SUST Journal of Science and Technology Vol. 35(2) 2025, 28 - 34

SUST Journal of Science and Technology

journal homepage: https://journals.sust.edu

Maintainability of LLM-Generated Code and Student's Understanding

Md. Mehedi Hasan [1*], Dr. Md Forhad Rabbi [1], Fazle Rabbi Rakib [2]

[1]*Department of Computer Science and Engineering,* [2]*Institute of Information and Communication Technology, Shahjalal University of Science and Technology, Sylhet – 3114, Bangladesh*

ARTICLE INFO

Keywords:

LLMs
CS education
Software maintainability
Program comprehension
Static analysis

ABSTRACT

Large Language Models (LLMs) are frequently used by students for programming assignments and capstone projects, yet their impacts on software maintainability and students' ability to explain and maintain their own codebases remain unclear in authentic educational settings. This study examines maintainability and code understanding as jointly examined outcomes in an undergraduate Web Technology course. We analyze 48 student GitHub repositories (3,043 commits) using commit-level SonarQube static analysis to track maintainability and risk-relevant indicators, including code smells, SQALE technical-debt measures, duplication, complexity, and security/reliability ratings. We complement repository-level measurements with student explainability (understanding) scores on a 0–10 scale ($n = 104$), reflecting students' ability to explain code segments from their projects. Across repositories, 5/48 (10.4%) failed the SonarQube quality gate. Security and reliability concerns were concentrated in a subset of projects. Our investigation exhibits 9/48 (18.75%) blocker-level security status and 2/48 (4.17%) high security issues, whereas reliability issues included 2/48 (4.17%) with blocker-level reliability and 6/48 (12.5%) with high reliability issues, while 32/48 (66.67%) fell in the medium range. In addition, security hotspots that require review were found on 41 repositories. On the explainability side, 33% of the students scored $\leq$ 5/10, indicating limited ability to explain their code and approximately 2% could not provide a meaningful explanation. Overall, the results suggest that a significant portion of student projects has maintainability and security/reliability risks and a notable subset of students demonstrate low explainability of their own code. These findings motivate instructional process and assessment practices that emphasize code review, refactoring and explanation alongside functional correctness, especially in courses where LLM-based tools are available.

1. Introduction

Large Language Models (LLMs) such as GPT-4, Cursor, Codex, GitHub Copilot, and Claude have altered how software is developed by rapid code generation, repair, and explanation. In university programming classes, students utilize LLM-powered tools (e.g., ChatGPT, Copi- lot) frequently to complete project/coding assignment. While LLMs might improve short-term productivity and help students complete tasks, there is growing concern that students may deliver workable code without proper understanding of the underlying logic, architecture, or design considerations. In software engineering, maintainability is a critical quality attribute as maintenance dominates software development lifecycle cost. For educational settings, the risk is twofold: (i) code may have technical debt even if it passes quality gate and (ii) learners may become less capable of explaining, debugging, and extending their own codebases. Despite rapid adoption, empirical evidence linking long-term maintainability outcomes with direct measures of student codebase understanding is still limited in authentic course contexts.

This paper addresses this gap by (i) performing commit-level longitudinal static analysis of student repositories using SonarQube and (ii) triangulating these quantitative

* Corresponding author.
E-mail address: mehedi-cse@sust.edu (Md. M. Hasan)

https://doi.org/10.63512/sustjst.2025.2003

Copyright© 2025 SUST. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

quality trajectories with student interviews and rubric-scored code comprehension tasks, enabling a student-centered evaluation of both maintainability and understanding.

Concretely, this study is guided by three research questions:

(RQ1) Repository-Level Maintainability Outcomes: To what extent do LLM-assisted student repositories exhibit maintainability, reliability, and security risks, as measured by commit-level static analysis?

(RQ2) Student Explainability Outcomes: How well can students explain and justify code segments drawn from their own repositories, and how does this vary across maintainability profiles?

(RQ3) LLM Use and Comprehension: Is there an observable relationship between how students report using LLMs (e.g., primarily for code generation versus interactive explanation and refactoring) and their demonstrated code comprehension?

2. Literature Review

Students in programming classes today frequently use LLM-based coding tools like GitHub Copilot and ChatGPT. Pitts et al. [2] examine LLM applications in programming education and highlight a significant tension: while these tools improve rapid output, students who rely on production without reflection develop poor debugging and design reasoning skills. Zi et al. [1] provide solid evidence for this gap, indicating that beginners generally struggle to understand the LLM-generated code they contributed, often stating that they would have arranged it differently. Stack Overflow's developer survey [19] and GitHub Copilot research [22] corroborate that generation-focused usage is the prevailing tendency among student and beginning engineers. Prather et al. [15] and Denny et al. [17] provide comprehensive overviews of the educational hurdles and opportunities that generative AI presents to computing educators, emphasizing the importance of developing evaluation mechanisms to account for over-reliance on generated code.

On the code quality side, static analysis research routinely demonstrates that LLM-generated code acquires maintainability and security problems even when functionally correct. GitClear [11] shows that the number of technical debt indicators in AI-assisted repositories will quadruple between 2022 and 2024. Abbassi et al. [6] define five types of inefficiencies in LLM-generated Python code: logic, performance, readability, maintainability, and blunders. Siddiq et al. [7] show that transformer-based code generation outputs have increased frequencies of code smells. Pearce et al. [8] reveal that about 40% of GitHub Copilot-generated code contains security vulnerabilities across prominent CWE categories, thereby validating the vulnerability hotspot prevalence identified in this analysis. The fundamental metric frameworks employed — cyclomatic complexity [3], object-oriented design metrics [5], and the ISO/IEC 25010 quality model [9] — offer the theoretical basis for interpreting SonarQube reports [10, 12, 13].

Code comprehension research indicates that structural complexity directly influences how readily code can be understood and maintained [3, 5]. Denny et al. [14] demonstrate that the manner in which students engage with LLMs is as significant as their usage; individuals who participate in prompt engineering and iterative improvement cultivate superior computational thinking compared to passive consumers of generated code. Existing research, however, measures understanding by self-report; none combine commit-level static analysis metrics with direct, rubric-scored explainability assessments within the same actual course dataset. This study addresses that gap.

This positions the present study's contribution relative to prior work along two axes. First, whereas Zi et al. [1] and Denny et al. [14] rely on self-reported or task-level comprehension measures without linking these to objective repository quality trajectories, this study pairs commit-level SonarQube metrics with independently rubric-scored explainability interviews for the same 104 students, enabling a within-cohort comparison of code quality and comprehension rather than treating the two as separate literatures. Second, whereas GitClear [11], Siddiq et al. [7], and Pearce et al. [8] characterize LLM-generated code quality and security risk at the level of isolated snippets or aggregate repositories, this study tracks quality longitudinally at the commit level across an entire semester (a semester typically lasts for 6 months, roughly 14 weeks of class) of authentic, graded coursework, which allows failing and passing trajectories to be distinguished rather than assessed only at a single snapshot.

3. Methodology

This study uses a mixed-methods, longitudinal methodology to look into how maintainability-relevant code quality signals connect to student codebase understanding in an authentic course scenario. The methodology combines (i) commit-level static analysis of student repositories using SonarQube and (ii) student understanding measurement via explainability scoring and interview-style comprehension assignments. All analyses were conducted on anonymized repository and participant identifiers, and all procedures were designed to be reproducible by an independent researcher.

3.1. Sample preparation

This research is undertaken as part of CSE344: Web Technology. The dataset includes 48 student-managed GitHub repositories that were produced iteratively over the course of a semester, with a total of 3,043 commits. Repository IDs are anonymised prior to analysis. Each repository was authored by one or more students enrolled in the course; where a repository reflected the work of more than one contributor, Git commit-author metadata (author identity, commit hash, and timestamp) was used to attribute individual code segments to specific students prior to the explainability interviews described in Section 3.4, so that each of the 104 interviewed students was assessed on code they had personally authored or substantially modified. Team sizes ranged from 1 to 3 students per repository. Inclusion criteria, participant consent, and anonymization procedures are detailed in Section 3.1.1.

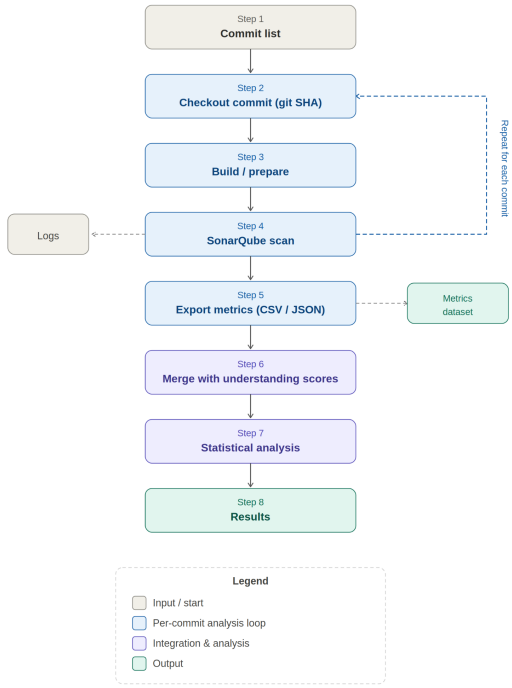


Figure 1. Methodology overview.

3.1.1. Ethics and Data Consent

This study was conducted with approval from the Head of the Department of Computer Science and Engineering, SUST, under the department's human-subjects research procedures. Participation in the interview and comprehension-assessment component was voluntary; students provided informed consent prior to the interview, were informed that participation (or non-participation) would not affect their course grade, and could withdraw at any time without penalty. Repository and participant identifiers were pseudonymized at the point of data collection and stored separately from any personally identifying information; only pseudonymized identifiers were used during SonarQube analysis and subsequent reporting. Inclusion in the quantitative (repository-level) phase required that a repository contain at least one commit during the semester; inclusion in the qualitative (interview) phase required enrollment in CSE344 and voluntary consent.

3.2. Material

In this study two types of material are utilized: software tolls for static analysis and instruments to asses student. For analyzing the source code, 48 students' GitHub repositories along with entire commit histories were collected. Sonarqube was used to perform static analysis, deployed as a Docker container via Docker compse to ensure consistent, easily reproduceable analysis environment across all repositories and commits. Analysis metric outputs and logs were stored in a PostgreSQL database, which will keep track of the previously analyzed projects among multiple sessions, help structured querying and tracking of quality indicators at the per-commit level.

For the student evaluation phase, a semi-structured interview process was designed with a code comprehension rubric used to rate student explainability responses on a 0-10 scale. The interview instrument also includes a self-report survey component documenting the frequency and purpose of students' LLM usage throughout the project, including generating, debugging, explaining, and refactoring tasks. Together, These materials allowed researchers to connect objective quality measurements with subjective understanding scores across the identical student sample.

3.3. Experimental procedure

We went through each repository's history one commit at a time, from oldest to newest. For every commit, we checked out the code and ran SonarQube on it to collect maintainability and other quality metrics. Each snapshot was saved together with the repository ID, the commit hash, and the commit timestamp.

Key metric categories include maintainability (code smells, maintainability rating, remediation effort), technical debt (debt ratio, SQALE), complexity (cyclomatic and cognitive complexity), duplication (duplicated lines/blocks), reliability (bugs), and security (vulnerabilities/hotspots).

3.4. Questionnaire formulation and administration

Semi-structured interviews were conducted with 104 students. At first they shared a report which contains how they utilized LLM in their projects (frequency and purpose: code generation, learning the concepts, debugging, explanation, refactoring). Then to measure their understanding each of them were asked to explain three code snippet from their repositories. Code snippets are taken from different maintainability profiles (low maintainability, high maintainability, duplication/buggy) then reflective questions about confidence in modifying/extending the codebase. Their level of understading was rated on a 0 - 10 scale, where 0 indicates no meaningful explanation and 10 indicates complete and accurate explanation of how the code works, structure and design decisions.

3.5. Sample size

Quantitative phase: 48 repositories and 3,043 commits, with team sizes ranging from 1 to 3 students per repository.

Qualitative phase: 104 interview participants (voluntary, consented). Sampling aims to represent a range of maintainability profiles identified in the quantitative phase.

4. Results and discussion

This section presents and interprets the findings from both the commit-level SonarQube static analysis of 48 student repositories and the structured explainability interviews conducted with 104 students. Results are organized by analysis standard-maintainability and quality metrics, followed by student explainability, before being integrated in a joint discussion of the relationship between code quality and code comprehension in an LLM-assisted course environment.

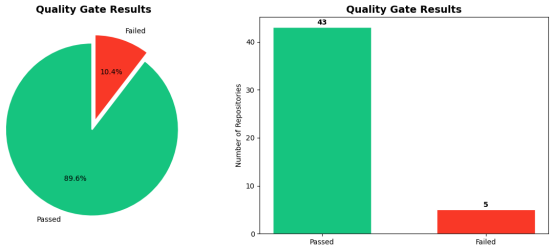


Figure 2. Quality gate results.

Table 1. SonarQube quality gate and security/reliability findings across 48 repositories

Metric / Category	Count (n=48)	Prop. (%)	Severity
Quality Gate: Failed	5	10.4%	—
Quality Gate: Passed	43	89.6%	—
Security Status: Blocker	9	18.75%	Blocker
Security Status: High	2	4.17%	High
Reliability Status: Blocker	2	4.17%	Blocker
Reliability Status: High	6	12.5%	High
Reliability Status: Medium	32	66.67%	Medium
Security Hotspots (present)	42	87.5%	Review req.

4.1. RQ1: Repository-Level Maintainability Outcomes

4.1.1. Quality Gate Status

The SonarQube Quality Gate was passed by 43/48 repositories (89.6%), with 5/48 (10.4%) failing (Figure 2; Table 1). Although the pass rate appears high, a Quality Gate pass reflects minimum threshold compliance rather than overall code health. Repositories with a small absolute size can pass while still carrying disproportionate technical debt and smell density. The five failing projects represent cases in which quality issues were pervasive enough to fall below even baseline standards.

4.1.2. Security Findings

9/48 repositories (18.75%) had a blocker-level security grade, while another 2/48 (4.17%) were classified as high severity, indicating that roughly one in every four projects had major security vulnerabilities. More broadly, **security hotspots were found in 42/48 repositories (87.5%)**, showing that security-sensitive code patterns were prevalent throughout the cohort, independent of Quality Gate outcome.

This prevalence is consistent with prior findings that LLM-generated code frequently reproduces known security anti-patterns [7, 8], and suggests that students neither recognized nor remediated these issues during development.

Figure 4 reports the distribution of security review ratings across the 48 repositories, where each repository is

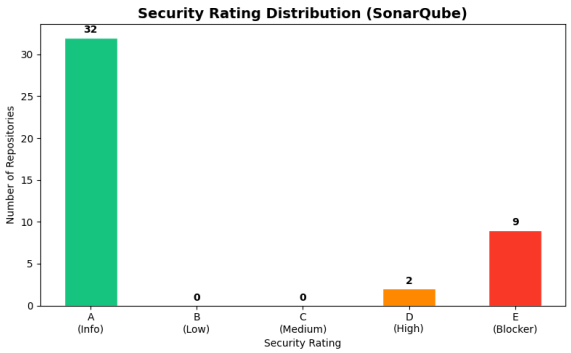


Figure 3. Security rating distribution.

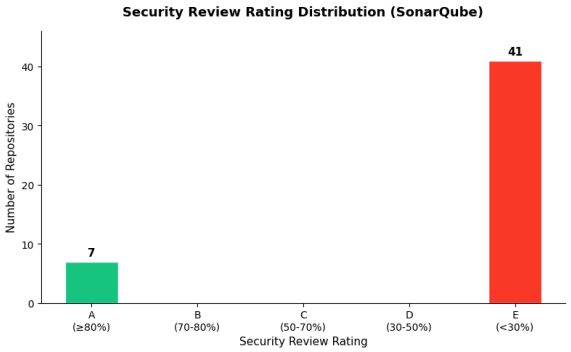


Figure 4. Security review rating distribution.

Table 2. Student explainability score distribution ($n = 104$, scale 0–10)

Score Band	Students (n=104)	Prop. (%)	Interpretation
0 (No explanation)	∼2	∼1.9%	Cannot explain the code
0.1–5.0 (Low)	∼31	30%	Partial/inadequate
≤ 5.0 (total)	33	32%	Limited explainability
5.1–7.0 (Moderate)	50	∼48%*	Adequate with gaps
7.1–10.0 (High)	21	∼20%*	Strong comprehension

classified by the proportion of flagged security hotspots that had been reviewed (rated A) versus left unreviewed (rated C or lower) at the time of final analysis. In the course context, a hotspot represents security-sensitive code (e.g., input handling, authentication logic, or database queries) that SonarQube flags for manual developer judgment rather than treating as an automatic defect. The distribution skews toward the lower (unreviewed) end, indicating that most students did not revisit or triage flagged hotspots even when their repositories otherwise passed the Quality Gate, consistent with the broader pattern of security-sensitive code being generated but not critically examined.

4.1.3. Reliability Findings

Reliability issues were broadly distributed: 2/48 (4.17%) reached blocker level, 6/48 (12.5%) were rated high severity, and the modal category was **medium severity at 32/48**

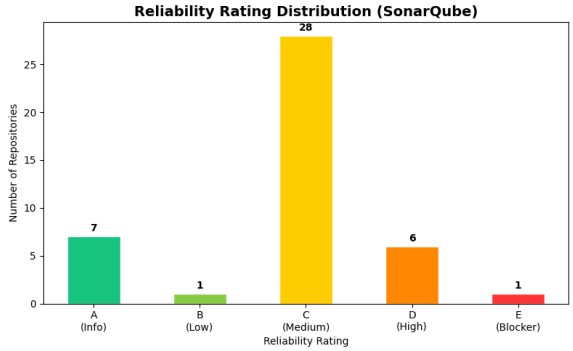


Figure 5. Reliability rating distribution.

(66.67%).

Figure 5 reports the distribution of SonarQube reliability ratings (Blocker, High, Medium) across the 48 repositories, where the rating reflects the most severe unresolved bug class detected in each codebase rather than the total bug count. While few projects had acute (blocker-level) correctness failures, the predominance of medium-severity bugs is notable — these manifest only under specific conditions and are precisely the defects that students relying on happy-path testing are least likely to detect.

4.1.4. Technical Debt and Code Smells

Code smells and SQALE technical debt were found in all 48 repositories. Failing repositories had consistently larger debt-to-size ratios, and their commit-level trajectories showed monotonic worsening with no noticeable recovery towards submission, indicating that refactoring was deprioritized throughout the development cycle.

4.2. RQ2: Student Explainability Outcomes

4.2.1. Score Distribution

33% of total students (n - 104) scored 5/10, and some scored lower than 5, which means a limited ability to explain code from their own repositories; around 2% scored at or near 0; they were completely blank and could not provide any explanation (Table 2). The study shows that almost one-third of the students scored at or below the midpoint on explaining the code written by them; it is evidence that there is an ownership gap among the participants. There might be other reasons too, like interview anxiety, unfamiliarity with the rubric, or code segments selected for review that a given student did not personally author (see Section 3.1.1). We treat the score distribution itself as the evidence for lower code ownership, but not the only possible explanation.

4.3. RQ3: LLM Use and Comprehension

Students who self-reported using LLMs primarily for *code generation* tended to fall in the low explainability band. In contrast, those who used LLMs interactively—for explanation or refactoring—generally scored higher. Although causality cannot be established from self-report data, the pattern suggests that *how* LLMs are used matters as much as whether they are used [1].

4.3.1. Measurement Caveats and Quantitative Comparison

Two measurement limitations should be noted when interpreting the pattern above. First, LLM usage frequency and purpose were captured entirely through student self-report in the interview protocol (Section 3.4); no independent artifact (e.g., IDE telemetry, prompt logs, or commit-message tagging) was used to verify whether, or how much, code in a given repository was actually LLM-generated. The association reported here should therefore be read as a relationship between reported usage pattern and comprehension score, not as a validated causal or even fully verified behavioral claim. Second, the comparison between generation-focused and interactive users is reported here descriptively rather than inferentially: LLM-usage style was recorded narratively during interviews rather than as a structured, per-student categorical field, so a Mann-Whitney U test could not be reliably applied to this sample; capturing usage style as a discrete variable is noted as a data-collection improvement for future work (Section 5). A Spearman rank correlation was computed between repository-level technical debt ratio (SonarQube `sqale_debt_ratio`) and each repository's average explainability score ($n = 48$ repositories, with team-averaged scores drawn from the $n = 104$ individual student explainability scores); the relationship was weak, negative, and not statistically significant ($\rho = -0.12$, $p = 0.43$), indicating no reliable association between this quality indicator and comprehension in the present sample. We flag both points as scope limitations addressed further in Section 5.

Placed alongside prior work, the 18.75% blocker-level security rate observed here is lower than the approximately 40% vulnerability rate Pearce et al. [8] report for GitHub Copilot-generated code snippets, though the two are not directly comparable: Pearce et al. evaluate isolated generated completions against known CWE categories, whereas this study evaluates whole student repositories that mix LLM-assisted and manually written code across a semester. The comparison nonetheless suggests that repository-level security risk in this cohort is broadly consistent with, rather than an outlier from, the security-risk profile reported for LLM-generated code elsewhere in the literature.

5. Limitations and Future Work

This study has several limitations. LLM usage was captured through self-report rather than independently verified via IDE telemetry or prompt logs, so the reported relationship between usage style and comprehension should be read descriptively rather than causally. The repository-to-student attribution procedure for group-authored repositories relies on commit metadata, which can misattribute pair-programmed or heavily reviewed code. A Spearman correlation between repository-level technical debt ratio and average explainability score was performed ($\rho = -0.12$, $p = 0.43$, $n = 48$ repositories) and did not reach statistical significance, so the quality-comprehension relationship discussed above should be read as suggestive rather than confirmed. A formal group comparison (e.g., a Mann-Whitney U test) between generation-focused and interactive LLM users could not be performed because usage style

was recorded narratively rather than as a structured categorical variable during data collection. Finally, the study is drawn from a single course (CSE344) at one institution and one semester, which may limit generalizability to other course designs, programming languages, or institutional contexts.

Future work should (i) replicate the quality-comprehension correlation with a larger, adequately powered sample to determine whether the weak, non-significant relationship observed here reflects a true absence of association or limited statistical power, (ii) capture LLM-usage style (generation-focused vs. interactive) as a structured, per-student categorical variable — alongside an independent measure of LLM involvement such as IDE-level generation logs — to enable a rigorous group comparison and complement self-reported usage, and (iii) replicate the design across additional course offerings and institutions to test whether the observed patterns generalize beyond this cohort.

6. Conclusion

This study demonstrates that while Large Language Models (LLMs) allow rapid code production among students, a substantial share of the resulting repositories carry unresolved maintainability, reliability, and security risk, and a non-trivial share of students are unable to explain the code they submitted. This research, utilizing commit-level SonarQube analysis on 48 repositories (3,043 commits), found that 18.75% of projects had blocker-level security vulnerabilities and 87.5% had security hotspots, despite an 89.6% Quality Gate pass rate. In parallel, 33% of students scored 5.0/10 or worse on explanation exercises, with 2% unable to explain their code at all. Taken together, these two independently measured signals repository level risk and individual explainability — are consistent with, though do not on their own prove, a pattern of students submitting code without fully reviewing or internalizing it. Because of this, the paper calls for better ways to assess project-based computing courses including automated quality checks integrated into grading, assignments where students explain their work aloud, and security-awareness training to encourage genuinely healthy code and durable learning. The principal novelty of this study is the joint, within-cohort measurement of both signals across an entire authentic semester, rather than treating code-quality analysis and comprehension assessment as separate research strands.

Acknowledgment

The authors thank the students of CSE344: Web Technology for their participation in the interview study and for consenting to the use of their repositories for academic research.

References

[1] Zi, Y. et al., *"I Would Have Written My Code Differently": Beginners Struggle to Understand LLM-Generated Code.* Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25). 2025, https://arxiv.org/abs/2504.19037

[2] Pitts, G. et al., *A Survey of LLM-Based Applications in Programming Education: Balancing Automation and Human Oversight.* arXiv preprint. 2025, https://arxiv.org/abs/2510.03719

[3] McCabe, T.J. and Butler, C.W. (1989). Design Complexity Measurement and Testing. *Communications of the ACM*, 32, 1415–1425. https://doi.org/10.1145/76380.76382

[4] Chen, L. et al., *Recent advances in large language models for code generation.* ACM Computing Surveys. 2024, 56(8); 1–40. https://doi.org/10.1145/3747588

[5] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," in *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994, https://doi.org/10.1109/32.295895

[6] Abbassi, R. et al., *A Taxonomy of Inefficiencies in LLM-Generated Python Code.* arXiv preprint. 2025, https://doi.org/10.48550/arXiv.2503.06327

[7] Siddiq, M.L. et al., *An Empirical Study of Code Smells in Transformer-Based Code Generation Techniques.* Proceedings of the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2022). 2022, https://doi.org/10.1109/SCAM55253.2022.00013

[8] Pearce, H. et al., *Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions.* Proceedings — 43rd IEEE Symposium on Security and Privacy (SP 2022). 2022; 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[9] ISO/IEC 25010:2021 Software Quality Model Committee. *Maintainability as a software quality attribute. ISO Standards Documentation.* 2021, https://www.iso.org/standard/78176.html

[10] Lenarduzzi, V., Saarimäki, N., Taibi, D., *On the Accuracy of SonarQube Technical Debt Remediation Time.* Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering (EASE). 2019, https://ieeexplore.ieee.org/document/8906700

[11] GitClear Research. *AI Copilot Code Quality: 2025 Data Suggests 4x Growth in Technical Debt.* 2024, https://www.gitclear.com/ai_assistant_code_quality_2025_research

[12] Lenarduzzi, V., Saarimäki, N., Taibi, D., *Some SonarQube issues have a significant but small effect on faults and changes: a large-scale empirical study.* Journal of Systems and Software. 2020, 170; 110750. https://doi.org/10.1016/j.jss.2020.110750

[13] SonarSource. *Cyclomatic complexity: Developer's guide.* 2025, https://www.sonarsource.com/resources/library/cyclomatic-complexity/

[14] Denny, P., Kumar, V., Giacaman, N., *Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language.* Proceedings of the 54th ACM Technical Symposium on Computer Science Education (SIGCSE '23). 2023; 1136–1142. https://doi.org/10.1145/3545945.3569823

[15] Prather, J. et al., *The Robots are Here: Navigating the Generative AI Revolution in Computing Education.* Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2023). 2023; 108–159. https://doi.org/10.1145/3623762.3633499

[16] Apache Software Foundation. *Apache Project Repository.* 2024, https://apache.org/

[17] Denny, P. et al., *Computing Education in the Era of Generative AI.* Communications of the ACM. 2024, 67(2); 56–67. https://doi.org/10.1145/3624720

[18] Microsoft. *Code metrics — Cyclomatic complexity. Visual Studio Documentation.* 2024, https://learn.microsoft.com/en-us/visualstudio/code-quality/

[19] Stack Overflow. *Developer Survey 2024: Developer tool usage and AI adoption trends.* 2024, https://survey.stackoverflow.co/

[20] Pylint Development Team. *Pylint: Code analysis and quality assessment tool.* 2024, https://www.pylint.org/

[21] PMD Project Team. *PMD: Static source code analyzer.* 2024, https://pmd.github.io/

[22] GitHub Copilot Research Team. *AI code generation adoption and impact study.* 2024, https://github.blog/research/

[23] Radon Project. *Python complexity metrics.* 2024, https://radon.readthedocs.io/en/latest/