real-time responsiveness, making them less reliable than local static code analysis tools, such as Pylint and Flake8. In addition, LLMs-based workflows often require iterative user feedback, are constrained by limited context windows, and can incur substantial costs for large-scale projects. Consequently, LLMs are best positioned as assistive tools that complement rather than replace traditional static code analysis tools. Broader adoption will depend on advances in latency reduction, offline inference, and mechanisms for seamless integration of user feedback into development environments.

VI. Conclusion

The ability of LLMs in detecting coding violations was assessed in this study, and their effectiveness was contrasted with traditional static code analysis tools. The results show that while LLMs, such as Claude and Gemini, may compete with or even outperform static code analysis tools in some areas; their efficacy varies considerably depending on the model and type of violation. Overall, Claude emerged as the top-performing LLM (F1-Score: 0.81), showcasing strengths in contextual understanding and recall, while static code analysis tools, such as Flake8 excelled in precision (1.00) but lagged in detecting nuanced violations. In terms of advantages and disadvantages, within the context of Python, LLMs can complement static code analysis tools in certain areas, but their non-deterministic nature and susceptibility to FNs make them unsuitable as standalone tools for security critical applications. Future work should examine whether these findings generalize to languages with stricter typing or different structural characteristics. Regarding the prompting strategies, structural prompts (P1) yielded the most balanced results, whereas CoT and role-based prompts showed mixed efficacy, underscoring the importance of prompt engineering. LLMs faced limitations in batch processing and long-context analysis, with individual snippet evaluation proving more reliable. Our contribution lies in the empirical evidence, the proposed taxonomy, dataset, and practical insights for integrating LLMs with existing tools.

In the future, we aim to expand this study by including more LLMs, additional prompt types, and a wider range of coding violations. Moreover, combining LLMs with static code analysis tools to create neuro-symbolic systems that reduce FP and enhance vulnerability detection in terms of integration and practical application is a crucial next step. Finally, we aim to develop a benchmark dataset specifically for static code analysis to support researchers in contributing effectively contribute to this rapidly evolving field.

References

AlOmar, E.A., and Mkaouer, M.W., 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, United States, pp.1-10.

Carandang, K.A.M., Arana, J.M., Casin, E.R., Monterola, C., Tan, D.S., Valenzuela, J.F.B., and Alis, C., 2025. Are LLMs reliable? An exploration of the reliability of large language models in clinical note generation. In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*. Association for Computational Linguistics, Stroudsburg, PA, USA, pp.1413-1422.

Flake8-Dev., 2025. *Flake8 : Is a Python Tool That Check the Style and Quality of Some Python Code*. Available from: https://github.com/pycqa/flake8 [Last accessed on 2025 Feb 17].

Guo, Q., Cao, J., Xie, X., Liu, S., Li, X., Chen, B., and Peng, X., 2023a. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp.1-13.

Guo, Z., Tan, T., Liu, S., Liu, X., Lai, W., Yang, Y., Li, Y., Chen, L., Dong, W., and Zhou, Y., 2023b. Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. *IEEE Transactions on Software Engineering*, 49(12), pp.5154-5188.

Haindl, P., and Weinberger, G., 2024. Does chatGPT Help novice programmers write better code? results from static code analysis. *IEEE Access*, 12, pp.114146-114156.

Hajipour, H., Hassler, K., Holz, T., Schönherr, L., and Fritz, M., 2024. CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models. In: *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, United States, pp.684-709.

Ignatyev, V.N., Shimchik, N.V., Panov, D.D., and Mitrofanov, A.A., 2024. Large language models in source code static analysis. In: *2024 Ivannikov Memorial Workshop (IVMEM)*. IEEE, United States, pp.28-35.

Jesse, K., Ahmed, T., Devanbu, P.T., and Morgan, E., 2023. Large language models and simple, stupid bugs. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, United States, pp.563-575.

Kedia, N.K., Kumari, H., and Mundra, S., 2023. A review paper on python for data science and machine learning. *Journal of Analysis and Computations*, 17(2), pp.97-103.

Li, H., Hao, Y., Zhai, Y., and Qian, Z., 2023a. Assisting static analysis with large language models: A ChatGPT experiment. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA: pp.2107-2111.

Li, H., Hao, Y., Zhai, Y., and Qian, Z., 2023b. *The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models*. Cornell University, United States.

Li, Z., Dutta, S., and Naik, M., 2025. *IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities*. Cornell University, United States, pp.1-24.

Liu, Z., Yang, Z., and Liao, Q., 2024. Exploration On Prompting LLM With Code-Specific Information For Vulnerability Detection. In: *Proceedings - 2024 IEEE International Conference on Software Services Engineering*, *SSE 2024*, pp.273-281.

Ma, W., Liu, S., Lin, Z., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., Li, L., and Liu, Y., 2024. *LLMs: Understanding Code Syntax and Semantics for Code Analysis*. Cornell University, United States.

Mohajer, M.M., Aleithan, R., Harzevili, N.S., Wei, M., Belle, A.B., Pham, H.V., and Wang, S., 2023. *SkipAnalyzer: A Tool for Static Code Analysis with Large Language Models*. Cornell University, United States.

Moratis, K., Diamantopoulos, T., Nastos, D.N., and Symeonidis, A., 2024. Write me This Code: An Analysis of ChatGPT Quality for Producing Source Code. *Proceedings - 2024 IEEE/ACM 21st International Conference on Mining Software Repositories, MSR 2024*, pp.147-151.

Mousavi, S.M., Alghisi, S., and Riccardi, G., 2025. *LLMs as Repositories of Factual Knowledge: Limitations and Solutions*. Cornell University, United States, pp.1-13.

Noever, D., 2023. *Can Large Language Models Find and Fix Vulnerable Software*? Cornell University, United States.

Omar, M., and Shiaeles, S., 2023. VulDetect: A novel technique for detecting software vulnerabilities using language models. In: *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, United States, pp.105-110.

Pearce, H., Tan, B., Ahmad, B., Karri, R., and Dolan-Gavitt, B., 2023. Examining zero-shot vulnerability repair with large language models. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, United States, pp.2339-2356.

Purba, M.D., Ghosh, A., Radford, B.J., and Chu, B., 2023. Software Vulnerability Detection using Large Language Models. In: *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, United States, pp.112-119.

Pylint-Dev., 2024. *Pylint: It's Not Just a Linter That Annoys You*. Available from: https://github.com/pylint-dev/pylint [Last accessed on 2025 Feb 17].

Raschka, S., Patterson, J., and Nolet, C., 2020. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information (Switzerland)*, 11(4), p.193.

Ságodi, Z., Siket, I., and Ferenc, R., 2024. Methodology for code synthesis evaluation of LLMs presented by a case study of ChatGPT and copilot. *IEEE Access*, 12, pp.72303-72316.

Souma, N., Ito, W., Obara, M., Kawaguchi, T., Akinobu, Y., Kurabayashi, T., Tanno, H., and Kuramitsu, K., 2023. Can chatGPT correct code based on logical steps. *Proceedings Asia-Pacific Software Engineering Conference, APSEC*, 2, pp.653-654.

Venkatesh, A.P.S., Sabu, S., Mir, A.M., Reis, S., and Bodden, E., 2024. The emergence of large language models in static analysis: A first look through micro-benchmarks. In: *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. ACM, New York, NY, USA, pp.35-39.

Wadhwa, N., Pradhan, J., Sonwane, A., Sahu, S.P., Natarajan, N., Kanade, A., Parthasarathy, S., and Rajamani, S., 2024. CORE: Resolving code quality issues using LLMs. *Proceedings of the ACM on Software Engineering*, 1(FSE), pp.789-811.

Yin, X., Ni, C., and Wang, S., 2024. Multitask-based evaluation of open-source LLM on software vulnerability. *IEEE Transactions on Software Engineering*, 50(11), pp.3071-3087.

Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., and Chen, Z., 2024. *A Survey on Large Language Models for Software Engineering*. Cornell University, United States.