From Overload to Insight: Bridging Code Search and Code Review with LLMs

Nikitha Rao
nikitharao@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Bogdan Vasilescu
vasilescu@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Reid Holmes
rtholmes@cs.ubc.ca
University of British Columbia
Vancouver, BC, Canada

Abstract

The software engineering (SE) research community has developed numerous tools to search and extract actionable insights from software artifacts, ranging from static analysis tools to testing frameworks and continuous integration pipelines (hereafter just "search tools"). Despite their potential, many of these search tools remain underutilized during code review, a critical process for ensuring software quality. Key challenges include the overwhelming volume of information generated by automated tools, high false-positive rates, and the need for manual configuration or interpretation, which disrupts the flow of review. In this paper, we propose a vision for an LLM-powered conversational agent designed to assist code reviewers by bridging the gap between human reviewers and search tools. This agent would summarize relevant insights, tailor them to the specific code change under review, and facilitate context-aware interactions. By enhancing the human-in-the-loop nature of code review, such a tool has the potential to amplify reviewer effectiveness, streamline the review process, and ultimately improve software quality.

CCS Concepts

• **Software and its engineering → Software maintenance tools**; *Integrated and visual development environments.*

Keywords

Code Review, Human-in-the-Loop, AI-Development Support

ACM Reference Format:
Nikitha Rao, Bogdan Vasilescu, and Reid Holmes. 2025. From Overload to Insight: Bridging Code Search and Code Review with LLMs. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728518

1 Introduction

Over the past several decades, the software engineering (SE) research community has made significant strides in developing tools and techniques to extract useful information from software artifacts (e.g., [6]). These tools span a wide range of categories, from static analysis tools that detect potential bugs, vulnerabilities, or code smells, to testing frameworks and continuous integration (CI) infrastructure that monitor the behavior of software during development [2, 8, 12]. Together, these tools can be thought of as a broad class of "search" tools, designed to mine and interpret various forms of information hidden within software systems, in source code, the history of changes to the repository, documentation, etc.

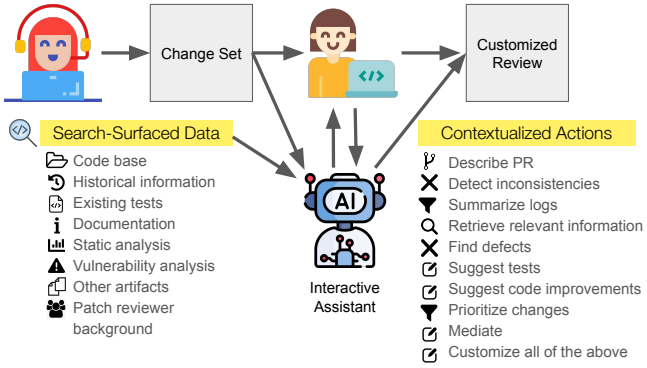

Figure 1: An LLM-based interactive assistant could sift through the large volume of information produced by code search tools, and create a customized code review experience.

Unfortunately, many of these tools go underutilized [7], especially during code review [5], one commonly-used checkpoint for ensuring that changes made to software systems do not degrade its quality [3, 9]. Two main obstacles hinder the effective use of search tools during code review. First, in contexts with a high degree of automation, many such tools are already invoked as part of CI pipelines [8, 12, 15]. The sheer volume of information generated by them, only a small fraction of which may be relevant to the particular code change under review, can be overwhelming for reviewers [17]. Moreover, many of these tools suffer from high false positive rates, causing alert fatigue [14, 16]. Second, if one needs to invoke tools on demand, the diversity of available tools places a heavy burden on the reviewer, who both has to know that they should invoke the search tool, along with the knowledge of how to use the search tool to use it to gather information pertinent to their current review task. Many search tools are not designed with the specific needs of code reviewers in mind, requiring manual configuration or interpretation that can disrupt the flow of the review. These two challenges combine to impede the velocity of code reviews, which are often constrained by tight timelines [4]. This has led modern code review practice to primarily focus on careful manual examination of compact diff-style changes and enabling collaborative discussion, instead of incorporating deeper consideration of more comprehensive tool-based analyses.

FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728518

The emergence of AI-supported programming tools, such as Microsoft's Copilot, is poised to further increase the importance of code review in the software development process. This is due to both the increased cadence with which code can be produced using AI [1], and the importance of having a quality gate to assess the correctness of both AI-generated and human-written changes.

However, while these AI-based approaches, often using large language models (LLMs), are inducing new pressure on the code review process, they can also improve how engineers perform code reviews. Our vision is that instead of expecting reviewers to manually sift through logs, warnings, and tool output, an LLM-powered conversational agent acts as a bridge between code reviewers and the plethora of available search tools, summarizing relevant insights, presenting them in a way that is tailored to the specific context of the code change under review, and allowing back-and-forth discussion (Figure 1). Central to this vision is the recognition that code review is fundamentally a human-in-the-loop process – our goal is not to replace human reviewers but to amplify their effectiveness through improved tooling. This LLM-powered assistance should enable reviewers to more completely, effectively, and quickly assess a given change. Concurrently, these advances will also enable developers to improve their changes before they are submitted for review, enabling the code review process to be more efficient and ultimately improving software quality.

2 Opportunity for AI-Enhanced Code Review

The current patch-centric approach to code review constrains reviewers to a narrow, change-focused view that often lacks broader context and project-wide implications of a change. Patches further focus reviewer effort on the exact change itself, without the additional tool-managed metadata and analyses commonly present during modern software development (e.g., code coverage information, static analysis feedback, security analyses). This format is not particularly amenable to rich human interaction or analyses that extend beyond the immediate contents of the patch itself. Reviewers are frequently left to manually piece together the wider impact of changes, cross-reference related parts of the code base, and consider other broad cross-cutting questions such as security and privacy. Each of these are time-consuming tasks that increase cognitive load and reduce reviewer efficiency.

However, we argue that code review offers one of the most interesting touch points where AI-based tools and humans could collaborate on cognitively demanding, highly technical software engineering tasks. Such collaboration can be both natural and potentially highly effective. *Natural* in that code review is already an interactive process, in which participants (historically, humans) seek consensus on the quality of a patch and the fate of a merge request through natural language dialogue. An LLM-based conversational agent would fit naturally in this process, providing a natural analysis interface through which developers can ask questions about the code change, and interact with analysis tools using their own domain understanding. One major downside of many analysis tools is that developers must know that a tool exists, how to get the data from it, and when to apply the tool. Being able to use natural language can free developers to focus on their intention and allow intelligent agents to manage the complexity of marshaling these tools for them [13].

And potentially *highly effective* in that LLMs have shown remarkable abilities at summarizing and synthesizing structured text, in addition to customizing responses [18]. By summarizing and synthesizing data from various analysis tools, the AI could intelligently augment the patch with minimal sets of important information to help the reviewer better understand the impact and context behind a change. In addition, the customization capability could enable the AI to personalize the information to the human actors involved in a way that makes it more useful. Taken together, AI augmentation could reduce the limitations of traditional patch-based code review by providing a more holistic, context-rich environment that extends far beyond the constraints of traditional patch-based workflows.

3 Realizing the Vision

There are an abundance of information sources that can help reviewers during the code review process. Furthermore, there are a number of underlying actions that an LLM-based conversational agent can perform on these information sources to provide meaningful insights and answer specific questions that a reviewer might have about the code change. Crucially, this is a reviewer-driven search process that is only enabled by the LLM: all of these data are not relevant for every patch (nor for every reviewer); providing a lightweight and intuitive mechanism for surfacing this information is the core idea underlying this work.

In the following, we expand on the sources of information and types of actions. In addition, we list a number of concrete ideas on how the LLM assistant could enhance the code review experience with information from search tools in Table 1. The list is not intended to be exhaustive, although we cover the main goals of the code review process reported in the literature [3], from *finding defects* to *knowledge transfer*; we also include two tasks part of the code review process, where we expect AI augmentation to be fruitful — deciding how to present the change set to the reviewer (*prioritization*) [11] and managing possible interpersonal conflicts between the submitter and reviewers (*mediation*) [10]. Rather, we seek to illustrate the potential for substantial advances in this area, and inspire future research. Please see the supplementary material DOI 10.5281/zenodo.15265736 for concrete examples of prompts and responses that demonstrate personalized support during the code review process based on reviewer background.

3.1 Sources of Information

🗁 *Code Base.* Often a code change alone may not provide enough context for an effective review, and having access to other files in the code base can be helpful, as the patch exists within this larger context. This extra context must be added judiciously, though, to prevent the context from overwhelming the change itself.

🕭 *Historical Information.* Code bases are constantly evolving, and all of these changes are recorded. These recordings, through version control histories, issue trackers, continuous integration logs, etc. provide access to historical information which can be used to suggest changes or provide hints based on common observed patterns. For example, these patterns could include previous code changes made by the same author, previous comments from the same reviewer (providing hints on the reviewer's commonly-held concerns), previous code changes similar to the one being reviewed (providing hints on concerns other reviewers have had for the changed code),

Table 1: A number of ways in which an LLM-based conversational agent could enhance the code review experience with (information from) search tools. *Information sources*: Code Base, Historical Information, Existing Tests, Documentation, Static Analysis, Vulnerability Analysis, Artifacts, Users. *Actions*: Describe PR, Detect Inconsistencies, Summarize Logs, Retrieve Information, Provide Suggestions.

Goals / Tasks	Sources of Information	Actions	Proposed Ideas
Finding defects			The agent retrieves and learns from previous bug fixes to detect and suggest repairs for any defects in the code, plus summarizes CI logs and build logs to identify breaking changes. It also learns from historical changes, to identify files that are often changed together, and suggests changes to dependencies if they are not updated.
Software testing			The agent detects inconsistencies between the code change and the corresponding tests, generates new tests when needed, and suggests changes to the existing ones. It also summarizes findings from executing the tests, and answers questions about them. The reviewer can ask the agent to generate a test that exercises specific lines of code by highlighting the code. Based on reviewer background, the agent can also answer questions about the testing framework and provide suggestions on the style/format of the test suite.
Code improvements - functional			The agent calls static analysis tools to detect null pointer exceptions or changes to data flow/control flow graphs, and vulnerability detection tools to detect security issues. It summarizes the findings from these tools. It retrieves information about alternative APIs and frameworks that can be used by querying web search tools, links to similar changes in the past, and suggests alternative implementations for the code change. Based on the reviewer's background (rather the lack thereof), the agent takes a more active role in identifying and alerting the reviewer of various issues that may be present in the code.
Code improvements - non-functional			The agent reads the code and documentation to learn the general style of the project, and applies these rules to the code change to ensure consistency. It also suggests comments and documentation updates given a patch. Moreover, the agent invokes program analysis tools to detect dead code, and removes it.
Updating other software artifacts			The agent detects inconsistencies between the code and other artifacts using historical repository information. Similarly, it retrieves relevant company policies to ensure none of them are being violated by the patch. The agent then alerts the reviewer if any inconsistencies are detected and suggests changes to address them.
Knowledge transfer			The agent retrieves and summarizes relevant API documentations for APIs present in the patch, personalized to the reviewer's background and expertise. It also has information about previous related change sets and other relevant parts of the code base, such that the reviewer has more context when reviewing the code change. It can help junior engineers to better understand the code base when reviewing the code change.
Prioritization			The agent identifies groups of similar changes (e.g., refactorings) and related changes (e.g., function definition and call sites), and presents the groups to the reviewer in a personalized way, ordered by familiarity with the change.
Mediation			The agent monitors the communication between the reviewer and the author, and suggests edits to language that can be perceived as toxic, pushback, etc.

and common code files that are often edited together (providing hints on whether a change is incomplete).

Existing Tests. The existing test suite can help ensure that changes made to the code do not break existing functionality. Test execution traces can also help guide the patch reviewer and patch writer towards new test cases that need to be added, or existing test cases that need to be updated. Exposing the dynamic outcome of these tests, specifically showing the tests relevant to a change and whether they continue to pass after the change, can further help the reviewer understand the risk associated with a code change.

Documentation. Documentation related to APIs being used in the code change can be useful if the reviewer is unfamiliar with them. Inspecting API documentation can also help validate whether they are being used appropriately.

Static Analysis. Warnings from static analysis tools can provide useful information and help identify issues with the code, such as null pointer exceptions, unexpected changes to data and control flow, and changes in project-relevant code quality metrics.

Vulnerability Analysis. Access to vulnerability datasets can be a useful for evaluating a change for commonalities with known vulnerabilities. Changes can also be evaluated against known vulnerability solutions to further reinforce reviewer feedback.

Other Artifacts. Software often has other supporting artifacts beyond code, including natural language specification documents, design documents, diagrams for various use cases, requirements documents, and policy documents related to privacy, the company goals, and other concerns. Source code often is intended to conform to these requirements, but the informal / unstructured nature of many of these kinds of documentation makes ensuring both adherence and consistency challenging. Having a system evaluate areas of support and contravention between a change and these documentation can help ensure the overall coherence of the change.

Patch Reviewer & Patch Writer. The patch reviewer's and patch writer's experience and preferences can be learned from the past interactions. This information can be useful for fetching relevant information that suits the needs of both stakeholders before and during code review. For example, a reviewer looking at code changes

on a file that they have never interacted with requires more support than a reviewer who has authored or contributed to the code file. Correspondingly, recommendations can also be made to the patch writer in advance of the reviewer actually looking at the change, giving them a chance to preemptively improve their submission.

3.2 Actions

Describe PR. A pull request (PR) often contains many different snippets of information, including code changes, the corresponding changes to the tests and documentation, commit messages, etc. Having the agent consume all this information to briefly summarize the changes in the PR provides a starting point for the reviewer.

Detect Inconsistencies. Code is rarely treated as an independent entity and often has dependencies to other code files, test files, documentation, specifications, etc. These dependencies can be learned from historical information. The agent should then be able to consume this information to determine if there are inconsistencies between a given code change and a commit message, a code change and the code comments, other documentation, and other artifacts.

Summarize Logs. Many tools can be used to perform various checks for code, such as static analysis or program analysis tools, logs from CI, security checks, performance measures, test metrics, and so on. The agent should consume these and extract the most relevant parts. For example, identifying new tests that do not add any new coverage, or alerting the reviewer of a security threat.

Retrieve Relevant Information. The supporting artifacts for a code change can be lengthy and difficult to consume in their raw format. For example, if the reviewer is not familiar with the library being used, going through the entire documentation can be tedious. However, having the agent summarize the most relevant parts of the documentation based on the reviewer's background can save time and effort. Additionally, having the agent summarizing other artifacts such as the design or requirements documents, company policies or relevant code and tests can help provide relevant context without overwhelming the reviewer.

Provide Suggestions. Having a checklist of things to look for during code review is beneficial; one might even automatically infer this checklist based on historical information (for example, style guidelines, or updating test suite), and other guidelines specific to the project. The agent can then provide suggestions based on the derived checklist to both the patch writer (to ensure that the PR is complete) and the reviewer (to ensure they do not miss anything).

4 The Road Ahead

In this paper, we have proposed a future vision for how the code review process can be positively supplemented by LLM-based approaches. While Section 2 may make it seem like we are proposing an overwhelmingly disparate set of information sources to be brought to bear on this problem, there are important commonalities across all of these information sources that enables progress to be made in a stepwise fashion. Several primary challenges face this work, although each can be tackled independently.

Natural Interaction. Each of the disparate search tools we propose to augment code review with have their unique interaction mechanisms. One research avenue for this work is to investigate whether a consistent and natural interaction layer can be applied on top of these search tools to reduce developer friction for invoking and interacting with these tools. Fortunately, the output from most search tools is itself text, upon which LLMs have demonstrated strengths. This is augmented by the context of the change, the patch, also being fully text-based, further easing input to the LLM.

Summarization and Distillation. Each of the search tools will return results in their own formats that must be filtered and tailored to both the specific patch under review, as well as the individual needs of the patch reviewer. Once again, the importance of this task further leans on the strength of LLMs to perform these kinds of tasks on structured text. The key challenge in this space is not in the summarization itself, but in finding useful facts that can help augment code review without overwhelming the reviewer with facts that do not improve the quality or speed of their review.

Trust & Explainability. A fundamental challenge facing this approach is one of trust. This can be thought most simply in terms of precision and recall. In terms of precision, the wealth of information search tools can surface about a change can easily overwhelm the developer. This suggests that considerable effort will be made to elide data that are not useful (e.g., the results should have high precision with few false positives). But this puts the approach in tension with recall. If augmenting code reviews with external information proves useful, developers will want to be able to trust that the tool will not hide information that could have improved their code review (e.g., the results should have high recall with few false negatives). Managing the trust of the patch-writer and patch-reviewer is likely to be more challenging than the technical aspects of interacting with and summarizing the search tool results.

Summary. Code review plays a longstanding multi-faceted role in modern software development. By its very nature, code review is a time-intensive human process that takes place in a complicated technical domain. In this work we have proposed deploying LLMs to augment the code review process without specific developer-provided training and tuning in a way that can surface this additional information, enabling reviewer time to be more effectively spent. Naturally, many challenges remain for this vision: can existing public models effectively fulfill this role, or do teams need to train their own? Can locally-hosted models be directly augmented with the necessary context, or will software development stacks gain yet another expensive service they need to pay for? How will the continually-evolving software development ecosystem hinder the kinds of feedback LLMs will propose within the code review? All of these important questions remain to be answered, but the promise itself is clear: enabling better code review decisions by extending discussions far beyond the patch itself.

Acknowledgements

The idea for this paper emerged at the 2024 Dagstuhl Seminar on Code Search, where we were inspired by discussions with Alexander Neubeck, Boris Bokowski, Bowen Xu, Christoph Treude, Ciera Jaspan, Crista Lopes, Dongsun Kim, Georgios Gousios, Jan Van den Bussche, Jens Krinke, José Cambronero, Julia Lawall, Jürgen Cito, Katie Stolee, Luca Di Grazia, Michael Pradel, Miryung Kim, Rijnard van Tonder, Satish Chandra, Svetlana Zemlyanskaya, Tobias Kiecker, and Tobias Welp. The authors also thank Vincent Hellendoorn for feedback on an earlier draft.

References

[1] 2024. Google Q3 earnings call: CEO's remarks. https://blog.google/inside-google/message-ceo/alphabet-earnings-q3-2024.

[2] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering (EMSE)* 21 (2016), 1143–1191.

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *International Conference on Software Engineering (ICSE)*. 712–721.

[4] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2016. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering (EMSE)* 21 (2016), 932–959.

[5] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. 2020. The silent helper: the impact of continuous integration on code reviews. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 423–434.

[6] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey. 2013. The MSR Cookbook: Mining a Decade of Research. In *International Conference on Mining Software Repositories (MSR)*. 343–352.

[7] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *International Conference on Software Engineering (ICSE)*. 672–681.

[8] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. 2019. Tool choice matters: JavaScript quality assurance tools and usage outcomes in GitHub projects. In *International Conference on Software Engineering (ICSE)*. 476–487.

[9] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering (EMSE)* 21 (2016), 2146–2189.

[10] Emerson Murphy-Hill, Ciera Jaspan, Carolyn Egelman, and Lan Cheng. 2022. The pushback effects of race, ethnicity, gender, and age in code review. *Commun. ACM* 65, 3 (2022), 52–57.

[11] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. 2018. What makes a code change easier to review: an empirical investigation on code change reviewability. In *International Conference on the Foundations of Software Engineering (FSE)*. 201–212.

[12] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, Vol. 1. 598–608.

[13] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 68539–68551. https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf

[14] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. 2020. Why can't Johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In *Symposium on Usable Privacy and Security (SOUPS)*. 221–238.

[15] Mairieli Wessel, Bruno Mendes De Souza, Igor Steinmacher, Igor S Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A Gerosa. 2018. The power of bots: Characterizing and understanding bots in OSS projects. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–19.

[16] Mairieli Wessel, Igor Wiese, Igor Steinmacher, and Marco Aurelio Gerosa. 2021. Don't disturb me: Challenges of interacting with software bots on open source software projects. *Proceedings of the ACM on Human-Computer Interaction* 5, CSCW2 (2021), 1–21.

[17] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. A conceptual replication of continuous integration pain points in the context of Travis CI. In *International Conference on the Foundations of Software Engineering (FSE)*. 647–658.

[18] Haopeng Zhang, Philip S. Yu, and Jiawei Zhang. 2024. A Systematic Survey of Text Summarization: From Statistical Methods to Large Language Models. arXiv:2406.11289 [cs.CL] https://arxiv.org/abs/2406.11289