SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

How Might We Improve Collaboration Amongst Software Engineers during Code Reviews?

Submitted by

LAI Zheng Hui Carey

Thesis Advisor

Dr. Simon PERRAULT

ISTD

A thesis submitted to the Singapore University of Technology and Design in fulfilment of the requirement for the degree of Master of Engineering in Innovation by Design (MIbD).

2025

Acknowledgements

"I know that I know nothing." ~ Plato

This thesis would not have been possible without:

the patient guidance of Professor Simon Perrault,

the design innovation classes of the Professor Arlindo Silva,

and the software engineers who have participated in this study.

Table of Contents

Chapter 1. Introduction

1.1 Background and Motivation

Code review has become a cornerstone of modern software engineering practice, serving as both a quality assurance measure and a collaborative learning opportunity for engineers. In distributed and agile teams, reviews are often the primary means of ensuring that code meets technical, organizational, and security standards before integration (Rigby and Bird 2013; Bacchelli and Bird 2013). Beyond defect detection, reviews foster knowledge transfer, improve team cohesion, and serve as a vehicle for mentoring less experienced developers (Sadowski et al. 2018; McIntosh et al. 2014).

Despite these benefits, conventional review practices face persistent challenges. Asynchronous reviews through pull requests are efficient but frequently suffer from communication breakdowns, delayed feedback, and difficulty in comprehending unfamiliar code (Kononenko et al. 2015). Synchronous review meetings, while offering richer interaction, can be hindered by interruptions, uneven participation, and the cognitive load of understanding complex code under time pressure (Herbsleb and Mockus 2003; Storey et al. 2020). These limitations highlight a need for tools and practices that better support collaboration, knowledge sharing, and code comprehension during review.

1.2 Problem Statement

Software engineers often encounter significant barriers to effective collaboration during code reviews. Surveys conducted for this research identified pain points including interruptions from external stakeholders, lack of technical or domain-specific knowledge, and communication difficulties across teams. Existing tools primarily focus on static text-based reviews, which make it challenging to visualize code structure or understand runtime behaviour. Recent work in software visualization suggests that dynamic, interactive representations can improve comprehension (Krause-Glau et al. 2024), but these approaches are rarely integrated into mainstream review workflows. Furthermore, current practices rarely leverage emerging HCI concepts, such as proxemics or multi-device interaction, to facilitate smoother collaboration (Greenberg et al. 2011; Ballendat, Marquardt, and Greenberg 2010). As a result, engineers may spend considerable time struggling to align on context rather than focusing on improving the quality and security of the code.

1.3 Research Questions

To address these issues, this thesis investigates how software visualization, proxemics, and lightweight security scanning can enhance collaborative code review. The study is structured around two primary research questions and one exploratory extension:

- **RQ1**: How do engineers at different organisations perceive the value of code visualization during code reviews?
- **RQ2**: Can proxemic triggers improve synchronous code reviews compared to existing workflows?
- **RQ3**: Can a simplified vulnerability scanner reduce knowledge and experience gaps in collaborative reviews without disrupting workflow efficiency?

1.4 Research Aims and Objectives

The aim of this research is to improve collaboration among software engineers during code reviews through the design and evaluation of novel interaction techniques. The specific objectives are:

1. To identify and validate pain points in collaborative software engineering practices through surveys.
2. To design and prototype incremental features that directly address these pain points.
3. To evaluate the usability and effectiveness of these prototypes with professional engineers.
4. To derive empirical insights into how visualization, proxemics, and security tools affect collaboration during review.

1.5 Contributions of the Thesis

The contributions of this thesis are threefold:

1. **Empirical Findings**: Surveys and usability studies documenting pain points and user perceptions of visualization, proxemics, and lightweight security scanning in code review.
2. **Prototyping of Sentinel**: An visual tool incorporating function call graphs, proxemic triggers, and a simplified security scanner to support collaborative review.
3. **Research Insights**: Analytical reflections on how emerging HCI concepts, such as proxemics, can be integrated into traditional software engineering practices.

Chapter 2. Literature Review

2.1 Modern Code Review: Practices and Implications

2.1.1 From Traditional Inspections to Lightweight Reviews

Traditional code reviews, such as Fagan inspections, were highly structured, face-to-face processes effective at defect detection but costly in fast-paced environments. Modern Code Review (MCR) evolved as a lightweight, asynchronous alternative, supporting small change sets and peer assessments. Rigby and Bird (2013) documented this shift across open-source and industrial settings, while Sadowski et al. (2018) demonstrated its large-scale institutionalisation at Google, where MCR supports quality control, mentoring, and knowledge transfer. However, MCR also removes the benefits of co-located, synchronous coordination and clarification.

2.1.2 Core Practices and Visualization Gaps

MCR practices have converged across tools and organisations: reviewers prefer small changes (<200 LOC), asynchronous workflows, and socially familiar collaborators (Bacchelli and Bird 2013; Sadowski et al. 2018; Badampudi et al. 2023). At Google, most reviews involve one reviewer, are completed within hours, and support onboarding in addition to correctness (Sadowski et al. 2018).

Yet, the benefits of visual support for comprehension in code reviews remains relatively unexplored. Review platforms like GitHub and Gerrit rely heavily on textual mechanisms such as inline diffs and comments. Bacchelli and Bird (2013) and Badampudi et al. (2023) found that visual artefacts such as function call graphs or architectural sketches are rarely used, even though they may support structural understanding. This motivates RQ1: could such visualizations help reviewers grasp complex changes more effectively?

2.1.3 Characteristics of MCRs

Across Microsoft, AMD, and open-source projects, Rigby and Bird (2013) observed consistent MCR norms: frequent, small, asynchronous reviews based on social familiarity. McIntosh et al. (2016) confirmed that reviewer participation and promptness predict fewer post-release defects. Sadowski et al. (2018) reinforced this convergence at scale.

This convergence implies two things: (1) that MCR tools operate within a narrow design space (asynchronous, textual, tool-mediated), and (2) that friction points (delays, incomplete context, comprehension gap) are common across contexts. This consistency creates opportunities for visual or spatial augmentations (RQ1, RQ2) without destabilising established workflows.

2.1.4 Impact on Software Quality

Empirical studies confirm that MCR improves software quality, particularly when reviews are timely and conducted by knowledgeable peers (McIntosh et al. 2016; Bacchelli & Bird 2013). At Google, Sadowski et al. (2018) found that engineers valued MCR for long-term maintainability. Yet, even with high tooling maturity, many decisions are made without the use of visual aids. Badampudi et al. (2023) highlight ongoing issues like reviewer fatigue and inadequate structural support, reinforcing the need for comprehension-enhancing tooling, as explored in RQ1.

2.2 Social Factors in Code Reviews

Across open-source and industrial contexts, MCR is not only shaped by tooling and technical workflows, but also deeply influenced by communication mechanics.

2.2.1 Communication, Distance, and Coordination Delay

Distributed review settings introduce unique barriers. Herbsleb and Mockus (2003) empirically demonstrated that geographically distributed software changes take approximately 2.5 times longer to complete than same-site changes, primarily because they involve more people and less informal communication. The size and frequency of communication in distributed teams is reduced, social networks are weaker, and cross-site coordination leads to longer modification request intervals. These findings highlight how asynchronous review workflows may exacerbate delays in globally distributed teams.

2.2.2 Summary and Implications

As such, review outcomes are not only shaped by review size or tool design (as discussed in Section 2.1), but by how communication is mediated. This motivates later chapters' explorations of how proximity-based triggers (RQ2) might support more fluid social coordination in co-located or hybrid teams.

2.3 Software Visualization for Comprehension and Collaboration

Software visualization has long been recognised as a means of aiding program comprehension, particularly in large and complex codebases. From static diagrams to immersive 3D representations, visualizations externalize abstract structures and relationships, supporting developers in navigating unfamiliar or intricate systems.

2.3.1 Foundations and Evolving Metaphors

Traditional visual metaphors such as SeeSoft (Marcus et al. 2003) laid the groundwork for line-oriented visualizations, mapping code metrics onto color and position. These early 2D approaches informed more expressive 3D representations, such as software cities and polymetric views, which leveraged spatial memory and perspective to encode structure, evolution, and ownership at scale (Marcus et al. 2003, Lanza and Ducasse 2003). Khan et al. (2011) extended this perspective by reviewing architecture-level visualization techniques, showing how hierarchical representations can aid comprehension but also suffer from scalability challenges in large systems.

Recent work has explored newer metaphors like software cities (Krause-Glau et al. 2024) or hierarchical edge bundles, balancing abstraction with interpretability (Padoan et al. 2024). These visualizations increasingly integrate both static and dynamic analysis data, enabling developers to reason not only about structure but also about runtime behaviour (Krause et al. 2024).

2.3.2 Comprehension, Review, and Onboarding

Empirical studies confirm that visualization tools support developers' mental model construction during onboarding and review. Padoan et al. (2024) found that although many practitioners recognize the value of software visualization for onboarding, few organizations integrate such tools systematically. This aligns with findings from Grabinger et al. (2024), who noted that developers often rely on textual and navigational cues alone, leading to cognitive overload and slower onboarding in large systems.

Specifically, within the context of code reviews, Krause-Glau et al. (2024) propose integrating static and dynamic analysis (including execution traces) into review visualizations, with the expectation that such integration can facilitate comprehension and error detection during review.

2.3.3 Function Call Graphs and Real-Time Interaction

Function call graphs (FCGs) are a particularly promising visualization for bridging code structure and comprehension. Tools such as PyCG for Python (Salis et al. 2021) and Project Map for C/C++ (Csaszar et al. 2020) enable developers to inspect call hierarchies dynamically while editing code, with minimal setup and hot reloads on code change. Li et al. (2024) extend this line of work with PyVisVue3D3, which extracts Python package, module, and class relationships and provides multiple interactive 3D layouts to support rapid comprehension of large codebases.

Such responsiveness is crucial during code reviews, where understanding a change's ripple effect across a call graph can clarify design rationale and surface potential side effects. Interactive FCGs allow reviewers to contextualize diffs within the broader program flow, potentially reducing misunderstandings and improving feedback precision.

2.3.4 Summary and Implications

Despite the clear value of software visualization, its integration into mainstream code review tools remains rare. Many systems rely solely on text-based diffs, leaving structural relationships implicit. This section highlights the potential of lightweight, embedded visualizations, such as FCG overlays, to support comprehension without disrupting established review workflows. The usability tests in later chapters examine whether such visualizations address the comprehension gap identified in Section 2.1 and 2.2.

2.4 Proxemic Interactions

Proxemic interaction, a concept derived from Edward Hall's theory of interpersonal distance, offers new affordances for designing spatially responsive systems that modulate behaviour based on distance, orientation, movement, identity, and location. While there is research interest in the benefits of proxemics to improve workplace productivity, the benefits of applying proxemics specifically to code reviews remains unexplored.

2.4.1 Evolution of Proxemics

Ballendat et al. (2010) demonstrated interaction techniques mediated by continuous movement or transitions across discrete proxemic zones, including implicit and explicit interactions with digital surfaces. Greenberg et al. (2011) extended this into proxemic interaction design, proposing that systems should dynamically adapt based on spatial cues, such as increasing informational fidelity as a user approaches a display. In collaborative settings, such systems can regulate access, awareness, and attention, providing seamless transitions from ambient to personal interaction. For example, a code review interface might become more detailed as a reviewer leans in or trigger shared debugging views as two developers approach a shared display.

2.4.2 Applications of Proxemics

Recent work has extended proxemic models to ecologies of devices and users. Marquardt et al.'s Proximity Toolkit (2011) enabled rapid prototyping of interactions involving people, devices, and environmental features like chairs or doors. Lunding et al. (2023) pushed this further into hybrid collaborative environments, such as co-located augmented reality, showing how proxemics can be used to guide transitions between individual and shared focus.

This growing body of work suggests that spatial awareness, which is traditionally neglected in developer tooling, can be repurposed to facilitate lightweight coordination, reduce verbal overhead, and support micro-collaboration moments like walkthroughs or handovers.

2.4.3 Summary and Implications

Proxemic interaction offers a promising lens for enriching collaborative developer environments by leveraging spatial awareness. Later chapters explore whether proximity-triggered views, designed based on these principles, can support more fluid social and cognitive coordination during code reviews (RQ2).

2.5 Challenges in Static Application Security Testing (SAST) Tools

Static Application Security Testing tools (SASTs) promise to improve code quality by detecting bugs and vulnerabilities before deployment. However, their widespread adoption remains limited, especially in modern CI/CD pipelines.

2.5.1 Usability and Developer Frustration

Many SASTs are perceived as overly rigid and noisy. Johnson et al. (2013) reported that high false positive rates and poor IDE integration discourage use, particularly among junior developers unfamiliar with interpreting security alerts. Developers often prefer lightweight tools or linters that offer quick, actionable feedback (Vassallo et al. 2018; Yang et al. 2019). Yang et al. (2019) found that traditional SASTs frequently interrupt developer flow, either through excessive warnings or irrelevant findings. In the context of code reviews, where time and cognitive resources are limited, developers deprioritize tools that are not precisely aligned with their immediate context.

2.5.2 Output Efficacy of SASTs

A recurring theme across studies is the uneven ability to interpret security warnings amongst engineers of different seniorities. Vassallo et al. (2018) highlight that inexperienced developers often lack the mental model to assess the relevance of a warning during review, especially when severity is unclear or improperly labelled. This suggests an opportunity for simplified SASTs that translate low-level static analysis into high-level review-relevant summaries.

2.5.3 Pipeline Integration Challenges

Saleh et al. (2024) note that enterprise SASTs in CI/CD pipelines are often too heavy for daily use. Tools like SonarQube and CodeQL have attempted to bridge this gap but often still suffer from limited customization and inconsistent severity labeling (Vassallo et al. 2018; Saleh et al. 2024). For a SAST to be useful during code review, especially by novice reviewers, it must be responsive, easy to configure, and compatible with fast-paced, iterative workflows.

2.5.4 Summary and Implications

Across these studies, the common theme is that SASTs are most effective when they are context-sensitive, trustworthy, and lightweight. Yet, most tools today fall short on one or more dimensions.

This motivates RQ3: Can a simplified vulnerability scanner reduce knowledge and experience gaps in collaborative reviews without disrupting workflow efficiency?

The prototype introduced in Chapter 4 is designed to test this hypothesis.

2.6 Research Gaps

Across the literature, three key gaps emerge.

Firstly, while modern code review (MCR) practices have converged around small, asynchronous changes, they remain almost entirely text based. Visual artefacts such as function call graphs are rarely integrated into review tools, leaving comprehension challenges unresolved (RQ1).

Secondly, review outcomes are shaped not only by tooling but also by communication and spatial coordination, yet existing platforms neglect embodied and proxemic interactions that could support synchronous alignment (RQ2).

Thirdly, while Static Application Security Testing (SAST) tools promise early defect detection, they are often noisy, poorly integrated, and difficult for less experienced developers to interpret. This highlights the need for lightweight, simplified scanners that bridge expertise gaps without disrupting workflows (RQ3).

These gaps motivate the iterative Research through Design approach presented in the following chapters.

Chapter 3. Research Approach

3.1 Research Methodology

This thesis is grounded in Research through Design (RtD), a well-established methodology within HCI, recognised for its ability to generate intermediate-level knowledge through the creation and evaluation of prototypes (Zimmerman et al. 2007; Gaver 2012). It has been identified as particularly well-suited for research at the intersection of HCI and software-intensive domains such as software engineering, because it allows novel prototypes to surface and interrogate emergent practices that cannot easily be studied through traditional controlled experiments (Wiberg 2014).

By adopting RtD, this thesis acknowledges that designing, deploying, and reflecting on prototypes can reveal tensions between code review theory and practice. It also bridges two traditions: the empirical software engineering and the broader context of HCI.

3.2 Iterative Workflow

The inquiry unfolded through an **iterative RtD cycle**:

1. **Exploration**: Surveys with professional engineers identified collaboration challenges in code reviews.
2. **Prototyping**: Successive iterations of the prototype introduced targeted interventions such as function call graph visualization, proxemic triggers, and a simplified vulnerability scanner.
3. **Evaluation**: Each prototype was studied with professional engineers using a mix of task-based usability studies, interviews, and reflection sessions.
4. **Reflection**: Insights from each cycle informed the design of subsequent iterations, progressively broadening the scope from comprehension to coordination and security.

This incremental structure aligns with RtD's principle of knowledge generation through situated evaluation rather than through a single monolithic study.

3.3 Methods in Context

Although unified under RtD, each iteration required **different empirical methods**, chosen pragmatically:

- **Surveys** (problem exploration): Likert-scale and open-ended responses identified pain points and informed the first design iteration.
- **Usability Testing** (prototyping cycles): Task-based sessions captured efficiency, breakdowns, and subjective feedback.
- **Interviews and Think-Aloud Protocols**: Provided qualitative insight into how participants perceived prototypes in relation to their workflows.
- **Thematic Analysis**: Applied to interview transcripts to identify recurring themes across studies.

Framing the research through RtD enables the contribution of empirical insights into how visualization, proxemics, and simplified scanning affect collaborative code review.

This approach provides the foundation for the next chapters, where each iteration of the prototype is described in detail - its design rationale, evaluation, and reflection.

Chapter 4. Design Iteration 1: Function Call Graphs

4.1 Survey and Motivation

To refine the problem space identified in the literature review, a focused survey was conducted with 13 professional software engineers. This survey aimed to understand day-to-day collaboration practices, pain points in code reviews, and opportunities for tool support.

4.1.1 Survey Demographics

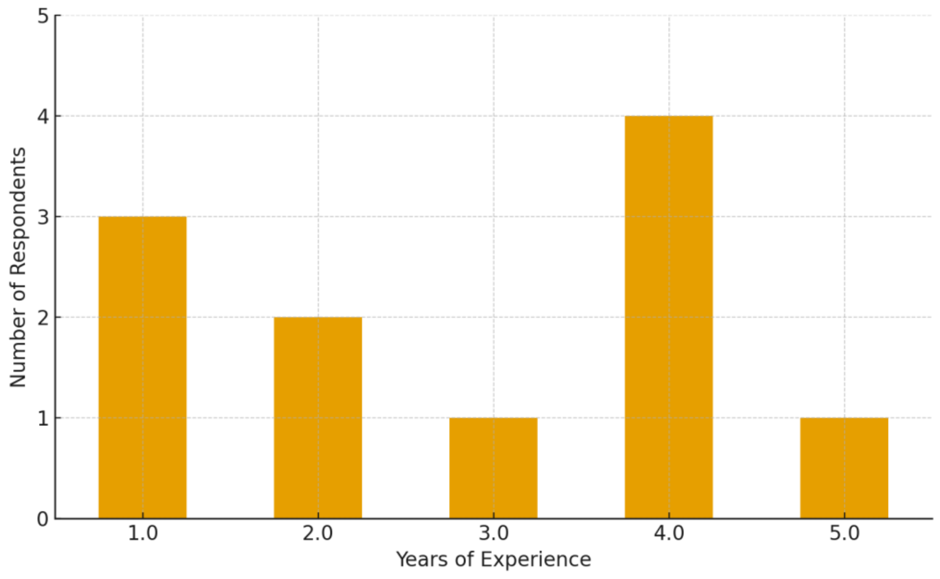


Figure 1: Experience Levels of 13 Full-time Software Engineers

All respondents were practicing software engineers. The majority fell within 1-5 years of professional experience, with roles and responsibilities spanning feature development, software testing, refactoring, production support, sprint planning, mentoring, and deployment tasks**.** This cohort represents early- to mid-career engineers with active involvement in collaborative workflows, especially code review.

4.1.2 Survey Findings

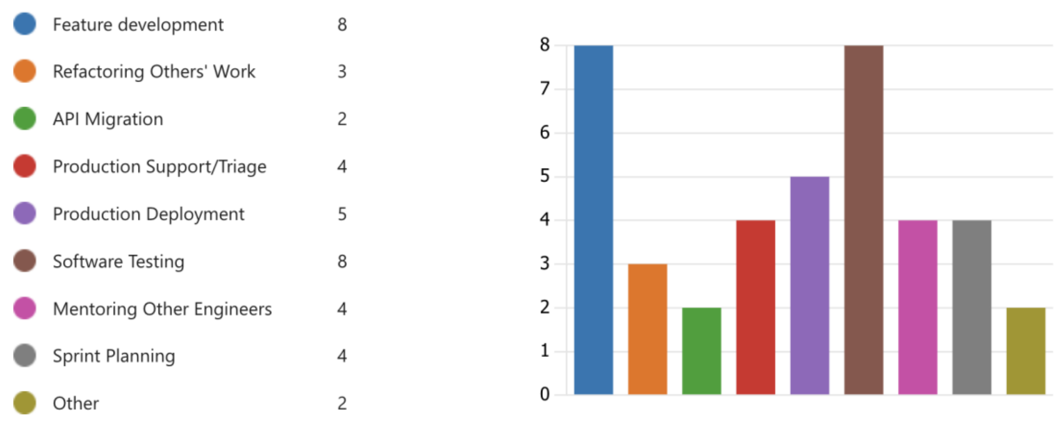


Figure 2: Roles and Responsibilities amongst 13 Full-time Software Engineers

- **Collaboration preferences:** Respondents reported relying on both **informal synchronous communication** (e.g., “popping by their desk,” face-to-face clarifications) and **asynchronous channels** (e.g., email, chat, design reviews). This echoes the literature on the coexistence of synchronous and asynchronous practices in MCR.
- **Collaboration challenges:** Engineers noted difficulty understanding intent (“difficulty understanding what others are trying to build”), misalignments in teamwork, scheduling issues, and occasional language/cultural barriers.
- **Workflow pain points:** A consistent theme was lack of technical or domain-specific knowledge, particularly during feature development and code review. Respondents described challenges in grasping the broader system implications of code changes, highlighting the burden of context-switching and incomplete comprehension.

4.1.2 Corroboration of Survey Findings with Literature

These survey findings corroborate the literature review in Sections 2.1-2.3: comprehension difficulties remain the most persistent barrier to effective review. As with prior studies, the survey shows that engineers value quick feedback but often lack the structural context needed to understand changes. This not only slows review cycles but also exacerbates knowledge gaps, as less experienced reviewers struggle to participate meaningfully.

4.1.3 Motivation for Function Call Graphs (FCGs)

To address these comprehension and context challenges, the first prototype feature explores the use of Function Call Graphs (FCGs). FCGs are a classical way of visualising the relationships between different function calls and the data flow between them. By externalising dependencies in a visual form, FCGs can help reviewers trace ripple effects across a system, situate diffs within the larger codebase, and support the construction of mental models. The survey evidence, aligned with gaps in existing tools, provides a strong justification for testing whether FCGs can improve understanding during review, directly addressing RQ1.

4.2 Prototype Details

4.2.1 Prototype Features

4.2.1.1 Commit Change Visualization

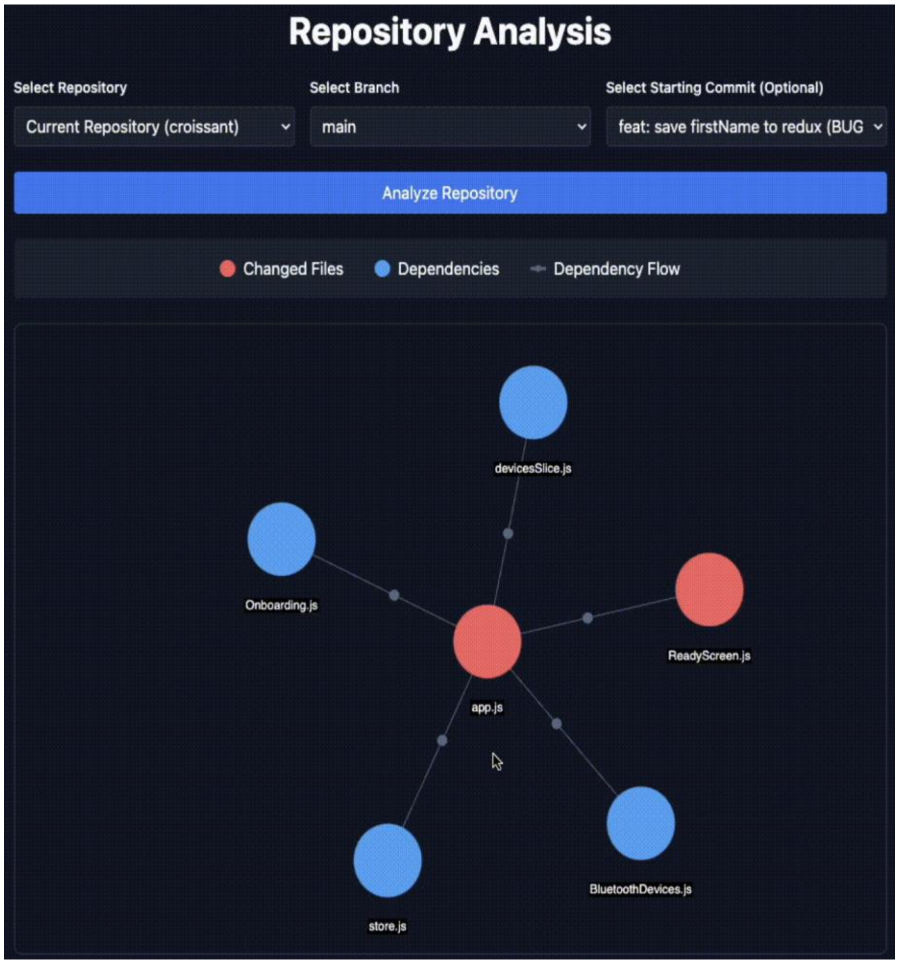


Figure 3: Engineers can select specific snapshot of a code base locally to visualise changes.

The system integrates seamlessly with existing Git workflows, allowing reviewers to select specific repositories, branches, or commits for analysis. This integration enables reviewers to understand not just what files have changed, but how those changes might affect other parts of the codebase. The visualization provides immediate visual feedback about change impact, helping reviewers identify potential areas of concern before they become problems. The interactive nature of the graph allows for exploratory analysis, where reviewers can follow dependency chains to understand the full scope of changes and their potential effects.

4.2.1.2 Spatial Relationships and Clustering in Function Call Graphs

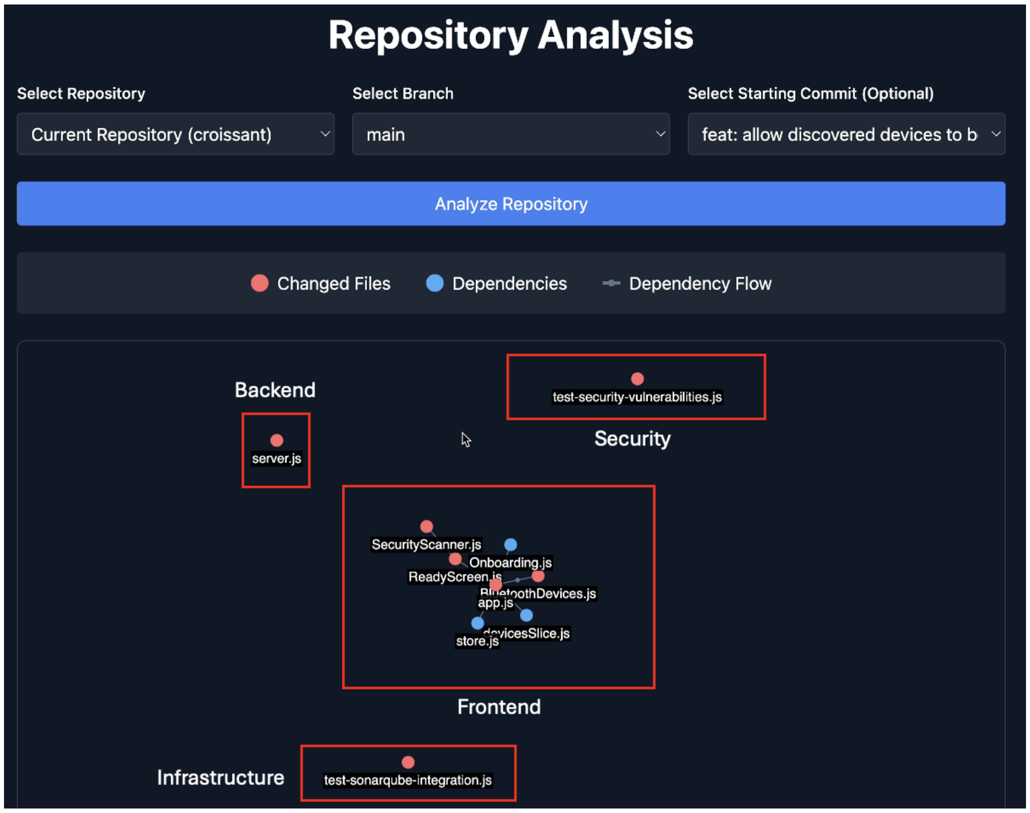


Figure 4. Spatial Relationship between Clusters of Files in the FCG

The spatial positioning of file clusters in the force-directed graph carries significant semantic meaning beyond visual organization. Files with strong dependencies naturally cluster together, forming cohesive modules that represent functional units within the codebase. The physical proximity of these clusters indicates architectural cohesion - related files that work together to accomplish specific features are positioned near each other, while loosely coupled files serving different concerns appear further apart.

This spatial clustering provides immediate visual understanding of the codebase's modular structure, helping reviewers quickly identify which parts are most tightly integrated and therefore most likely to be affected by changes.

The force-directed algorithm's optimization for visual clarity means that highly interconnected files form distinct, recognizable groups, while files with minimal dependencies appear more isolated. This distance-based representation aids in impact analysis, as changes within tightly clustered groups are more likely to have cascading effects on other files in the same cluster, while changes to isolated files are less likely to propagate throughout the system.

4.2.2 Technical Architecture

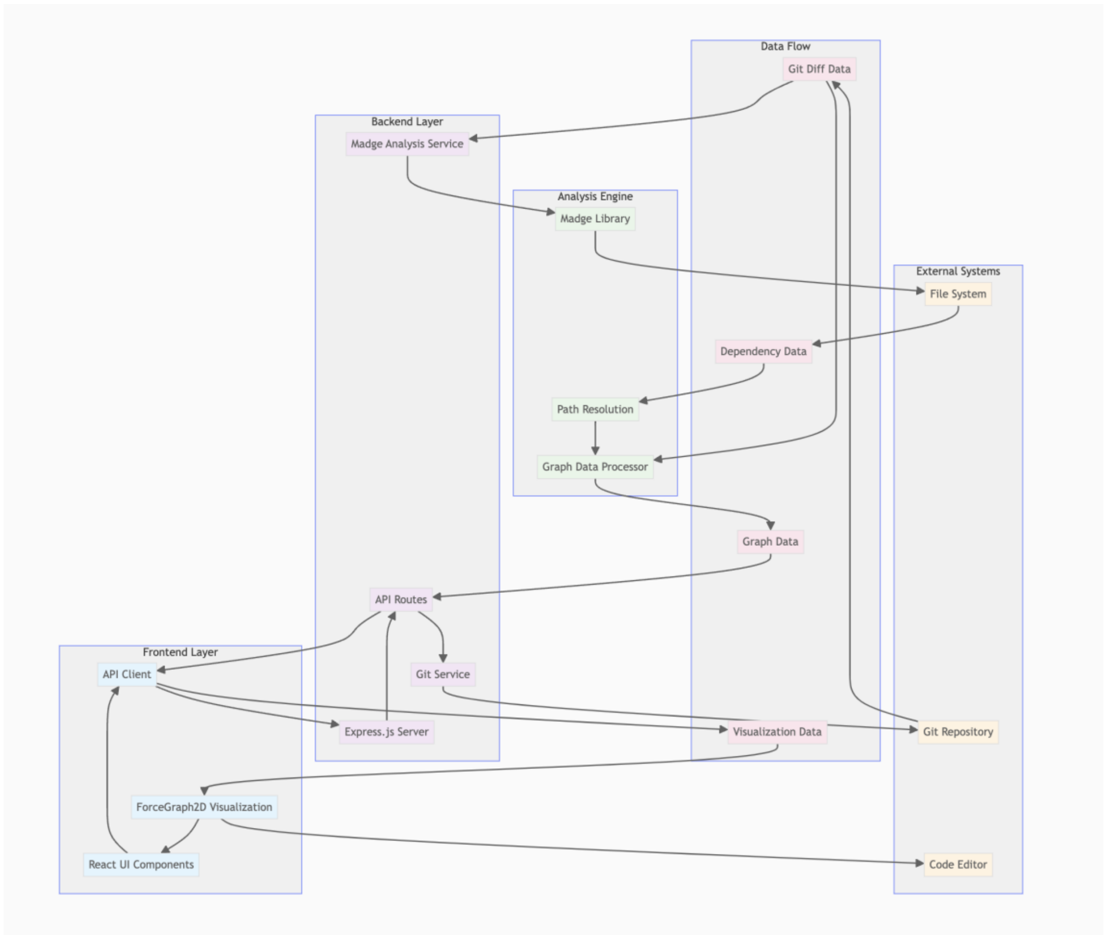


Figure 5: Technical Architecture of FCG

The implementation is done entirely in the JavaScript Web Engine Node.js.

Madge serves as the foundation for static analysis, providing reliable dependency detection across JavaScript and TypeScript codebases. React Force Graph 2D handles the complex task of interactive graph visualization, while D3.js provides the underlying data manipulation and force simulation capabilities.

The Express.js backend provides a clean RESTful API that separates the analysis logic from the presentation layer, enabling future enhancements and alternative visualization approaches. Git integration is handled through native system calls, ensuring compatibility with existing development workflows and version control practices.

4.3 Experimental Method

4.3.1 Participant Recruitment

2 software engineers participated in the first iteration study.

Participant A1 had two years of experience as a virtual reality Engineer (Unity3D) in a research lab and one year as a backend engineer at a telemedicine startup. His professional background meant he was familiar with collaborative code practices in both research and industry settings.

Participant A2 had 4 years of experience as a backend engineer at a social media conglomerate.

4.3.2 Study Design

The study followed a task-based usability format. The participants were asked to complete a series of review-related tasks with the Function Call Graph (FCG) prototype:

- **File identification** across repositories.
- **Context switching** between different repositories.
- **Visualising commit change snapshots** in a repository based on verbal task instructions.

The study was conducted as a think-aloud session combined with a semi-structured interview.

4.3.3 Data Collection

Data was collected qualitatively through:

- **Observation** of task performance and navigation patterns.
- **Interview responses** focusing on the perceived value of FCGs, ease of contextualisation, and workflow fit.

The participants explicitly compared the manual workflow (file identification through selection on GitHub) with the prototype's visualisation support, providing feedback on usability and perceived productivity.

4.3.4 Data Analysis

Interview data was analysed thematically. The following two themes were identified:

- **Comprehension support**: FCGs helped "reduce the time taken to analyze the code base" by showing surrounding context.
- **Collaboration potential**: The participant noted that visualisation could speed up onboarding and reduce explanation overhead when seeking help from peers.

These themes provided formative insights for evaluating the role of FCGs in code review comprehension and for refining subsequent iterations of the Sentinel prototype.

4.4 Results

The evaluation of the Function Call Graph (FCG) prototype was conducted with A1 and A2. Although limited in scale, the sessions generated rich qualitative insights.

4.4.1 Comprehension Support

A1 reported that the FCG visualisation was especially effective in surfacing structural context:

- *"Really good visualization… especially what files would change."*
- *"This visualization graph really helps to reduce the time… to analyze the code base."*

A2 expressed nuanced views about the prototype:

- *"I do appreciate a bigger view, how everything kind of fits in."*
- *"Conceptually, it's an interesting idea, but practically, if the graph is very dense, then it could be very noisy for me. I don't know."*

These comments reinforce the literature review findings that code reviews often suffer from comprehension bottlenecks. The prototype helped externalise surrounding context, reducing the need for manual navigation across repositories. However, a more longitudinal study is needed to validate the efficacy of FCG visualization on large code bases.

4.4.2 Collaboration and Onboarding

A1 emphasised that the FCG could streamline collaborative review, reducing the time and effort needed to bring colleagues up to speed:

- *"Take much less time to get both of us on board… since there's a visual representation of the feature branch."*
- *"I don't need to spend as much time explaining what files were changed… I can just look at the visual graph."*

This suggests that visualisation could support not only individual comprehension but also joint understanding during peer support and onboarding.

4.4.3 Design takeaways for Iteration 1

1. **Make context legible:** Consider the use of a "Context Card" (branch/commit/author, last updated).
2. **Show usages of changed functions:** A separate tab for visualising the usages of different functions could be also useful during code reviews.
3. **Measure what matters next.** Given the participant's emphasis on time savings, future studies should capture time-to-context**,** steps-to-first-useful-view, and number of clarifying explanations reduced during pair help.

Chapter 5. Design Iteration 2: Proxemics

5.1 Motivation

The first iteration demonstrated that FCGs can help engineers build a clearer mental model of a codebase and reduce the time required to understand structural context. However, the evaluation also highlighted that reviews often involve joint activity: engineers working together to interpret changes, onboard newcomers, or resolve ambiguities. In these cases, verbal explanations were still necessary to align understanding, even when visualisations reduced individual comprehension burdens.

The literature review in Section 2.4 introduced proxemic interaction as a design space within HCI that adapts system behaviour based on users' spatial relationships. Proxemics has been shown to support smooth transitions between personal and shared focus, and to reduce coordination overhead in collaborative settings. Yet, it has not been applied to software engineering workflows such as code review.

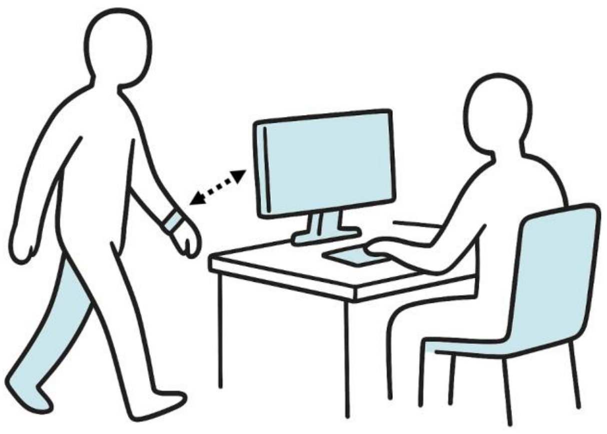


Figure 6. Proxemic Triggering of FCG Visualization

This motivated the second design iteration to explore the use of proxemic triggers to support synchronous code reviews. By automatically parsing review context such as the relevant branch and commit when an engineer approaches a shared display, the prototype aimed to reduce effort for context switching and lower the burden of verbal explanation. This directly addresses RQ2: Can proxemic triggers improve synchronous code reviews compared to existing workflows?

5.2 Prototype Description

5.2.1 Technical Architecture

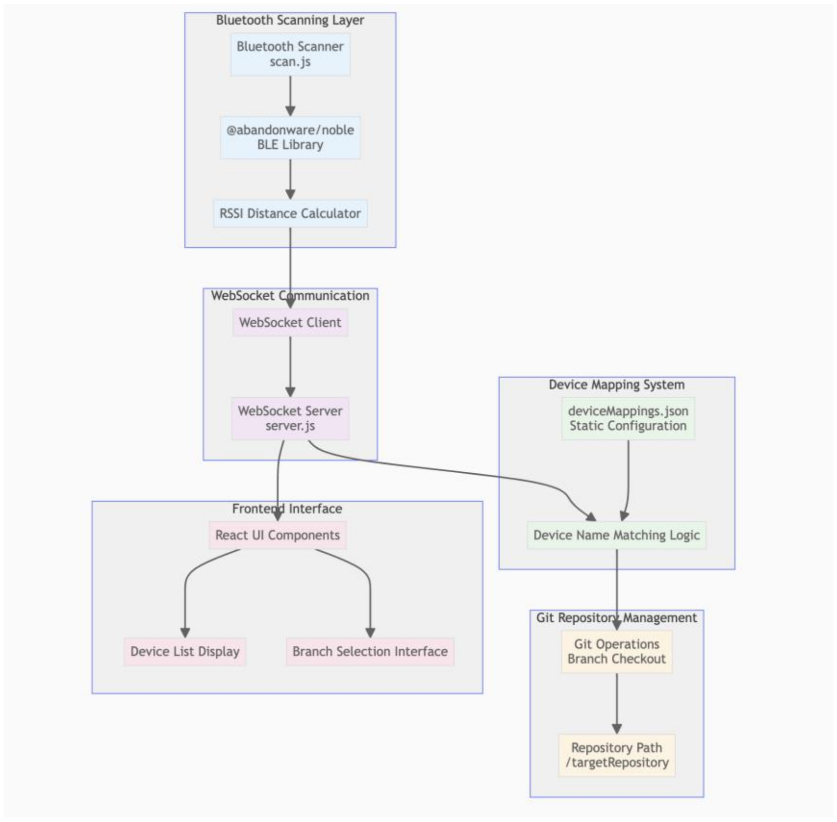

Bluetooth Scanning Layer
Bluetooth Scanner scan.js
@abandonware/noble BLE Library
RSSI Distance Calculator
WebSocket Communication
WebSocket Client
WebSocket Server server.js
Device Mapping System
deviceMappings.json Static Configuration
Device Name Matching Logic
Frontend Interface
React UI Components
Device List Display
Branch Selection Interface
Git Repository Management
Git Operations Branch Checkout
Repository Path /targetRepository

Figure 7. Technical Architecture of Proxemic Feature

The proxemic prototype was implemented using **Bluetooth-based sensing** to infer proximity between an approaching engineers' device (such as a smart watch), and the seated engineer's display. Each developer's device broadcast its identifier, which was mapped to the most recently active feature branch and commit in their local repository. When a device moved within range of the display, the system automatically retrieved this branch-commit context and displayed the corresponding Function Call Graph (FCG) and diff snapshot.

This pipeline consisted of three steps:

1. **Sensing**: Detect presence and proximity of approaching engineer's devices.
2. **Context mapping**: Match device to user, user to branch, and branch to last commit.

3. **Display triggering**: Push corresponding FCG and commit diff to the shared review screen.

The proxemics feature is implemented as a small, event-driven pipeline that turns Bluetooth Low Energy (BLE) presence into a deterministic “load this repository/branch” action. A headless scanner process (*scan.js*) runs on the host and uses the JavaScript library *@abandonware/noble* to continuously scan for BLE advertisements. On each device discovery event, it extracts *peripheral.advertisement.localName* and *rssi*, computes an approximate distance in metres from *rssi* and sends a JSON payload *{ deviceName, timestamp, distanceProxyInMetres }* over a persistent WebSocket connection to the backend at on localhost.

The backend (*server.js*) hosts both an HTTP API and a WebSocket server. Each time it receives a device payload over WebSocket, it first rebroadcasts the message to all connected frontends so the UI can update in real time. It then attempts to resolve the device to a Git branch using a static configuration file (*deviceMappings.json*). Matching is tolerant to cosmetic differences (e.g., symbol/spacing) by normalizing both the configured and observed device names with a simple regular expression comparison. If a mapping is found, the backend constructs a shell command to target the development repository directory and performs Git operations in that context. With the updated requirement, the target path is */targetRepository*. The intended flow is to change directory into a */targetRepository, perform git reset --hard HEAD* to discard local changes, then checkout to a targeted mapped branch and pull new changes.

These commands are executed via Node’s *child_process.exec*, with errors surfaced back to the WebSocket client as *{ type: 'error', message }* if anything fails (e.g., invalid branch). This keeps the switching logic centralized on the backend and decoupled from the scanner.

On the client side, the React app opens a WebSocket to the backend and maintains an in-memory list of observed devices. Each incoming payload updates or inserts a device record, storing the latest timestamp and distance proxy. Devices are rendered in descending proximity (closest first) by sorting on *distanceProxyInMetres*. A dedicated Bluetooth mapping UI (*public/BluetoothDevices.js*) supports manual mapping flows when needed (e.g., associating selected device names to branches per engineer), but the proxemics path works automatically when a device appears that already exists in *deviceMappings.json*.

This design is deliberately simple and resilient: presence detection happens out-of-process; transport is real time via WebSockets; resolution is a pure lookup; and the branch switch is a short-lived Git operation scoped to */targetRepository*. The approach ensures that walking into range with a known device deterministically checks out the intended branch, while the UI provides immediate feedback on detections and their proximity.

5.2.2 Automatic Context Handover

Approaching the display loaded the relevant branch and last commit associated with the approaching engineer, eliminating the need for manual navigation.

Visual encodings were carried over from Iteration 1's FCG prototype. The nodes represented functions or files, edges represented call dependencies, and colours indicated changed elements. This provided continuity while introducing new proxemic features.

5.3 Method

5.3.1 Participant Recruitment

5 pairs of professional software engineers were recruited to evaluate the proxemic prototype.

Each pair consisted of a junior engineer and a senior engineer. Each pair of engineers had prior experience with collaborative development and code review in industry settings, and were invited to participate in a code review session**,** reflecting the synchronous collaboration scenarios that proxemic triggers are designed to support.

5.3.2 Study Design

The study followed a **task-based observational format**. Pairs of participants were asked to conduct review activities under two subtasks:

1. **Task 1**: Using manual selection of repository, branch and commit to visualise the FCG and jointly discuss the structural impact of the change using the FCG visualisation (as in design iteration 1)
2. **Task 2**: 1 engineer wearing a smart watch approaches the display of a seated engineer, automatically triggered the loading of the relevant branch, commit, and function call graph.

Participants were encouraged to use think-aloud protocols, verbalising their reasoning and impressions as they worked through each condition.

5.3.3 Data Collection

Data was collected qualitatively through a survey to compare the manual flow in task 1 and with the proxemic flow in task 2.

5.4 Results

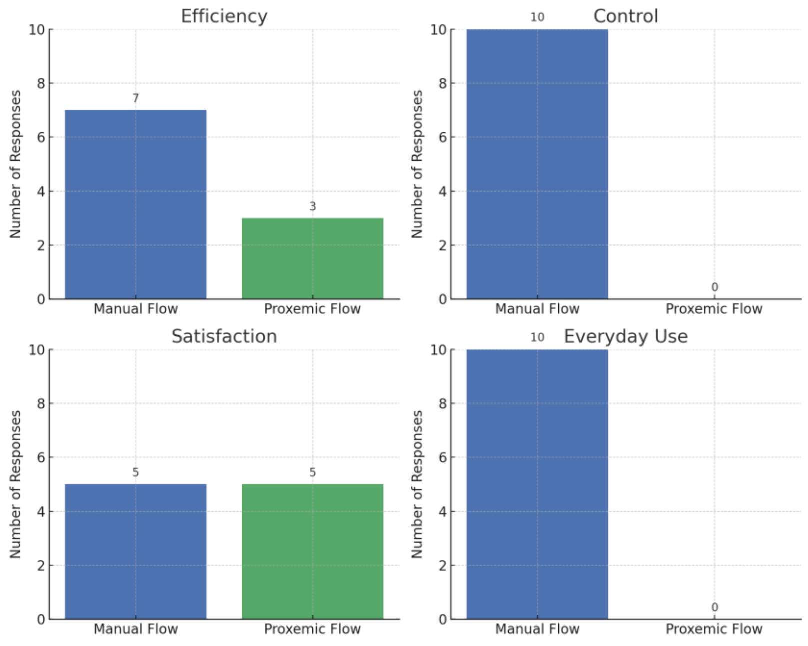


Figure 8. Experiment Results of FCG Flow vs Proxemic Flow. The scores are representing the number of engineers who voted for that characteristic.

- **Efficiency:** Most respondents judged the manual interface more efficient, though a minority found proxemics faster for context switching.
- **Sense of Control:** Every participant reported that the manual interface provided a greater sense of control, underscoring trust and predictability as key factors in review workflows.
- **Satisfaction:** Satisfaction was evenly split. While some participants valued the seamlessness of proxemics, others preferred the stability of manual control.

- **Everyday Use:** All participants indicated that for daily work, they would rely on the manual flow. Proxemic triggers were seen as situational rather than routine.

In addition to comparative interface questions, participants rated the proxemic prototype on a six-item Likert scale covering transparency, intuitiveness, predictability, learnability, efficiency, and complexity. Overall, responses indicated moderate agreement that the system kept participants informed of its behaviour and that the interface was intuitive.

However, ratings for predictability and efficiency were more mixed, with several respondents signaling concern that the automation introduced unpredictability. While participants acknowledged that the proxemic interface avoided unnecessary complexity, they were less convinced that it consistently supported fast and efficient work. These results align with the forced-choice questions: proxemics was perceived as *satisfying and novel*, but trust and predictability concerns limited its appeal for everyday use.

5.5 Reflections on Application of Proxemics to Code Reviews

One interesting observation was that the overall demographic preference for the FCG flow over the proxemic flow was independent of seniority level or engineering specialisation. All participants playing the role of the seated engineer expressed frustration at unwanted interruptions and screen changes:

- *“I don't think I will use this because I don't want people to trigger screen changes just because they walked past me. It could be quite annoying if not managed well.”*
- *“I suppose it can frustrate the seated engineer if there are unwanted automatic screen changes.”*
- *“Unwanted screen takeovers, especially when I’m in the middle of a meeting.”*

Another reason for the lack of support for the proxemic flow was that the physical office layouts of these engineers could not afford proxemic transitions:

- *“Engineers sit next to each other in my workplace.”*
- *“There is no space in my office to use this - my engineering manager sits right behind me.”*

Thus, proxemic triggers alone do not constitute a viable replacement for established review workflows. Iteration 2 confirmed that while proxemics can reduce coordination overhead in synchronous collaboration, it does not address the expertise and knowledge gaps that emerged consistently in both the literature review (Sections 2.1-2.5) and earlier experiments. Even with improved context handover, less experienced reviewers may still struggle to interpret vulnerabilities or provide meaningful feedback. This gap motivates Iteration 3: the exploration of a simplified vulnerability scanner, designed to reduce knowledge disparities during code reviews without disrupting workflow efficiency, thereby addressing RQ2.

Chapter 6. Design Iteration 3: Lightweight Vulnerability Scanner

6.1 Motivation

The first two design iterations explored comprehension (Function Call Graphs) and coordination (proxemic triggers), but neither resolved the persistent expertise gap in code review. Less experienced engineers often struggle to identify vulnerabilities or assess risky patterns, limiting their contributions.

As noted in Section 2.5, existing SAST tools are noisy, rigid, and poorly integrated into day-to-day workflows. They are typically designed for compliance in CI/CD pipelines rather than lightweight, collaborative review, making them unsuitable for supporting novices.

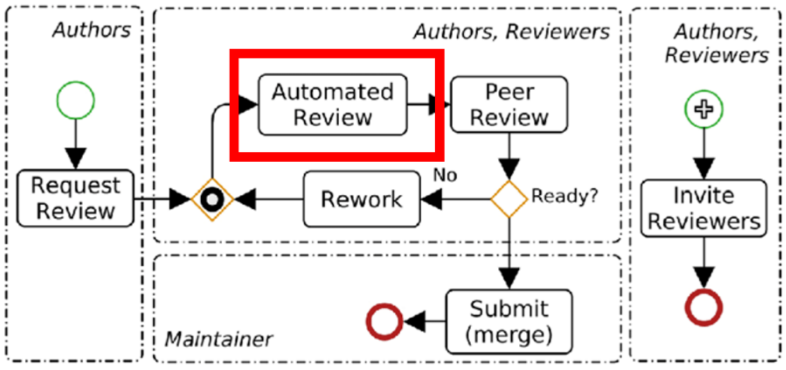


Figure 9. A diagram of Code Review Process adapted from Santos, Eduardo & Nunes, Ingrid. (2018).

A survey done with a full-stack software engineer with 3 years of experience in the civil service sector corroborated similar concerns - during peak periods when the deadline of large feature releases approaches, engineers often rely on co-located, synchronous code reviews, as the automated checks are not helpful in catching vulnerabilities or validating code quality.

This motivated Iteration 3: a simplified vulnerability scanner embedded in the review workflow. By surfacing clear, lightweight warnings without overwhelming detail, the prototype aimed to reduce knowledge barriers for junior reviewers while maintaining workflow efficiency - directly addressing **RQ3**.

6.2 Prototype Description

6.2.1 Technical Architecture

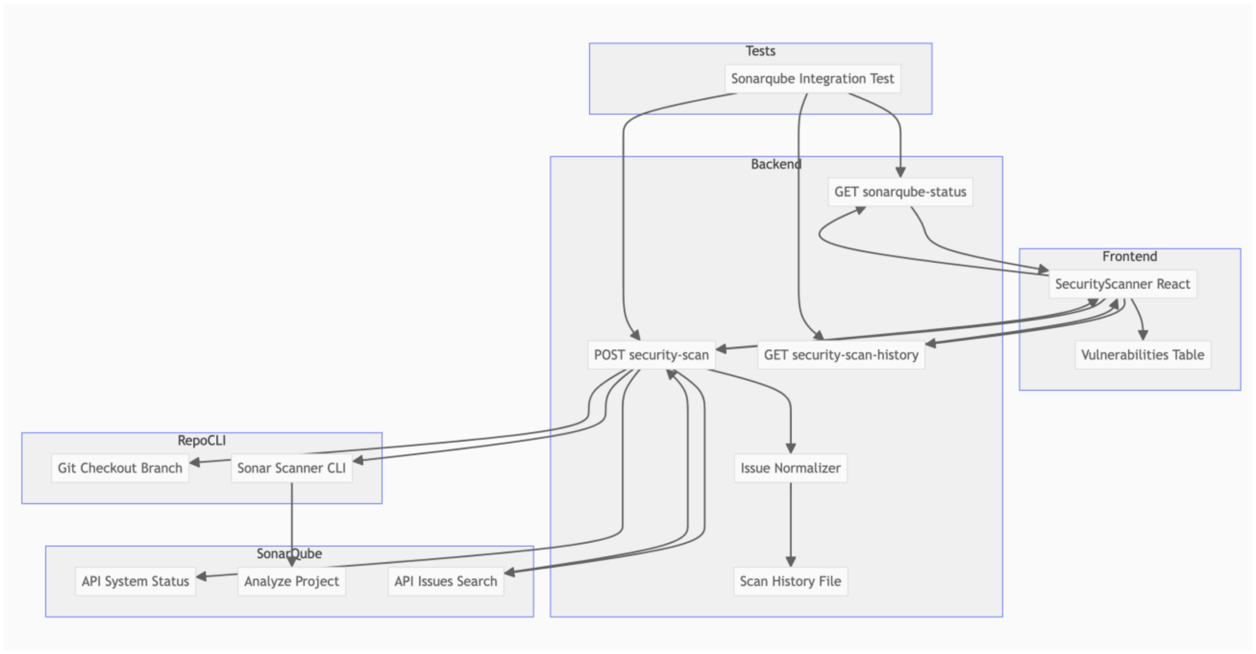


Figure 10. Technical Architecture of Lightweight Vulnerability Scanner

The prototype was implemented as a lightweight static analysis tool integrated into the review interface. It scanned code diffs for common patterns of insecure practice (e.g., unsanitised inputs, insecure API calls) and surfaced results directly within the review environment.

6.2.2 Features

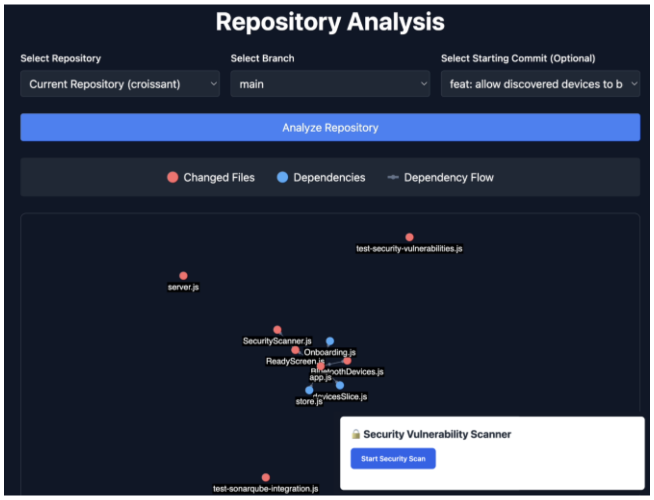


Figure 11. Positioning of the lightweight security scanner

Situated at the bottom right of the screen, it acts as a streamlined, one-click widget that analyzes exactly the repository and branch you select. From the UI, starting a scan sends a single request to the backend, which checks out the target branch and invokes SonarScanner with a unique project key.

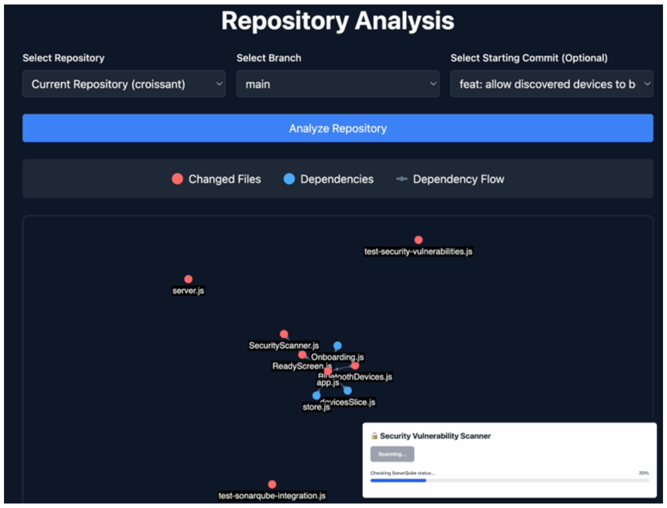
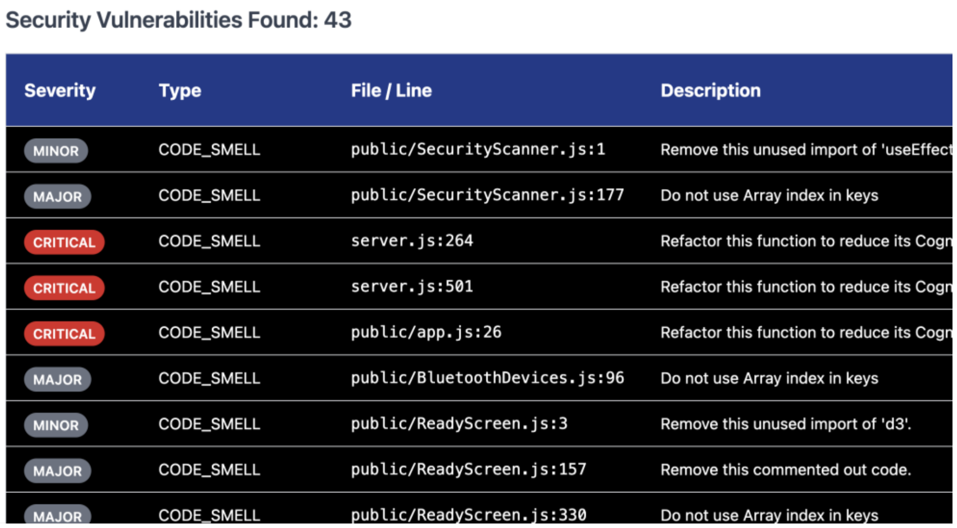


Figure 12. Lightweight scanner in action

While analysis runs, the frontend shows live progress and rotating status messages, then renders an expandable view of a structured results table on completion: findings are color-coded by severity and include the rule type, file and line, short description, rule ID, first-detected timestamp, and status.


Security Vulnerabilities Found: 43

Severity	Type	File / Line	Description
MINOR	CODE_SMELL	public/SecurityScanner.js:1	Remove this unused import of 'useEffect
MAJOR	CODE_SMELL	public/SecurityScanner.js:177	Do not use Array index in keys
CRITICAL	CODE_SMELL	server.js:264	Refactor this function to reduce its Cogn
CRITICAL	CODE_SMELL	server.js:501	Refactor this function to reduce its Cogn
CRITICAL	CODE_SMELL	public/app.js:26	Refactor this function to reduce its Cogn
MAJOR	CODE_SMELL	public/BluetoothDevices.js:96	Do not use Array index in keys
MINOR	CODE_SMELL	public/ReadyScreen.js:3	Remove this unused import of 'd3'.
MAJOR	CODE_SMELL	public/ReadyScreen.js:157	Remove this commented out code.
MAJOR	CODE_SMELL	public/ReadyScreen.js:330	Do not use Array index in keys

Figure 13. Vulnerabilities and code improvements detected by lightweight security scanner

One key characteristic of this lightweight scanner is that it provides a description of the vulnerability detected in simple, jargon-free English and directs the engineer to the exact line(s) of concern. For faster remediation, the backend attaches succinct fix suggestions mapped from rule IDs. Each scan appends JSON metadata to a local history file, including timestamp, repo path, branch, duration, *vulnerabilitiesFound, hotspotsFound, totalFindings*, and raw scanner output, so recent runs are retrievable via an API.

Configuration is centralized in sonar-project.properties: it scopes sources, applies broad exclusions (e.g., node_modules, dist, migrations), sets encoding and server URL, and explicitly enables JavaScript security analysis. Operationally, the integration is resilient to empty or malformed API responses (returning an empty issues list rather than failing) and surfaces clear errors for common faults.

6.3 Method

6.3.1 Participant Recruitment

Two software engineers participated in the evaluation of the simplified vulnerability scanner.

Participant C1 was a junior engineer with 2 years of frontend engineering experience and limited security experience.

Participant C2 other a senior engineer with 4 years of backend engineering experience at a social media conglomerate.

6.3.2 Study Design

The evaluation was conducted through semi-structured interviews following hands-on use of the prototype. Each participant was asked to interact with the scanner while reviewing code diffs and then reflect on its usefulness, clarity, and impact on their workflow.

6.3.3 Data Collection

Data consisted of interview transcripts capturing participants' perceptions of:

- The clarity and interpretability of scanner warnings.
- The extent to which the tool supported or disrupted their normal review flow.
- Differences in how junior vs. senior engineers relied on or dismissed scanner feedback.

6.3.4 Data Analysis

Interview transcripts were analysed thematically. Codes were grouped around comprehension support**,** workflow integration, and trust in results**,** with attention to contrasts between the junior and senior participants.

6.4 Results

6.4.1 Perceived Usefulness

Both participants agreed that security checks are important, particularly for developers without deep security expertise. C1, a frontend engineer, noted: *"I've never really dealt with vulnerabilities… and I don't know any other way, I guess I would use it"*. This highlights the scanner's potential value as an onboarding aid for juniors who lack familiarity with security practices.

C2 echoed the need but questioned why such checks were not already part of continuous integration(CI) pipelines: *"Why is it not like a GitHub triggered CI... it just runs constantly"*. For him, the key value of a local scanner would be quick, lightweight feedback before pushing to CI: *"If this thing takes like 30 seconds to run, I'm not going to wait." And specified that such checks should take "less than five seconds, 90% of the time"*. This reinforces that efficiency is critical for adoption; a scanner must operate in the background or provide near-instant feedback to avoid workflow disruption.

6.4.2 Form Factor and Integration

C1 was flexible on delivery form, suggesting a web interface, desktop client, or VS Code extension would all be acceptable: *"The web is fine. The desktop app is also fine, and I guess like a VS Code extension or something"*.

C2, however, was more sceptical of introducing new local tools when CI systems already provide automated scanning, indicating that integration with existing developer environments is crucial for adoption.

6.5 Reflection

The interviews provide a clear perspective on **RQ3**: Can a simplified vulnerability scanner reduce knowledge and experience gaps in collaborative reviews without disrupting workflow efficiency?

For junior engineers (C1), the scanner holds clear value as an onboarding and support tool, helping surface vulnerabilities that they would otherwise overlook. This suggests that simplified, natural-language warnings can help bridge expertise gaps and encourage more meaningful participation in reviews.

For senior engineers (C2), the scanner's usefulness is conditional. They emphasised that adoption depends on speed (<5 seconds) and seamless integration into existing workflows. In their view, a lightweight scanner has merit only as a pre-CI check, not as a replacement for enterprise-scale security pipelines.

Together, these insights suggest that simplified scanners should not attempt to replicate the breadth of CI-integrated SASTs. Instead, their contribution lies in providing lightweight, actionable guidance to less experienced reviewers, while offering seniors an optional but efficient pre-commit safeguard. This iteration therefore underscores the need to design vulnerability scanning as an augmentation to collaborative review, not a disruption - directly addressing RQ3.

Chapter 7. Conclusion

7.1 Revisiting the Research Questions

This thesis set out to investigate how collaboration in code reviews can be improved through novel interaction techniques, guided by three research questions.

RQ1 asked: *How do engineers perceive the value of code visualization during code reviews?*

Findings from Iteration 1 showed that Function Call Graphs (FCGs) improved comprehension by externalising structural context, reducing time-to-understanding, and easing onboarding. Participants valued the visualisation for situating diffs within a larger system. At the same time, scalability and visual noise in large graphs emerged as limitations, pointing to the need for targeted integration of FCGs into review tools.

RQ2 asked: *Can proxemic triggers improve synchronous code reviews compared to existing workflows?*

Iteration 2 revealed that proxemic interaction is compelling in theory but limited in practice. While participants appreciated seamless transitions and reduced explanation overhead in specific contexts (e.g., demos, quick onboarding), quantitative results showed that the manual flow was overwhelmingly preferred for efficiency, control, and daily use. Proxemic triggers may therefore complement but cannot replace conventional workflows.

RQ3 asked: *Can a simplified vulnerability scanner reduce knowledge and experience gaps in collaborative reviews without disrupting workflow efficiency?*

Iteration 3 demonstrated that a lightweight scanner can bridge expertise gaps for juniors, providing accessible, natural-language warnings that increase participation. Seniors, however, demanded sub-five-second performance and seamless integration, seeing the tool primarily as a pre-commit safeguard rather than a CI replacement. The scanner's role is therefore as an augmentation rather than a disruption to collaborative workflows.

7.2 Contributions of the Thesis

This work makes three primary contributions:

1. **Empirical Findings:**
    - Surveys and interviews revealed persistent challenges in comprehension, coordination, and expertise distribution during code reviews.
    - Iterative evaluations showed how visualisation, proxemics, and simplified scanning each addressed these gaps in situated but limited ways.
2. **Design Contributions:**
    - Development of a web-based FCG visualisation integrated with Git.
    - A proxemic trigger system linking Bluetooth sensing to review context.
    - A lightweight vulnerability scanner embedded in the review interface.
      These prototypes serve as *intermediate knowledge artefacts*, highlighting the design trade-offs between automation and control, workflow fit and novelty, and comprehensiveness versus simplicity.
3. **Methodological Contribution:**
    - Demonstration of Research through Design (RtD) in software engineering research. Iterative cycles of problem exploration, prototyping, and reflection revealed insights not accessible through controlled experiments alone, including socio-technical tensions around trust, transparency, and adoption.

7.3 Limitations

Several limitations constrain the scope of the findings:

- **Scale:** Participant numbers were modest, with each iteration engaging only a handful of engineers. Findings should be seen as exploratory rather than generalisable.
- **Fidelity:** Prototypes were functional but not production-grade. Performance issues and visual scalability limited ecological validity.

- **Context:** Studies were conducted in controlled or simulated settings, not deployed in long-term, real-world team environments.

7.4 Future Work

Building on these limitations, several directions for future research emerge. First, there is a need for **longitudinal field studies** to evaluate how Function Call Graphs, proxemic triggers, and simplified vulnerability scanners are adopted and integrated into real-world production workflows over time. Such studies would provide stronger evidence of their sustained utility and highlight challenges that short-term evaluations cannot capture.

Secondly, future work should address the **scalability of visualisations**. Techniques such as layering, filtering, and AI-assisted summarisation could help reviewers manage complexity and avoid visual overload when analysing large codebases.

Thirdly, proxemic triggers would benefit from refinement through **stronger user control mechanisms**. Features such as opt-in confirmations or adaptive thresholds could reduce unpredictability, balancing automation with the trust and transparency that reviewers require.

Fourth, the **simplified vulnerability scanner** could be extended to integrate more tightly with CI/CD pipelines while preserving its lightweight, accessible outputs for junior engineers. This would allow the tool to complement enterprise-scale security infrastructure while still serving as a bridge for less experienced reviewers.

Finally, the rapid advancement of **large language models (LLMs)** presents an opportunity to complement both visualisation and scanning. LLMs may help bridge comprehension and expertise gaps in more flexible ways, for instance by generating contextual explanations or summarising vulnerabilities in natural language.

Bibliography

Ballendat, T., Marquardt, N., & Greenberg, S. (2010). Proxemic interaction: designing for a proximity and orientation-aware environment. ACM International Conference on Interactive Tabletops and Surfaces, 121–130. doi:10.1145/1936652.1936676

Greenberg, S., Marquardt, N., Ballendat, T., Diaz-Marino, R., & Wang, M. (2011). Proxemic interactions: the new ubicomp? Interactions, 18(1), 42–50. doi:10.1145/1897239.1897250

Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. 2013 35th International Conference on Software Engineering (ICSE), 712–721. doi:10.1109/ICSE.2013.6606617

Krause-Glau, A., Damerau, L., Hansen, M., & Hasselbring, W. (2024a, October). Visual Integration of Static and Dynamic Software Analysis in Code Reviews via Software City Visualization. 2024 IEEE Working Conference on Software Visualization (VISSOFT), 144–149. doi:10.1109/VISSOFT64034.2024.00028

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. Proceedings of the 11th Working Conference on Mining Software Repositories, 192–201. doi:10.1145/2597073.2597076

Rigby, P. C., & Bird, C. (2013). Convergent contemporary software peer review practices. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 202–212. doi:10.1145/2491411.2491444

Sadowski, C., Söderberg, E., Church, L., Sipko, M., & Bacchelli, A. (2018). Modern code review: a case study at google. Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, 181–190. doi:10.1145/3183519.3183525

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., & Godfrey, M. W. (2015). Investigating code review quality: Do people and participation matter? 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 111–120. doi:10.1109/ICSM.2015.7332457

Herbsleb, J., & Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. Software Engineering, IEEE Transactions On, 29, 481–494. doi:10.1109/TSE.2003.1205177

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering, 21(5), 2146–2189. doi:10.1007/s10664-015-9381-9

Badampudi, D., Unterkalmsteiner, M., & Britto, R. (2023). Modern Code Reviews—Survey of Literature and Practice. ACM Trans. Softw. Eng. Methodol., 32(4). doi:10.1145/3585004

Tsay, J., Dabbish, L., & Herbsleb, J. (2014). Influence of social and technical factors for evaluating contribution in GitHub. Proceedings of the 36th International Conference on Software Engineering, 356–366. doi:10.1145/2568225.2568315

Baltes, S., & Diehl, S. (2018). Towards a theory of software development expertise. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 187–200. doi:10.1145/3236024.3236061

Grønbæk, J. E., Knudsen, M. S., O'Hara, K., Krogh, P. G., Vermeulen, J., & Petersen, M. G. (2020, April). Proxemics beyond Proximity: Designing for Flexible Social Interaction Through Cross-Device Interaction. Conference on Human Factors in Computing Systems - Proceedings. doi:10.1145/3313831.3376379

Lunding, M. S., Grønbæk, J. E. S., Grymer, N., Wells, T., Houben, S., & Petersen, M. G. (2023). Reality and Beyond: Proxemics as a Lens for Designing Handheld Collaborative Augmented Reality. Proceedings of the ACM on Human-Computer Interaction, 7(ISS). doi:10.1145/3626463

Grabinger, L., Hauser, F., Wolff, C., & Mottok, J. (2024). On Eye Tracking in Software Engineering. SN Computer Science, 5(6), 729. doi:10.1007/s42979-024-03045-3

Császár, I.-A., & Slavescu, R. R. (2020). Interactive call graph generation for software projects. 2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP), 51–58. doi:10.1109/ICCP51029.2020.9266149

Padoan, F., Santos, R. D. S., & Medeiros, R. P. (2024). Charting a Path to Efficient Onboarding: The Role of Software Visualization. Proceedings of the 2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering, 133–143. doi:10.1145/3641822.3641873

Krause-Glau, A., Damerau, L., Hansen, M., & Hasselbring, W. (2024b). Visual Integration of Static and Dynamic Software Analysis in Code Reviews via Software City Visualization. 2024 IEEE Working Conference on Software Visualization (VISSOFT), 144–149. doi:10.1109/VISSOFT64034.2024.00028

Marcus, A., Feng, L., & Maletic, J. I. (2003). 3D representations for software visualization. Proceedings of the 2003 ACM Symposium on Software Visualization, 27-ff. doi:10.1145/774833.774837

Lanza, M., & Ducasse, S. (2003). Polymetric views - a lightweight visual approach to reverse engineering. IEEE Transactions on Software Engineering, 29(9), 782–795. doi:10.1109/TSE.2003.1232284

Li, C., Pei, Y., Shen, Y., Lu, J., Fan, Y., Linghu, X., … Wang, K. (2024). PyVisVue3D3: Python visualization from hierarchy tree to call graph. SoftwareX, 26. doi:10.1016/j.softx.2024.101689

Khan, T., Barthel, H., Ebert, A., & Liggesmeyer, P. (2011). Visualization and evolution of software architectures. OpenAccess Series in Informatics, 27, 25–42. doi:10.4230/OASIcs.VLUDS.2011.25

Salis, V., Sotiropoulos, T., Louridas, P., Spinellis, D., & Mitropoulos, D. (2021). PyCG: Practical Call Graph Generation in Python. Proceedings of the 43rd International Conference on Software Engineering, 1646–1657. doi:10.1109/ICSE43902.2021.00146

Marquardt, N., Diaz-Marino, R., Boring, S., & Greenberg, S. (2011). The proximity toolkit: prototyping proxemic interactions in ubiquitous computing ecologies. Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, 315–326. doi:10.1145/2047196.2047238

Christakis, M., & Bird, C. (2016, August). What developers want and need from program analysis: an empirical study. 332–343. doi:10.1145/2970276.2970347

Schreiber, A., Sonnekalb, T., & Kurnatowski, L. V. (2021). Towards Visual Analytics Dashboards for Provenance-driven Static Application Security Testing. Proceedings - 2021 IEEE Symposium on Visualization for Cyber Security, VizSec 2021, 42–46. doi:10.1109/VizSec53666.2021.00010

Yang, J., Tan, L., Peyton, J., & A Duer, K. (2019). Towards Better Utilizing Static Application Security Testing. 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 51–60. doi:10.1109/ICSE-SEIP.2019.00014

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? 2013 35th International Conference on Software Engineering (ICSE), 672–681. doi:10.1109/ICSE.2013.6606613

Saleh, S., Madhavji, N., & Steinbacher, J. (2024). A Systematic Literature Review on Continuous Integration and Deployment (CI/CD) for Secure Cloud Computing. Proceedings of the 20th International Conference on Web Information Systems and Technologies, 331–341. doi:10.5220/0013018500003825

Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., & Gall, H. C. (2018, March). Context is king: The developer perspective on the usage of static analysis tools. 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 38–49. doi:10.1109/SANER.2018.8330195

Wiberg, M. (2014). Methodology for materiality: interaction design research through a material lens. Personal Ubiquitous Comput., 18(3), 625–636. doi:10.1007/s00779-013-0686-7

Gaver, W. (2012). What Should We Expect From Research Through Design? Conference on Human Factors in Computing Systems - Proceedings. doi:10.1145/2207676.2208538

Zimmerman, J., Forlizzi, J., & Evenson, S. (2007). Research through design as a method for interaction design research in HCI. doi:10.1145/1240624.1240704

Santos, E., & Nunes, I. (2018). Investigating the effectiveness of peer code review in distributed software development based on objective and subjective data. Journal of Software Engineering Research and Development, 6. doi:10.1186/s40411-018-0058-0