A Systematic Survey on Large Language Models for Static Code Analysis

Hekar A. Mohammed Salih[1†] and Qusay I. Sarhan[2]

[1]Department of Computer Science, College of Science, University of Duhok, Zakho Street 38 AJ, Duhok, Kurdistan Region - Iraq

[2]Department of Computer Science, College of Science, University of Duhok, Zakho Street 38 AJ, Duhok, Kurdistan Region - Iraq

***Abstract*—*Static* code analysis plays a pivotal role in improving software quality, security, and maintainability by detecting vulnerabilities, errors, and programming issues in source code without executing it. Recent advancements in artificial intelligence, especially the development of large language models (LLMs), such as ChatGPT, have enabled transformational opportunities in this domain. Thus, it is essential to explore this emerging field of research from many perspectives. This systematic survey focuses on the use of LLMs for static code analysis, detailing their applications, advantages, contexts, limitations, etc. The study examines research papers published on the topic from reputable literature databases to answer several research questions regarding the state-of-the-art use of LLMs in static code analysis. In addition, different research gaps and challenges were identified and discussed alongside many directions. The results of this study demonstrate how LLMs can enhance static code analysis and address existing limitations, paving the way for developers and researchers to employ LLMs for a more affordable and effective software development process.**

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

I. Introduction

Static code analysis is a crucial activity in software development, which is aimed at detecting possible vulnerabilities, defects, or other issues related to code quality without executing the program (Louridas, 2006). Traditional static analysis techniques are based on rules and heuristics defined to analyze the source code; however, they often face challenges with modern complex systems and new coding styles. Static code analysis is essential in several domains including software engineering, cybersecurity, and the Internet of Things (IoT, henceforth). For example, in software engineering, static code analysis is critical for the prompt resolution of potential code threats, which in turn makes it easy to develop software programs that are competent, high-quality, reliable, and error-free (Ramamoorthy, et al., 2024). In the area of cybersecurity, it helps locate and understand vulnerabilities in sensitive information systems, such as banking and healthcare systems (Hassan, Sarhan and Beszédes, 2024). In the field of IoT, where interconnected devices are used extensively in critical areas, such as healthcare, smart cities, and automated industry, the risk is higher than ever. With the diverse resource and connection constraints, static code analysis ensures that IoT applications meet high performance and security needs. Static code analysis helps to a system's robustness and effectiveness, thereby enhancing the overall safety and efficiency of contemporary digital ecosystems (Kotenko, Izrailov and Buinevich, 2022).

Large language models (LLMs, henceforth), such as ChatGPT and many others have triggered a new interest in their application for improving static code analysis. LLMs have a unique feature that allows them to understand source code because they are trained on extensive code and natural language datasets. In addition to that, LLMs can recognize many different types of code issues, especially those that require an in-depth comprehension of context or intricate reasoning. Furthermore, by synthesizing information from multiple code snippets, LLMs may be able to recognize several issues that offer a great deal of information on the code's readability, maintainability, and compliance with software design paradigms. On the other hand, traditional methods, such as abstract syntax tree (AST, henceforth) analysis, data flow analysis, and static symbolic execution may not be able to identify certain types of faults compared to LLMs.

By incorporating more advanced artificial intelligence (AI) models, especially LLMs, there is a remarkable transformation in the methods of software development in the fast-evolving field of software engineering. Generative AI technologies are widely adopted because they have clear advantages, such as high productivity, high accuracy, and rapid development cycles. Industry research suggests that by 2027, around 70% of professional software

ARO-The Scientific Journal of Koya University
Vol. XIII, No. 1 (2025), Article ID: ARO.12082. 15 pages
DOI: 10.14500/aro.12082
Received: 03 March 2025; Accepted: 27 May 2025
Regular review paper; Published: 22 June 2025
†Corresponding author's e-mail hekar.mohammedsalih@uod.ac

developers will use AI-assisted coding tools for standard programming activities, such as code creation, debugging, and optimization (Sikand, et al., 2024). The use of LLMs for static code analysis, or in combination with traditional methods, holds great potential for developing new tools in the software industry. Given their knowledge of natural and programming languages, LLMs can find bugs, provide specific suggestions in code, and even improve a developer's productivity.

This systematic survey represents the first comprehensive review focused exclusively on LLM-based static code analysis. Through rigorous examination of many related studies, the survey provides insights and establishes a novel taxonomy for this rapidly evolving field, while identifying critical limitations and proposing actionable research directions.

Summing up, the key contributions of this survey are listed below:

- First dedicated survey: This survey provides the first dedicated, systematic examination of LLMs for static code analysis, establishing a comprehensive foundation for understanding their applications, capabilities, and limitations in this specific domain. Unlike broader surveys of LLMs in software engineering, our work focuses exclusively on static code analysis tasks.
- Systematic methodology: This survey concentrates on the research that were recently published on the topic by using systematic techniques for inclusion and exclusion of the criteria set by the clearly articulated questions.
- Identification of critical challenges: This survey highlights the limitations of LLMs in static analysis, such as (high false positive rates, context window constraints, and computational costs as key adoption barriers).
- Statistical synthesis of LLMs applications in static code analysis: The survey provides a comprehensive statistical synthesis of prior research, including the prevalence of LLMs, such as ChatGPT-4 and its alternatives, as well as traditional static code analysis tools that have been widely used. It also covers programming language distribution, software engineering tasks coverage, evaluation metrics, and prompting strategies in significant detail.
- Future research directions: This survey identifies many possible research directions that need to be studied and addressed by researchers. Thus, this work serves as a quality reference for researchers and developers, bridging the gap between LLMs and static code analysis while setting a foundation for future advancements.

This systematic survey was prompted by the reasons listed below:

- Traditional static code analysis constraints: Traditional approaches, such as AST analysis, data flow analysis, and symbolic execution often fail to detect coding errors and coding quality issues as they are not designed to understand codes. Such techniques may be enhanced by LLMs which offer improved comprehension and reasoning capabilities over the code.
- LLMs for static code analysis: The use of LLMs in coding activities is rapidly increasing, making it necessary to evaluate their effectiveness in performing static code analysis in terms of functional correctness, security, and maintainability. A survey would facilitate the evaluation of their strengths, limitations, and potential areas for development in this area.
- Gaps in existing research: A significant deficiency exists in survey research that particularly examines the application of LLMs in static code analysis. A systematic survey would address this gap by aggregating insights on how LLMs might improve static code analysis and pinpointing areas for further advancement.

The remainder of this survey paper is structured as follows: Section II presents the related works for this survey. Section III describes the details of the research methodology that has been used to conduct this survey systematically. Section IV presents the results and outcomes of this survey. Section V presents the future directions of research in the selected topic. Section VI outlines threats to validity and the measures taken to address them. Finally, the conclusions of the survey are provided in Section VII.

II. Related Works

The use of LLMs in software engineering has transformed several research areas, including static code analysis, code creation, optimization, testing, maintenance, and security. This section summarizes the progress and related works that have been performed in the literature divided in several categories, as follows:

A. Code Generation and Optimization

The application of LLMs to the generation and completion of code has gained significant attention in recent years. The authors (Zheng, et al., 2023a) offer a comprehensive review of the development of LLMs for code generation and their astounding success in this task. In addition, they discussed the impact of model size and the quality of data on code generation, and called for more comprehensive ways to enhance these models. In (Zheng, et al., 2023b), the incorporation of LLMs into software engineering and their efficiency in code summarization or repair tasks were discussed significantly. The study highlights the innovative changing possibilities of LLMs in boosting developer efficiency and automating mundane coding tasks, while calling for further research to solve issues, such as model explainability and optimization for specific tasks. LLMs have proven to be useful in the area of code optimization as well as by increasing the overall efficiency of the code, such as in the cases of execution duration and memory use. The authors (Gong, et al., 2025), provide a systematic literature review pinpointing different trends and obstacles in LLMs-based code optimization. The paper emphasizes the superiority of general-purpose LLMs, such as GPT-4 for general optimization tasks, although other models excel in specific optimization tasks. The primary issues include reconciling model complexity with practical applicability and attaining cross-linguistic generalizability. The authors suggest further research avenues, including model reduction and multilingual optimization, to improve the efficiency and resilience of LLM-based code optimization methods for more dependable and scalable solutions.

B. Software Testing and Debugging

The application of LLMs in automating software testing activities, including test case development, program repair, and bug detection, has been thoroughly investigated in many studies. The authors (Wang, et al., 2024), conducted a comprehensive review on the applications of LLMs in activities, such as unit test case development and test oracle creation. While noting the effective capabilities of LLMs for the generation of diverse test inputs and their use in testing, they identified the obstacles that hinder achieving appropriate coverage and the test oracle challenge. They also discussed automating and optimizing software testing and debugging processes with the novel capabilities of LLMs while noting the imperfection of existing studies, such as the lack of integration into actual developer's work and lack of tools for deeper evaluation.

C. Software Maintenance and Management

LLMs have been put to use for the automation of several tasks within software maintenance and management, including rewriting code, creating documentation, and managing the software's lifecycle. The authors (Zhang, et al., 2023a), discussed the impact of LLMs on coding tasks, such as code creation, summarization, and program repair. The author vividly describes the increasing LLM-based software engineering research, driven by deep learning and the availability of open-source code on repositories. The authors (Hou, et al., 2023), conducted a systematic review and identified 85 distinct software engineering jobs in which LLMs have demonstrated efficacy, notably in software development and maintenance. These studies provide evidence that different tasks can be automated using LLMs, which drives software quality improvement, but at the same time creates new challenges, such as data dependence, model size, and generalization. The authors suggest the focus should move toward developing domain-oriented LLMs, along with more detailed evaluation processes.

D. Security and Vulnerability Detection

The utilization of LLMs for detecting and addressing security issues in software systems is a rapidly growing area of study. The authors (Chen, et al., 2024) provided a systematic analysis that classifies the types of attacks and how LLMs can be used to detect them. Furthermore, they provided several defense strategies that can be used to prevent such attacks. The authors (Zhou, et al., 2024) conducted a literature review on the employment of LLMs for discovering and fixing software vulnerabilities. This study shows that encoder-only models, such as CodeBERT, performed best for detection tasks, while decoder-only models, such as GPT-4 excelled at repair tasks. These papers jointly highlight the ability of LLMs for enhancing software security.

E. Natural Language Processing in Software Engineering

Like many domains, the intersection of natural language processing (NLP, henceforth) and software engineering has been very promising with respect to research for different activities, such as code summarization, code translation, and even code repair. The authors in (Zhang, et al., 2023b), provided a detailed understanding of processes related to coding by reviewing work on language model-based processing of code, including modern changes that improve performance, such as moving from statistical models to pre-trained transformers and LLMs. The research focuses on the efficiency of LLMs, such as Codex and GitHub Copilot, within the scope of code generation and comprehension. Moreover, it tackles issues, such as the need for thorough evaluation strategies, benchmarks, and the need for better practical features for codes. The authors in (Salem, et al., 2024), investigated the role of language models in the intersection of spoken languages and computer languages, particularly in automated processes that include writing of code, code refactoring, and debugging. Such studies together demonstrate the promise the application of LLMs has for changing the workflow of software engineering to foster higher productivity of software developers and enable the automation of complex processes.

F. Sustainability and Reusability in Software Engineering

The environmental impact and the sustainability of using LLMs in software engineering have also been addressed in many studies. The authors (Hort, Grishina and Moonen, 2023), examined the diffusion of software source codes and other studies artifacts and found out that only 27% of relevant studies provide sufficient artifacts for reuse. The paper highlights the significant energy consumption associated with training such large models and argues for more transparency regarding hardware specifications and training durations. The authors advocate for the dissemination of pre-trained models to mitigate unnecessary training and foster sustainable methods in software engineering. This research highlights the necessity of mitigating the environmental effect of LLMs and enhancing model reusability to promote sustainability in the domain.

In recent years, the incorporation of LLMs into software engineering has attracted considerable research interest, with various surveys investigating their applications in tasks, such as code generation, optimization, and testing. However, this systematic survey differentiates itself by providing a concentrated, thorough, and rigorous analysis of LLMs specifically for static code analysis, highlighting their applications, challenges, and other aspects often overlooked in general surveys. Thus, this survey represents a significant contribution to the field of software engineering, particularly in the niche area of static code analysis. To achieve this, the survey follows a systematic approach making the analysis of the related works to be relevant and accurate, which in return increases the trustworthiness of the obtained results. Thus, this study can be considered as the most comprehensive and up-to-date study on the use of LLMs in static code analysis, which is beneficial for developers and researchers alike.

III. Research Methodology

This survey follows established guidelines for conducting systematic literature surveys which were presented in (Petersen, Vakkalanka and Kuzniarz, 2015). Fig. 1 illustrates the five steps in the study's process. The first step defines the purpose and scope of the study and defines its objectives along with the research questions (RQs) to be answered. The second step termed the search strategy, focuses on devising a method for searching relevant articles related to the topic under investigation. In the third step, the identified papers are screened and filtered. The fourth step involves data extraction, whereby the selected papers are reviewed and relevant data that meets the objectives of the research is collected. The results are documented during the fifth step of the process. Subsequently, the following sections explain these steps in detail.

A. Identification of Research Objectives and Questions

1. Research objectives: This systematic survey seeks to examine the published works on involving static code analysis with LLMs by searching, assessing, and categorizing state-of-the-art contributions. This is done so that developers and researchers can understand the answers to particular questions and subsequently enhance their efforts in development and research.
2. RQs: Several primary RQs have been defined and answered in this survey. Each RQ deals with a different dimension of the topic of the study, as follows:
   - RQ1: What are the most frequently used LLMs for static code analysis tasks?
   - RQ2: What are the traditional static code analysis tools that are used to assess the quality of LLMs for static code analysis?
   - RQ3: What are the major programming languages used in research for specific static code analysis tasks that involve the use of LLMs?
   - RQ4: What is the range of software engineering activities that have been targeted by static code analysis with LLMs?
   - RQ5: Which evaluation metrics are most used in quantifying the accuracy and utility of LLMs in static code analysis?
   - RQ6: What prompt design strategies are most effective for optimizing LLMs performance in static code analysis?
   - RQ7: What are the common challenges and limitations in leveraging LLMs for static code analysis?

B. Search Strategy

1. Literature sources: Well-known standard online databases, such as IEEE Xplore, Elsevier Science Direct, and ACM Digital Library. that index most of the papers relevant to the scope of this survey were selected as literature sources.

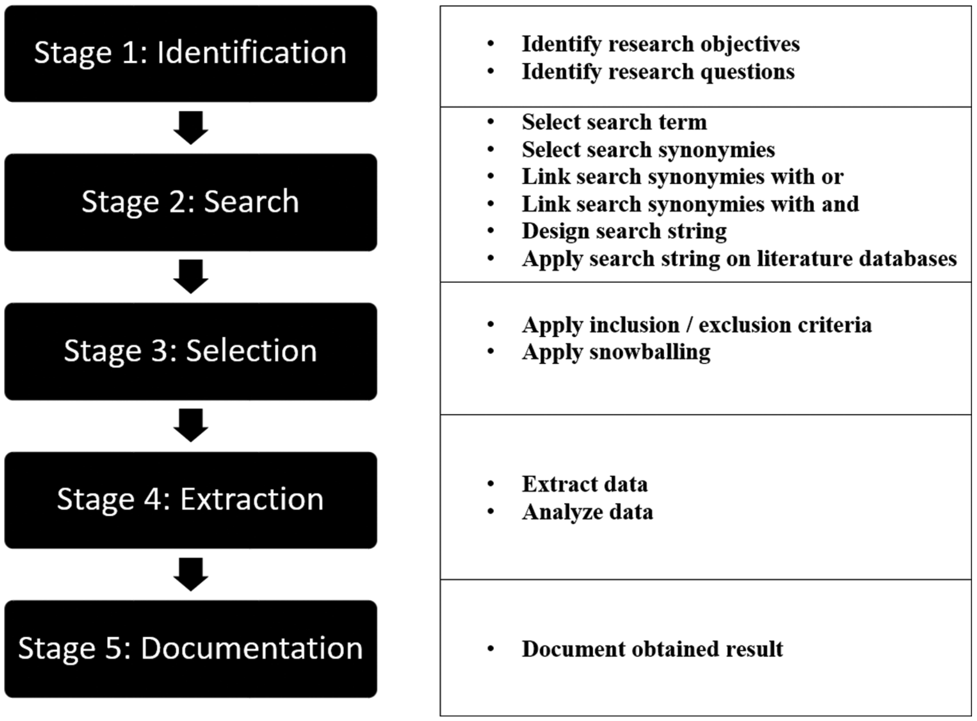

Stage 1: Identification
• Identify research objectives
• Identify research questions
Stage 2: Search
• Select search term
• Select search synonymies
• Link search synonymies with or
• Link search synonymies with and
• Design search string
• Apply search string on literature databases
Stage 3: Selection
• Apply inclusion / exclusion criteria
• Apply snowballing
Stage 4: Extraction
• Extract data
• Analyze data
Stage 5: Documentation
• Document obtained result

Fig. 1. The employed survey process.

2. Search string: Using the database literature sources, the following search string was used to locate the papers relevant to this survey:

   "(*Generative AI OR LLM OR Large Language Model) AND (static code analysis*)"

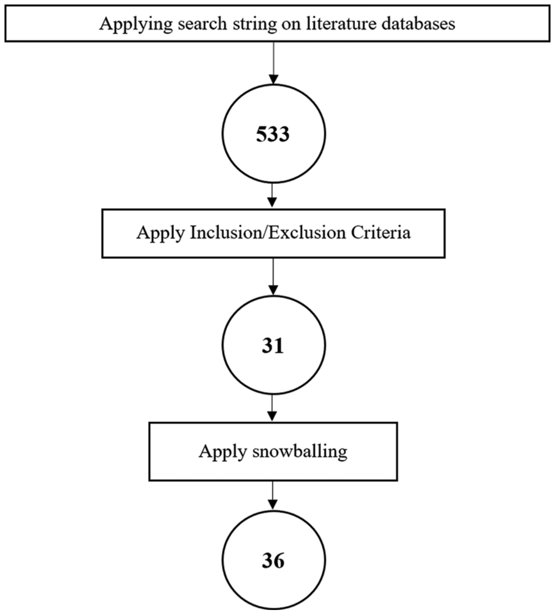
All terms of the search string were linked with each other using Boolean operators (Brereton, et al., 2007). The Boolean "OR" was employed to link synonyms or related terms that refer precisely or broadly to different aspects of the study topic and the Boolean "AND" was used to link the major terms.

C. Paper Selection

1. Paper inclusion/exclusion criteria: A set of inclusion and exclusion criteria were established and employed to decide whether a paper is relevant to this study or not. These criteria, which are listed below, have been applied based on the titles, abstracts, and full-text reading of the collected papers.
   a. Inclusion criteria:
   - Papers related directly to static code analysis using LLMs.
   - Papers published over the past 2 years (2023–2024). According to our search and exploration of the literature, papers on static code analysis using LLM started in 2023.

   b. Exclusion criteria:
   - Papers not published in English
   - Papers not peer reviewed (e.g., grey literature)
   - Papers not published electronically
   - Papers that are duplicates of other papers
   - Papers without clear results and evidence.
2. Snowballing: To reduce the risk of missing some relevant papers, the snowballing search technique (Wohlin, 2014) was applied to the remaining papers. In snowballing, the reference list of each paper is checked with the inclusion/exclusion criteria. Then, the paper selection process is applied recursively to the papers that have been found. Fig. 2 shows the number of included and excluded papers at each stage of the paper selection process.

All the papers used in this study are listed below: (Sikand, et al., 2024), (Amburle, et al., 2024), (Li, et al., 2023), (Rahmaniar, 2024), (Hajipour, et al., 2024), (Wadhwa, et al., 2024), (AlOmar and Mkaouer, 2024), (Yuan, et al., 2024), (Mahyari, 2024), (Fang, et al., 2023), (Mathews, et al., 2024), (Li and Shan, 2023), (Hossain, et al., 2024), (Mohajer, et al., 2023), (Gonzalez-Barahona, 2024), (Omar and Shiaeles, 2023), (Guo, et al., 2023a), (Ságodi, Siket and Ferenc, 2024), (Venkatesh, et al., 2024), (Moratis, et al., 2024), (Souma, et al., 2023), (Pearce, et al., 2023), (Bajpai, et al., 2024), (Yin, Ni and Wang, 2024), (Bairi, et al., 2024), (Akuthota, et al., 2023), (Ignatyev, et al., 2024), (Gupta, et al., 2023), (Liu, Yang and Liao, 2024), (Villmow, et al., 2023), (Jesse, et al., 2023), (Guo, et al., 2023b), (Di, et al., 2023), (Haindl and Weinberger, 2024), (Ardito, Ballario and Valsesia, 2023), (Purba, et al., 2023).


Applying search string on literature databases
533
Apply Inclusion/Exclusion Criteria
31
Apply snowballing
36
Fig. 2. Results of the paper selection process.

D. Data Extraction and Analysis

To address the RQs, data were taken from the chosen papers and thoroughly examined. An Excel document with several fields was made, especially for this study to include the extracted data. As presented in Table I, each field contains a data item and a value. It is worth mentioning that the Excel document can be accessed by clicking on the link given here: Extraction form (Google Drive).

IV. Results

All of the papers selected were thoroughly examined to address the RQs that were identified for this survey. Based on the results, each RQ is represented by a brief title and is covered in the next subsections.

A. Popular LLMs for Static Code Analysis (RQ1)

Table II presents the most used LLMs in research. It is clear that OpenAI's models, such as ChatGPT-4 and ChatGPT-3.5-turbo, lead the academic landscape, with ChatGPT-4 being referenced in 16 studies, underscoring its versatility and widespread adoption. ChatGPT models, particularly ChatGPT-4 and ChatGPT-3.5-turbo, are popular LLMs because they are the best performers at tasks, are easy to reach through OpenAI APIs, and have better contextual understanding and reasoning. With researchers, working on AI-powered static analysis and software engineering, they remain a top choice especially when economical options, such as ChatGPT-3.5-turbo are available (Gupta, et al., 2023; Acl, 2024). This is because the researchers can find fairly priced options for a variety of tasks including bug detection, code generation, and even vulnerability scans. LLMs are known to have a wide range of applications in static code analysis. They aid in enhancing error detection, warning verification, as well as static analysis test translations across programming languages. LLMs are also known to enhance precision and efficacy by reducing false positives and false negatives in bug detection, malicious code detection, and even vulnerability detection. Their scope of usage expands within programming education for real-time error checking and explanation purposes and even cybersecurity for aiding in the detection and remediation of vulnerabilities.

TABLE I
Data Extraction Form

Data item	Value	RQs
Paper number	Paper ID.	None
Paper title	Title of the study.	None
Used LLMs	LLMs used in the studies.	RQ1
Static code analysis tools	Static code analysis tools used in the studies with LLMs for code evaluation.	RQ2
Programming languages	Programming languages involved in the studies utilizing LLMs for static code analysis.	RQ3
Type of tasks	Tasks of static code analysis performed in the studies using LLMs.	RQ4
Evaluation metrics	Metrics used to assess the effectiveness of LLMs for static code analysis.	RQ5
Prompt designs	Strategies of effective prompt designs for LLMs used in static code analysis tasks.	RQ6
Limitations and challenges	Limitations and Challenges of performing static code analysis using LLMs.	RQ7

LLMs: Large language models, RQs: Research questions

Fig. 3 displays the companies that contributed toward LLMs development which suggests the scope of activity and engagement. OpenAI is leading the list by boosting 8 LLMs, much higher than the other prominent companies including Google, Hugging Face, Meta Microsoft, and Salesforce who have only made 2 LLMs each. This highlight gap clearly indicates openness toward capital and optimism toward supporting the research and development of LLMs. Hence, further solidifying the statement of America being proactive on AI research. The data reveals the rapid changing competitive landscape of LLMs development where OpenAI holds the front line in production, and the other groups have a significantly lesser but steady presence. This gap could mean a change in primary research objectives, budgetary spending, or competitive practices for progressing AI technology.

B. Baseline Static Code Analysis Tools (RQ2)

Table III presents the use of different static code analysis tools outlined in the academic articles, with a particular focus on their application and frequency. Programming mistake

Table II
Popular LLMs for Static Code Analysis

S. No.	LLM	Corresponding papers	Total papers
1.	ChatGPT-4	(Amburle, et al., 2024), (Li, et al., 2023), (Rahmaniar, 2024), (Hajipour, et al., 2024), (Wadhwa, et al., 2024), (AlOmar and Mkaouer, 2024), (Sikand, et al., 2024), (Yuan, et al., 2024), (Mahyari, 2024), (Fang, et al., 2023), (Mathews, et al., 2024), (Li and Shan, 2023), (Hossain, et al., 2024), (Mohajer, et al., 2023), (Gonzalez-Barahona, 2024), (Omar and Shiaeles, 2023)	16
2.	ChatGPT-3.5-turbo	(Guo, et al., 2023a), (Mahyari, 2024), (Fang, et al., 2023), (Ságodi, Siket and Ferenc, 2024), (Mohajer, et al., 2023), (Gonzalez-Barahona, 2024), (Venkatesh, et al., 2024), (Omar and Shiaeles, 2023), (Moratis, et al., 2024)	9
3.	ChatGPT-3.5	(Amburle, et al., 2024), (Souma, et al., 2023), (Hajipour, et al., 2024), (Yuan, et al., 2024), (Pearce, et al., 2023), (Li and Shan, 2023), (Hossain, et al., 2024)	7
4.	CodeLlama	(Amburle, et al., 2024), (Fang, et al., 2023), (Bajpai, et al., 2024), (Li and Shan, 2023), (Yin, Ni and Wang, 2024), (Omar and Shiaeles, 2023)	6
5.	ChatGPT-2	(Bairi, et al., 2024), (Pearce, et al., 2023), (Akuthota, et al., 2023)	3
6.	GitHub Copilot	(Rahmaniar, 2024), (AlOmar and Mkaouer, 2024), (Ságodi, Siket and Ferenc, 2024)	3
7.	CodeGen	(AlOmar and Mkaouer, 2024), (Ignatyev, et al., 2024), (Venkatesh, et al., 2024)	3
8.	ChatGPT-3	(Bairi, et al., 2024), (Wadhwa, et al., 2024)	2
9.	Mixtral	(Li, et al., 2023), (Bajpai, et al., 2024)	2
10.	Google Bard	(Rahmaniar, 2024), (Gupta, et al., 2023)	2
11.	Codex	(Liu, Yang and Liao, 2024), (Ignatyev, et al., 2024)	2
12.	Polycoder	(Liu, Yang and Liao, 2024), (Ignatyev, et al., 2024)	2
13.	DeepSeek-Coder	(Bajpai, et al., 2024), (Yin, Ni and Wang, 2024)	2
14.	WizardCoder	(Yin, Ni and Wang, 2024), (Omar and Shiaeles, 2023)	2
15.	Mistral	(Yin, Ni and Wang, 2024), (Omar and Shiaeles, 2023)	2
16.	ChatGPT-1	(Bairi, et al., 2024)	1
17.	StarCoder	(Amburle, et al., 2024)	1
18.	CodeQL	(AlOmar and Mkaouer, 2024)	1
19.	CodeFuse	(Pearce, et al., 2023)	1
20.	AI21 Jurassic-1	(Liu, Yang and Liao, 2024)	1
21.	BERT	(Villmow, et al., 2023)	1
22.	CODEDOCTOR	(Jesse, et al., 2023)	1
23.	INCODER	(Jesse, et al., 2023)	1
24.	GRAPHCODEBERT	(Jesse, et al., 2023)	1
25.	StarChat-Beta	(Fang, et al., 2023)	1
26.	Phi-2	(Yin, Ni and Wang, 2024)	1
27.	OpenAI Davinci	(Venkatesh, et al., 2024)	1
28.	Vicuna	(Omar and Shiaeles, 2023)	1

LLMs: Large language models

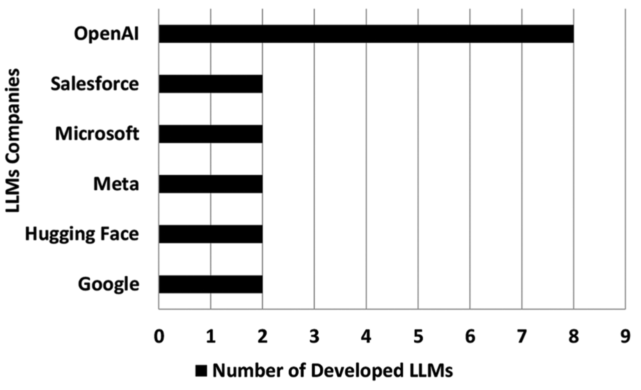


Fig. 3. Company contributing to large language models development.

Table III
Static Code Analysis Tools

S. No.	Tool	Corresponding papers	Total papers
1.	PMD	(Souma, et al., 2023), (Sikand, et al., 2024), (Guo, et al., 2023b)	3
2.	SonarQube	(Guo, et al., 2023b), (Mohajer, et al., 2023)	2
3.	Simian	(Souma, et al., 2023)	1
4.	Custom static analysis tool	(Di, et al., 2023)	1
5.	Static analyzer	(Haindl and Weinberger, 2024)	1
6.	PyTorch	(Haindl and Weinberger, 2024)	1
7.	LLB	(Li and Shan, 2023)	1
8.	FindBugs	(Guo, et al., 2023b)	1
9.	Coverity	(Guo, et al., 2023b)	1
10.	TECA	(Guo, et al., 2023b)	1
11.	Rust-code-analysis	(Ardito, Ballario and Valsesia, 2023)	1
12.	CodeQL	(Mohajer, et al., 2023)	1
13.	Infer	(Gonzalez-Barahona, 2024)	1
14.	PyCG	(Omar and Shiaeles, 2023)	1
15.	HeaderGen	(Omar and Shiaeles, 2023)	1
16.	TypeEvalPy	(Omar and Shiaeles, 2023)	1

detector (PMD, henceforth) emerges as the most widely cited tool, appearing in three papers, suggesting its effectiveness in code analysis and bug detection. SonarQube, cited in two papers, illustrates its capability in continuously monitoring of code quality. Simian, Static Analyzer, and Pytorch receive single mentions, reflecting their specialized use cases. More advanced tools, such as PyCG and HeaderGen indicate growing attention toward domain and language-specific static analysis, especially in Python and C++.

PMD is considered one of the most widely used static code analysis tools due to several critical factors. It covers a number of programming languages, such as Java, JavaScript, Apex, and PLSQL, which makes it very flexible and appropriate for a wide range of tasks. Another factor is that PMD is highly configurable which allows a great deal of customization enabling developers to create new rule sets or change existing ones to particular coding standards and best practices (AlOmar and Mkaouer, 2024). This flexibility ensures that each development team can adjust the processes according to their specific needs. In addition, PMD is available as an open-source tool which enables people to improve it and make it better and more sophisticated than ever. The ability to detect code duplication, dead variables, and potential bugs assists to improve the quality and longevity of the code. PMD also works with other popular build tools and IDEs, which makes it easy to use in various development environments. Together, all these factors combined with an active community and frequent releases increase PMD's recognition as one of the best static code analysis tools.

C. Target Programming Languages (RQ3)

Table IV presents the focus of research regarding the target programming languages for static code analysis using LLMs. The Java language was noted more often than other languages, as it appeared in 22 of the examined papers, while Python appeared in 17 and the C language in 14. Other languages, such as C++ had a moderate representation, 10 papers.

In contrast, languages, such as Swift, Kotlin, and Go appear in only 2 papers each, while niche or specialized languages, including Solidity, Ruby, Rust, Verilog, PHP, Objective-C, SQL, Perl, Scala, and R, are mentioned in just 1 paper each. These findings highlight the prominence of widely adopted languages, such as Java and Python in research contexts, particularly in leveraging LLMs for tasks, such as bug detection, vulnerability identification, and code comprehension through static code analysis. Java and Python are the most studied programming languages in static code analysis due to their widespread use, mature ecosystems, and suitability for analysis. Java's prevalence in enterprise systems and Android development, coupled with Python's dominance in data science, Machine Learning, and web development, ensures their relevance in improving code quality and security (Rahmaniar, 2024). Comprehensive static code analysis tools, including PMD and SonarQube for Java, as well as Pylint and Bandit for Python, bolstered by engaged communities and enormous resources, enhance both languages. The object-oriented structure and strong typing of Java enable correct interpretation and analysis of the program under consideration. In comparison, Python's dynamic features create unique challenges that nurture the ingenuity of researchers. Furthermore, the active interest from industry and academia, underlines the importance of these programming paradigms for static code analysis research. The limited scope of representation of the other general languages or domain-specific languages suggests a reasonable avenue for subsequent research regarding the potential of LLMs in using them.

D. Common Static Code Analysis Tasks (RQ4)

Numerous tasks in the collected papers have been identified regarding the static code analysis and applied LLMs are presented in Table V. The greatest number of papers that is 12, has been published in the area of security weaknesses and the attempts that are done to discover and resolve these issues in the code. Static behavior analysis and code quality estimation and control are fundamental parts of 10 papers that concern the understanding of program behavior and

TABLE IV
Programming Language used in Studies

S. No.	Programming language	Corresponding papers	Total Papers
1.	Java	(Amburle, et al., 2024), (Li, et al., 2023), (Souma, et al., 2023), (Rahmaniar, 2024), (Gupta, et al., 2023), (Di, et al., 2023), (Wadhwa, et al., 2024), (Sikand, et al., 2024), (Yuan, et al., 2024), (Pearce, et al., 2023), (Mahyari, 2024), (Jesse, et al., 2023), (Ignatyev, et al., 2024), (Bajpai, et al., 2024), (Li and Shan, 2023), (Ságodi, Siket and Ferenc, 2024), (Guo, et al., 2023b), (Ardito, Ballario and Valsesia, 2023), (Mohajer, et al., 2023), (Gonzalez-Barahona, 2024), (Akuthota, et al., 2023), (Moratis, et al., 2024)	22
2.	Python	(Amburle, et al., 2024), (Souma, et al., 2023), (Rahmaniar, 2024), (Gupta, et al., 2023), (Di, et al., 2023), (Hajipour, et al., 2024), (Wadhwa, et al., 2024), (AlOmar and Mkaouer, 2024), (Pearce, et al., 2023), (Liu, Yang and Liao, 2024), (Guo, et al., 2023a), (Jesse, et al., 2023), (Fang, et al., 2023), (Hossain, et al., 2024), (Mohajer, et al., 2023), (Venkatesh, et al., 2024), (Omar and Shiaeles, 2023)	17
3.	C	(Amburle, et al., 2024), (Gupta, et al., 2023), (Wadhwa, et al., 2024), (AlOmar and Mkaouer, 2024), (Liu, Yang and Liao, 2024), (Guo, et al., 2023a), (Mahyari, 2024), (Villmow, et al., 2023), (Fang, et al., 2023), (Hossain, et al., 2024), (Guo, et al., 2023b), (Yin, Ni and Wang, 2024), (Venkatesh, et al., 2024), (Akuthota, et al., 2023)	14
4.	C++	(Wadhwa, et al., 2024), (Guo, et al., 2023a), (Mahyari, 2024), (Jesse, et al., 2023), (Bajpai, et al., 2024), (Ságodi, Siket and Ferenc, 2024), (Guo, et al., 2023b), (Yin, Ni and Wang, 2024), (Venkatesh, et al., 2024), (Akuthota, et al., 2023)	10
5.	JavaScript	(Souma, et al., 2023), (Rahmaniar, 2024), (Wadhwa, et al., 2024), (Mahyari, 2024), (Fang, et al., 2023), (Hossain, et al., 2024)	6
6.	C#	(Haindl and Weinberger, 2024), (Mahyari, 2024), (Bajpai, et al., 2024), (Mathews, et al., 2024)	4
7.	Swift	(Wadhwa, et al., 2024), (Mahyari, 2024)	2
8.	Kotlin	(Wadhwa, et al., 2024), (Mahyari, 2024)	2
9.	Go	(Mahyari, 2024), (Venkatesh, et al., 2024)	2
10.	Solidity	(Amburle, et al., 2024)	1
11.	Ruby	(Wadhwa, et al., 2024)	1
12.	Rust	(Wadhwa, et al., 2024)	1
13.	Verilog	(Liu, Yang and Liao, 2024)	1
14.	PHP	(Mahyari, 2024)	1
15.	Objective-C	(Mahyari, 2024)	1
16.	SQL	(Mahyari, 2024)	1
17.	Perl	(Mahyari, 2024)	1
18.	Scala	(Mahyari, 2024)	1
19.	R	(Mahyari, 2024)	1

TABLE V
Addressed Tasks in Academic Paper on Static Code Analysis using LLMs

S. No.	Type of task	Corresponding papers	Total papers
1.	Security vulnerability detection	(Li, et al., 2023), (AlOmar and Mkaouer, 2024), (Liu, Yang and Liao, 2024), (Guo, et al., 2023a), (Villmow, et al., 2023), (Mathews, et al., 2024), (Li and Shan, 2023), (Hossain, et al., 2024), (Yin, Ni and Wang, 2024), (Venkatesh, et al., 2024), (Akuthota, et al., 2023), (Moratis, et al., 2024)	12
2.	Static behavior analysis	(Amburle, et al., 2024), (Di, et al., 2023), (Fang, et al., 2023), (Bajpai, et al., 2024), (Hossain, et al., 2024), (Guo, et al., 2023b), (Ardito, Ballario and Valsesia, 2023), (Mohajer, et al., 2023), (Gonzalez-Barahona, 2024), (Omar and Shiaeles, 2023)	10
3.	Code quality assurance	(Souma, et al., 2023), (Di, et al., 2023), (Sikand, et al., 2024), (Yuan, et al., 2024), (Pearce, et al., 2023), (Jesse, et al., 2023), (Ságodi, Siket and Ferenc, 2024), (Guo, et al., 2023b), (Ardito, Ballario and Valsesia, 2023), (Mohajer, et al., 2023)	10
4.	Bug detection	(Gupta, et al., 2023), (Sikand, et al., 2024), (Liu, Yang and Liao, 2024), (Ignatyev, et al., 2024), (Bajpai, et al., 2024), (Purba, et al., 2023)	6
5.	Syntax understanding	(Amburle, et al., 2024)	1
6.	Variable misuse detection	(Haindl and Weinberger, 2024)	1
7.	Adherence to green coding rules	(Rahmaniar, 2024)	1
8.	Logical reasoning in error detection	(Hajipour, et al., 2024)	1
9.	Identifying violations (naming conventions)	(Jesse, et al., 2023)	1

the quality control standards, respectively. Six papers are dedicated to bug detection highlighting the ongoing approach of seeking errors in different phases of software development and correcting them.

There are relatively few studies looking at other more specific issues, such as understanding syntax, detecting variable misuses, and reasoning for errors. These suggest very specific research areas. It also shows research activities, such as compliance with the green coding initiative, breaches of naming conventions, and the reasoning logic behind error detection, all of which are performed by individual researchers. These definitional boundaries mark some of the newer and broader static code analysis challenges where researchers are exploring sustainable coding practices and improving sustainability and maintainability in software development. The complex nature of static code analysis and its importance in various software engineering fields is exhibited in these different sets of tasks.

E. Evaluation Metrics (RQ5)

Table VI presents the common metrics used to evaluate the abilities of LLMs for static code analysis. Accuracy is the most often reported metric, cited in 11 papers, which illustrates its importance in measuring the effectiveness of these AI models. Several papers refer to the F1 Score, manual verification, precision, and false positive rate as terms of importance each in the balance of capturing problems and warnings. The specific metrics, such as recall, match rate, and true positive rate provide deeper insight into how well these models perform in detecting important issues (Liu, Yang and Liao, 2024). Some metrics are domain-specific metrics, such as Green Code Compliance Percentage, Rule Satisfaction Rates, and Security Correctness that focus on sustainability and security. Syntax error counts, repair success rates, and functional correctness measures highlight the significance of evaluating code quality and the effectiveness of automatic code repair. The complexity metrics, such as Cyclomatic Complexity, Cognitive Complexity, and Weighted Methods per Class assess the structural and maintainability aspect of the code. Novel approaches to code comparison and evaluation are captured by distinctive measures, such as the DiffBLEU score, Levenshtein Distance, and Jaccard Index. As for the measurement, the most frequent usage is for the F1 score and accuracy, which can provide a relatively objective and comprehensive evaluation. Accuracy is defined as the number of correctly identified issues (including both true positives and true negatives) divided by all predictions made, thus offering a complete picture of a tool's effectiveness.

The datasets used in static code analysis often show an uneven distribution between genuine issues and non-problems because they contain many more instances of the latter (negatives). This disparity renders accuracy inadequate since a tool could achieve elevated accuracy by only predicting the majority class (non-issues) and fail to detect actual problems (Ignatyev, et al., 2024). The F1 score addresses this challenge because it merges both accuracy and recall into one single value (Yin, Ni and Wang, 2024). The precision metric defines the ratio of actual problem instances correctly identified from total expected problem instances to address false positives; recall defines the ratio of correctly detected real issues to prevent false negatives. The F1 score serves as the harmonic mean of precision and recall and provides a good balance between them, which makes it particularly valuable for evaluating static code analysis tools since both types of errors (false positives and negatives) can produce major consequences. The integrated use of accuracy and the F1 score together can be used to assess both the reliability and effectiveness of static code analysis. The range of metrics applied shows both the extensive research in static code analysis and its broad application to every field of software engineering.

Fig. 4 highlights how academic papers are distributed across research categories, with performance-related studies leading the way due to 51 papers focusing on tool and model efficiency assessment. Code quality ranks as the second most investigated domain, with 18 papers, indicating a significant focus on maintainability and adherence to best practices. Categories, such as code analysis, code complexity, and code structure, each represented by 5–7 papers, highlight targeted efforts to understand software behavior and architecture. Less commonly addressed topics, including similarity, security, code size, process success, bug analysis, and test coverage, each account for fewer than five papers, indicating niche but essential areas of study. This distribution underscores the diverse priorities within the field of static code analysis and software engineering research.

F. Common Prompting Strategies (RQ6)

Table VII categorizes various prompting strategies for LLMs across the collected papers. Among the prompting techniques, Structured Prompting is the most widely used, appearing in 8 papers. This technique likely involves designing prompts with a defined structure to enhance a model's performance. Standardized or Basic Prompting follows closely, with 7 associated papers, highlighting its role as a fundamental approach in prompt design. Few-shot

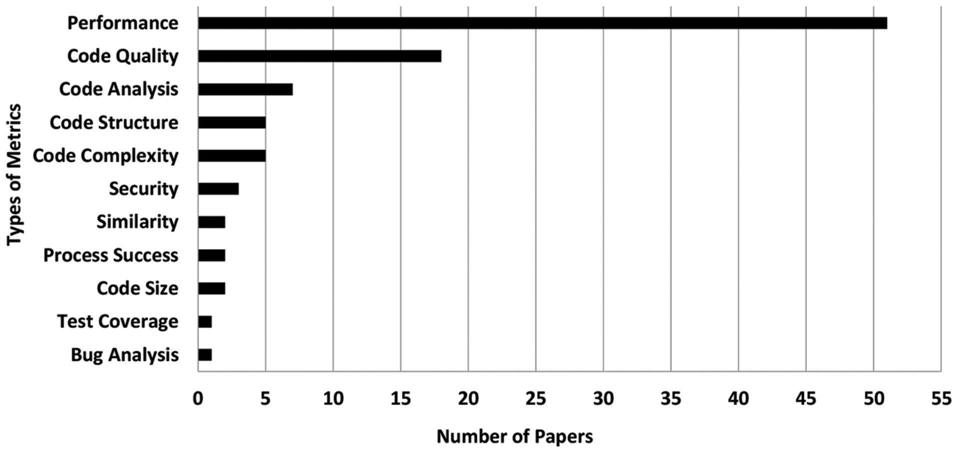


Fig. 4. Number of papers categorized by type of evaluation metrics.

TABLE VI
Commonly used evaluation metrics

S. No.	Metric	Corresponding papers	Total papers
1.	Accuracy	(Amburle, et al., 2024), (Gupta, et al., 2023), (Hajipour, et al., 2024), (Wadhwa, et al., 2024), (Haindl and Weinberger, 2024), (Guo, et al., 2023a), (Villmow, et al., 2023), (Yin, Ni and Wang, 2024), (Gonzalez-Barahona, 2024), (Akuthota, et al., 2023), (Moratis, et al., 2024)	11
2.	F1 score	(Amburle, et al., 2024), (Hajipour, et al., 2024), (Guo, et al., 2023a), (Bajpai, et al., 2024), (Yin, Ni and Wang, 2024), (Gonzalez-Barahona, 2024), (Venkatesh, et al., 2024), (Akuthota, et al., 2023)	8
3.	Manual verification	(Li, et al., 2023), (Gupta, et al., 2023), (Sikand, et al., 2024), (Pearce, et al., 2023), (Mathews, et al., 2024), (Ságodi, Siket and Ferenc, 2024), (Mohajer, et al., 2023)	7
4.	Precision	(Guo, et al., 2023a), (Bajpai, et al., 2024), (Hossain, et al., 2024), (Yin, Ni and Wang, 2024), (Gonzalez-Barahona, 2024), (Venkatesh, et al., 2024), (Akuthota, et al., 2023)	7
5.	False positive	(Haindl and Weinberger, 2024), (Li and Shan, 2023), (Guo, et al., 2023b), (Yin, Ni and Wang, 2024), (Mohajer, et al., 2023), (Venkatesh, et al., 2024)	6
6.	Recall	(Guo, et al., 2023a), (Yin, Ni and Wang, 2024), (Gonzalez-Barahona, 2024), (Venkatesh, et al., 2024), (Akuthota, et al., 2023)	5
7.	Match rate	(Mahyari, 2024), (Fang, et al., 2023), (Ignatyev, et al., 2024), (Bajpai, et al., 2024), (Omar and Shiaeles, 2023)	5
8.	Number of rule violations	(Souma, et al., 2023), (Yuan, et al., 2024), (Mohajer, et al., 2023)	3
9.	Time to fix issues	(Sikand, et al., 2024), (Mathews, et al., 2024), (Li and Shan, 2023)	3
10.	True positive	(Li and Shan, 2023), (Guo, et al., 2023b), (Venkatesh, et al., 2024)	3
11.	Syntax error counts	(Di, et al., 2023), (Gonzalez-Barahona, 2024)	2
12.	Success rate of LLM	(Sikand, et al., 2024), (Hossain, et al., 2024)	2
13.	False negative	(Haindl and Weinberger, 2024), (Venkatesh, et al., 2024)	2
14.	Security correctness	(Liu, Yang and Liao, 2024), (Li and Shan, 2023)	2
15.	Soundness	(Omar and Shiaeles, 2023)	1
16.	Completeness	(Omar and Shiaeles, 2023)	1
17.	Jaccard index	(Amburle, et al., 2024)	1
18.	Number of prompts	(Souma, et al., 2023)	1
19.	Green code compliance percentage	(Rahmaniar, 2024)	1
20.	Rule satisfaction rates	(Di, et al., 2023)	1
21.	Plagiarism detection	(Di, et al., 2023)	1
22.	Pass@k	(Hajipour, et al., 2024)	1
23.	Number of discovered vulnerabilities	(AlOmar and Mkaouer, 2024)	1
24.	Top-k accuracy	(Yin, Ni and Wang, 2024)	1
25.	Cyclomatic complexity	(Yuan, et al., 2024)	1
26.	Cognitive complexity	(Yuan, et al., 2024)	1
27.	Correctness metrics	(Pearce, et al., 2023)	1
28.	Coverage metrics	(Pearce, et al., 2023)	1
29.	Functional correctness	(Liu, Yang and Liao, 2024)	1
30.	Repair success rate	(Liu, Yang and Liao, 2024)	1
31.	BLEU	(Mahyari, 2024)	1
32.	Mean	(Jesse, et al., 2023)	1
33.	Bug/patch ratio	(Ignatyev, et al., 2024)	1
34.	Logical lines of code	(Ságodi, Siket and Ferenc, 2024)	1
35.	Number of statements	(Ságodi, Siket and Ferenc, 2024)	1
36.	McCabe cyclomatic complexity	(Ságodi, Siket and Ferenc, 2024)	1
37.	Nesting level	(Ságodi, Siket and Ferenc, 2024)	1
38.	Coding smells	(Ságodi, Siket and Ferenc, 2024)	1
39.	ABC metric	(Ardito, Ballario and Valsesia, 2023)	1
40.	Weighted methods per class	(Ardito, Ballario and Valsesia, 2023)	1
41.	Number of public methods	(Ardito, Ballario and Valsesia, 2023)	1
42.	Number of public attributes	(Ardito, Ballario and Valsesia, 2023)	1
43.	Class operation accessibility	(Ardito, Ballario and Valsesia, 2023)	1
44.	Class data accessibility	(Ardito, Ballario and Valsesia, 2023)	1
45.	Area under the curve	(Akuthota, et al., 2023)	1

TABLE VII
Distribution prompting strategies

S. No.	Prompt type	Corresponding papers	Total papers
1.	Structured prompting	(Souma, et al., 2023), (Liu, Yang and Liao, 2024), (Fang, et al., 2023), (Ignatyev, et al., 2024), (Mathews, et al., 2024), (Mohajer, et al., 2023), (Venkatesh, et al., 2024), (Moratis, et al., 2024)	8
2.	Standardized/basic prompting	(Hajipour, et al., 2024), (Guo, et al., 2023a), (Mahyari, 2024), (Fang, et al., 2023), (Bajpai, et al., 2024), (Li and Shan, 2023), (Ságodi, Siket and Ferenc, 2024)	7
3.	Few-shot prompting	(Amburle, et al., 2024), (Wadhwa, et al., 2024), (AlOmar and Mkaouer, 2024), (Sikand, et al., 2024), (Li and Shan, 2023), (Gonzalez-Barahona, 2024)	6
4.	Zero-shot prompting	(Amburle, et al., 2024), (Wadhwa, et al., 2024), (Sikand, et al., 2024), (Gonzalez-Barahona, 2024)	4
5.	Iterative prompting	(Souma, et al., 2023), (Pearce, et al., 2023), (Purba, et al., 2023), (Omar and Shiaeles, 2023)	4
6.	One-shot prompting	(Sikand, et al., 2024), (Gonzalez-Barahona, 2024), (Omar and Shiaeles, 2023)	3
7.	Task-specific prompting	(Rahmaniar, 2024), (Yin, Ni and Wang, 2024)	2
8.	CoT prompting	(Hajipour, et al., 2024), (Guo, et al., 2023a)	2
9.	Natural language descriptions	(Pearce, et al., 2023), (Ignatyev, et al., 2024)	2
10.	High-level query language	(Di, et al., 2023), (Bajpai, et al., 2024)	2
11.	Temporal and spatial context	(Bajpai, et al., 2024)	1
12.	Retrieval-augmented generation	(Li and Shan, 2023)	1
13.	Scenario-based prompting	(Mahyari, 2024)	1
14.	Correction prompt	(Hajipour, et al., 2024)	1

Prompting, where the model is given limited examples, is addressed in 6 papers, while Zero-shot Prompting, where the model is given no examples, is used in 4 papers. Iterative Prompting, which involves multi-step refinement, appears in 4 papers. One-Shot Prompting has 3 associated papers, indicating a growing interest in specific task-tailored or stepwise reasoning approaches. Less frequently studied techniques include Task-Specific Prompting, Chain-of-Thought (CoT, henceforth) Prompting, Natural Language Descriptions, and High-Level Query Language, each appeared in 2 papers. Niche methods, such as Temporal and Spatial Context, Retrieval-Augmented Generation, Scenario-Based Prompting, and Correction Prompts are mentioned in 1 paper each, reflecting emerging or specialized areas of research in LLM prompting strategies in the case of static code analysis.

Structured Prompting, Standardized/Basic Prompting, and Few-shot Prompting are the most widely used prompting strategies due to their effectiveness, simplicity, and adaptability to the static code analysis unique challenges. Structured Prompting is advantageous as it offers a coherent, systematic framework for directing models in code analysis, which corresponds effectively with the ordered characteristics of programming languages. This approach enables the breakdown of complex code analysis tasks into smaller components, which improve the model's ability to detect issues, such as syntax errors, code smells, or security vulnerabilities (Jesse, et al., 2023). Standardized/Basic Prompting is chosen because it is straightforward and can be duplicated because it involves specific directions that apply from one codebase to another and from one analytical task to another (Mathews, et al., 2024). This sets up a trustworthy standard for evaluating static code analysis techniques. The method is exceptionally effective for static code analysis because it allows models to learn from limited examples, which makes it very useful when datasets are small or when shifting between coding languages or styles. These strategies together meet the needs for static code analysis precision, scalability, and adaptability, which make them the most common approaches in research. This distribution shows the scope and distinctiveness of prompt engineering strategies that are being studied in the related works.

G. Common Limitations and Challenges (RQ7)

In this survey, the following challenges and constraints on the use of LLMs in static code analysis are identified:

1. High false positive rates: In the static code analysis, LLMs tend to produce many false positives that require a thorough human verification process to detect actual issues. Some of the related challenges include; how to detect Null Dereferences and Resource Leaks among others (AlOmar and Mkaouer, 2024), (Guo, et al., 2023b), (Mohajer, et al., 2023), (Gonzalez-Barahona, 2024), (Venkatesh, et al., 2024).
2. Token and context limitations: LLMs have limitations on the input size which makes them not very effective in handling large codebases or complex dependencies at a time. For example, dealing with imported module code or checking through large files is challenging (Amburle, et al., 2024), (Wadhwa, et al., 2024), (Sikand, et al., 2024), (Yuan, et al., 2024), (Liu, Yang and Liao, 2024), (Jesse, et al., 2023), (Fang, et al., 2023), (Mathews, et al., 2024), (Li and Shan, 2023), (Mohajer, et al., 2023), (Gonzalez-Barahona, 2024).
3. Data limitations: It is a common issue to find that the quality and diversity of training data sets directly impact the model's generality and accuracy. The reliance on databases, such as CVE, which are prone to errors, decreases the accuracy of vulnerability detection (Amburle, et al., 2024), (Rahmaniar, 2024), (Hajipour, et al., 2024), (Haindl and Weinberger, 2024), (Guo, et al., 2023a).
4. Non-deterministic outputs: Uncertainty about how many times LLMs will produce different outcomes from multiple executions for the same input creates problems when integrating them into static code analysis frameworks. For instance, making coding integration and development workflows deterministic is difficult due to the variability result (Rahmaniar, 2024), (Bajpai, et al., 2024).

5. Computational costs: The high resource requirements pose a significant problem for implementing large-scale code analysis with LLMs and the need for fine-tuning that complicates this issue further. Flow-sensitive pointer information and extensive call graphs demand substantial processing resources to address them (Bajpai, et al., 2024), (Venkatesh, et al., 2024), (Omar and Shiaeles, 2023).
6. Model hallucinations and assumptions: Sometimes LLMs produce wrong or overconfident outcomes because of built-in biases and limited understanding of context. For instance, when doing bug detection tasks, they frequently hallucinate or miss dependencies (Amburle, et al., 2024), (Li and Shan, 2023), (Gonzalez-Barahona, 2024).
7. Scalability and adaptability issues: LLMs struggle to generalize across different languages, ecosystems, or complex design patterns. For instance, they have a limited capacity to preserve language-specific semantics while doing cross-language analysis (Gupta, et al., 2023), (Villmow, et al., 2023).

V. Future Directions of Research

Examining the collected papers reveals that there are many research gaps to be addressed in using LLMs for static code analysis, which can be summarized as follows:

1. Improving prompt engineering
   - Researchers can develop automated prompt generating strategies to tailor prompts for certain activities (such as vulnerability detection and bug fixing) (Li, et al., 2023).
   - To increase accuracy and reasoning in challenging tasks, researchers can investigate iterative prompting and CoT prompting (Li, et al., 2023).
   - Researchers can examine task-specific prompt templates for various analytical tasks and programming languages (Liu, Yang and Liao, 2024).
2. Fine-tuning LLMs for specific tasks
   - To enhance the performance, researchers can fine-tune LLMs using domain-specific datasets (such as code vulnerabilities and static analysis warnings) (Hossain, et al., 2024).
   - To improve accuracy and efficiency, researchers can develop hybrid models that incorporate LLMs in traditional static analysis techniques (Omar and Shiaeles, 2023).
   - To allow LLMs to manage several static code analysis tasks at once, researchers can investigate multi-task learning (Yin, Ni and Wang, 2024).
3. Handling complex and obfuscated code
   - Researchers can develop techniques to de-obfuscate code before analysis using LLMs (Fang, et al., 2023).
   - Researchers can integrate graph-based representations (e.g., control flow graphs and data flow graphs) to help LLMs understand complex code structures (Hossain, et al., 2024).
   - Researchers can explore multi-agent systems where LLMs collaborate with other tools to analyze interconnected code components (Bajpai, et al., 2024).
4. Reducing false positives and improving precision
   - Researchers can combine LLMs with rule-based systems or ML classifiers to filter out false positives (Guo, et al., 2023b).
   - Researchers can use ranking mechanisms to prioritize the most likely true positives for developer review (Mohajer, et al., 2023).
   - Researchers can develop explainable AI techniques to help developers understand why a particular issue was flagged (Wadhwa, et al., 2024).
5. Expanding language and framework support
   - Researchers can extend LLM-based static analysis to less common programming languages (e.g., Rust, Kotlin, Swift, Arduino, etc.) (Guo, et al., 2023a).
   - Researchers can develop cross-language LLMs-based analysis tools that can handle multi-language projects (Guo, et al., 2023a).
   - Researchers can explore support for domain-specific languages (e.g., solidity for smart contracts) (Ardito, Ballario and Valsesia, 2023).
6. Integration with development workflows
   - Researchers can develop IDE plugins that leverage LLMs for real-time code analysis and feedback (Bajpai, et al., 2024).
   - Researchers can create automated tools for continuous monitoring of code quality and vulnerabilities using LLMs (Akuthota, et al., 2023).
   - Researchers can explore collaborative workflows where LLMs assist developers in debugging, refactoring, and code review (Bajpai, et al., 2024).
7. Addressing ethical and security concerns
   - Researchers can develop secure coding guidelines for LLM-generated code to prevent vulnerabilities (Purba, et al., 2023).
   - Researchers can investigate adversarial training to make LLMs more robust against malicious inputs (Li and Shan, 2023).
   - Researchers can ensure data privacy by avoiding the use of proprietary or sensitive code in LLM training datasets (Akuthota, et al., 2023).
8. Scaling to large codebases
   - Researchers can develop chunking strategies to break down large codebases into manageable segments for analysis (Bairi, et al., 2024).
   - Researchers can use hierarchical models that analyze code at multiple levels of granularity (e.g., file-level, function-level, etc.) (Ignatyev, et al., 2024).
   - Researchers can explore distributed computing techniques to scale LLM-based analysis to enterprise-level projects (Venkatesh, et al., 2024).
9. Benchmarking and standardization
   - Researchers can produce uniform datasets and metrics that enable effective assessment of LLM performance in static code analysis tasks (Yin, Ni and Wang, 2024).
   - Researchers can establish assessment tools that report on both primary product behavior and secondary qualities including code readability and maintainability (Ságodi, Siket and Ferenc, 2024).

- Researchers can foster open-source contributions to aggregate research data and materials (Omar and Shiaeles, 2023).

10. Enhancing explainability and developer trust
    - Researchers can design tools that give thorough explanations for the LLM-produced analysis findings (Wadhwa, et al., 2024).
    - Researchers can use interactive interfaces where developers can request further context or clarification from LLMs (Bajpai, et al., 2024).
    - Researchers can study how human-loop systems function to cooperate between developers and LLMs to improve analysis product accuracy (Mohajer, et al., 2023).
11. Exploring hybrid approaches
    - Researchers can improve precision by integrating LLMs with symbolic execution, abstract interpretation, or formal methods (Omar and Shiaeles, 2023).
    - Researchers can integrate LLMs with graph-based models (e.g., GraphCodeBERT) for better representation of code dependencies (Hossain, et al., 2024).

The future of static code analysis using LLMs lies in improving prompt engineering, fine-tuning models for specific tasks, reducing false positives, and integrating LLMs into development workflows. By combining LLMs with traditional static analysis tools and exploring hybrid approaches, researchers can unlock the full potential of LLMs for different static code analysis tasks.

VI. Threat to Validity

The results of surveys may be impacted by many factors. Thus, to prevent validity risks, the following steps were taken into consideration for this paper:

- Finding related papers: The availability of all relevant papers cannot be guaranteed. To find the related papers, a search string containing several word synonyms was employed, and many well-known literature databases were employed. There could yet be some missing papers, though. The snowballing search strategy was used to mitigate this issue by lowering the likelihood of missing related papers.
- Accuracy of data extraction: When extracting data from the chosen papers, several errors might happen. The data extraction procedure was carried out by hand to address this threat. The spreadsheet also made advantage of Microsoft Excel's automatic mining and filtering features. After that, the outcomes of the two approaches were contrasted to identify any differences and create a final Excel document with all the correct extracted data.
- Study reproducibility: A further threat is that if this study is carried out or replicated, other researchers could get similar findings. Every stage of the research approach used and carried out in this study was thoroughly explained to address this threat (see Section III).

VII. Conclusion

This survey highlights the transformative role of LLMs in static code analysis, offering a fresh perspective on leveraging LLMs to address longstanding challenges in software quality, security, and maintainability. By synthesizing insights from a diverse range of studies, we identified how LLMs excel in tasks, such as vulnerability detection, bug fixing, and code quality assurance, often complementing or surpassing traditional static analysis tools. Key strengths include their ability to understand complex code semantics and adapt to diverse programming languages and contexts. However, integrating LLMs into static code analysis workflows is not without challenges. High false positive rates, token size constraints, and computational costs remain significant barriers to widespread adoption. Looking ahead, future research should focus on refining prompt engineering, exploring underrepresented programming languages and niche tasks, and improving scalability to support large codebases. In addition, expanding benchmarks and real-world case studies will further validate the practical utility of LLMs in static code analysis. As the field evolves, LLMs have the potential to redefine software engineering practices, making development processes more efficient, secure, and adaptive to the complexities of modern codebases and software applications. By bridging the gap between traditional static code analysis methods and AI-driven solutions, this study aims to inspire researchers and practitioners to unlock the full potential of LLMs, contributing to the ongoing advancement of secure and high-quality software development.

References

Acl, A., 2024. *An Empirical Study of LLM for Code Analysis : Understanding Syntax and Semantics*. *ACL ARR*. Available from: https://openreview.net/forum?id=yezazwj1yf> [Last assessed on 2025 Jan 04].

Akuthota, V., Kasula, R., Sumona, S.T., Mohiuddin, M., Reza, M.T., and Rahman, M.M., 2023. Vulnerability detection and monitoring using LLM. In: *Proceedings of 2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering, WIECON-ECE 2023*, IEEE, United States, pp.309-314.

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&amp;T)*. IEEE, United States, pp.1-10.

Amburle, A., Almeida, C., Lopes, N., and Lopes, O., 2024. AI based code error explainer using gemini model. In: *2024 3rd International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*. IEEE, United States: pp.274-278.

Ardito, L., Ballario, M., and Valsesia, M., 2023. Research, Implementation and Analysis of Source Code Metrics in Rust-Code-Analysis. In: *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, United States, pp.497-506.

Bairi, R., Sonwane, A., Kanade, A., Vageesh, D.C., Iyer, A., Parthasarathy, S., Rajamani, S., Ashok, B., and Shet, S., 2024. Codeplan: Repository-level coding using LLMs and planning. *Proceedings of the ACM on Software Engineering*, 1, pp.675-698.

Bajpai, Y., Chopra, B., Biyani, P., Aslan, C., Coleman, D., Gulwani, S., Parnin, C., Radhakrishna, A., and Soares, G., 2024. Let's fix this together: Conversational debugging with github copilot. In: *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, United States, pp.1-12.

Brereton, P., Kitchenham, B.A., Budgen, D., Turner, M., and Khalil, M., 2007. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4), pp.571-583.

Chen, Y., Sun, W., Fang, C., Chen, Z., Ge, Y., Han, T., Zhang, Q., Liu, Y., Chen, Z., and Xu, B., 2024. *Security of Language Models for Code: A Systematic Literature Review*. Vol. 1. Available from: https://arxiv.org/abs/2410.15631 [Last assessed on 2025 Jan 04].

Di, P., Li, J., Yu, H., Jiang, W., Cai, W., Cao, Y., Chen, C., Chen, D., Chen, H., Chen, L., Fan, G., Gong, J., Gong, Z., Hu, W.,… & Zhu, X., 2023. CodeFuse-13B: A pretrained multi-lingual code large language model. *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. ICSE, New Delhi, pp.418-429.

Fang, C., Miao, N., Srivastav, S., Liu, J., Zhang, R., Fang, R., Asmita, Tsang, R., Nazari, N., Wang, H., and Homayoun, H., 2023. *Large Language Models for Code Analysis: Do LLMs Really do Their Job*? Available from: https://arxiv.org/abs/2310.12357 [Last assessed on 2025 Jan 04].

Gong, J., Voskanyan, V., Brookes, P., Wu, F., Jie, W., Xu, J., Giavrimis, R., Basios, M., Kanthan, L., and Wang, Z., 2025. *Language Models for Code Optimization: Survey, Challenges and Future Directions*. Vol. 1. ACM Computing Surveys. [arxiv Preprint]. Available from: https://arxiv.org/abs/2501.01277 [Last assessed on 2025 Jan 04].

Gonzalez-Barahona, J.M., 2024. Software development in the age of LLMs and XR. In: *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*. ACM, New York, USA, pp.66-69.

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. In: *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.

Gupta, N.K., Chaudhary, A., Singh, R., and Singh, R., 2023. ChatGPT: Exploring the capabilities and limitations of a large language model for conversational AI. In: *2023 International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT)*. IEEE, United States, pp.139-142.

Haindl, P., and Weinberger, A.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.

Hassan, H.B., Sarhan, Q.I., and Beszédes, Á., 2024. Evaluating python static code analysis tools using FAIR principles. *IEEE Access*, 12, pp.173647-173659.

Hort, M., Grishina, A., and Moonen, L., 2023. An exploratory literature study on sharing and energy use of language models for source code. In: *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, United States, pp.1-12.

Hossain, A.A., Mithun Kumar, P.K., Zhang, J., and Amsaad, F., 2024. Malicious code detection using LLM. In: *NAECON 2024 - IEEE National Aerospace and Electronics Conference*. IEEE, United States, pp.414-416.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H., 2023. *Large Language Models for Software Engineering: A Systematic Literature Review*. pp.1-79. Available from: https://arxiv.org/abs/2308.10620 [Last assessed on 2025 Jan 04].

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.

Kotenko, I., Izrailov, K., and Buinevich, M., 2022. Static analysis of information systems for IoT cyber security: A survey of machine learning approaches. *Sensors (Basel)*, 22(4), p.1335.

Li, H., and Shan, L., 2023. LLM-based vulnerability detection. In: *2023 International Conference on Human-Centered Cognitive Systems (HCCS)*. IEEE, United States, pp.1-4.

Li, H., Hao, Y., Zhai, Y., and Qian, Z., 2023. 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, USA, pp.2107-2111.

Liu, Z., Yang, Z., and Liao, Q., 2024. Exploration on prompting LLM with code-specific information for vulnerability detection. *Proceedings - 2024 IEEE International Conference on Software Services Engineering, SSE 2024*. IEEE, United States, pp.273-281.

Louridas, P., 2006. Static code analysis. *IEEE Software*, 23(4), pp.58-61.

Mahyari, A.A., 2024. Harnessing the power of LLMs in source code vulnerability detection. In: *MILCOM 2024 - 2024 IEEE Military Communications Conference (MILCOM)*. IEEE, United States, pp.251-256.

Mathews, N.S., Brus, Y., Aafer, Y., Nagappan, M., and McIntosh, S., 2024. *LLbezpeky: Leveraging Large Language Models for Vulnerability Detection*. Available from: https://arxiv.org/abs/2401.01269 [Last assessed on 2025 Jan 04].

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*. Available from: https://arxiv.org/abs/2310.18532 [Last assessed on 2025 Jan 04].

Moratis, K., Diamantopoulos, T., Nastos, D.N., and Symeonidis, A., 2024. Write me this code: An analysis of chatGPT quality for producing source code. In: *Proceedings - 2024 IEEE/ACM 21st International Conference on Mining Software Repositories, MSR 2024*, pp.147-151.

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.

Petersen, K., Vakkalanka, S., and Kuzniarz, L., 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64(5), pp.1-18.

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.

Rahmaniar, W., 2024. ChatGPT for software development: Opportunities and challenges. *IT Professional*, 26(3), pp.80-86.

Ramamoorthy, J., Gupta, K., Kafle, R.C., Shashidhar, N.K., and Varol, C., 2024. A novel static analysis approach using system calls for linux IoT malware detection. *Electronics*, 13(15), p.2906.

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.

Salem, N., Hudaib, A., Al-Tarawneh, K., Salem, H., Tareef, A., Salloum, H., and Mazzara, M., 2024. A survey on the application of large language models in software engineering. *Computer Research and Modeling*, 16(7), pp.1715-1726.

Sikand, S., Mehra, R., Sharma, V.S., Kaulgud, V., Podder, S., and Burden, A.P., 2024. Do generative AI tools ensure green code? An investigative study. In: *Proceedings of the 2nd International Workshop on Responsible AI Engineering*. ACM, New York, USA, pp.52-55.

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. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 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, USA, pp.35-39.

Villmow, J., Campos, V., Petry, J., Abbad-Andaloussi, A., Ulges, A., and Weber, B., 2023. How well can masked language models spot identifiers that violate naming guidelines? In: *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, United States, pp.131-142.

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. In: *Proceedings of the ACM on Software Engineering*. Vol. 1. ACM, United States, pp.789-811.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q., 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4), pp.911-936.

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *ACM International Conference Proceeding Series*. ACM, United States.

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.

Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., and Lou, Y., 2024. Evaluating and improving chatGPT for unit test generation. *Proceedings of the ACM on Software Engineering*. 1(FSE), pp.1703-1726.

Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., and Chen, Z., 2023a. *A Survey on Large Language Models for Software Engineering*. Available from: https://arxiv.org/abs/2312.15223 [Last assessed on 2025 Jan 04].

Zhang, Z., Chen, C., Liu, B., Liao, C., Gong, Z., Yu, H., Li, J., and Wang, R., 2023b. *Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code*. pp.1-99. Available from: https://arxiv.org/abs/2311.07989 [Last assessed on 2025 Jan 04].

Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M., and Chen, J., 2023a. A survey of large language models for code: Evolution, benchmarking, and future trends. *ACM Transactions on Software Engineering and Methodology*, 31(2), pp.1-44.

Zheng, Z., Ning, K., Zhong, Q., Chen, J., Chen, W., Guo, L., Wang, W., and Wang, Y., 2023b. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2), p.50.

Zhou, X., Cao, S., Sun, X., and Lo, D., 2024. *Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead*. Vol. 1. Available from: https://arxiv.org/abs/2404.02525 [Last assessed on 2025 Jan 04].