IJHSR
ijhighschoolresearch.org
REVIEW ARTICLE

Large Language Models in Automated Code Review: A Review

Ethan C. Lee

Interlake High School, 16245 NE 24th St, Bellevue, Washington, 98008, USA; ethan7sky@gmail.com

ABSTRACT: Code review is traditionally conducted by human reviewers, leading to its time- and labor-intensive nature, and assisted by tools such as linters and static analyzers, which are limited by their rule-based functionalities. Given that code review is essential to and widespread in modern software engineering, as it helps to ensure high-quality codebases, recent literature now examines the possibility of integrating large language models (LLMs) to analyze source code and automate the code review process. This paper reviews and synthesizes recent works regarding this endeavor, specifically investigating current approaches regarding frameworks and benchmarks to assess model performance, varying architectural and dataset design choices, and hybrid implementations of LLM-based code review. By noting current challenges, such as the lack of domain-specific integration, and possible future directions, such as human-in-the-loop systems, this paper finds that LLM-based code review has large potential to create increasingly intelligent and streamlined codebases.

KEYWORDS: Computer Science, Software Engineering, Machine Learning, Large Language Models, Software Quality Assurance, Automated Code Review.

Introduction

With the advent of new tools such as large language models (LLMs) in parallel with the growing size and complexity of modern software systems, recent works have sought to create automated tools to aid in this process.[1] One important part of modern software systems is software quality assurance (SQA), and a specific subcomponent within SQA is a process called code review. Code review ensures high-quality code by ensuring readability, detecting bugs, facilitating knowledge transfer within teams, and providing numerous other benefits.

Currently, code review is undertaken by human reviewers and rule-based tools. However, this traditional approach presents numerous drawbacks. For one, human reviewers are in high demand and expensive.[2] Furthermore, the complexity of software systems leads this process to be very time-intensive, with tools such as linters and static analyzers struggling to understand code in context and remaining limited to finding low-level mistakes (such as syntax errors).[2] To demonstrate, modern projects can demand up to tens of thousands of reviews each month.[3] There exist various strategies to tackle this issue, such as tools that recommend the best reviewer for a given change (using recommender systems)[4] and revising code before submitting it for review.[5]

In recent years, the application of LLMs in software engineering has been proven possible in multiple adjacent subdomain tasks, such as code generation and completion by LLMs such as Github Copilot, PolyCoder, and CodeT5.[1] Successes in adjacent domains prove that LLMs are more effective at understanding code, promoting research in the past few years investigating their application to code review. Recent studies conclude that LLMs in SQA have strong potential for scalability within the next decade.[6] Considering cost-effectiveness, one study quantifies that the usage cost of an LLM for code review is up to 99.46% lower than the highest average salary worldwide for a software engineer (i.e., $110,140 in the United States).[7]

In this review, we aim to address the limitations of traditional tools employed, which struggle with contextual understanding and the capacity to detect high-level and nuanced errors, and explore the recent advancement of LLM-driven code review tools. Specifically, this review aims to answer the following research questions:

- What are the recent evaluative, architectural, and training strategies that have been employed to design LLM-based code review tools?
- How are LLMs integrated into hybrid systems of code review?
- What challenges and future directions remain for their adoption in real-world applications?

Noting that recent literature is focused on a specific part of code review, such as security risk detection[8,9] or for a specific purpose, such as data science projects,[2] we aim to provide a holistic synthesis of these tools, from dataset design considerations to methods of assessing the models. We also further explore how these new tools can integrate into existing systems, specifically those involving human reviewers and traditional tools.

In what follows, we first outline the traditional tools and frameworks used for evaluation. Then we examine different architectures and dataset designs that boost performance, proposed hybrid systems that include humans in the loop or integrate static analyzers, and, finally, the challenges and opportunities that are present for LLM-based code review.

DOI: 10.36838/IJHSR89.70

Background, Frameworks, and Benchmarks

Background:

Here, we provide a background of code review and the traditional process it consists of. First, we define code review. To reiterate, code review describes the process of systematically reviewing changes to source code to ensure high-quality codebases. Commonly, code review is informal and widely practiced among large tech companies such as Microsoft and Google,[10,11] and is also asynchronous and focuses on code changes specifically.[11] In a typical review process, the author first submits their change onto a platform (tool-based review) such as CodeFlow or Git, then reviewers examine the change and initiate discussions, and finally, modifications are proposed.[11] This cycle repeats until the code meets requirements and is ready to be integrated into the larger codebase.

Code review has many benefits that make it essential to modern software systems. Besides simply detecting errors, code review can also ensure readability and consistency in codebases, aid in code transparency, knowledge transfer within software teams, ensure shared ownership, etc.[10] Interestingly, the tools that facilitate code review, such as the aforementioned Git, can collect data and create readily available databases for code review.[12] This is useful as it allows for the creation of other code review tools, such as training data for LLMs. For these reasons, code review is not merely used to avoid errors, but serves pivotal roles as a part of the foundation for modern software systems.

Unfortunately, due to its complex nature, code review can be time-consuming and resource-intensive. One study, for instance, finds that the median times for the review process can range from 14.7 to 19.8 hours,[13] although some companies, such as Google, have faster turnaround times by implementing less rigorous code review.[11]

Specifically, these times are greatly influenced by the reviewers encountering code that they are unfamiliar with. In other words, reviewers are presented with code without the context of the changes, leading them to often express confusion upon being presented with code changes without additional rationale and context.[14] When reviewers are presented with code without additional context, they are forced to perform unnecessary tasks such as running the code, sending clarification emails, and usually reading much more code than that submitted for review.[10] As logically follows, on the other hand, review requests take less time for reviewers familiar with the code, or in cases of small code changes and files with previous errors.[14] Surprisingly, it is also more common for reviewers to evaluate code in new contexts than it is for developers to work on new code.[10]

Another important aspect of code reviews is the traditional tools that are used. Common tools include linters and static analysis tools, which are rule-based and automatically flag issues such as syntax errors, style violations, security vulnerabilities, and areas of poor performance.[15] These tools benefit the manual code review process by offering low-level insights and detections to reviewers, which also means that they are incapable of replacing manual review.[16]

The limitations of traditional tools are rooted in their inability to identify complex, content-dependent issues;[15] for instance, static analysis tools search for specific patterns or rules in the source code, similar to antivirus programs, implying that they fail to find errors if certain patterns or rules are not specified.[16] In fact, studies find that warnings raised by static analysis tools are rarely removed during code review; rather, the broader integration of code into software systems is seen as a higher priority by reviewers.[14]

Overall, the limitations presented by traditional code review and the developments for LLM-based methods establish the grounds for the integration of LLMs into SQA. In particular, the limitations of current tools, such as static analyzers, as well as the labor and time-intensive nature of manual inspection, demonstrate the need for context-aware, faster, and scalable tools.

Frameworks and Benchmarks:

While there is no universal standard for code review, there are multiple frameworks that ensure best practices in the code review process. These general guidelines are often agreed upon by consensus among reviewers. These frameworks include process-oriented, programming-language-specific, architectural, and tool-based views,[17] and the extensiveness of these frameworks reflects the multiple scopes and diverse nature of code review practices.

While there are some common frameworks for standardizing traditional code review, there are a limited number of benchmarks to assess LLM capabilities in code review.[18] Many recent studies utilize common metrics that are used to assess any LLMs, not just in the application of code review. These include accuracy, precision, recall, and F1 score. One instance of this application is present in automated vulnerability detection in code.[9]

However, aside from these common metrics, some benchmarks have emerged that specialize in automated code review. For example, StackEval and StackUnseen are capable of assessing performance across different programming languages.[19] SWE-BENCH is a benchmark consisting of GitHub issues and pull requests aimed at assessing model performance on editing codebases, which is similar to code review but not engineered specifically for that purpose.[20] Additionally, benchmarks dedicated specifically to code review include CodeReview,[5] CodeReviewer,[21] Review-Explaining,[22] and Code-Review-Assist,[23] as compiled by Hu *et al.*[24] These benchmarks often utilize metrics such as Exact Match (EM), BLEU, and ROUGE to compare model-generated and reference human-written review comments.[24] However, current datasets face risks such as dataset contamination between training and test data and overreliance on similarity-based metrics (like BLEU and Exact Match), even though correct answers can vary between LLM-generated outputs.[24]

Overall, there remains a lack of a unified benchmark for evaluating LLM-based code review, especially quantitative ones. This is clearly demonstrated through the trend of implementing qualitative metrics to assess the outputs of LLM-based code review tools, relying on human evaluators

to determine the performance of models. While this method is not without merit, it demonstrates that more research is necessary concerning methods to reliably and systematically evaluate performance.

LLM-Based Code Review Tools: Structure

Model Architecture and Training Techniques:

Now, we will analyze model architectures and training techniques that have seen success in code review tasks. One key takeaway from synthesizing multiple works is that encoder-decoder architectures are most fruitful in constructing LLM-based code review tools. This is because the architecture enables accurate text outputs in various code review functionalities, such as comment generation and code quality estimation. In fact, empirical comparisons between encoder-decoder architectures and encoder-only architectures find that encoder-decoder models significantly outperform the latter. To be specific, T5-based models, which are based on the encoder-decoder architecture, have been found to significantly outperform other deep-learning architectures proposed in the previous year by Tufano *et al.*[5,25] Similarly, CodeReviewer is another recently proposed model that receives diffs (differences in original and changed code) as inputs and successfully utilizes a similar encoder-decoder architecture.[21]

Another approach is code review tools that are fine-tuned versions of large, existing models. Research has been conducted into parameter-efficient fine-tuning to minimize costs and complexity. One such example is Carllm and its variants, which were fine-tuned on base LLMs, including LLaMA and Magicoder.[26] The Carllm models were found to significantly outperform other baseline models such as CodeBert and CodeReviewer, with one model obtaining a 73.16% F1-Score and another noting 71.75% accuracy, in comparison to a maximum F1-Score of 68.44% and maximum accuracy of 67.80% among the baselines.[26] Another specific example is LLaMA-Reviewer, which applies fine-tuning strategies such as low-rank adaptation and prefix-tuning to a small LLaMA model, significantly reducing the number of trainable parameters and saving storage.[27] The study concludes that LLaMA-Reviewer achieves a superior recall of 83.50% and and matches baselines in F1-Score at 70.49%, while also achieving a high precision of 88.61% after threshold adjustment, exceeding all baselines.[27] Overall, both models demonstrate promise in fine-tuned LLMs, which are easier to construct and deploy compared to completely original LLMs. Table 1 displays the discussed code review models, both encoder-decoder architectures and fine-tuned models, with more detail.

Table 1: Comparison of the aforementioned LLM-based code review models. The results demonstrate that parameter-efficient training methods for fine-tuned, decoder-only transformers attain high performance, especially in tasks such as issue detection and comment generation.

Model Name	Base Architecture	Training Strategy	Capabilities	Key Metrics
T5 Text-To-Text-Transfer Transformer (small)[5]	Encoder-decoder transformer (6L/6L, 8-headed attention, 60M params), SentencePiece tokenizer	Pretrained with StackOverflow and CodeSearchNet, fine-tuned on 167k mined triplets from GitHub and Gerrit	Code change recommendation, comment-based code generation, and review comment generation	Code→Code: EM = 5% Code + Comment→Code: EM = 14% Code→Comment: EM = 2%, CodeBLEU = 0.80
Code-Reviewer[21]	Encoder-decoder transformer (12L/12L, 12-headed attention, 223M params), RoBERTa tokenizer	Initialized from CodeT5, pretrained on GitHub PR diffs and review comments, fine-tuned per task.	Code quality estimation, review comment generation, and comment-based code refinement	Code→Quality: F1 = 71.5%, accuracy = 73.9% Code → Comment: BLEU -4 = 5.32, Human eval = 3.20-3.60 (out of 5) Code + Comment → Code: EM = 30.3%, BLEU = 82.6
Carllm and variants[26]	Fine-tuned decoder-only transformers (LLaMA, LLaMA2, CodeLLaMA, and Magicoder)	Applied low-parameter fine-tuning (LoRA) on ~19.5k chain-of-thought data instances from 1793 GitHub projects	Code issue detection, issue localization, and review comment generation	Code→Issue Detection: F1 = 73.16%, accuracy = 71.75% (CodeCarllm 13B) Code→Comment: clear = 28.5-36.3%, unclear = 16.4-26.0%
LLaMA-Reviewer[27]	Fine-tuned decoder-only transformer (LLaMA 6.7B)	Utilized parameter-efficient fine-tuning (LoRA, zero-unit prefix-tuning) on code-review instruction data (Code Alpaca)	Review necessity prediction, review comment generation, and code refinement	Code→Necessity Prediction: F1 = 70.49%, recall = 83.50%, precision = 60.99% Code→Comment: BLEU-4 = 5.70, Tufano = 5.04 Code→Code: BLEU-4 = 82.27, Tufano = 78.23

Furthermore, these two architectures elicit inquiries addressing the larger tradeoff that exists between efficient, fine-tuned models that are more realistic to deploy and larger, complex models that have a higher performance ceiling at the cost of higher compute and storage costs. Unfortunately, such a comparative analysis is underexplored in current literature. An analysis of this tradeoff would be valuable in determining the exact relationship between performance and compute cost among different architectural choices for LLMs designed for code review.

Dataset Design:

Research finds that data curation and design choices result in massive accuracy gains and can shape model behavior. For instance, Ersoy and Erşahin introduce Code2Prompt, an LLM-powered code review tool trained on data-science-specific repositories from GitHub over subdomains including NLP, computer vision, and recommender systems, outperforming general LLMs such as GitHub Copilot and DeepCode significantly.[2] On the other hand, CodeReviewer, proposed by Li *et al.*, is trained on open-source projects found from GitHub, and its multilingual benchmark and ablation studies demonstrate the potential of general models trained on large and diverse datasets.[21]

Interestingly, many innovative techniques to strategically alter datasets have also resulted in improved model performance. For example, Lin *et al.* implemented oversampling by

DOI: 10.36838/IJHSR89.70

repeatedly exposing LLMs to review comments from more experienced reviewers during the training process.[12] The findings concluded that giving more weight to higher-quality comments was correlated with more insightful explanations and higher accuracy overall.[12] Surprisingly, the effectiveness of this oversampling technique was proven by outperforming CodeReviewer[21] by using the same dataset, but applying oversampling to achieve the said results.

Another innovative example is Carllm, which uses data from GitHub repositories and organizes code review comments into a chain-of-thought structure before applying it as training data, resulting in better reasoning and explainability capabilities in the end.[26] Lastly, Fan *et al.* explore the incorporation of both inputs (to learn the structures of code over time) and outputs (to determine whether issues the model flagged were valid or false positives) in model training.[28]

These decisions, such as oversampling and data structuring, stress the emerging experimentation of data preprocessing techniques in code review models, which yet remains underexplored. Existing research, as described above, suggests that code review models may collectively see performance improvements from datasets that incorporate reviewer experience and other specifications.

Hybrid Integration

Human in the Loop:

As a general trend, current literature agrees that LLM-based code review tools will not be able to completely automate code review and replace human reviewers. Indeed, it is essential to maintain human reviewers in the loop for two main reasons. For one, completely automating code review would remove vital benefits such as ensuring code transparency, team-wide communication, and knowledge transfer. Additionally, without human oversight, overlooked flaws in LLM-based code review tools could cause harm of massive scale. For these reasons, many researchers have logically focused on maintaining human reviewers and situating LLMs as enhanced, complex-aware tools.

There are various methods that models utilize to maintain human reviewers in the loop. Carllm is a model that improves cohesive interactions with the reviewer by incorporating user-interaction capabilities during the code review process, enabling reviewers to continuously refine outputs as desired.[26] This is valuable because this system maintains the critical-thinking skills of the reviewer. On the other hand, Tufano *et al.* aim to simulate the real code review process and allow easier reviewer integration into its outputs by exploring models with the dual ability to generate comments from code and vice versa.[5] Lastly, Zhou *et al.* propose Edit Progress (EP), which captures the LLM's partial improvements over the original code until the final, revised code.[29] In contrast to other models that output only once, this approach makes LLM-based tools more interactive for reviewers, increasing transparency behind decisions in the process and avoiding a fully automated system.

Broadly, Heander *et al.* propose an AI agent-based architecture in which tools can act as supportive agents to ensure code quality while maintaining team benefits of code review, including knowledge transfer, shared ownership, and process learning.[30] By avoiding complete automation and keeping humans in the loop, this approach keeps key team benefits from code review and minimizes risks such as false positives with human oversight. However, this integration remains theoretical and not tested, raising questions about how agent-based systems would affect the responsibilities of roles such as reviewers and authors.

Combining LLMs with Traditional Tools:

Research has also explored the possibility of hybrid systems to combine the benefits of context-aware LLMs with traditional tools. For instance, Jaoua *et al.* propose a combination of these learning based systems (LLMs) with knowledge-based systems (static analyzers), finding improved accuracy and coverage of review comments primarily by applying Retrieval Augmented Generation (at the inference stage) while exploring hybrid integrations at all three stages of data preparation, inference, and post-inference.[31]

Similarly, integrating static analyzers is the Intelligent Code Analysis Agent (ICAA), proposed by Fan *et al.*, which combines the rule-based and limited, traditional static analyzers with LLMs to dynamically detect both low-level bugs and high-level logic errors in code.[28] By implementing the ReAct Bug Detection Agent, which makes decisions to split code, run static analyzers, and other decisions in multiple steps, as well as a code-intention consistency checker that compares the predicted intention of code with its actual application, this model takes advantage of both the reasoning skills of LLMs and the reliable nature of static analyzers. Ultimately, the research found a notable reduction in false positive rates and a recall of 60.8%, comparable to other bug detectors.

Discussion

Challenges:

We will now discuss a few key challenges that are faced when creating LLM-based code review tools. The first challenge, present across multiple works, was the lack of quantitative metrics for assessing outputs, especially for generated code review comments, due to their open-ended nature. Instead, many models turned to the application of qualitative metrics, which lack the same level of rigor as quantitative metrics and are more easily unknowingly influenced by bias and human subjectivity. Additionally, alternative traditional metrics employed to assess LLMs in general, such as Exact Match, proved inadequate in fully assessing outputs such as those for code refinement and comment generation tasks. Concrete steps moving forward include developing metrics that combine multiple existing metrics, such as accuracy and F1-Score, with more qualitative aspects, such as comment readability and reasoning quality, whether they be captured automatically or human-evaluated, to better capture the holistic capabilities of models.

Additionally, one key challenge is generalizability. Due to the widespread adoption of code review as a common practice across various domains, many LLM-based code review tools will find difficulties in generalizing to specific fields due to their varying terminologies, standards, and workflows.[1] As

LLM-based code review tools are relatively new, they have not yet been integrated into specific fields, and this capability has not been thoroughly assessed. Furthermore, as most current models are trained on large, open-source repositories, it is difficult to identify which data is underrepresented. The generalizability of these models to other fields also puts an emphasis on explainability to ensure transparency behind any model decisions. A possible recommendation is to conduct further research into domain-specific fine-tuning, given its demonstrated effectiveness in existing code review models.

Lastly, another consideration is the ethical side. In training bias, for instance, coding practices that are more common in one developer community could be overrepresented, perpetuating biases.[15] Furthermore, automating code review risks overreliance, potentially leading to developers trusting the system instead of applying their critical thinking and understanding skills.[15] Lastly, security concerns also arise from collecting data from large databases such as GitHub. Although some solutions to these concerns include anonymized fine-tuning and private cloud deployments, they are still costly and understudied.[1] For example, models trained on source code from open repositories can be vulnerable to membership inference attacks (MIAs), in which attackers can determine whether the training data included a specified selection of code, eliciting privacy concerns.[32] Due to the possibility of data leaks and other risks associated with collecting code from real-world applications, this challenge also requires thoughtful consideration and accentuates the need for data governance frameworks.

Limitations of This Study:

The limitations of this review are rooted in the aforementioned, existing challenges within the domain of LLM-based code review. To begin, the review does not quantitatively compare different models and architectures, but rather examines the metrics reported in each study for a broader qualitative comparison. This is due to a lack of standard benchmarks for examining the different models across studies. Furthermore, the literature explored in this review originates predominantly from the last few years, with the possibility of missing long-term developments past this timeframe. Finally, the literature explored exists in academia and remains theoretical. With a lack of real-world implementation in societal domains, the true effectiveness of the deployment of code review models remains a gap in this study.

Future Prospects:

The field of LLM-based code review is still widely understudied, as indicated by the fact that most papers regarding this tool originate from the last two to three years. This fact highlights the vast opportunities that yet remain for research and experimentation. A few directions hinted at earlier in this review include exploring reviewer-LLM interactions to streamline the code review process, delving deeper into cost-performance tradeoffs between different architectures, and exploring how well these models perform for specific fields. Additionally, given that most research focuses on functionalities such as bug detection or review comment generation, there is much potential in exploring the integration of LLM capabilities with other functionalities, such as ensuring readability in code. Finally, more research is needed in creating standardized, quantitative metrics that effectively capture the quality of LLM code review outputs.

Beyond academia, automated code review has large potential in its application to industry. This review finds that deploying these tools is feasible and beneficial given their low compute costs, hybrid systems such as human-in-the-loop practices, and efficient fine-tuning strategies. Additionally, these models are anticipated to greatly accelerate software development by serving as an assistive role, not a fully automated system, within processes such as pull request reviews and code standardization. Maintaining human oversight, LLMs can assist developers with tasks such as bug detection and review comment generation. As evaluation metrics become unified and models continue seeing performance gains, LLM-based code review models will assuredly become increasingly viable for use in professional environments.

■ Conclusion

This review has analyzed the current approaches to LLM-based code review tools, specifically regarding model architectures, dataset designs, hybrid integrations, benchmarks, and a discussion of prevalent challenges and potential prospects. Overall, recent developments are characterized by best performance in encoder-decoder models, high potential in fine-tuning models, various innovative dataset design strategies, possibilities of integration with human-in-the-loop systems or utilizing static analyzers, and a lack of quantitative metrics for open-ended output tasks. Additionally, the recent surge in performance improvements and creative, innovative solutions over the past few years indicates that this topic has high potential and many research directions, as well as the possibility of integrating LLM-based code review tools into specific fields. In conclusion, while key challenges such as security and generalizability remain, LLM-based code review tools show large promise in creating intelligent and efficient software systems.

■ Acknowledgments

I would like to acknowledge and express my gratitude towards my mentors, Dr. Siddharth Krishnan and Dr. Plinio Zanini, as well as Indigo Research, for their guidance and support in writing this paper.

■ References

1. Patil, A. Advancing Software Quality: A Standards-Focused Review of LLM-Based Assurance Techniques. *arXiv* **2025**, 1–6. DOI: 10.48550/arXiv.2505.13766
2. Ersoy, P.; Erşahin, M. A Deep Dive into LLM-Powered Code Review Tools: A Comparative Analysis. *Computational Intelligence and Machine Learning* **2024**, *5* (2), 1–5. DOI: 10.36647/CIML/05.02.A001
3. Yang, X.; Kula, R. G.; Yoshida, N.; Iida, H. Mining the Modern Code Review Repositories: A Dataset of People, Process and Product. *Proceedings of the 13th International Conference on Min-*

DOI: 10.36838/IJHSR89.70

ing Software Repositories; ACM: Austin, Texas, 2016; pp 460–463. DOI: 10.1145/2901739.2903504

4. Chouchen, M.; Ouni, A.; Mkaouer, M. W.; Kula, R. G.; Inoue, K. WhoReview: A Multi-Objective Search-Based Approach for Code Reviewers Recommendation in Modern Code Review. *Applied Soft Computing* **2021**, *100*(1), 1–42. DOI: 10.1016/j.asoc.2020.106908
5. Tufano, R.; Masiero, S.; Mastropaolo, A.; Pascarella, L.; Poshyvanyk, D.; Bavota, G. Using Pre-Trained Models to Boost Code Review Automation. *Proceedings of the 44th International Conference on Software Engineering*; ACM: Pittsburgh, Pennsylvania, 2022; pp 2291–2302. DOI: 10.1145/3510003.3510621
6. He, J.; Shi, J.; Zhuo, T. Y.; Treude, C.; Sun, J.; Xing, Z.; Du, X.; Lo, D. From Code to Courtroom: LLMs as the New Software Judges. *arXiv* **2025**. DOI: 10.48550/arXiv.2503.02246
7. Pornprasit, C.; Tantithamthavorn, C. Fine-Tuning and Prompt Engineering for Large Language Models-Based Code Review Automation. *Information and Software Technology* **2024**, *175*, 1–12. DOI: 10.1016/j.infsof.2024.107523.
8. Tehrani, M. G.; Sultanow, E.; Buchanan, W. J.; Houmani, M.; Fodja, C. H. D. LLM vs. SAST: A Technical Analysis on Detecting Coding Bugs of GPT4-Advanced Data Analysis. *arXiv* **2025**. DOI: 10.48550/arXiv.2506.15212
9. Khare, A.; Dutta, S.; Li, Z.; Solko-Breslin, A.; Alur, R.; Naik, M. Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. *Proceedings of the 2025 IEEE Conference on Software Testing, Verification and Validation (ICST)* **2025**, 1–12. DOI: 10.1109/ICST62969.2025.10988968
10. Bacchelli, A.; Bird, C. Expectations, Outcomes, and Challenges of Modern Code Review. *Proceedings of the 35th International Conference on Software Engineering (ICSE)* **2013**, 712–721. DOI: 10.1109/ICSE.2013.6606617
11. Sadowski, C.; Söderberg, E.; Church, L.; Sipko, M.; Bacchelli, A. Modern Code Review: A Case Study at Google. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*; ACM: Gothenburg, Sweden, 2018; pp 181–190. DOI: 10.1145/3183519.3183525.
12. Lin, H. Y.; Thongtanunam, P.; Treude, C.; Charoenwet, W. Improving Automated Code Reviews: Learning from Experience. *Proceedings of the 21st International Conference on Mining Software Repositories*; ACM: Lisbon, Portugal, 2024; pp 278–283. DOI: 10.1145/3643991.3644910
13. Rigby, P. C.; Bird, C. Convergent Contemporary Software Peer Review Practices. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*; ACM: Saint Petersburg, Russia, 2013; pp 202–212. DOI: 10.1145/2491411.2491444
14. Davila, N.; Nunes, I. A Systematic Literature Review and Taxonomy of Modern Code Review. *arXiv* **2021**, 1–36. DOI: 10.48550/arXiv.2103.08777
15. Mathew, K. Automating Code Review Systems Using Natural Language Processing. *International Journal of Emerging Trends in Computer Science and Information Technology* **2025**, 593–603. DOI: 10.56472/ICCSAIML25-165
16. Stefanović, D.; Nikolić, D.; Dakić, D.; Spasojević, I.; Ristić, S. Static Code Analysis Tools: A Systematic Literature Review. *Proceedings of the 31st DAAAM International Symposium*; Vienna, Austria, 2020; pp 565–573. DOI: 10.2507/31st.daaam.proceedings.078
17. Nelson, S.; Schumann, J. What Makes a Code Review Trustworthy? Proceedings of the *37th Annual Hawaii International Conference on System Sciences*; Big Island, HI, 2004; pp 1–10. DOI: 10.1109/HICSS.2004.1265711
18. Jiang, J.; Wang, F.; Shen, J.; Kim, S.; Kim, S. A Survey on Large Language Models for Code Generation. *arXiv* **2024**, 1–70 DOI: 10.48550/arXiv.2406.00515
19. Shah, N.; Genc, Z.; Araci, D. Stackeval: Benchmarking LLMs in Coding Assistance. *Proceedings of the 38th Conference on Neural Information Processing Systems*; 2024; pp 36976–36994. DOI: 10.48550/arXiv.2412.05288
20. Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; Narasimhan, K. SWE-Bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv* **2024**. DOI: 10.48550/arXiv.2310.06770
21. Li, Z.; Lu, S.; Guo, D.; Duan, N.; Jannu, S.; Jenks, G.; Majumder, D.; Green, J.; Svyatkovskiy, A.; Fu, S.; Sundaresan, N. Automating Code Review Activities by Large-Scale Pre-Training. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; ACM: Singapore, Singapore, 2022; pp 1035–1047. DOI: 10.1145/3540250.3549081.
22. Widyasari, R.; Zhang, T.; Bouraffa, A.; Maalej, W.; Lo, D. Explaining Explanations: An Empirical Study of Explanations in Code Reviews. *ACM Transactions on Software Engineering and Methodology* **2025**, *34* (6), 1–30. DOI: 10.1145/3708518
23. Sghaier, O. B.; Sahraoui, H. A Multi-Step Learning Approach to Assist Code Review. *Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*; IEEE, 2023; pp 450–460. DOI:10.1109/SANER56733.2023.00049
24. Hu, X.; Niu, F.; Chen, J.; Zhou, X.; Zhang, J.; He, J.; Xia, X.; Lo, D. Assessing and Advancing Benchmarks for Evaluating Large Language Models in Software Engineering Tasks. *arXiv* **2025**. DOI: 10.48550/arXiv.2505.08903
25. Tufano, R.; Pascarella, L.; Tufano, M.; Poshyvanyk, D.; Bavota, G. Towards Automating Code Review Activities. *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*; IEEE, 2021; pp 163–174.
26. Yu, Y.; Rong, G.; Shen, H.; Zhang, H.; Shao, D.; Wang, M.; Wei, Z.; Xu, Y.; Wang, J. Fine-Tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review. *ACM Transactions on Software Engineering and Methodology* **2025**, *34* (1), 1–26. DOI: 10.1145/3695993.
27. Lu, J.; Yu, L.; Li, X.; Yang, L.; Zuo, C. Llama-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. *Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*; IEEE, 2023; pp 647–658.
28. Fan, G.; Xie, X.; Zheng, X.; Liang, Y.; Di, P. Static Code Analysis in the AI Era: An In-Depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents. *arXiv* **2023**. DOI: 10.48550/arXiv.2310.08837
29. Zhou, X.; Kim, K.; Xu, B.; Han, D.; He, J.; Lo, D. Generation-Based Code Review Automation: How Far Are We? *Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*; IEEE, 2023; pp 215–226. DOI: 10.1109/ICPC58990.2023.00036
30. Heander, L.; Söderberg, E.; Rydenfält, C. Support, Not Automation: Towards AI-Supported Code Review For Code Quality and Beyond. *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*; ACM: Trondheim, Norway, 2025; pp 591–595. DOI: 10.1145/3696630.3728505
31. Jaoua, I.; Sghaier, O. B.; Sahraoui, H. Combining Large Language Models with Static Analyzers for Code Review Generation. *Proceedings of the 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*; IEEE, 2025; pp 174–186. DOI: 10.48550/arXiv.2502.06633
32. Yang, Z.; Zhao, Z.; Wang, C.; Shi, J.; Kim, D.; Han, D.; Lo, D. Gotcha! This Model Uses My Code! Evaluating Membership Leakage

Risks in Code Models. *IEEE Transactions on Software Engineering* **2024**, *50* (12), 3290–3306. DOI: 10.1109/TSE.2024.3482719

Author

Ethan C. Lee is a junior at Interlake High School in Bellevue, Washington. He is interested in pursuing a career in computer science and software engineering, particularly within the subfields of artificial intelligence and machine learning.

DOI: 10.36838/IJHSR89.70