COMPRENDIA: AI-Augmented Code Comprehension

Costain Nachuma
Department of Computer Science
Idaho State University
Pocatello, Idaho, USA
costainnachuma@isu.edu

Minhaz F. Zibran
Department of Computer Science
Idaho State University
Pocatello, Idaho, USA
zibran@isu.edu

arXiv:2608.10290v1 [cs.SE] 10 Aug 2026

***Abstract*—COMPRENDIA is an Eclipse plugin that integrates structural dependency visualization with LLM-powered code explanation on a shared interactive graph for Java program comprehension. The tool rests on four pillars: (1) a multi-edge-type dependency graph with live search and multiple layouts; (2) LLM explanations grounded in Graph-Aware Callee Pruning (GACP), an auditable strategy that selects relevant callees using the same graph the developer navigates; (3) a clone-detection overlay that highlights duplication and suggests extract-to-parent refactoring opportunities; and (4) a CVE risk overlay powered by OSV.dev. GACP uses graph distance, inheritance collapse, and edge-type weighting to produce prompts that are reproducible across LLM families and traceable to visible graph nodes. We demonstrate COMPRENDIA on a Java project containing known clones and vulnerabilities, showing how the unified graph substrate supports comprehension while keeping the developer in control. Screencast: https://youtu.be/1wlh_RYehzA**

***Index Terms*—program comprehension, large language models, prompt construction, dependency graphs, code clones, software refactoring, IDE plugin, Eclipse**

I. INTRODUCTION

Modern developers rely increasingly on large language models (LLMs) to write code [1], [2]. Likewise, they are using LLMs to explain unfamiliar code regions inside their integrated development environment (IDE). The dominant interaction paradigm, shared by Copilot Chat, Cursor, and Claude Code, treats the developer's text selection as the LLM's entire view of the project [3], [4]. This is empirically insufficient: even short methods routinely call into project-internal helpers whose behaviour is essential to a complete explanation, and a selection-only prompt forces the model to guess at those helpers from their names alone. Recent empirical work on AI-assisted programming finds that the gap between what the model is shown and what it needs to reason over is a frequent source of hallucinated identifiers and shallow, name-driven explanations [5], [6], in line with longstanding findings on the cognitive demands of code comprehension [7]–[9].

Tools that attempt to close this gap rely on embedding-similarity retrieval [10]–[12] or perplexity-based prompt compression [13]–[16]. Both families share three limitations: their selection criteria are *approximate* (a similarity score, not a structural guarantee), *not reproducible* across LLM families, and *not auditable* by the developer. We argue that the IDE's own dependency graph, which the developer can see, navigate, and trust, should be the relevance signal.

This paper presents COMPRENDIA, an Eclipse plugin for software comprehension that integrates dependency-graph visualization, LLM-assisted code explanation, and clone-refactoring discovery onto a single graph substrate. COMPRENDIA makes three contributions, each grounded in the team's prior published empirical work [5], [6], [17] and unified by a single design principle: *the visualization the developer sees is the data structure the tool reasons over.*

a) Contribution 1: A multi-edge-type structural substrate: COMPRENDIA renders an Eclipse Java project's dependency graph as an interactive visualization with five structural edge types (inheritance, interface, field-type, method-parameter, method-return). Edge-type filters, four layout algorithms, and in-graph search are exposed through a single toolbar. This extends Zibran's management-oriented [18]–[20] clone-visualization line [21] from a single-edge overlay to a comprehensive structural graph, complementing code-city work [22] and industrial program-comprehension research [23].

*b) Contribution 2: Graph-Aware Callee Pruning (*GACP*):* COMPRENDIA introduces GACP, an algorithm that uses the same dependency graph as the relevance signal for prompt construction. GACP (i) admits callees within k graph hops, (ii) collapses near-duplicate siblings sharing an inheritance ancestor into a single delegate, and (iii) allocates the remaining budget under an edge-type-weighted score. Every emitted callee carries an *inclusion reason* mapping one-to-one onto a visible graph node, making decisions *auditable* and *reproducible*. Section IV gives the full specification.

c) Contribution 3: Clone-refactoring discovery as a graph overlay: COMPRENDIA integrates the CLOMAN clone-detection engine [21] with the dependency graph by painting clone-group members as red-highlighted nodes. Clicking a group focuses the graph on its members and reveals extract-to-parent refactoring opportunities [24]. Because the overlay shares the same substrate, clicking a cloned class yields both its graph context and a GACP-enriched explanation via the inheritance-collapse phase.

All three contributions share a single substrate: the project dependency graph that GACP queries, the clone overlay

paints onto, and the developer reads to verify the tool's reasoning. COMPRENDIA also exposes a CVE risk badge via OSV.dev [17], [25], but we exclude this from the contribution claim as its graph integration is forthcoming work. The plugin, benchmark, and replay harness are publicly available [26].

II. RELATED WORK

A. IDE-Native LLM Code Explanation

Several tools embed LLM-powered code explanation directly in the developer editor. GitHub Copilot Chat [3] and Cursor [4] send the text selection for the developer to the model with no additional project context; the model must infer callee behaviour from names alone. Sourcegraph Cody [10] and RepoFuse [11] go further by retrieving project-internal code via embedding similarity, ranking candidate snippets by cosine distance to the selection. In all cases the inclusion decision is approximate (a similarity score above a threshold), not reproducible across embedding models, and not auditable by the developer: there is no visible structure the user can inspect to verify why a snippet was included. Patel et al. [27] offer a valuable conceptual framework, arguing that unreviewed AI-generated code causes long-term *knowledge erosion*; GACP's auditable citations operationalize a concrete, tool-level countermeasure to exactly this risk.

B. Prompt Compression for Long Contexts

A complementary line of work starts from too much context and compresses it. LLMLingua [13] and its long-context extension LongLLMLingua [15] use a small language model's perplexity scores to drop low-information tokens, with question-aware compression for long contexts. RECOMP [14] trains extractive and abstractive compressors to select or rewrite retrieved passages before they enter the prompt. These techniques effectively reduce token costs, but their compression decisions are model-internal: different perplexity estimators or compressor checkpoints yield different prompts, and the developer cannot trace a retained token back to a project-level structure. GACP differs in kind: its only relevance signal is the dependency graph the developer already sees, making every inclusion decision traceable to a visible graph node and reproducible across LLM families.

C. Structural Visualization in IDEs

Structural visualization tools help developers navigate large codebases. CodeCity renders software metrics as 3D cityscapes [22]; Lattix, Structure101, and Eclipse Zest (used by Project Usus [28]) render dependency matrices, layered diagrams, or general-purpose graphs within the Eclipse platform. Quante [23] evaluates dynamic object-process graphs for industrial program comprehension. Zibran [21] proposes clone analysis and visualization with respect to inheritance hierarchies and call graphs as necessary support for clone refactoring. COMPRENDIA continues this line: it extends Zibran's clone-visualization substrate from a standalone clone view to a multi-edge-type dependency graph that supports navigation, AI explanation, and refactoring discovery on the same surface. None of the tools above combine structural visualization with LLM-driven code explanation.

TABLE I
POSITION OF COMPRENDIA AMONG RELATED APPROACHES.

	AI-explanation	Graph-grounded	IDE-native	Auditable
Copilot Chat / Cursor	✓	–	✓	–
Cody / RepoFuse	✓	–	✓	–
LLMLingua / RECOMP	–	–	–	–
CodeCity / Zest	–	✓	✓	✓
COMPRENDIA (GACP)	✓	✓	✓	✓

D. Position of COMPRENDIA

Table I summarises the gap. COMPRENDIA is, to our knowledge, among the first IDE-native systems to use the project's dependency graph as the relevance signal for prompt construction, making callee-inclusion decisions auditable and reproducible.

III. TOOL OVERVIEW

A. User-Facing Workflow

A developer opens an unfamiliar Eclipse Java project and right-clicks *Visualize Dependencies*. COMPRENDIA parses the workspace incrementally via JDT and renders a dependency graph inside the IDE: nodes are classes, interfaces, and enums; edges encode five structural relationships (inheritance, interface, field-type, method-parameter, method-return). The developer toggles edge types, switches among four layout algorithms (spring, tree, radial, grid), and uses in-graph search to locate a class of interest. Double-clicking a node opens the corresponding source file.

To understand a method, the developer highlights its body and clicks *Explain with AI*. COMPRENDIA invokes GACP (Section IV) to select project-internal callees from the graph, assembles a prompt containing the selection and the chosen callee snippets, and routes the request to the configured LLM backend. The explanation appears in the InsightPanel with callee citations the developer can click to navigate to the referenced source. Because each cited callee maps one-to-one onto a visible graph node, the developer can verify the tool's reasoning against the same structure used to produce it.

Two overlays enrich the graph without leaving it. Running *Detect Clones* invokes the CloMan clone-detection engine [21] via a reflection bridge; clone-group members are highlighted in red, and clicking a group focuses the graph on its members, exposing candidate extract-to-parent refactoring opportunities. Independently, parsing the project's `pom.xml` and querying the OSV.dev vulnerability database [29] paints CVE-severity borders on affected nodes (red for critical, orange for high), grounded in our prior finding that 62.89% of latest Maven releases carry transitive vulnerabilities [17].

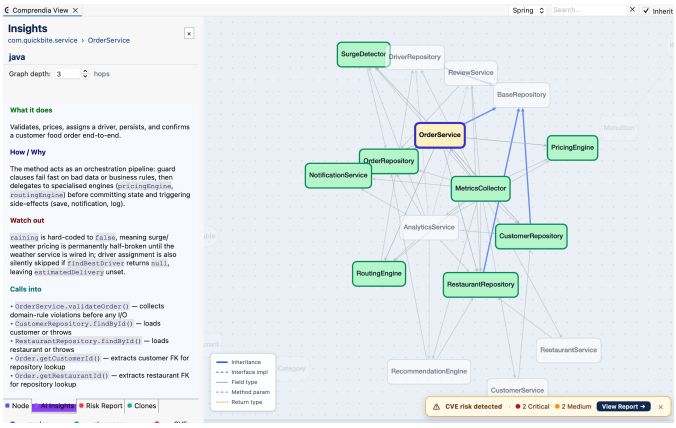

Fig. 1. COMPRENDIA showing the dependency graph with `OrderService` selected. The InsightPanel (left) displays the GACP-grounded AI explanation with callee citations. The toolbar (top) exposes edge-type filters, layout selection, and search. The CVE risk banner (bottom) reports detected vulnerabilities.

Fig. 2. Clone overlay: CloMan detects two clone groups (6 occurrences). The four-class `BaseRepository` group (k=4) is highlighted in red; non-clone nodes are grayed. The tooltip shows code metrics for the selected member. This is the group GACP's Phase 2 collapses in Section V.

B. Architecture and Infrastructure

COMPRENDIA is organized in five layers. The *UI layer* hosts the graph view (Cytoscape.js [30] in an SWT Browser), the InsightPanel, and the toolbar. The *analysis layer* builds the graph via JDT AST parsing [31], runs GACP, and bridges CloMan for clone detection. The *AI layer* exposes a single `AiGateway` interface with three production backends (Anthropic Claude, OpenAI, Ollama) plus a deterministic mock for offline use; all receive the same GACP-assembled prompt, enabling the cross-LLM comparison in Section V. The *data layer* reads JDT models, queries OSV.dev [29] for CVE data, and accesses git history via JGit [32]. An opt-in telemetry service logs feature activations and task timestamps to a local JSONL file for controlled user studies [33].

C. Practical Use Scenarios

COMPRENDIA targets three recurring needs. A developer onboarding onto an unfamiliar module clicks *Explain with AI* and receives an explanation whose callee citations resolve to the graph already on screen, rather than an opaque summary. A maintainer preparing a refactoring sprint runs *Detect Clones* to surface duplicate implementations and their common-ancestor candidates before writing code. A team triaging a dependency advisory uses the CVE overlay to see which classes touch a flagged package. For researchers, GACP's parameterized, auditable selection offers a controllable variable for studying how context-selection strategy affects LLM-assisted comprehension.

IV. GRAPH-AWARE CALLEE PRUNING

This section specifies GACP, a pure, static function that chooses project-internal callees for the LLM-explanation prompt. Given the enclosing class FQN, AST-extracted callees (filtered to project-internal symbols via JDT), a read-only `GraphIndex` over the dependency graph, and a token `Budget`, it returns a ranked list of `GacpCallee` records. Each record carries the callee's FQN, graph distance, snippet mode (`FULL_BODY` or `SIGNATURE_ONLY`), rendered prompt fragment, and a structured `inclusionReason` (e.g., *"distance-1 INHERITANCE; full body"*) that maps one-to-one onto a visible graph node. The function contains no randomness, no LLM calls, and no UI dependencies.

A. Phase 1: Reachability Filter

Phase 1 admits a candidate callee if its declaring class lies within $K_{\max}$=2 hops from the enclosing class on the dependency graph; self-calls (distance 0) are admitted directly. The cutoff is a hard structural admission rule, not a soft penalty. Two hops covers a selection's direct callees and their immediate structural context while bounding prompt size; deeper callees add tokens faster than comprehension signal. Candidates are de-duplicated by declaring class so that Phases 2 and 3 operate at owner granularity. The snippet mode is distance-dependent: distance-$\leq$1 callees are rendered as `FULL_BODY`, distance-2 as `SIGNATURE_ONLY`.

B. Phase 2: Topological Redundancy Elimination

Any group of $\geq$2 surviving candidates whose declaring classes share an *immediate* inheritance ancestor A is replaced with A itself (rendered as `FULL_BODY`) plus a single concrete delegate. The "immediate" qualifier guards against false collapses through `java.lang.Object`. This phase yields the largest savings on clone-touching selections. For example, when the developer requests an explanation inside one of four sibling repository classes sharing `BaseRepository`, Phase 2 collapses the four near-duplicate method bodies into the `BaseRepository` contract (rendered in full) plus one concrete delegate, producing an explanation that describes the shared contract rather than repeating sibling implementations (Section V).

C. Phase 3: Edge-Type-Weighted Budget Allocation

The final phase ranks survivors by edge-type weight divided by squared graph distance, favoring inheritance and interface edges (weight 1.0) over field-type (0.6) and method-parameter or method-return (0.4) relationships. Candidates are emitted

TABLE II
BENCHMARK FIXTURES FROM QUICKBITE.

Selection	Category	Calls into
`placeOrder`	no-collapse	7 helpers across 4 pkgs
`recommendationEngine`	no-collapse	5 model + util helpers
`pricingEngineSurge`	no-collapse	4 engine/util helpers
`baseRepoCollapse`	collapse k=4	4 sibling repositories
`reviewableCollapse`	collapse k=3	3 `Reviewables`
`abstractUserCollapse`	collapse k=2	2 user subclasses

TABLE III
PROMPT TOKENS BY MODE. IDENTICAL ACROSS ALL THREE LLM FAMILIES.

Selection	SEL	LEG	SIG	BODY
`placeOrder`	197	249	297	604
`recommendEngine`	158	199	214	386
`pricingSurge`	158	199	213	364
`baseRepoCollapse`	144	186	181	193
`reviewCollapse`	132	161	164	195
`userCollapse`	108	134	143	159
Mean (all 6)	150	188	202	317
Mean (no-collapse)	171	216	241	451
Mean (collapse)	128	160	163	182

greedily under the token budget, skipping degenerate snippets. Given fixed inputs, GACP returns the same output on every invocation: Phase 1 is $O(|V|+|E|)$; Phases 2 and 3 are $O(|C|\log|C|)$ on the surviving set, which is single-digit in practice. Section V confirms identical prompts across three LLM families.

V. DEMONSTRATION

We illustrate GACP's behaviour on six method-body selections from the QUICKBITE open-source Java benchmark (33 classes, 6 packages), replayed against three LLM families (Claude Sonnet 4.6, GPT-4o-mini, Llama 3 via Ollama). The harness compares four modes per fixture: SELECTION-ONLY (no callee context), LEGACY (dedup-and-cap at 5 callees), GACP-SIG (GACP with signature-only rendering), and GACP-BODY (GACP with full bodies for distance-1 callees, the shipping default). All modes share the same JDT AST visitor and prompt scaffolding; they differ only in callee filtering and rendering. Table II lists the six fixtures: three exercise Phase 1's reachability filter without engaging collapse; three engage Phase 2 with group sizes $k = 2, 3, 4$.

A. Token Cost

Table III reports prompt-token cost per fixture. Values are identical across all three LLM families, confirming that the algorithm's output is gateway-independent.

GACP-SIG trades a modest token premium (+7.5% overall) for graph-grounded inclusion decisions. On collapse fixtures the cost is near-parity (+1.5%), because Phase 2 replaces k sibling bodies with one parent contract plus one delegate. On the k=4 case, GACP-SIG actually *saves* 2.7% (186→181 tokens).

TABLE IV
PHASE 2 RESHAPE: LEGACY EMITS k SIBLINGS; GACP EMITS PARENT + DELEGATE.

Fixture	LEGACY emits	GACP emits
`baseRepo` (k=4)	4 sibling repositories	`BaseRepository` + `CustomerRepository`
`review-able` (k=3)	`MenuItem`, `Restaurant`, `Driver`	`Reviewable` + `MenuItem`
`abstract-User` (k=2)	`Customer`, `Driver`	`AbstractUser` + `Customer`

On no-collapse fixtures, GACP-SIG admits more callees than LEGACY's cap: `placeOrder` emits 7 vs. 5, including two structurally proximate helpers that the cap drops (+19.3%). Where the cap is not binding (`recommendationEngine`, `pricingEngineSurge`), the two modes emit identical callee sets.

GACP-BODY roughly doubles the prompt on no-collapse fixtures (+109.3%) by including full method bodies. On collapse fixtures, Phase 2 limits the increase to +13.7%.

Table IV shows the reshape on the three collapse fixtures. Phase 2 fires on all three and replaces k near-duplicate siblings with the common ancestor plus one concrete delegate; this is algorithm-only and LLM-independent. Prompt-token counts are identical across all three LLM families for every fixture×mode combination, confirming that GACP's output is a pure function of the graph and the selection.

VI. LIMITATIONS AND FUTURE WORK

a) Single benchmark project: The evaluation draws from one subject system, QUICKBITE, chosen for its seeded clone groups at three sizes ($k = 2, 3, 4$). The algorithm is project-agnostic; running it on additional codebases is on our follow-on agenda.

b) Coarse accuracy rubric: The keyword-match accuracy score saturates to 2/2 across all modes and LLM families. Quality claims in Section V therefore rest on the algorithm-output comparison (callee-set composition and collapse behaviour), which is LLM-independent. A behaviour-test rubric would discriminate more finely but is follow-on work.

c) Token estimator: Prompt tokens are estimated as character-count/4. Absolute counts under provider-specific tokenizers differ by ±10–15%, but relative comparisons, which are the basis of every claim, are preserved.

d) No human-comprehension claim: This paper measures the algorithm's effect on the prompt, not whether developers comprehend code faster. A controlled user study on QUICKBITE is designed but postdates this submission.

e) Personalization and graph granularity: GACP treats every developer as equivalent and operates on a class-level dependency graph. Method-level call graphs and expertise-adaptive pruning are future directions.

f) Baseline and parameter scope: Section V compares GACP against Selection-only and a fixed-cap Legacy baseline, not against an embedding-retrieval system (e.g., Cody, RepoFuse); a head-to-head comparison on the same six fixtures is

future work. We likewise did not sweep GACP's design parameters ($K_{\max}$, edge-type weights); confirming that collapse behaviour is stable under alternative settings is a natural next step.

VII. Conclusion

This paper presented COMPRENDIA, an Eclipse plugin that integrates dependency-graph visualization, LLM-assisted code explanation, and clone-refactoring discovery onto a single graph substrate. The unifying design principle is that the visualization the developer sees is the data structure the tool reasons over: GACP queries the graph for callee distances, the clone overlay paints onto the graph, and the developer reads the graph to verify both. GACP's callee-inclusion decisions are auditable against visible graph nodes and reproducible across LLM families, distinguishing COMPRENDIA from embedding-retrieval and perplexity-based approaches whose relevance signals are model-internal.

On the QUICKBITE benchmark, GACP's inheritance-collapse phase reshapes prompts on clone-touching selections from k sibling duplicates to a parent-contract-plus-delegate form, and the graph-distance cutoff admits structurally proximate helpers that a fixed cap drops. The plugin, benchmark fixtures, and replay harness are publicly available [26].

References

[1] C. Nachuma, M. F. Rabbi, A. I. Champa, and M. F. Zibran, "Analyzing chatgpt assistance in programming," in *Software Engineering and Management: Theory and Applications: Volume 17*. Springer, 2025, pp. 203–215.

[2] H. Li, H. Zhang, and A. E. Hassan, "The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering," 2025. [Online]. Available: https://arxiv.org/abs/2507.15003

[3] GitHub, "GitHub Copilot Chat," https://github.com/features/copilot, 2026, accessed: May 2026.

[4] Anysphere, "Cursor: The AI-First Code Editor," https://cursor.com, 2026, accessed: May 2026.

[5] A. I. Champa, M. F. Rabbi, C. Nachuma, and M. F. Zibran, "ChatGPT in Action: Analyzing Its Use in Software Development," in *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*, Lisbon, Portugal, 2024, pp. 182–192.

[6] C. Nachuma and M. Zibran, "When AI Teammates Meet Code Review: Collaboration Signals Shaping the Integration of Agent-Authored Pull Requests," in *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*, Brazil, 2026, pp. 1–5.

[7] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *36th international conference on software engineering*, 2014, pp. 378–389.

[8] T. LaToza and B. Myers, "Hard-to-answer questions about code," in *Evaluation and usability of programming languages and tools*, 2010, pp. 1–6.

[9] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A Comprehensive Model for Code Readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018.

[10] Sourcegraph, "Cody: AI Coding Assistant That Knows Your Codebase," https://sourcegraph.com/cody, 2026, accessed: May 2026.

[11] M. Liang, X. Xie, G. Zhang, X. Zheng, P. Di, W. Jiang, H. Chen, C. Wang, and G. Fan, "RepoFuse: Repository-Level Code Completion with Fused Dual Context," *CoRR*, vol. abs/2402.14323, 2024, arXiv preprint.

[12] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using Pre-Trained Models to Boost Code Review Automation," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2291–2302.

[13] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu, "LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023, pp. 13 358–13 376.

[14] F. Xu, W. Shi, and E. Choi, "RECOMP: Improving Retrieval-Augmented LMs with Context Compression and Selective Augmentation," in *12th International Conference on Learning Representations*, 2024.

[15] H. Jiang, Q. Wu, X. Luo, D. Li, C.-Y. Lin, Y. Yang, and L. Qiu, "LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression," in *62nd Annual Meeting of the Association for Computational Linguistics*, 2024, pp. 1658–1677.

[16] R. Ravi, D. Bradshaw, S. Ruberto, G. Jahangirova, and V. Terragni, "LLMLOOP: Improving LLM-Generated Code and Tests through Automated Iterative Feedback Loops," in *IEEE International Conference on Software Maintenance and Evolution, Tool Demo Track*, 2025.

[17] C. Nachuma, M. M. Hossan, A. K. Turzo, and M. F. Zibran, "Decoding Dependency Risks: A Quantitative Study of Vulnerabilities in the Maven Ecosystem," in *Proceedings of the 22nd International Conference on Mining Software Repositories (MSR)*, 2025, pp. 270–280.

[18] M. F. Zibran and C. K. Roy, "IDE-based real-time focused search for near-miss clones," in *Proceedings of the 27th ACM Symposium On Applied Computing (SAC)*. ACM, 2012, pp. 1235–1242.

[19] C. Roy, M. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future," in *proceedings of the IEEE CSMR-18/WCRE-21 Software Evolution Week (SEW'14), Vision Keynote*. IEEE, 2014, pp. 18–33.

[20] M. F. Zibran, "Towards implementation of an integrated clone management infrastructure," in *Proceedings of the 10th IEEE International Workshop on Software Clones (IWSC)*. IEEE, 2016, pp. 60–61.

[21] M. F. Zibran, "Analysis and Visualization for Clone Refactoring," in *Proceedings of the 9th IEEE International Workshop on Software Clones (IWSC)*, Montréal, Canada, 2015, pp. 47–53.

[22] R. Wettel and M. Lanza, "CodeCity: 3D Visualization of Large-Scale Software," in *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE Companion)*, 2008, pp. 921–922.

[23] J. Quante, "Do Dynamic Object Process Graphs Support Program Understanding? A Controlled Experiment," in *16th IEEE International Conference on Program Comprehension*, 2008, pp. 73–82.

[24] A. AlOmar, W. Mkaouer, and A. Ouni, "Toward the Automatic Classification of Self-Affirmed Refactoring," *J. of Systems and Software*, 2022.

[25] C. Improta, "Poisoning Programs by Un-Repairing Code: Security Concerns of AI-Generated Code," in *Proceedings of the International Symposium on Software Reliability Engineering Workshops*, 2023.

[26] C. Nachuma and M. F. Zibran, "Comprendia: AI-Augmented Code Comprehension (ICSME 2026 Artifact)," https://doi.org/10.5281/zenodo.20436759, 2026.

[27] M. S. Patel, R. Dhawan, L. Bramhanwade, and D. J. Mishra, "Code Comprehension Gaps in AI-Assisted Development: A Conceptual Framework for Knowledge Erosion Patterns," *IEEE Access*, vol. 14, pp. 66 740–66 757, 2026.

[28] Project Usus Contributors, "Project Usus: Eclipse Plug-Ins for OO Design Metrics and Dependency Visualization," https://github.com/usus/usus-plugins/wiki, 2017, accessed: May 2026.

[29] Google Open Source Security Team, "OSV: Open Source Vulnerabilities Database," https://osv.dev/, 2026, accessed: May 2026.

[30] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, "Cytoscape.js: a graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 2016.

[31] Eclipse Foundation, "Eclipse Java Development Tools (JDT)," https://www.eclipse.org/jdt/, 2026, accessed: May 2026.

[32] Eclipse Foundation, "Eclipse JGit: Java Implementation of Git," https://www.eclipse.org/jgit/, 2026, accessed: May 2026.

[33] M. Basha, A. Ribeiro, J. Javahar, G. Rodriguez-Perez, and C. de Souza, "CodeWatcher: IDE Telemetry Data Extraction Tool for Understanding Coding Interactions with LLMs," in *IEEE International Conference on Software Maintenance and Evolution, Tool Demo Track*, 2025.