Research Paper

Deep Learning-based Code Reviews: A Paradigm Shift or a Double-Edged Sword?

- **College:** College of Computing
- **Major:** Computer Science
- **Supervisor:** W.K. Chan
- **Student Name:** CHEN Xian
- **Student Number:** [redacted]

Nov. 21st, 2025

1 Introduction

In contemporary software engineering, code review has long been recognized as a foundational practice for ensuring software quality. Beyond serving as an initial safeguard for defect detection, code review functions as a crucial mechanism for knowledge sharing, maintaining engineering standards, and fostering collective code ownership within development teams. From Fagan's early formal inspections [1] to the lightweight practices embodied in modern platforms such as GitHub and GitLab, the code review process has undergone decades of evolution. Yet, the rapid expansion of software system complexity and the widespread adoption of Continuous Integration and Continuous Deployment (CI/CD) pipelines have introduced unprecedented challenges. Large-scale projects now generate tens of thousands of pull requests annually, placing significant cognitive load and time pressure on reviewers. Prior studies have shown that under conditions of excessive workload or reviewer fatigue, the quality of review decisions deteriorates, increasing the likelihood that defects escape into production environments.

To alleviate these challenges, the software engineering research community has long pursued automated support tools. Early approaches centered on static analysis tools [2]. While static analysis tools excel at detecting syntactic issues and style violations, their high false-positive rates and limited contextual understanding restrict their ability to provide deep, human-like feedback.

In recent years, breakthroughs in deep learning—especially the emergence of Large Language Models (LLMs) [3] such as GPT4 [4] and GitHub Copilot—have fundamentally reshaped the landscape of automated code review. Unlike rule-based tools, LLMs built on Transformer architectures possess powerful generative capabilities that allow them to understand source code semantically and produce natural-language feedback, often accompanied by concrete suggestions for fixes. These models have therefore been heralded as potential AI co-reviewers capable of transforming existing review workflows.

Although LLMs perform impressively in benchmark evaluations, their real-world effects on developer behavior, review quality, and reviewer cognition are still poorly understood. Industrial reports often emphasize productivity gains, but whether such gains translate into higher-quality code reviews or whether they introduce risks such as automation bias remains unclear. Against this backdrop, the study by Rosalia Tufano et al., "Deep Learning-based Code Reviews: A Paradigm Shift or a Double-Edged Sword?" [5], makes a significant contribution. Instead of focusing solely on model performance, the authors investigate the human and socio-technical dimensions of LLM assisted code review, revealing how AI integration reshapes this essential collaborative practice. In addition, the definition of "cost" in this context remains contested. It is often assumed that automation saves time, yet in code review, reading, understanding, and verifying AI generated suggestions some of which may be hallucinated is itself a time-consuming cognitive activity. If the time required to validate AI suggestions exceeds the benefits they provide, then the purported "efficiency gains" may be a false premise. Therefore, their study breaks down the problem into three core dimensions: (1) Review Quality: Does AI assistance enable reviewers to identify more, or more severe, software defects? Or does the presence of AI lead reviewers to overlook certain critical issues? (2) Review Cost: Is reviewing code with AI-provided initial comments truly more time-efficient than performing a

manual review from scratch? This involves precise measurement of total duration, code inspection time, and comment-writing time. (3) Reviewer's Confidence: Does AI involvement increase reviewers' certainty regarding the final submission?

Their experiment recruited 29 professional developers with substantial industry experience, systematically injected 48 defects into six Java and Python projects, and employed three review conditions: manual code review (MCR) without assistance, AI-assisted review (ACR) using GPT-4-generated comments, and comprehensive review (CCR) simulating an ideal AI system. The research team also developed a custom VS Code plugin capable of capturing millisecond-level behavioral data, enabling a comprehensive evaluation across three dimensions above. The results challenge prevailing optimism about AI-augmented programming. In both the ACR and CCR conditions, AI-generated comments induced strong anchoring effects, directing reviewers' attention toward AI-highlighted lines and reducing exploration of the remaining code. Although AI assistance increased the total number of issues identified, this gain was driven largely by low-severity findings, with no improvement in detecting high-severity functional defects. Moreover, AI assistance did not reduce review time; even when presented with "perfect" suggestions, reviewers had to invest substantial cognitive effort to verify them. AI involvement also did not enhance reviewers' confidence, because it did not substantially improve the understanding of the underlying code logic.

2 Paper Analysis

2.1 Evaluation of the key strengths

The Excellence of Participant Composition

The most notable strength of this study lies in the high-quality composition of its participants. Empirical research in software engineering has long suffered from concerns about the validity of using "students as subjects." Students generally lack experience maintaining industrial-scale codebases, and their cognitive patterns tend to focus on simple "bug finding" rather than systematic "quality maintenance." Tufano et al. successfully recruited 29 professional developers, the vast majority of whom had several years of full-time development experience (average 11.4 years) and covered both reviewer and contributor roles. This sample selection greatly enhances the generalizability of the study's conclusions to real world industrial contexts.

Innovative Experimental Design of the CCR Treatment Group

The introduction of the "Comprehensive Code Review" (CCR) group is the methodological highlight of this study. When evaluating AI tools, a common defense is that "current models are not good enough; once the model is perfect, the problem will be solved." However, the CCR group simulated a hypothetical scenario in which a technological singularity had already occurred by providing manually constructed "perfect AI comments." The experimental results showed that even in this ideal scenario, reviewers did not save time and still neglected unmarked areas. This design successfully disentangled issues of "model capability" from "human-computer interaction," strongly demonstrating that the current bottleneck is not merely AI accuracy but also the inherent cognitive mechanisms of humans when verifying automated outputs. This counterfactual reasoning experimental design sets a new benchmark for research on HCI (Human-Computer Interaction) in software engineering.

2.2 Key weaknesses of the research content

Controversy over the Realism and Complexity of Injected Defects

Although the researchers made efforts to simulate realistic defects, artificially injected bugs (Injected Bugs) are inherently different from naturally occurring "organic bugs." Previous studies have noted that synthetic defects often reside on the code's "main paths" and usually have relatively local contextual dependencies, making them easier to detect. In contrast, high risk vulnerabilities in real-world software often involve complex cross-module interactions or deep business logic violations, which are currently the hardest for code generation models to capture. If the injected defects in the experiment are too "atomic" or lack deep contextual dependencies, the performance of human reviewers (MCR) may be artificially inflated, potentially masking AI's advantages or disadvantages in handling certain types of complex defects. Moreover, although six projects were included, most of them consist of algorithm snippets or utility libraries (e.g., Rosetta Code) and lack the large architectural context and historical technical debt typical of industrial-scale software, which may limit the generalizability of the conclusions to very large-scale software systems.

Lack of Longitudinal Consideration

This experiment is a typical cross-sectional study, capturing participants' performance in a single task "snapshot." However, human adaptation to new technologies is a dynamic learning process. Developers' trust in AI assistants is not static but undergoes continuous calibration (Trust Calibration) based on experience. Initially, developers may spend extra time on verification due to novelty or distrust, but as proficiency increases, they may learn to recognize AI "hallucinations" and improve verification efficiency. The current experimental design cannot capture this "learning curve" effect. Therefore, the observed "no time savings" conclusion may reflect the adaptation period costs rather than the true long term steady state efficiency.

Statistical Power Limitations of the Sample Size

Although the sample of 29 professional developers is of high quality, at a statistical level, especially when divided into three treatment groups (approximately 9 to 10 participants each) and two programming languages, the statistical power may be insufficient to detect small effect sizes. The study noted some trends (e.g., slight changes in confidence ratings) that did not reach statistical significance, which may be due to a Type II error caused by limited sample size.

3 Future Work

3.1 Self-reflection

I believe that to truly unravel the enigma of "AI code review," research must shift from the current focus on "performance evaluation" to a deeper investigation of "cognitive interaction." The existing experiments reveal the phenomena (What) but have not fully explained the underlying mechanisms (Why), nor have they provided actionable solutions (How).

The application of AI in code review is currently at a delicate turning point. It holds the potential to become a powerful tool for improving code quality, but if poorly designed, it could also foster cognitive laziness. Future research must move beyond a narrow focus on

“model performance competitions” and shift toward more human-centered designs of “human-AI symbiotic systems”. Only by deeply understanding and respecting the cognitive patterns of human developers can we truly harness this double-edged sword.

3.2 Method

Longitudinal Adaptation Study

To address the limitations of studies in that dimension, a primary future direction is to conduct long-term research. Such a study would involve selecting a mid-sized software team and integrating AI review tools into their daily workflow, with continuous observation over a period of three to six months. The core metric would track the evolution of verification time over this period. It is hypothesized that efficiency may initially decline due to distrust and the adaptation process, but as developers gain a clear understanding of the AI’s capabilities and limitations, they would learn to quickly disregard low quality suggestions, leading to improved efficiency in later stages. Additionally, longitudinal research should consider the risk of deskilling: prolonged reliance on AI for style or quality checks could potentially diminish junior developers’ ability to identify such issues manually, which would need to be assessed through baseline testing conducted before and after the study.

Construction and Testing of Complex Organic Defects

To address the problem of overly simplistic injected defects, future experimental materials should be drawn from the regression testing histories of real open-source projects. Using bug mining techniques, extracted complex logical errors that have led to serious production incidents often spanning multiple files or depending on specific runtime states. Testing AI-assisted review on these “high-difficulty” samples is essential to truly evaluate whether LLMs have the potential to function as “senior engineers” or are merely “advanced Linters.”

References

[1] Fagan M. Design and code inspections to reduce errors in program development[M]//Software pioneers: contributions to software engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011: 575-607.

[2] Emanuelsson P, Nilsson U. A comparative study of industrial static analysis tools[J]. Electronic notes in theoretical computer science, 2008, 217: 5-21.

[3] Naveed H, Khan A U, Qiu S, et al. A comprehensive overview of large language models[J]. ACM Transactions on Intelligent Systems and Technology, 2025, 16(5): 1-72.

[4] Achiam J, Adler S, Agarwal S, et al. Gpt-4 technical report[J]. arXiv preprint arXiv:2303.08774, 2023.

[5] Tufano R, Martin-Lopez A, Tayeb A, et al. Deep Learning-based Code Reviews: A Paradigm Shift or a Double-Edged Sword?[J]. arXiv preprint arXiv:2411.11401, 2024.