HeatMetro: A 3D Code Visualization for Test Debt Analysis in Evolving Systems

Braulio Nayap Maldonado Casilla[1], Sergio Mogollon[1], Paul Parizaca-Mozo[1], and Edgar Sarmiento-Calisaya[1]

[1]Universidad Nacional de San Agustín de Arequipa
Arequipa – Peru

{bmaldonadoc, smogollon, pparizaca, esarmientoca}@unsa.edu.pe

Abstract. *Traditional test coverage metrics often fail to reveal test debt. Additionally, it is difficult to manually uncover code critical for testing, particularly in complex and rapidly changing components of large systems. However, few tools help developers visually explore test debt. In this work, we introduce HeatMetro, a web-based implementation of the code city metaphor that creates an interactive 3D software visualization and operationalizes a context-aware code coverage criterion for the visual analysis of test debt in large Java and Go projects. The feasibility of HeatMetro is evaluated through a study with developers and projects, which indicates promising results, confirming the tool's effectiveness in identifying and explaining critical components for testing.*

1. Introduction

Developing and evolving large software systems often entails increased complexity and continuous evolution. The fragility of these systems was evident in recent incidents, such as the "CrowdStrike Falcon failure", which demonstrates that defective code can trigger catastrophic consequences [Amusuo et al. 2025]. Furthermore, the cost of poor software quality has reached multi-billion-dollar levels, driven primarily by the difficulty of maintaining legacy systems [Krasner 2021]. As systems evolve, the volume of information generated by frequent changes to support new features, bugs, and enhancement requests increases. This phenomenon, also known as *code churn*, refers to the volume and frequency of code changes over time, and it is strongly correlated with defect introduction [Hilton et al. 2018]. Making the management of effective test suites challenging.

Test coverage metrics are commonly used to assess test suite quality. However, traditional global coverage metrics often fail to reveal the accumulation of test debt [Aragão et al. 2022], leading to a false sense of security. For example, a project with 90% global coverage may still contain significant, untested, or buggy code, particularly in large projects. Therefore, a high global coverage percentage can hide significant defects in critical components. This disconnection often occurs because standard tools produce static reports that obscure the effectiveness of tests in critical components, particularly in complex and rapidly changing components of large systems. Aragon et al. [Aragão et al. 2022] attribute this phenomenon to inadequate decisions, such as the lack of tests in core features, revealing that the most dangerous technical debt resides precisely at the intersection of these critical components and insufficient coverage – *test debt*.

Recent studies have investigated how coverage changes with code evolution, particularly the coverage of patches (changed statements) [Hilton et al. 2018]

[Miranda et al. 2025]. On the other hand, the industry has identified a need for a new coverage metric to guide developers in identifying which code must be tested and which should be left untested [Ivankovic et al. 2024] [Brandt and Ramírez 2025]. However, most existing approaches focus on coverage metrics derived from local, file-based, or 2D views of a project, and rarely consider historical information or provide explanations. As code evolves at an increasing rate, software developers face the challenge of managing critical components that have accumulated test debt. This effort is time-consuming and requires deep knowledge of the project structure, hindering effective comprehension, visualization, and exploration [Strandberg 2017] [Miranda et al. 2025]. Therefore, it is essential to prioritize testing of components with a higher probability of leading to software execution failures [Hernandes et al. 2025].

To address these challenges, we introduce HeatMetro, a web-based implementation of the code city metaphor [Wettel and Lanza 2008] for visual analysis of test debt in Java and Go projects. Our tool is based on a novel *context-aware code coverage* criterion that relies on code coverage, source code metrics, the abstract syntax tree (AST), and code churn (evolutionary data) information of a project to create an interactive 3D software visualization in an automated way, then, highlight and explain critical components that should be tested, effectively separating real test debt from the noise of global metrics. To support future research, we make our approach publicly available on https://github.com/ShinjiMC/HeatMetro-Tool.

We demonstrate the feasibility of HeatMetro through usage scenarios involving 6 open-source projects and an empirical study with 12 developers. The results are encouraging, reporting a high accuracy (F1-scores > 0.80) and a System Usability Scale (SUS) score of 79.0, confirming the tool's effectiveness in identifying quality risks and efficiently prioritizing maintenance efforts. Therefore, our research advances software engineering by providing a novel approach to handle the complexities of software evolution and test debt analysis from an architectural perspective in highly evolving large projects.

This work is organized as follows: Section 2 describes related work. Section 3 details the approach and architecture of HeatMetro. Section 4 presents usage scenarios from real-world Java and Go projects. Section 5 presents developers' feedback on our tool and discusses the results. Finally, we conclude and outline future work in Section 6.

2. Related Work

Several tools have been proposed to assist developers in understanding different software aspects by providing visual representations, such as graphs, matrices, and 3D models, as reported in [Chotisarn et al. 2020]. These tools help map, track, and analyze software structure, dependencies, and history.

In the context of repository evolution, tools such as Git-Truck [Højelse et al. 2022] visualize the hierarchical evolution of Git repositories using treemaps. Although this approach is useful for specific authorship analyses, its 2D representation presents scalability and readability limitations in large systems. On the other hand, Tim [Strandberg et al. 2022] was developed as a heatmap-based tool for exploring test results in continuous integration environments, though its approach lacks deep integration with the system's architecture.

Regarding structural representation, CodeCity [Wettel and Lanza 2008] is a pop-

ular metaphor for representing large software systems as 3D cities, supporting software evolution from an architectural perspective by modeling classes as buildings and packages as districts. This technique has been shown to reduce the time required for software comprehension tasks by leveraging spatial orientation. Following this line, the metaphor was adapted for the web ecosystem with JSCity [Viana et al. 2015], a tool that visualizes JavaScript systems in 3D to support restructuring and refactoring tasks. Similarly, GoCity [Brito et al. 2019] was developed to support Go developers. To support collaboration, ExplorViz [Hasselbring et al. 2020] was proposed to facilitate efficient program comprehension in software development teams. Recently, this metaphor has also been implemented in interactive software visualization tools using Virtual Reality (VR) in an immersive 3D environment [Romano et al. 2019]. More recently, [Mortara et al. 2024] proposed the VariCity approach, an extension of code city, to visualize zones with a high density of object-oriented variability implementations, thereby facilitating the identification of quality-critical classes that exhibit potential technical debt.

In the context of coverage analysis, [Dreef et al. 2023] proposed Morpheus, a matrix-based approach for exploring the relationship between tests and methods under test. While this tool improves precision in test suite analysis by detailing which tests cover which methods, the matrix view loses the intuitive spatial reference of the system architecture, making it difficult to detect global architectural patterns. Therefore, recent efforts have shifted towards defining more dynamic coverage criteria. [Ivankovic et al. 2024] introduced *productive coverage*, a novel approach to code coverage that guides developers to uncovered code that should be tested by (unit) tests, based on its utility and execution context. [Brandt and Ramírez 2025] proposed *refined code coverage*, which uses machine learning models trained on source code metrics and AST data to predict the need for testing specific lines. While these approaches improve metric precision, they typically rely on tabular reports or complex machine learning pipelines and lack an interactive visual interface for exploring these risks within the system's architecture.

Existing approaches mainly focus on static structures or specific implementations details, often overlooking historical changes and dynamic coverage information. During test exploration, these tools usually depend on tabular reports that lack spatial context, making it harder to navigate complex projects efficiently. If data are not presented well, time and attention may be wasted when making critical decisions [Ahmad et al. 2022]. However, HeatMetro integrates the code city metaphor with an evolutionary timeline and operationalizes a *context-aware code coverage* criterion for visual analysis of test debt in large projects, highlighting and explaining critical components that should be tested.

3. HeatMetro

Instead of developer intuition, our *context-aware code coverage* relies on structural and evolutionary (historical) information about a project to determine which component is critical: code coverage, source code metrics, the AST, and code churn. HeatMetro implements a continuous inspection workflow that decouples data mining from visual representation, thereby transforming heterogeneous metrics into an interactive 3D city model, as illustrated in Figure 1. The solution adopts a separation-of-concerns pattern, divided into two operational phases: historical data synthesis (*Builder*) and visualization deployment (*Deployer*). This design isolates intensive processing from the web infrastructure, thereby optimizing performance and enabling analysis of large projects in polyglot environments.

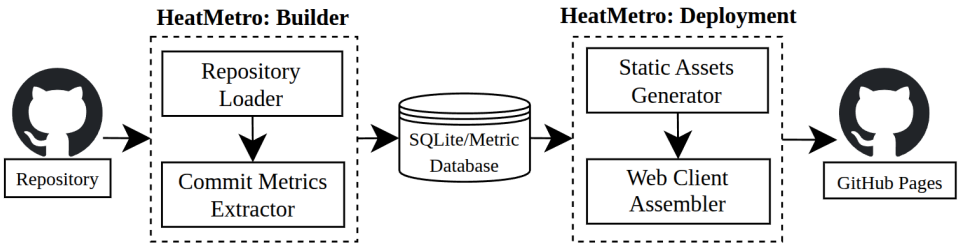


Figure 1. General Flow Diagram of HeatMetro

3.1. Construction and Extraction Module (Builder)

The HeatMetro Builder is the platform's analytical core. Its primary function is to implement our *context-aware code coverage* by extracting and consolidating structural and evolutionary metrics. The workflow, as depicted in Figure 2, begins when the user specifies the GitHub repository URL, the analysis scope (branch or tag), and provides the static analysis configuration file `sonar-project.properties`. This file is fundamental because it enables data synchronization with SonarCloud[1] and defines exclusion rules that clearly distinguish source code from test files. To ensure efficiency, the module uses the GitHub GraphQL API, implementing a shallow clone and selective mining strategy that restricts analysis to commits that involve source code changes, excluding documentation.

To support multiple ecosystems, a detection mechanism based on the Strategy Pattern is implemented. The system inspects the repository root and dynamically loads the corresponding analyzers based on the system building tool used in the project (`pom.xml` or `build.gradle` for Java, and `go.mod` for Go projects).

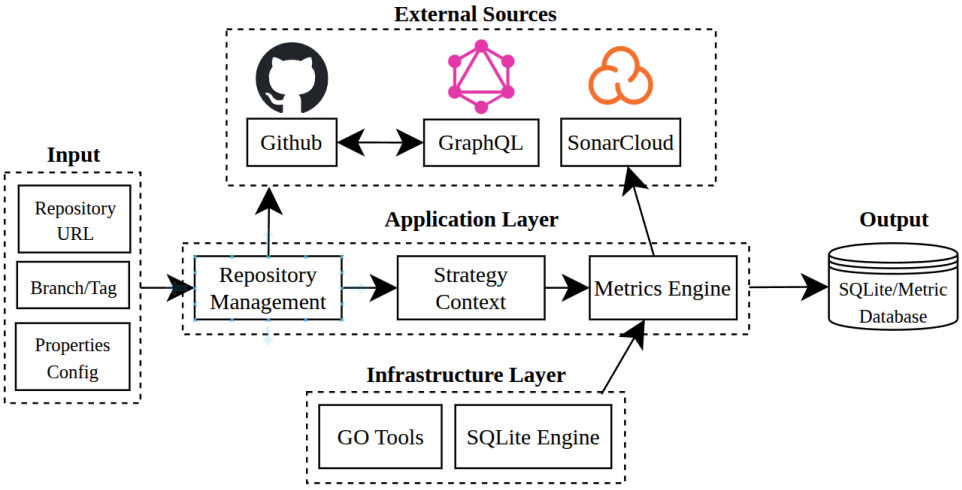


Figure 2. Internal Architecture of HeatMetro Builder

The extraction process implements a hybrid system to compute the required indicators for the *context-aware code coverage*. On one hand, it establishes a bridge with SonarCloud to obtain standardized test coverage and complexity metrics. Simultaneously, local computation engines analyze the version history to compute code churn and authorship fragmentation, and construct the AST to determine the city topology.

[1] `https://sonarcloud.io/`

To identify critical components—those requiring mandatory testing—the algorithm applies a relevance filter based on industry thresholds configured in the system:

- **Structural Dimension:** Components whose Coupling Between Objects (CBO) exceeds a value of 9 [Shatnawi 2010] or whose Cyclomatic Complexity exceeds 10 points are marked as risky [McCabe and Watson 1996].
- **Evolutionary Dimension:** Files with a code churn greater than 20% of the code base [Walkinshaw and Minku 2018, Ostrand et al. 2004] or an authorship fragmentation (Gini Coefficient) greater than 0.75 are identified as volatile [Vasa et al. 2009, Avelino et al. 2016].

By combining these datasets, the *Builder* generates a unified data artifact that isolates the real test debt, discarding noise from trivial components and persisting the information in a self-contained SQLite database for subsequent visualization.

3.2. Integration and Deployment Flow (Deployer)

To transform and publish the results, the deployment module was designed as a lightweight, automated process orchestrated via CI/CD workflows (e.g., GitHub Actions). Its objective is to convert the relational model generated into an accessible web experience without requiring server-side processing, as illustrated in Figure 3.

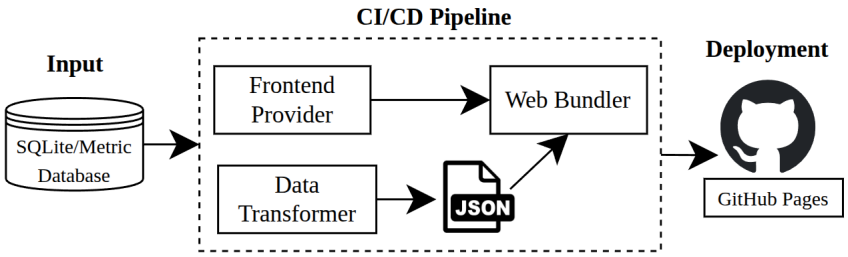


Figure 3. Architecture of the automated deployment flow

The process starts when the database file is detected (`.db`). A conversion engine serializes the data into JSON documents, generating two key structures: structural snapshots to reconstruct project hierarchy (buildings and districts) and transposed time series for trend analysis. Concurrently, the system injects this data into the visualization client (*Frontend Provider*) and forces a synchronization with the publication branch (`gh-pages`). As a result, the interactive report is immediately available via a public URL, enabling developers to visually assess the test debt exposed by our *context-aware code coverage*, thereby facilitating the fast detection of critical code areas that require mandatory coverage.

4. Usage Examples

To demonstrate the practical applicability of HeatMetro for analyzing test debt, we selected a heterogeneous corpus of six open-source projects (See Table 1 to reach the 3D visualization of the projects). This dataset ensures diversity across three key dimensions: ecosystem, covering Go (*sourcegraph/conc*, *ikawaha/kagome*) and Java (*stleary/JSON-java*); scale, ranging from compact frameworks to industry libraries like

apache/commons-dbcp; and quality maturity, spanning from high-coverage environments (*apache/commons-cli*) to legacy systems with urgent debts (*SERG-Delft/jpacman*).

Repository	Lang.	Contrib.	History			Size		Quality	Live View
			Commits	Tags	Br.	LOC	Meth.	Cov. %	
sourcegraph/conc	Go	≈ 60k	236	3	4	2,451	116	99.1	Link
SERG-Delft/jpacman	Java	17	374	12	6	2,177	252	36.0	Link
ikawaha/kagome	Go	16	847	87	2	5,711	235	86.7	Link
apache/commons-cli	Java	57k	2,110	35	4	11,069	1,094	97.4	Link
stleary/JSON-java	Java	103	1,449	25	10	12,574	1,017	87.3	Link
apache/commons-dbcp	Java	170k	3,515	76	11	30,761	3,913	68.4	Link

Table 1. Corpus of selected repositories for validation

This diversity confirms the adaptability of our approach across different lifecycles and lays the groundwork for the visual analysis, as follows.

4.1. Structural Representation

The inspection begins in Day Mode, which establishes the system's spatial reference frame using the Code City metaphor. To facilitate cognitive understanding of the architecture, HeatMetro employs the visual mapping scheme detailed in Figure 4: directories are rendered as districts (yellow), files as blocks (green), and structures (classes) as buildings (blue). The geometric dimensions directly reflect static metrics: height represents the number of methods (NOM), base area indicates the number of attributes (NOA), and color saturation is proportional to the volume of lines of code (LOC) affecting the specific component (directory, file, or structure).

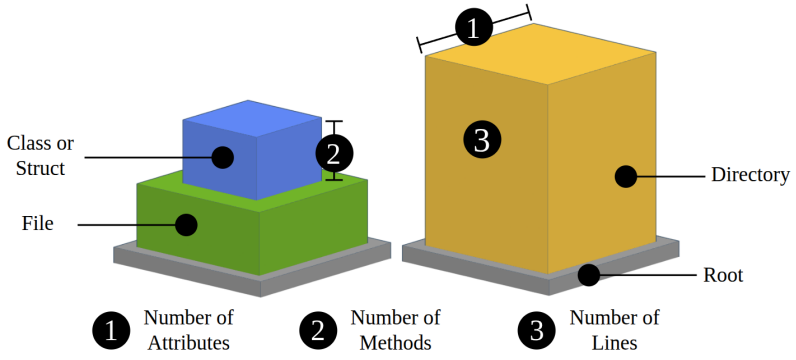
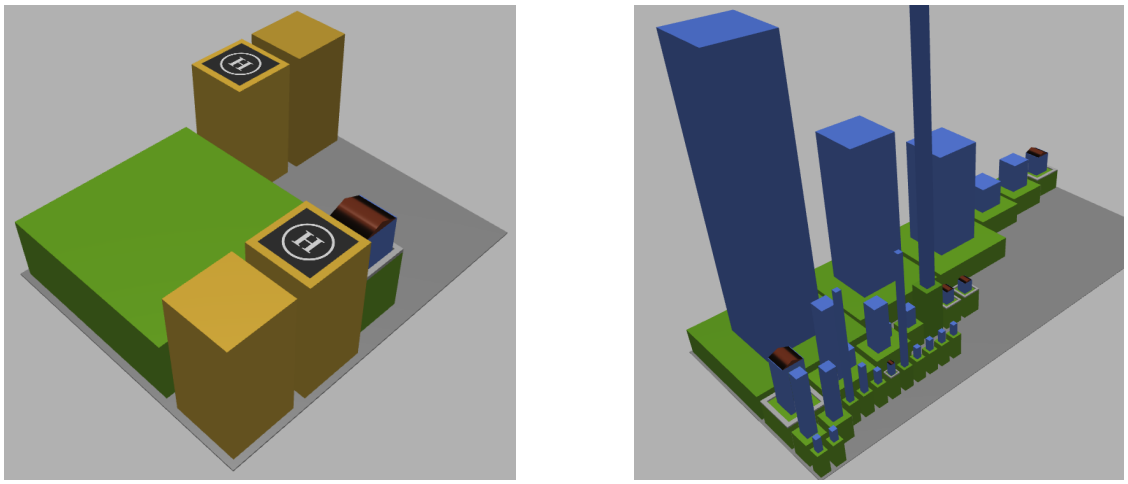


Figure 4. Visual Mapping Schema

Figure 5 contrasts the global projection of two distinct architectural patterns. The *sourcegraph/conc* project (a) exhibits a highly modularized topology, clearly segmented by districts (directories). In contrast, the *stleary/JSON-java* project (b) reveals a flatter hierarchy characterized by loose files within the source root and a greater heterogeneity of building structures. The interface further integrates an interactive timeline that allows navigation between versions (*commits*) and visualizes the project's history (volume) with vertical bars.

To quality assurance, the system integrates visual indicators of completeness (Figure 6). When a component reaches 100% test coverage, its geometry evolves to highlight an optimal state: directories deploy heliports on their surface; regarding files, they display

health symbols (red crosses) if they act as standalone units (without classes), whereas if they contain structures, the specific classes are crowned with residential roofs. This mechanism provides immediate positive feedback on the project's health status.



(a) *sourcegraph/conc* (GO) (b) *stleary/JSON-java* (Java)

Figure 5. Global Day View

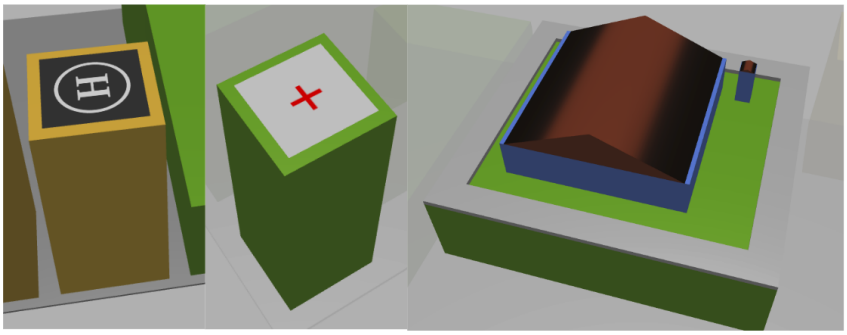


Figure 6. Completeness Indicators by Component Type

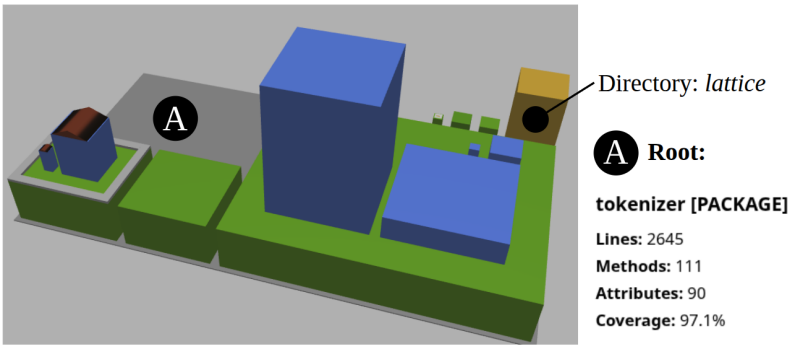
4.2. Exploration and Evolution Mechanisms

Prior to risk analysis, software developers must navigate complexity and understand the system's history. HeatMetro addresses this through two complementary mechanisms.

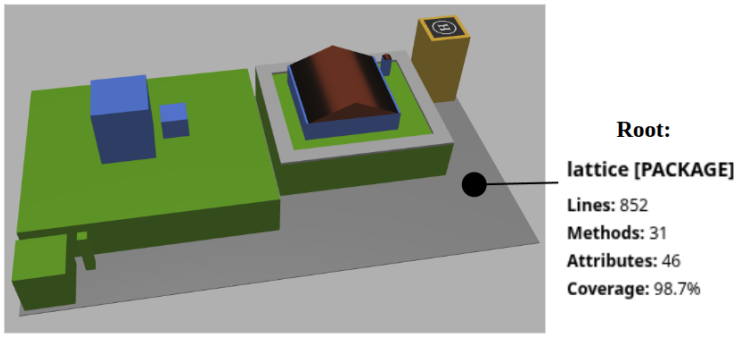
Hierarchical Navigation: To manage rendering performance in large projects, we implement on-demand loading (*Lazy Loading*). Figure 7 illustrates this strategy by using the *ikawaha/kagome* project: the user can descend from a high-level district view (a) into specific sub-directories like *lattice* to reveal their internal details (b). Furthermore, interacting with a specific file triggers a *fading effect* (c), focusing attention exclusively on the selected component while dimming the surrounding context.

Evolutionary Timeline: The historical dimension is crucial to understanding the maturity of the code. As shown in Figure 8 using the project *sourcegraph/conc*, the tool allows for reproducing the architectural evolution, transitioning from an initial construction phase (a) to an intermediate expansion characterized by a significant increase in volume

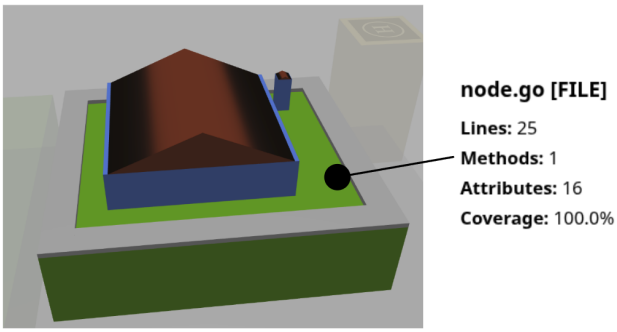
(b). The progression culminates in a structured final state (c), where visual indicators of completeness indicate a strategic prioritization of testing during the final stages.



(a) High-level district view.



(b) Internal component expansion.



(c) File selection focus.

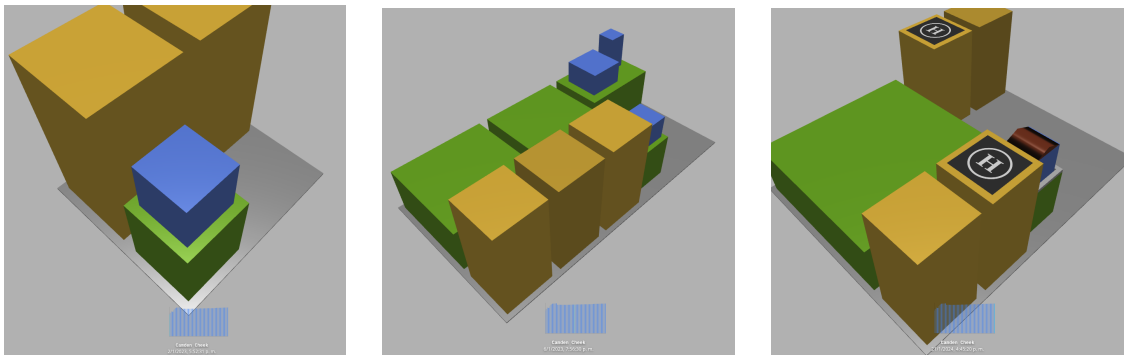
Figure 7. Progressive Modular Exploration in *ikawaha/kagome* (GO)

4.3. Test Debt Analysis

The core of our proposal is *night mode*, designed to visually operationalize our *context-aware code coverage* and reveal test debt. In this view, semantic coloring is replaced with a risk-based thermal map that enables users to filter out noise and focus on critical components. As illustrated in Figure 9, source files (1) exhibit a heatmap coloration ranging from red (close to 0% coverage) to white (close to 100%). This scheme contrasts sharply with test files (2), which are rendered in muted tones to divert user attention from non-production code.

Complementing the thermal map, the system implements selective filters to isolate specific risk factors. As depicted in Figure 10, applying the *cyclomatic complexity* filter

instantly illuminates critical components with a *glow effect* while keeping trivial code obscure. This discrimination is evident in both the *jpacman-framework* (a) and *commons-cli* (b) projects, where the tool successfully identifies high-risk targets within the *sprites* and *help* directories, respectively. By visually directing attention solely to glowing components, the interface validates the efficacy of our approach, enabling developers to filter out noise and focus exclusively on the underlying *test debt* hidden behind global metrics.



(a) Initial State (b) Intermediate State (c) Current State

Figure 8. Historical Evolution of *sourcegraph/conc* (GO)

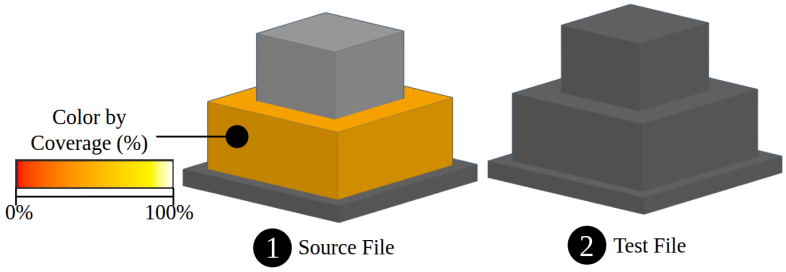
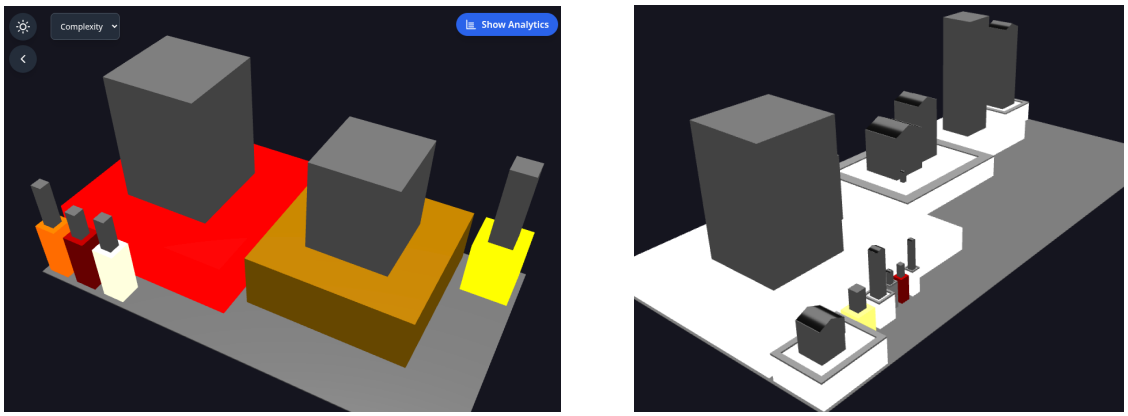


Figure 9. Night Mode Visual Mapping



(a) *jpacman-framework/sprites* (Java) (b) *commons-cli/help* (Java)

Figure 10. Global Night View with Active Filter

4.4. Quantitative Inspection and Criterion Validation

Once risk hotspots are visually identified, a dashboard (Figure 11) enables quantification of the impact of our *context-aware code coverage*. This dashboard updates dynamically

to break down precise metrics for selected components and contrast the global reality against the critical one.

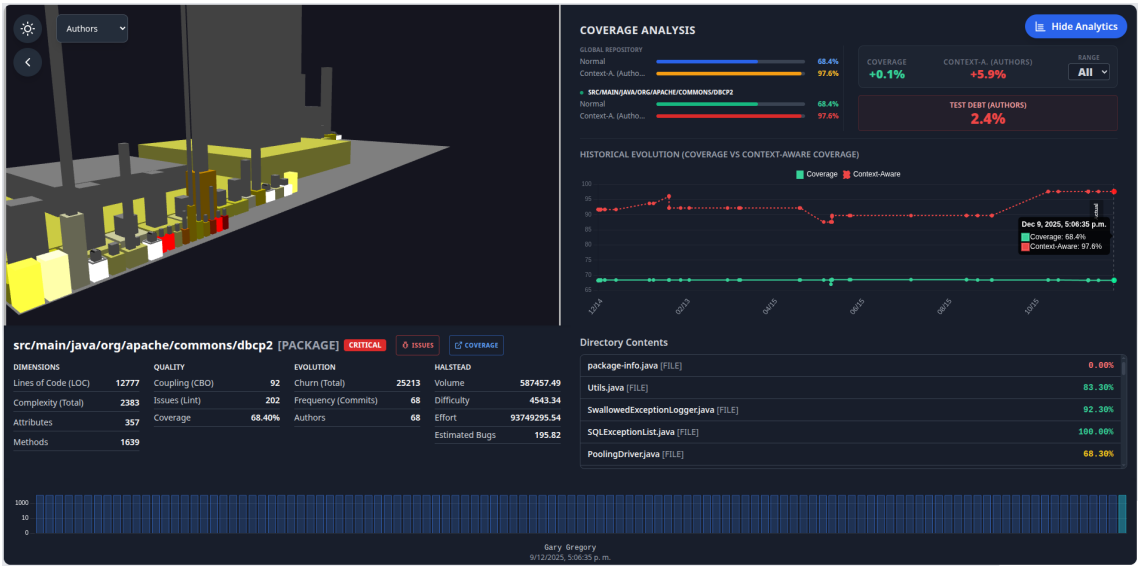


Figure 11. Analytical Dashboard and Metrics in *commons-dbcp* (Java)

Numerical validation is evidenced by the discrepancy between global coverage and the proposed criterion (Figure 12). In the case of *apache/commons-dbcp*, the chart reveals a *context-aware code coverage* of 97.6% compared to a modest 68.4% global coverage. This data aligns with *night mode* observations: although a large portion of the codebase remains untested, developers have correctly prioritized critical areas (high risk). This insight, which traditional metrics fail to reveal, demonstrates how the tool helps distinguish between dangerous low coverage and efficient test debt management.

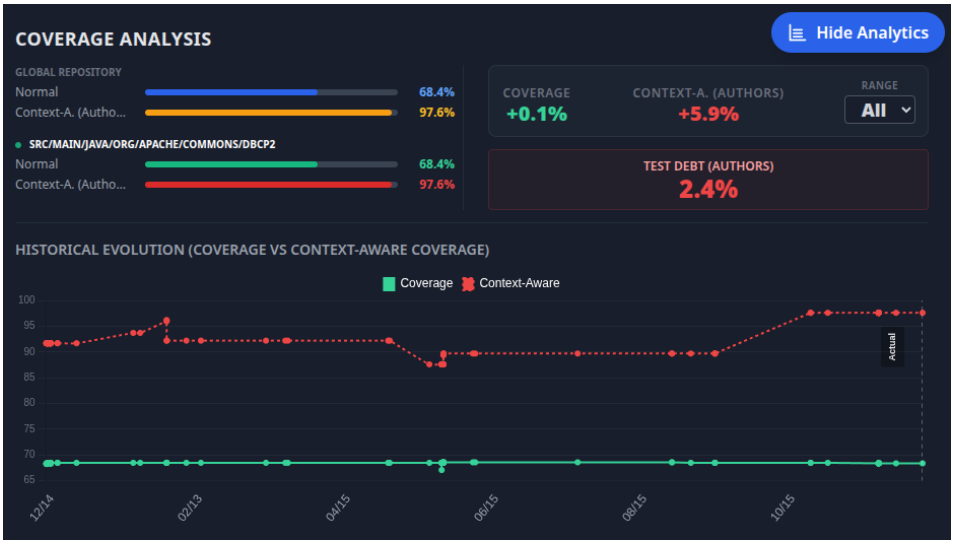


Figure 12. Global vs. *Context-Aware Code Coverage* in *commons-dbcp* (Java)

5. Evaluation and Discussion

To assess the practical applicability and efficacy of HeatMetro, we conducted an empirical evaluation involving software developers. This section outlines the participants' profile,

the experimental design, and a detailed analysis of the results obtained.

5.1. Study Design

To ensure the technical validity of the experiment, we recruited 12 developers with heterogeneous profiles, comprising Junior (58.3%), Semi-Senior (33.3%), and Senior (8.3%) levels. The inclusion criteria prioritized specific technical skills for quality inspection over tenure: the sample had high familiarity with version control systems (Git), practical experience in software testing, and previous exposure to 3D visualization tools to mitigate the *wow effect* during the evaluation. This qualification ensured that all candidates had the skills to interpret coverage and evolution visualizations accurately.

The experimental environment utilized the complete corpus of six projects detailed in Section 4. We structured a within-subjects experiment using a Latin Square design to mitigate potential learning and fatigue effects. Participants completed a set of specific analysis tasks across these repositories (full task details are available at https://github.com/ShinjiMC/HeatMetro-Tool/wiki). The core task was designed to evaluate the efficacy of our proposed metric: participants were instructed to employ HeatMetro's filters to identify critical components exhibiting significant coverage gaps—indicative of test debt—despite the project's high global metrics. To evaluate accuracy, participants cross-referenced their findings against a provided validation set, enabling them to verify their responses and receive immediate feedback on errors or omissions. Subsequently, they evaluated whether the *context-aware code coverage* view effectively highlighted testing areas that remained obscured in the standard global report.

5.2. Results

To measure the tool's effectiveness, we assessed the participants' responses using information retrieval metrics: *precision*, *recall*, *F1-score*, and *correctness*. These indicators collectively assess the accuracy of the identified critical components and the tool's ability to prevent omissions and minimize false positives [Dreef et al. 2023]. The detailed performance metrics extracted across all experimental tasks are summarized in Table 2.

Tarea	Precision	Recall	F1-Score	Correctness (%)
1.1	0.94	0.80	**0.86**	76.4%
1.2	0.83	0.83	**0.83**	83.3%
2.1	0.94	0.80	**0.86**	75.8%
2.2	0.91	0.71	**0.80**	69.4%
3.1	1.00	1.00	**1.00**	100.0%
3.2	0.67	0.67	**0.67**	66.7%
4	0.95	0.79	**0.86**	76.4%

Table 2. Detailed performance and analytical utility metrics per task.

The quantitative results demonstrate robust analytical utility, with high *correctness* and *F1-scores* across all tasks, proving that developers successfully correlated system topology with our *context-aware code coverage* criterion. Following the tasks, a standardized System Usability Scale (SUS) questionnaire was applied to measure user satisfaction and perceived cognitive load [Weichbroth and Giedrowicz 2024]. Figure 13 depicts a granular analysis of the response distribution for the ten items of the instrument.

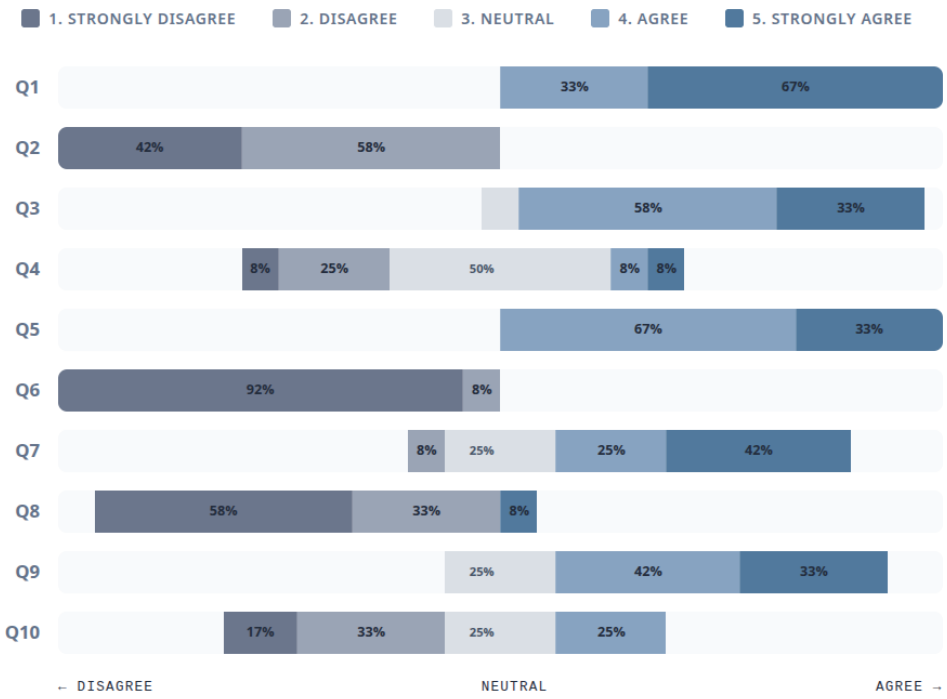


Figure 13. Distribution of User Opinion (SUS) for HeatMetro

The data analysis reveals a strong acceptance pattern. Positive valence items showed absolute consensus on critical aspects; notably, questions Q1 (frequency of use) and Q5 (integration of functions) reached 100% agreement. This indicates that developers perceived HeatMetro not as an isolated utility but as a cohesive system ready for seamless workflow integration. Conversely, negative valence items concentrated at rejection levels, evidencing low perceived friction. For question Q2 (unnecessary complexity), 100% of participants disagreed, while for Q6, 92% strongly disagreed that the system presented inconsistencies. This challenges the assumption that 3D visualizations inherently introduce complexity. Consequently, HeatMetro achieved a global SUS score of 79.0. According to the reference framework [Weichbroth and Giedrowicz 2024], this exceeds the acceptability threshold, positioning the tool as *acceptable* and within the *excellent* range.

5.3. Discussion

The results demonstrate both the analytical efficacy and feasibility of the proposed approach. A SUS score of 79.0 confirms that, despite the information density inherent in large systems, the cognitive load of 3D navigation remains manageable. Beyond usability, the accuracy—evidenced by F1-scores consistently exceeding 0.80 and precision peaking at 1.00 (task 3.1)—validates our core hypothesis: participants successfully and accurately correlated the system's topology with the test debt representation provided by HeatMetro.

By operationalizing our *context-aware code coverage* criterion, developers distinguished between safe untested code and critical components. This is corroborated by an average correctness above 75% and strong recall ($\sim$0.80), proving the visual metaphor minimizes critical omissions. This finding aligns with industrial actionability needs, empowering developers to make informed decisions rather than relying solely on aggregated

metrics. Consequently, our results show that the tool is highly effective not only for inspection but also for accurate quality analysis and prioritizing maintenance tasks.

Therefore, the study highlights the need to transition from static visualizations to analysis tools that integrate code evolution and quality metrics. HeatMetro's ability to identify test debt in critical components indicates that upcoming development tools should prioritize the automatic correlation of diverse data sources.

5.4. Threats to Validity

Construct Validity: There is a risk that the positive SUS rating is influenced by the aesthetic novelty of the 3D environment, a phenomenon known as the *wow effect*. While participants' familiarity with 3D interfaces partially accounted for this factor, it could still inflate perceived usability, independent of the tool's pragmatic utility. Furthermore, our study exclusively assesses visual comprehension; the accuracy of the *context-aware code coverage* metric still needs formal validation against a ground truth of known defects.

Internal Validity: Participants' prior 3D navigation dexterity was not controlled for; users with higher spatial notion might perceive the interface as more intuitive, introducing a satisfaction bias independent of the tool's design. Alongside this evaluation bias, a methodological bias arises because HeatMetro implicitly trusts outsourced SonarCloud metrics, thereby inheriting any inaccuracies in their calculations or parsing failures.

External Validity: Generalizing our findings to a continuous industry environment is constrained both by the controlled experimental sessions ($N = 12$, predominantly junior developers) and by HeatMetro's dependency on third-party maintainer policies (SonarCloud and GitHub APIs). However, the use of real, heterogeneous repositories of varying complexity ensures the tool was assessed against architectural structures representative of modern industry standards.

6. Conclusion

Managing test debt in continuously evolving software systems remains a critical challenge, as traditional global metrics often obscure the risks latent in complex, high-churn components. We address such a challenge using HeatMetro, a web-based visualization tool that operationalizes the code city metaphor and a *context-aware code coverage* criterion for visual analysis of test debt in large Java and Go projects, by integrating structural topology with evolutionary data. Our approach effectively isolates critical testing gaps from trivial code, providing an analysis interface that empowers developers to distinguish between safe untested code and dangerous technical debt.

The evaluation, involving 12 developers across a heterogeneous corpus of 6 open-source projects, yielded a SUS score of 79.0 and F1-scores exceeding 0.80, validating the tool's analytical and practical effectiveness in identifying quality risks and efficiently prioritizing maintenance efforts. Therefore, our work enables researchers and practitioners to evolve components and tests in a timely and cost-effective manner, overcoming the limitations of traditional reports that do not discriminate the relevance of the code.

Future work will focus on conducting a sensitivity analysis to calibrate the heuristic thresholds across different project domains. To quantitatively validate the *context-aware code coverage*'s accuracy against known defects, we plan to execute comparative studies against standard industry tools, integrating new evaluation measures such as

NASA-TLX for cognitive load, user efficiency (time-on-task), and technical robustness alongside our current framework. Finally, we will pursue longitudinal studies to assess long-term industrial impact and completely mitigate the potential *wow effect*, extending support to emerging programming ecosystems to generalize our findings.

References

Ahmad, A., Leifler, O., and Sandahl, K. (2022). Data visualisation in continuous integration and delivery: Information needs, challenges, and recommendations. *IET software*, 16(3):331–349.

Amusuo, P. C., Patil, P. V., Cochell, O., Le Lievre, T., and Davis, J. C. (2025). A unit proofing framework for code-level verification: A research agenda. In *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 36–40. IEEE.

Aragão, B. S., Andrade, R. M., Santos, I. S., Castro, R. N., Lelli, V., and Darin, T. G. (2022). Testdcat 3.0: catalog of test debt subtypes and management activities. *Software Quality Journal*, 30(1):181–225.

Avelino, G., Passos, L., Hora, A., and Valente, M. T. (2016). A novel approach for estimating truck factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE.

Brandt, C. and Ramírez, A. (2025). Towards refined code coverage: A new predictive problem in software testing. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 613–617. IEEE.

Brito, R., Brito, A., Brito, G., and Valente, M. T. (2019). Gocity: code city for go. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 649–653. IEEE.

Chotisarn, N., Merino, L., Zheng, X., Lonapalawong, S., Zhang, T., Xu, M., and Chen, W. (2020). A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558.

Dreef, K., Palepu, V. K., and Jones, J. A. (2023). Exploring granular test coverage and its evolution with matrix visualizations. *Information and Software Technology*, 155:107085.

Hasselbring, W., Krause, A., and Zirkelbach, C. (2020). Explorviz: Research on software visualization, comprehension and collaboration. *Software Impacts*, 6:100034.

Hernandes, V., Carvalho, A., Santos, E., Soares, Y., Oliveira, H., Barros, A., Soares, R., Lima, A., Ferreira, R., Martins, G., et al. (2025). A method for regression testing plan ordering for non-automated executions in black box testing. In *Congresso Ibero-Americano em Engenharia de Software (CIbSE)*, pages 120–134. SBC.

Hilton, M., Bell, J., and Marinov, D. (2018). A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 53–63.

Højelse, K., Kilbak, T., Røssum, J., Jäpelt, E., Merino, L., and Lungu, M. (2022). Gittruck: Hierarchy-oriented visualization of git repository evolution. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 131–140. IEEE.

Ivankovic, M., Petrovic, G., Kulizhskaya, Y., Lewko, M., Kalinovcic, L., Just, R., and Fraser, G. (2024). Productive coverage: Improving the actionability of code coverage. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 58–68.

Krasner, H. (2021). The cost of poor software quality in the us: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2:3.

McCabe, T. J. and Watson, A. H. (1996). Structured testing: A testing methodology. *NIST Special Publication*, 500:235.

Miranda, C., Avelino, G., and Santos Neto, P. (2025). Test co-evolution in software projects: A large-scale empirical study. *Journal of Software: Evolution and Process*, 37(7):e70035.

Mortara, J., Collet, P., and Dery-Pinna, A.-M. (2024). Visualization of object-oriented software in a city metaphor: Comprehending the implemented variability and its technical debt. *Journal of Systems and Software*, 208:111876.

Ostrand, T. J., Weyuker, E. J., and Bell, R. M. (2004). Where the bugs are. *ACM SIGSOFT software engineering notes*, 29(4):86–96.

Romano, S., Capece, N., Erra, U., Scanniello, G., and Lanza, M. (2019). The city metaphor in software visualization: feelings, emotions, and thinking. *Multimedia Tools and Applications*, 78(23):33113–33149.

Shatnawi, R. (2010). A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on software engineering*, 36(2):216–225.

Strandberg, P. E. (2017). Software test data visualization with heatmaps–an initial survey. *Report, no. MDH-MRTC318/2017-1-SE*.

Strandberg, P. E., Afzal, W., and Sundmark, D. (2022). Software test results exploration and visualization with continuous integration and nightly testing. *International Journal on Software Tools for Technology Transfer*, 24(2):261–285.

Vasa, R., Lumpe, M., Branch, P., and Nierstrasz, O. (2009). Comparative analysis of evolving software systems using the gini coefficient. In *2009 IEEE international conference on software maintenance*, pages 179–188. IEEE.

Viana, M., Moraes, E., Barbosa, G., Hora, A., and Valente, M. T. (2015). Jscity–visualização de sistemas javascript em 3d. In *III Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, pages 73–80.

Walkinshaw, N. and Minku, L. (2018). Are 20% of files responsible for 80% of defects? In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10.

Weichbroth, P. and Giedrowicz, M. (2024). Sus-lib: An automated tool for usability evaluation based on the software usability scale from user feedback. *arXiv preprint arXiv:2410.09534*.

Wettel, R. and Lanza, M. (2008). Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922.