Semantic Dependency Visualization on Code Review: A Comparative Analysis

Vinícius Resende Barbosa
Informatics Center, Federal University of Pernambuco
Recife, Brazil
vrb@cin.ufpe.br

Ykaro dos Santos Albuquerque
Informatics Center, Federal University of Pernambuco
Recife, Brazil
ysa@cin.ufpe.br

Paulo Borba
Informatics Center, Federal University of Pernambuco
Recife, Brazil
phmb@cin.ufpe.br

Abstract

In collaborative software development contexts, developers work in parallel, integrating code changes into a shared repository. Before integration, the changes are often subjected to peer review, when other developers analyze the code, offer suggestions, and identify potential issues. Reviewing is often challenging because it is difficult to semantically understand how changes fit together, and predict the impact they might have on system behavior. The challenge is increasing with the rise of AI generated code, and the often greater volume, complexity and team cognitive debt of the code to be reviewed. It's particularly difficult to detect subtle and harmful semantic dependencies between the changes, which are not easily detected with traditional review tools. To help reviewers, this paper introduces InView, a tool that extends GitHub's pull request interface with information about semantic dependencies between the changes to be integrated. Understanding these dependencies during review might help developers detect harmful and unexpected interference between the changes, which lead to hard to detect and resolve bugs. InView applies multiple static analysis techniques to detect semantic code dependencies, and presents the results through an integrated code diff and interactive dependency visualizations. We evaluate InView through interviews and a controlled experiment with developers performing code review tasks on pull requests using both InView and GitHub. Results show that InView improves the detection of semantic dependencies but increases code review time, while being perceived by developers as more effective for understanding code changes.

Keywords

Code review, Semantic dependencies, Pull request, Code visualization, Static analysis

1 Introduction

In collaborative environments, developers work in parallel, implementing new features, fixing bugs, and making other kinds of changes to the code base. The changes are then integrated into a shared repository. However, before integration, projects that value code quality often subject the changes to verification practices. One of these practices is peer code review, in which other developers analyze the modifications to be integrated in order to ensure that the implementation conforms to coding and quality standards, offering suggestions and identifying potential problems [1, 5, 19].

During the review process, to detect bugs and prevent them from being introduced into the system, reviewers need to understand the effects behind each change— and how they relate to other changes— while also keeping in mind all the changes made by other developers [15]. This task can be costly, as it is difficult to semantically understand how various changes relate and predict the impact they may have on the system's behavior, especially when many changes are submitted in a short time window, making code review a non-trivial process [10, 11].

The code review challenges are further increased by growing adoption of LLM (large language models) agents for code generation and the creation of Pull Requests (PRs) directly in project repositories [8, 9]. While these tools might increase developer productivity, they also significantly increase the volume and review complexity of integrated changes. This can lead developers to approve and integrate code whose behavior they do not fully understand [13], and increase cognitive debt. Consequently, these tools might introduce bugs due to subtle and harmful semantic dependencies between the integrated changes, potentially contributed from more than one agent. Such bugs can be difficult for reviewers to detect.

Motivated by the need to reduce these problems, this work introduces InView, a tool built on the GitHub PR interface with the goal of assisting reviewers in understanding semantic dependencies—interference between the modifications to be integrated that can alter the expected behavior of the system. Understanding these dependencies during code review can be difficult, so the introduction of an assistance tool may help developers better predict possible negative impacts on the final behavior of the system, through the identification of unwanted interactions between the changes, which can cause bugs that are difficult to detect and resolve.

InView employs static analysis techniques [2, 6, 7] to detect semantic dependencies between parts of the code modified by different developers or agents, such as the data flow of an assignment modified by one developer to an expression changed by another, and presents the results through three distinguishing features: Three-way Diff, Dependency Log, and Dependency Graph. The first is a diff that integrates the changes made by the PR with the changes already integrated in the receiving branch, showing the expected final version of the code after the merge. This diff contrasts with the one shown by GitHub, which includes and highlights only the PR contributions, hiding any changes already integrated into the repository, but which are not present in the PR branch. The second feature is a list reporting the dependencies found by the analyses and the third feature is a graph with the semantic dependencies between the modified parts of the code; reviewers can interact with the graph to navigate the diff and analyze the impact of each change made by a developer with the changes made by others.

To assess the impact of InView on the code review process, we conduct a controlled experiment with 12 developers performing code review tasks on two PRs, each with a different tool (GitHub

and InView). Each participant was subsequently interviewed so that we could understand their experience using InView.

The experiment was designed to investigate the tool's effects on code review, with respect to dependency detection, review effort, and reviewers' perceptions; overall quality of review decisions is outside our scope. We seek to answer the following research questions: (RQ1) Does InView improve detection of semantic dependencies during code review? (RQ2) Does InView affect time spent on review? (RQ3) How do developers perceive InView's impact on code review?

The results show that InView improves the detection of semantic dependencies but increases code review time, while being perceived by developers as more effective for understanding code changes. The main contributions of the paper are the following: the design and implementation of InView; the evaluation of InView through semi-structured interviews and a controlled experiment.

2 Motivating Examples

To illustrate the inherent complexity of the code integration process, this section details two scenarios that explore different workflows using the GitHub PR interface. These scenarios highlight the potential semantic interferences that can emerge from concurrent changes, demonstrating situations where review based solely on textual differences proves insufficient to guarantee the behavioral integrity of the system after the merge.

2.1 First Example: Pull Requests Submitted in Sequence

Consider the code shown in Figure 1 as the base version. This code stores an input string in the variable `input` and uses that string to initialize a new variable `t` of the `Text` class. After that, it calls the `removeComments()` method to remove comments from the content of `t` (parts of the string between "//" and a line break, as in `.tex` documents).

```
3  public class Main {
4    public static void main(String[] args){
5      String input = args[0];
6      Text t = new Text(input);
7      t.removeComments();
8      System.out.println(t.getText());
9    }
10 }
```

Figure 1: Base Code of Motivating Example

We then analyze a situation where two developers— Previous and Current— are working from the same version of the `Main` class illustrated in Figure 1. Previous adds a call to the method `normalizeWhiteSpace()` (responsible for removing duplicate whitespace in strings stored in `Text` class objects) in the region before the `removeComments()` call, and opens a PR with his additions. Meanwhile, Current, who is still working with the version without Previous's changes, adds a call to the `removeDuplicateWords()` method (responsible for removing duplicate words in strings stored in `Text` class objects) in the region after the `removeComments()` call, tests the resulting code and verifies that the system behaves as expected. Current then opens a PR with his version of the code to the remote repository. Only after that[1] a third developer accepts Previous's PR and proceeds to review Current's PR. Since the Previous and Current changes were in different areas of the code,[2] no textual conflict is reported by GitHub. In Figure 2 we see what is displayed to the reviewer in the Files Changed tab of the GitHub PR screen, so that they can assess whether the Current's changes should be accepted or not.

```
3  public class Main {
4    public static void main(String[] args){
5      String input = args[0];
6      Text t = new Text(input);
7      t.removeComments();
8      t.removeDuplicateWords();
9      System.out.println(t.getText());
10   }
11 }
```

Figure 2: Diff Displayed by the Files Changed Tab From GitHub on the First Example

Note that in the code shown to the reviewer (illustrated in Figure 2), Previous's addition is not displayed, despite having been recently integrated into the remote repository. This happens because GitHub presents the reviewer with a diff between the base version of the code (the commit from which Current created its branch) and Current's version [12], thus omitting the changes made by Previous. In this way, the reviewer is not informed of Previous's changes, nor consequently knows what implications they may have on the final behavior of the system. Therefore, they accept the PR only based on the analysis of one developer's code, potentially introducing bugs into the system because they lack visibility of how the code in this PR may affect or be affected by code from other developers integrated after Current opened their PR.

After merging the PR from Current, besides the change in Figure 2, the resulting code also has a call to `normalizeWhiteSpace()`, added by Previous in the region before the `removeComments()` call (as shown in Figure 3). As a consequence, the resulting code presents undesirable behavior. For example, in a case where `main` is executed with the input string "`the␣the␣dog`", the method call inserted by Current removes the second "`the`" from the string after duplicate spaces were removed. The resulting string is then "`the␣dog`", which has duplicated whitespace. This breaks the intent of Previous, who had inserted a call to the `normalizeWhiteSpace()` method precisely to prevent the system from printing messages with this issue. This scenario occurs because Previous's changes were not seen by the reviewer when accepting Current's PR, nor did he remember or were aware of what had been integrated previously. Thus, the reviewer assumed that Current's additions made sense with the existing code, when in fact they broke the intent of Previous's

[1] If it were before, Current could have pulled and merged Previous changes
[2] One before and other after the call to `removeComments()`.

changes. The PR was accepted because, with the omission of Previous's additions in GitHub's interface, the reviewer did not realize that the changes from the two developers were incompatible. In practice, the situation is even riskier because the repository may contain changes from several developers who contributed other changes, omitting even more information from the reviewer.

2.2 Second Example: Current Integrates Previous's Changes

Now consider a similar situation except that Previous's PR is accepted before Current opens theirs. Problems may still arise. Suppose Current updates their local repository with the latest changes from remote but does not carefully analyze them. Semantic dependencies between the pulled code and their modifications may go unnoticed. In this scenario, Current incorporates Previous's changes before opening their PR. As shown in Figure 3, the diff presented to the reviewer includes Previous's addition, but it is not highlighted because it already exists in the main branch. This can obscure harmful interactions between the changes, creating a false sense of confidence during review.

```
3  public class Main {
4    public static void main(String[] args){
5      String input = args[0];
6      Text t = new Text(input);
7      t.normalizeWhiteSpace();
8      t.removeComments();
9      t.removeDuplicateWords();
10     System.out.println(t.getText());
11   }
12 }
```

Figure 3: Diff Displayed by the Files Changed Tab From GitHub on the Second Example

Such issues can degrade code quality, as defects introduced by unnoticed dependencies may persist even in projects with strong testing [14, 21, 22] or review practices [5, 15]. Detecting and fixing them later can be hard and require rework, impact productivity and system reliability. It is also important to note that this limitation is not exclusive to GitHub. Other tools, including platforms such as IDE extensions, also prioritize textual diffs, leading to fragmented views that require manual effort to track interactions across changes. To address this, InView provides a visualization of the merged code that highlights all changes, including those already integrated. It also exposes dependencies between them— for example, showing how Current's changes may affect Previous's— aiming to help reviewers detect unintended interference. While such issues may be easier to identify in small examples or when context is fresh, they become significantly harder in larger pull requests involving multiple developers and limited review time.

3 InView

To address the problems illustrated in the previous section, InView provides a PR reviewer with information that makes them aware of changes integrated to the repository *after* the PR branch was created, highlighting potentially harmful dependencies between these changes and the PR changes. In this way, the reviewer is better positioned to decide whether the PR should be accepted. InView does that through three features that display extra information for the code reviewer: Three-Way Diff, Dependency Log, and Dependency Graph. In the current implementation, these features are available in a "Dependencies" tab added to the GitHub PR screen. In the following we explain these three features in more detail.

3.1 Features

3.1.1 Three-Way Diff. The first feature consists of a diff between the base version of the code (the commit from which the developer who opened the PR started working) and the PR merge version. This includes changes to the base that are already in the repository and those being added by the PR, as illustrated in Figure 4, which shows the Three-Way Diff generated from the motivating example.

```
3  public class Main {
4    public static void main(String[] args){
5      String input = args[0];
6      Text t = new Text(input);
7      t.normalizeWhiteSpace();
8      t.removeComments();
9      t.removeDuplicateWords();
10     System.out.println(t.getText());
11   }
12 }
```

Figure 4: InView's Three-Way Diff of Motivating Example

Unlike the code presented in Figures 2 and 3, here the change by Previous is included. Furthermore, to indicate to the reviewer who made each modification, additions previously made by Previous are highlighted in pink, while those included by Current in the PR are highlighted in green. In this case, Previous represents only one developer, but if several other PRs had already been accepted before the analysis of Current's PR, the changes of all would appear in pink.

Figure 5: Partial Dependency Log of Motivating Example

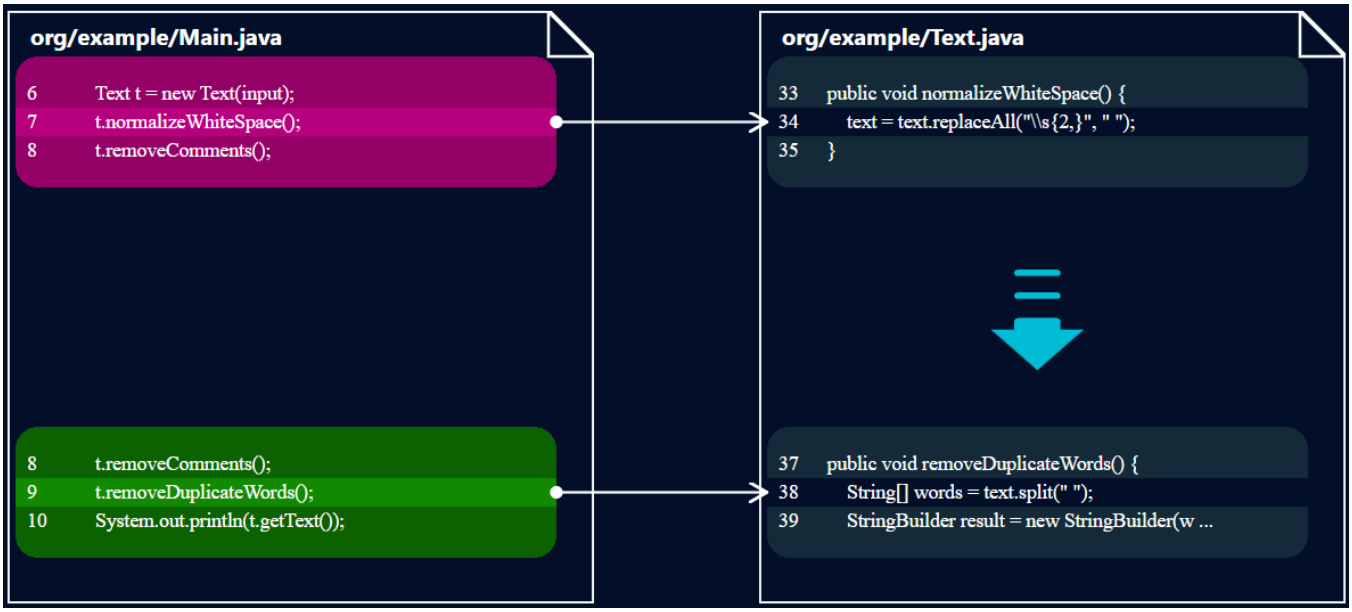

Figure 6: Dependency Graph of Motivating Example

3.1.2 Dependency Log. In addition to this richer diff, InView brings the Dependency Log. Through static analysis algorithms [2, 6, 7] applied to the merge version of the code, with information on which lines were modified by Previous and Current, InView identifies code dependencies between Previous and Current changes; these can potentially interfere with each other, as they were independently carried out, completely unaware of each other. These dependencies are presented to the reviewer in an interactive panel in the interface. In Figure 5 we can visualize the dependency log shown in the motivating example scenario, where the analyzes identified seven dependencies (only two are shown for space reasons). The label at the top of each item indicates that they are Data Flow (DF) dependencies, signaling that the changes from one developer involve an assignment to a variable that is read by the changes from the other developer. In the caption of each item, InView shows the file name and the line where the dependency begins, followed by the file name and line where the dependency ends. The difference between the line numbers in the figure and the lines in the examples is due to the report's operation: here the reported lines correspond to the final location of each method call chain, that is, in the first item of the report, line 34 is contained in the body of the method introduced by Previous while line 38 belongs to the method of Current. These are the dependencies that are indicated by the tool as a suggestion of what the reviewer should pay attention to during the review.

DF dependencies are not the only ones that can be reported by InView. The tool also covers Overriding Assignment (OA) dependencies, which occur when an assignment made by a developer is overridden by another simultaneous assignment, silently canceling its effect on the final program state. The combination of these two analyses allows InView to capture a more comprehensive view of potential interference, exposing risks that an isolated or purely textual analysis might miss.

3.1.3 Dependency Graph. To analyze one of the dependencies in the log, a code reviewer can click on one of the items presented, accessing a graph that represents the dependency to be analyzed, as illustrated in Figure 6, which displays the dependency graph visualized by the reviewer when clicking on the first dependency in the report in Figure 5. The nodes at the left represent the method calls in `Main` added by Previous (pink node) and Current (green node), signaling a potentially harmful dependency between these changes. Each node shows the line involved in the dependency in its center, with two lines of context, one above and one below. The colors of the nodes represent the author of each change, following the same color pattern shown in Figure 4. The nodes are grouped into files. Since the changes were in `Main.java`, we see those nodes grouped inside the `Main` file icon. The bodies of the called methods—which are in `Text.java`— are grouped on the right. Given that both changes correspond to the addition of method calls, the nodes have a white arrow next to them, called a "call" arrow, indicating a call dependency between each method call and its declaration. Finally, there is a DF arrow (the blue arrow), that goes from the node called by "Previous" to the node called by "Current", showing the code reviewer that there is a DF dependency between these nodes; `text` is assigned on the top node and read in the bottom node. The graph then represents the structure of the detected dependency, what changes caused it, and what parts of the code need to be analyzed to determine if the dependency is problematic. PRs with a long method call chain would not alter the graph structure, as only the first and last lines of the sequence of calls are represented. The reviewer can also click on one of the central lines of the nodes in the dependency graph to be redirected to the diff line indicated by the clicked node. In this example, the reviewer would pay attention to the interaction between lines 34 and 38 of Text (see Figure 7), observing a potential interference. They would click on one of the

nodes at the left column to understand the context and reflect on whether the interference was intentional or an accident that needs to be fixed.

```
33  public void normalizeWhiteSpace() {
34    text = text.replaceAll("\\s{2,}", "␣");
35  }
36
37  public void removeDuplicateWords() {
38    String[] words = text.split("␣");
39    ...
40  }
```

Figure 7: Methods Called by Previous and Current

3.2 General Architecture

To support these funcionalities, InView is composed of three main components: a GitHub App, a database server, and a browser extension.

3.2.1 GitHub App. The GitHub App enables automated access to repository data through the GitHub API and webhook. When installed in a repository, it monitors pull requests and retrieves code from the involved branches. The same component executes static analyses (Overriding Assignment and Data Flow) [2, 6, 7] on PR changes and send the results to the database. Each result includes dependency type, affected elements (classes, methods, lines), and authorship of changes. The app is analysis-agnostic: any technique producing results in the expected format can be integrated.

3.2.2 Browser Extension. The browser extension presents the results directly in the GitHub interface through a new "Dependencies" tab. It renders the three-way diff, dependency log and dependency graph. This approach was chosen to integrate with the standard GitHub workflow, avoiding the need for external tools or IDE plugins. It also allows direct comparison with the default interface. To support complete analysis, the extension can include related files not modified in the PR but involved in dependencies.

3.2.3 Database Server. The database server stores analysis results in a MongoDB database and exposes them through an API consumed by the browser extension. Persisting results avoids re-running analyses on each access and improves performance. The separation of this component also enhances modularity, allowing independent evolution of the app and the visualization layer.

4 Methodology

To evaluate the impact of InView on the code review process, we combine a controlled experiment with semi-structured interviews to understand the following aspects compared to the default GitHub interface: developers' ability to detect semantic dependencies, review time, and overall user perception. The combination of both quantitative and qualitative methods enables methodological triangulation, strengthening the validity of our conclusions. The study was reviewed and approved by the relevant institutional ethics committee prior to data collection.

4.1 Experimental Design

We conducted a controlled, within-subject study in which participants performed code review tasks using two tools: InView and GitHub. To mitigate learning effects, we employed a balanced incomplete crossover design, where each participant reviewed two different PRs, each using a different tool. Counterbalancing also mitigates order effects by ensuring that neither tools consistently benefits from being used first or second. The assignment of tools and PRs was counterbalanced across participants to ensure that each tool and PR combination was equally represented. This design prevents participants from reviewing the same PR twice, reducing bias due to familiarity and contamination between treatments, while still enabling direct comparison between tools.

4.2 Participants

The study involved 12 participants from 8 teams across 7 companies recruited through an open call to both industry contacts and university mailing lists targeting software developers with experience in code review. Participation was voluntary, and no incentives were provided. Participants had different levels of experience in software development— varying from 1 to 11 years— and familiarity with code review, many practicing it weekly. Prior experience with GitHub was common, while none had prior exposure to InView.

4.3 Materials

4.3.1 Pull Requests. We designed two synthetic pull requests[3] based on real-world examples from an existing dataset of merge scenarios [23]. These PRs were adapted and transformed into smaller, self-contained repositories and files to reduce the effort required for participants to understand the system context, thereby minimizing unnecessary cognitive load and allowing them to focus on the review task in reduced time. The first PR is more complex, containing two relevant semantic dependencies across seven modified files, whereas the second PR is simpler, involving a singe relevant dependency across two modified files. Each PR was constructed to include semantic dependencies between the integrated changes, as one of our goals is to understand whether InView can help with this kind of situation. In PRs with no semantic dependencies, InView reports no relationships beyond those in GitHub, making the review equivalent to using the "Files Changed" tab interface. Scenarios with no dependencies were then not included, as they would not provide meaningful evidence for the study.

4.3.2 Tools. Participants performed reviews using two tools: InView and the standard GitHub interface, accessed through its web interface as available at the time of the study (march-april 2026).

4.4 Experiment Procedure

Each experiment session lasted from 50 to 90 minutes and consisted of four phases illustrated in Figure 8. First, participants answered a short questionnaire capturing demographic information and prior experience. Next, participants performed two code review tasks, each using a different tool and a different PR. The order of tool-PR combinations was counterbalanced across participants to mitigate learning and order effects. Using a different PR for each task also

[3] Full PRs are available on the online appendix (see Section Artifact Availability)

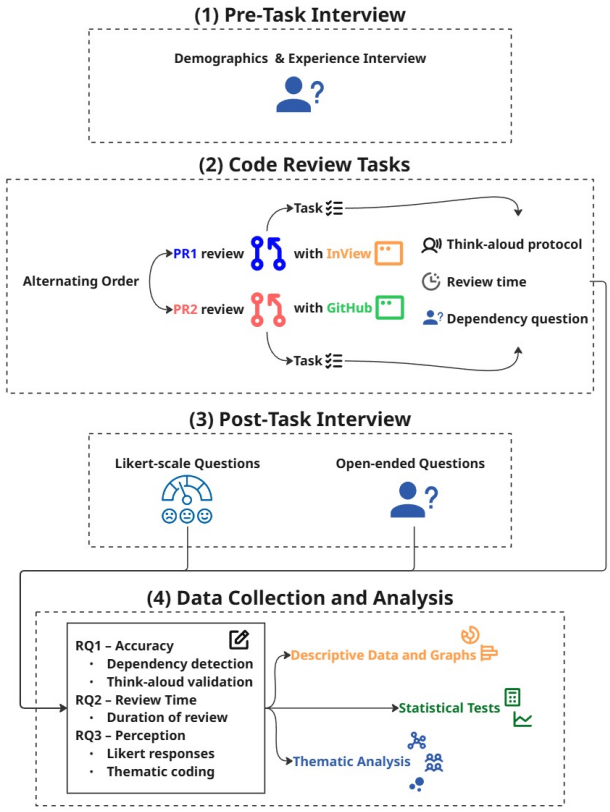

Figure 8: Experiment Procedure Pipeline

prevented familiarity with a review task from influencing performance under the other treatment. Before using InView, participants received a brief tutorial on its interface. After that, participants answered a set of semi-structured interview questions about their experience with each tool[4]. Finally, the data collected across each phase was compiled and analyzed through descriptive data and graphs, statistical tests and thematic analysis. During the review tasks, participants were asked to answer specific questions about the code, designed to assess their understanding of semantic dependencies, while following the think-aloud protocol.

4.5 Data Collection and Measures

4.5.1 RQ1 - Dependency Detection. To assess dependency detection, participants were asked task-specific questions about the code whose correctness depended on identifying semantic dependencies during code review. We then evaluated whether they correctly incorporated semantic dependencies into their reasoning. Rather than focusing solely on the final answer, we considered an answer correct if the participant demonstrated an understanding of the relevant dependency and its effects on the program's logic. This assessment was performed independently by two researchers who reviewed the recorded sessions, including participants' think-aloud explanations and interaction traces. Incorrect answers due to unrelated factors, such as arithmetic mistakes, were not penalized, provided that the participant's reasoning correctly reflected the effect of the dependencies. Disagreements were resolved through discussion and consensus. For example, in one task involving an overriding assignment, participants were considered correct if they recognized that a variable's value was modified by a later operation of another developer, even if they miscalculated the final value yielded by the method. This approach allows us to isolate the ability to detect and reason about dependencies, which is the primary focus of this research question. Accordingly, our evaluation focuses on dependency awareness rather than the overall quality or correctness of participants' review decisions.

4.5.2 RQ2 - Review Time. RQ2 was assessed based on the total time required to complete the review of each PR using each tool. The time of each review was measured using session recordings, considering the interval between the end of the task context explanation (when the participant assumed control of the review) and the moment the final answer to the review question was provided. This setup enables direct comparison of review time across tools as each participant reviewed different PRs under a balanced experimental design.

4.5.3 RQ3 - Perception. Participants rated each tool using Likert-scale questions assessing usefulness, usability and confidence, as well as giving justification and other perceptions for each topic. Additionally, qualitative feedback was collected through open-ended interview questions.

4.6 Data Analysis

4.6.1 Quantitative Analysis. We analyzed the quantitative data by comparing dependency detection accuracy and time metrics across tools. Descriptive statistics were used to assess that.

4.6.2 Qualitative Analysis. We analyzed qualitative data using thematic analysis. An initial codebook was derived from the research questions and iteratively extended through inductive coding. Two researchers independently coded a subset of the data and resolved disagreements about the meaning of the codes through discussion, following a negotiated agreement approach. To enable comparison across tools, each coded segment was assigned a polarity (positive, neutral or negative), reflecting the participant's evaluation. We then analyzed the distribution of polarity within each theme to identify differences in perception between tools.

5 Results and Discussion

This section presents the results of the empirical evaluation of InView in comparison with the default GitHub interface. We structure the analysis according to our three research questions: (RQ1) the detection of semantic dependencies during code review, (RQ2) review time, and (RQ3) developers' perceptions and experiences with each tool. For each research question, we present the quantitative results and discuss them in light of the qualitative evidence obtained through semi-structured interviews and thematic analysis.

5.1 RQ1: Dependency Detection Accuracy

RQ1 investigates whether InView improves developers' ability to detect semantic dependencies during code review. To evaluate this,

[4]See Section Artifact Availability to access all interview questions

participants were asked to perform review tasks on two pull requests, each containing relevant dependencies that required cross-file reasoning. For each task, we assessed whether participants correctly identified these dependencies based on their explanations during the think-aloud process.

We report the results separately for each PR due to differences in the number and nature of dependencies involved. For PR1, which contains two key dependencies, we analyze the number of dependencies identified by each participant. For PR2, we assess whether participants correctly incorporated the dependency into their reasoning. We then compare the detection capability across tools and discuss the observed patterns.

5.1.1 Results for PR1. For PR1, which contains two relevant dependencies, we evaluated participants based on the number (0, 1 or 2) of correctly identified dependencies. Participants using InView were more likely to identify at least one dependency (66.7%, 4/6) compared to those using GitHub (33.3%, 2/6). Conversely, failure to detect any dependency was more frequent with GitHub (66.7%, 4/6) than with InView (33.3%, 2/6). However, complete detection of both dependencies was rare for both tools, occurring only once with GitHub. The isolated instance of complete detection is likely due to a particularly meticulous manual inspection strategy or a fortuitous concentration of effort on the specific file interactions where the dependencies occurred, as the participant's background in both development and code review did not stand out from the sample median. This case represents a unique occurrence that was not replicated in any other review throughout the study, suggesting it is an isolated instance unrelated to the inherent features of the GitHub interface. When considering an ordinal score (0–2), InView achieved a higher mean (0.67) and median (1) compared to GitHub ($mean = 0.50$, $median = 0$), indicating that participants using InView more frequently reached at least a partial understanding of the dependencies. While these results suggest that InView improves dependency awareness, they also indicate that providing information does not automatically guarantee the complete identification of all interferences in more complex contexts.

5.1.2 Results for PR2. For PR2, which involved a single key dependency, we evaluated whether participants correctly identified and incorporated the dependency into their reasoning. Participants using InView achieved a detection rate of 100% (6/6), whereas those using GitHub achieved a detection rate of 66.6% (4/6). Notably, no participant failed to identify the dependency when using InView, while two participants (33.3%) failed to detect it when using GitHub. These results indicate that InView consistently supports the identification of dependencies in simpler scenarios, while the default GitHub interface was associated with a higher rate of missed dependencies. The results for PR2 are consistent with the ones obtained for PR1, but showing stronger advantage towards InView, likely because PR2 involve less complex dependencies that are more easily checked by the reviewer following InView's report.

5.2 RQ2: Review Time

We assess RQ2 in terms of the time required to complete the review of each PR using each tool. Due to the balanced incomplete crossover design, each PR was reviewed by different participants under each tool. As a result, comparisons between tools for a given PR involve independent samples. Therefore, we applied the Mann-Whitney U test to assess differences in review time distributions.

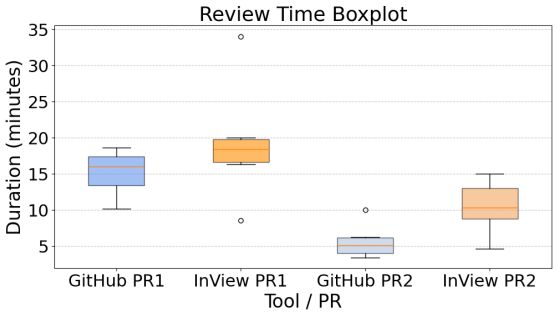

Figure 9: Time Distribution for both PRs (GitHub vs InView)

5.2.1 PR1 Results. For PR1, as observed in the left half of Figure 9, participants using InView required a mean time of 19.3 minutes ($median = 18.4$), compared to 15.2 minutes ($median = 16$) when using GitHub, corresponding to a decrease of approximately 21%. A two-tailed Mann-Whitney U test indicated that this difference is not statistically significant ($U = 10.0, p = 0.23, \alpha = 0.05$). However, the effect size was moderate (Cliff's $\delta = 0.44$), suggesting a tendency for longer review times when using InView.

5.2.2 PR2 Results. For PR2, participants using InView required a mean time of 10.4 minutes ($median = 10.3$), whereas those using GitHub completed the task in 5.6 minutes on average ($median = 5.1$), representing a decrease of approximately 46%. The distribution of review times is shown in the right half of Figure 9. A two-tailed Mann-Whitney U test revealed a statistically significant difference between tools ($U = 5.0, p = 0.04, \alpha = 0.05$). The effect size was large (Cliff's $\delta = 0.72$), indicating that review times were consistently higher when using InView. The lower average review times for PR2 can be explained by its lower complexity compared to PR1. PR1 includes more relevant dependencies and files, requiring broader exploration and more extensive reasoning. In contrast, PR2 contains only one dependency across two files, leading to a more focused analysis. The reduced scope likely lowers the cognitive effort, resulting in faster reviews for PR2 and reflecting the intended difference in task complexity between the two PRs.

5.3 RQ3: Reviewer Perception

To assess developers' perceptions of the tools, we administered a set of post-task questions[5] combining Likert-scale and open-ended responses. The Likert questions evaluated different aspects of tool usage, including perceived helpfulness for code review (Q1), ease of use (Q2), intuitiveness (Q3), understanding of the graph visualization (Q4, for InView only) and ease of navigation across code elements (Q5). In addition, we collected qualitative feedback through open-ended questions regarding participants' confidence in their review decisions (Q6), their willingness to adopt the tool in their

[5]All questions are detailed on the online appendix (see Section Artifact Availability).

daily workflow (Q7), and suggestions for improvement (Q8). Each Likert response was accompanied by a justification, which was analyzed qualitatively to better understand the reasoning behind participants' ratings.

5.3.1 Quantitative Perception (Likert). Figure 10 presents the distribution of responses to the Likert-scale questions for both tools.

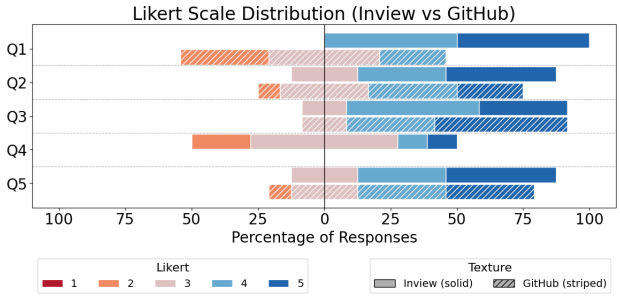

Figure 10: Likert Scores Distribution (InView vs GitHub)

Based on these distributions, we can observe the following about: **Helpfulness (Q1):** Participants rated InView as significantly more helpful for code review than GitHub. InView achieved a mean score of 4.5 (*median* = 4.5), compared to 2.9 (*median* = 3) for GitHub; **Usability and Navigation (Q2, Q5):** In terms of ease of use and navigation, both tools received generally positive evaluations, though InView obtained slightly higher mean scores than GitHub. This suggests that participants were able to effectively use and navigate InView despite its additional features; **Intuitiveness (Q3):** The perceived intuitiveness of the tools was comparable, with mean scores of 4.2 for InView and 4.3 for GitHub. This indicates that participants did not perceive InView as significantly less intuitive than the default interface; **Graph Understanding (Q4):** Understanding the graph visualization in InView presented more variability. Among participants who used the graph, the responses ranged from difficult understanding to complete comprehension, with a mean score of 3.1 (median = 3). This suggests that while some participants were able to effectively interpret the graph, others experienced challenges, particularly in more complex scenarios.

Overall, these results indicate that InView is perceived as helpful and usable, but its graph-based representation introduces additional cognitive demands. Such demands are expected to be reduced over time, as soon as users become familiar with InView features.[6] This additional cognitive demand finding aligns with the findings from RQ2, where increased review times were observed with InView; the increase, however, is likely due not only to the complexity of the graph-based representation but also to the deeper code investigation triggered by InView's report of dependencies.

5.3.2 Qualitative Insights (Thematic Analysis). To further understand developers' perceptions and experiences with the tools, we conducted a thematic analysis of the qualitative data collected from open-ended questions (Q6-Q8) and the justifications provided for the Likert-scale responses. The analysis followed an iterative coding process, combining deductive codes derived from the research questions with inductively generated codes emerging from the data.

Initially, two researchers independently coded a subset of the responses using a preliminary codebook. New codes were added as needed and iteratively refined through discussion until consensus was reached. The finalized codes were organized into higher-level themes representing key aspects of developers' experiences, such as understanding, usability, and cognitive effort. To enable comparison between tools, each code was also assigned a sentiment polarity (positive, neutral or negative), reflecting whether the experience described was beneficial, indifferent or detrimental.

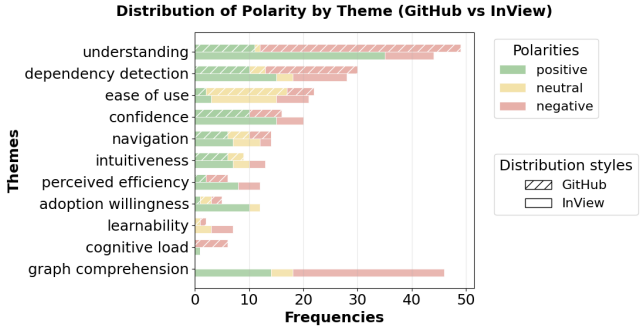

Figure 11: Polarity Distribution (InView vs GitHub)

Table 1 summarizes the main themes identified in the analysis. The complete codebook, including detailed code definitions is available in the online appendix [4]. Figure 11 presents the distribution of sentiment polarity across themes for both tools. The results reveal clear differences in how participants perceived InView and GitHub, particularly in terms of understanding, dependency detection, and overall usefulness. InView received substantially more positive codes for understanding (35 vs 9), whereas GitHub was predominantly negative (37 vs 11), indicating that participants perceived InView as more effective for code comprehension. A similar trend is observed for dependency detection, reinforcing the findings from RQ1. However, the Graph Comprehension theme had more negative codes (28 vs 14), suggesting that while the graph aids understanding for some users, it introduces challenges for others. Perceptions for efficiency were mixed, with InView showing more positive codes, despite the longer review times observed in RQ2. Usability-related themes (ease of use, navigation, and intuitiveness), were balanced or slightly positive for both tools, indicating that usability alone does not explain differences in dependency detection capabilities. Finally, InView was associated with higher confidence and a strong willingness to adopt (10 positive, 2 neutral, 0 negative), whereas GitHub showed more neutral or negative responses.

5.4 Discussion and Implications

Taken together, the qualitative results help explain the trade-offs observed in the quantitative analysis. InView improves understanding and dependency awareness, which aligns with the higher accuracy observed in RQ1. While navigating the graph-based representation requires deeper code investigation, contributing to the longer

[6] For the experiment, they had only a very brief explanation of the features.

Table 1: Overview of qualitative themes

Theme	Description
Understanding	How the tool supports comprehension of code and relationships between elements
Dependency Detection	Ability to identify and reason about dependencies in the code
Perceived Efficiency	Perceived impact on review time and navigation effort
Cognitive Load	Mental effort required to use the tool and understand the code
Ease of Use	Perceived difficulty or ease of interacting with the tool
Navigation	Ability to move across files and locate relevant information
Intuitiveness	How easily users understand the interface without guidance
Learnability	Effort required to become familiar with the tool
Graph Comprehension	Ability to interpret and use the graph visualization
Confidence	Confidence in review decisions and perceived support from the tool
Adoption	Willingness to use the tool in practice

review times observed in RQ2, the strong positive perceptions of usefulness, confidence, and adoption suggest that reviewers value the additional insights provided by InView. Participants also suggested improvements to InView. While some suggestions focused on usability, such as cosmetic refinements to the interface, other have implications for scalability. In particular, participants proposed grouping dependencies by code region or by the affected program element (e.g. the same variable), which could reduce the effort required to inspect many reported dependencies.

Overall, these results highlight a trade-off between improved awareness and increased review effort. From a practitioner perspective, InView is likely most beneficial in scenarios where understanding interactions between concurrent changes is critical, such as complex refactoring, tightly coupled modifications, or large feature integrations, where semantic dependencies might be difficult to infer from syntactic diffs alone. In contrast, for small or largely independent changes, the additional overhead of inspecting dependencies may not provide sufficient benefit over traditional diff-based review. From a research perspective, these findings suggest opportunities for adaptive dependency visualization techniques that selectively surface semantic relationships only when they are likely to provide value, thereby reducing cognitive overhead while preserving the benefits of enhanced dependency awareness.

Finally, in more realistic development settings, developers often review PRs after pulling and running code locally, providing additional context beyond our controlled setup. In such cases, the main difference would likely lie in prior knowledge of the code base rather than in how reviewers interact with the tool. Participants also frequently rely on running tests as part of their workflow. While this may help detect some issues, tests often do not reliably capture semantic dependencies between concurrent changes [14, 21, 22], which is the type of information InView is designed to surface.

6 Threats to Validity

We identified the following threats to validity that affect this study.

Regarding internal validity, a potential concern is the presence of learning effects, as participants performed two review tasks sequentially. To mitigate this, we employed a balanced incomplete crossover design, ensuring that each participant reviewed different pull requests with each tool and task order was counterbalanced. Using different PRs also reduces contamination between treatments by preventing familiarity with one review task from influencing performance of the other tool. Besides that, assigning distinct PRs and balancing the order of treatments also contributed to mitigate carryover effects, where exposure to one interface may have influenced participants' review strategy in subsequent tasks. Additionally, participants receive a brief training session on InView to reduce bias due to unfamiliarity with the tool. The think-aloud protocol may have influenced participants' behavior or increased task duration, however, it was necessary to capture reasoning and was applied consistently across conditions.

In terms of construct and conclusion validity, our measures may not fully capture the studied phenomena. RQ1 was assessed through a task-specific question acting as an oracle, which, while objective, may not reflect all aspects of understanding. To mitigate this, we validated responses using think-aloud data. RQ2 was measured through task completion time, which may be affected by individual differences. Perceptions (RQ3) were collected via Likert-scale and qualitative responses, which are inherently subjective. We addressed this through triangulation with thematic analysis.

Finally, external and qualitative validity may affect the generalizability and interpretation of the results. The study involved 12 developers recruited via open calls, which may not represent broader populations. Moreover, the use of synthetic pull requests adapted from real scenarios improves experimental control but may not fully reflect real-world complexity— larger PRs with many files may increase the cognitive effort required to inspect all reported dependencies. Also, thematic analysis may be subject to researcher bias. To mitigate this, we employed independent coding, iterative refinement, and consensus among researchers. While additional validation techniques such as member checking were not performed, the consistency between qualitative and quantitative findings increases confidence in the results.

7 Related Work

This section presents the work related to the development of InView, discussing other methods and tools for supporting code review.

Workflow Impact. The effectiveness of code review is also influenced by how integrations are performed, as investigated by Paixao

and Maia [17] in their study on the impact of rebasing. The research shows that the practice of rebasing can be detrimental to the review process by linearizing the history of commits and "flattening" the history of collaboration. This makes changes that were developed concurrently and independently appear to have been written sequentially, obscuring the fact that the authors were not aware of each other's changes at the time of writing.

Comparison with Industrial and Academic Platforms. In addition to relating to studies involved in the context of semantic interference and code review, the solution introduced in this research can be compared with previously proposed analysis ecosystems that seek to balance scalability and practical utility.

Menarini et al. [16] developed Getty, a tool that implements the concept of Semantics-Assisted Code Review (SCR) by mining likely program invariants. These invariants are extracted using a dynamic call graph analyzer, which runs test suites across different code versions to generate a "behavioral summary" that summarizes the effects of each commit. In an evaluation of 6 open-source projects, the authors demonstrated that invariant differentials helped expose test gaps in 32 out of 100 randomly selected commits, all with 100% reported code coverage. These results align with the idea that suites with high code coverage may be insufficient to guarantee test effectiveness. The same work also performed a study with 18 developers, it was observed that Getty fundamentally altered the review process: while control teams suffered from fatigue caused by linear and massive code reading, Getty users adopted a hypothesis-driven process. InView is related to Getty in its aim to reduce the cognitive load and effort required to understand changes, however, the tools support different forms of reasoning. Getty enables reviewers to reason about the expected runtime behavior in a hypothesis-driven manner. In contrast, InView supports dependency-oriented reasoning, helping reviewers understand how concurrent modified code elements influence one another through explicit semantic dependencies. Instead of invariants, InView reports semantic dependencies, and instead of behavioral summaries, InView uses an enriched diff and graphical visualizations to represent these dependencies.

In contrast, Tricorder [20], developed by Google, is a software analysis ecosystem focused on building a robust data environment around static analyses. It leverages a microservices architecture to automatically integrate various inspection tools into the developers' workflow, processing thousands of results daily and displaying them as comments (robocomments) directly in the code review tool. The platform prioritizes actionable, low-latency checks based on bug patterns detected via AST (Abstract Syntax Tree) matching using analyzers such as ErrorProne and ClangTidy. Unlike InView, Tricorder prioritizes detecting isolated problems in a single code version, without focusing on the interaction between parallel changes. While InView uses graphs and a diff to highlight the propagation of semantic dependencies between changes from different developers, Tricorder communicates its findings through point-in-time text messages. Thus, InView fills a gap in visualizing semantic dependencies that Tricorder, by its nature geared towards the scalability of local checks, does not address.

Another tool in the same context is Emergo [18], designed to improve the maintainability of preprocessor-based software product lines. Emergo helps developers identify dependencies between functionalities that may be affected during maintenance tasks. The solution is based on the concept of "Emerging Interfaces" to capture and expose the connections between the maintained functionality and other parts of the system through context-sensitive data flow analysis of conditional compilation directives. To aid understanding, Emergo provides table and graph-based visualizations integrated into the Eclipse development environment. Although Emergo also uses graphical representations to manage code dependencies, its scope is restricted to modularity and separation of concerns in systems that use preprocessors. InView differentiates itself by transposing the analysis of semantic dependencies to the context of reviewing PRs on GitHub, focusing on interferences between simultaneous modifications in collaborative workflows.

8 Conclusion

This paper investigates the impact of InView on code review by comparing it with the default GitHub interface through a controlled study with 12 developers. Our results show that InView improves developers' ability to identify semantic dependencies, as evidenced by higher accuracy in dependency-related tasks and supported by qualitative feedback indicating enhanced understanding of code relationships. We observed that providing this deeper contextual awareness is accompanied by an increase in review time. Qualitative findings indicate that the graph visualization promotes a more thorough exploration of the code, which benefits comprehension but also requires a period of familiarization from the user. Developers reported higher confidence in their decisions and a strong willingness to adopt InView, suggesting that the perceived benefits of improved understanding balances the time investment.

These findings highlight a dynamic in tool support for code review: providing richer representations to foster context awareness can enhance comprehension but may introduce additional time demands. From a design perspective, this suggests that tools like InView should balance expressiveness with usability, particularly by improving the clarity and learnability of visual representations such as graphs. More generally, our results reinforce the importance of supporting developers in reasoning about code relationships, which are often difficult to identify using traditional review interfaces. For future work, we intend to evaluate InView using real PRs and make general improvements to the tool (usability, scalability and scope).

Artifact Availability

The InView repository [3] and other artifacts (codebook, interview questions, transcriptions, demographics, PR descriptions) [4] are available on the referenced urls under the license Creative Commons Attribution 4.0 International (CC BY 4.0).

Acknowledgements

We acknowledge the members of the Software Productivity Group, in particular Galileu Santos, Matheus Barbosa and Rodrigo Bonifácio, for their support in the understanding and integration of static analyses, as well as Léuson da Silva for contributing to the examples. We also acknowledge INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence), CNPq (grants 408817/2024-0 and 401032/2025-6) and FACEPE (grant IBPG-1892-1.03/25).

References

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. 712–721.

[2] Matheus Barbosa, Paulo Borba, Rodrigo Bonifacio, and Galileu Santos. 2022. Semantic conflict detection with overriding assignment analysis. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering (SBES '22)*. 435–445.

[3] Vinicius Resende Barbosa, Ykaro dos Santos Albuquerque, and Paulo Borba. 2026. *InView GitHub repository.* https://github.com/Vinicius-resende-cin/InView

[4] Vinicius Resende Barbosa, Ykaro dos Santos Albuquerque, and Paulo Borba. 2026. Online Appendix for Semantic Dependency Visualization on Code Review: A Comparative Analysis. doi:10.5281/zenodo.19902162

[5] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. 2017. Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft. *IEEE Transactions on Software Engineering* 43, 1 (2017), 56–75.

[6] Galileu de Jesus, Paulo Borba, Rodrigo Bonifácio, and Matheus de Oliveira. 2026. Comparing static analyses for improved semantic conflict detection. *Automated Software Engineering* (2026).

[7] Galileu Santos De Jesus, Paulo Borba, Rodrigo Bonifácio, and Matheus Barbosa De Oliveira. 2024. Lightweight Semantic Conflict Detection with Static Analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24)*. 343–345.

[8] DORA. 2024. *Accelerate state of devops 2024.* Technical Report. Google Cloud. https://dora.dev/research/2024/dora-report/

[9] DORA. 2025. *State of AI-assisted Software Development 2025.* Technical Report. Google Cloud. https://dora.dev/research/2025/dora-report/

[10] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2019. Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 49–60.

[11] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2021. An exploratory study on confusion in code reviews. *Empirical Softw. Engg.* 26, 1 (Jan. 2021), 48 pages.

[12] GitHub. 2026. *About comparing branches in pull requests.* https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-comparing-branches-in-pull-requests

[13] Lo Gullstrand Heander, Emma Söderberg, and Christofer Rydenfält. 2026. Code review as decision-making - building a cognitive model from the questions asked during code review. *Empirical Software Engineering* 31, 3 (03 Jan 2026), 54.

[14] Toni Maciel, Paulo Borba, Léuson Silva, and Thaís Burity. 2024. Explorando a detecção de conflitos semânticos nas integrações de código em múltiplos métodos. In *38th Brazilian Symposium on Software Engineering (SBES 2024)*. 181–191.

[15] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. 2018. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software* 35, 4 (2018), 34–42.

[16] Massimiliano Menarini, Yan Yan, and William G. Griswold. 2017. Semantics assisted code review: an efficient toolchain and a user study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. 554–565.

[17] Matheus Paixao and Paulo Henrique Maia. 2019. Rebasing in Code Review Considered Harmful: A Large-Scale Empirical Investigation. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 45–55.

[18] Márcio Ribeiro, Társis Tolêdo, Johnni Winther, Claus Brabrand, and Paulo Borba. 2012. Emergo: a tool for improving maintainability of preprocessor-based product lines. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development Companion (AOSD Companion '12)*. 23–26.

[19] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. 181–190.

[20] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 598–608.

[21] Léuson Silva, Paulo Borba, Toni Maciel, Wardah Mahmood, Thorsten Berger, João Moisakis, Aldiberg Gomes, and Vinícius Leite. 2024. Detecting semantic conflicts with unit tests. *Journal of Systems and Software* 214 (2024).

[22] Léuson Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. 2020. Detecting Semantic Conflicts via Automated Behavior Change Detection. In *36th IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)*. 174–184.

[23] victorlira. 2026. ref-dataset. https://github.com/spgroup/ref-dataset. Version: commit 3a79a38017db4c052e56ac7383b6cde01281fc61.