Exploring Technical Debt in AI-assisted Code Generation

Laura Quiroga-Sánchez
Dept. of Computer Science
University of British Columbia
Kelowna, Canada
lquiroga@student.ubc.ca

Mariya Putwa
Dept. of Computer Science
University of British Columbia
Kelowna, Canada
mputwa@student.ubc.ca

Prateek Balani
Dept. of Computer Science
University of British Columbia
Kelowna, Canada
pbalani@student.ubc.ca

Richard Pillaca Burga
Dept. of Computer Science
University of British Columbia
Kelowna, Canada
rikepilb@student.ubc.ca

***Abstract*—In today's rapidly evolving technological landscape, generative AI tools are increasingly being adopted to streamline software engineering tasks and boost developer efficiency, yet their impact on long-term code maintainability and quality remains unclear. This study addresses this gap by examining how AI-assisted code differs from purely human-written code in terms of three main aspects: levels of technical debt, refactoring patterns, and types of code smells. To investigate these differences, we curated a dataset of 10 Python repositories that included AI-assisted contributions and 10 comparable human-written repositories. We used SonarQube to assess technical debt and code smells through its standardized tagging system, and PyRef to analyze refactoring activity and code evolution. Our analysis reveals no statistically significant differences between the two groups, challenging the expectation that GenAI-generated code inherently introduces more technical debt. However, with only 20 repositories, our findings may not fully capture broader industry trends, and the small sample size limits the statistical power to detect subtler differences. We view this as a first step toward a more comprehensive understanding of GenAI's impact on long-term code quality, encouraging further studies with larger and more diverse datasets.**

Key words: AI-Assisted Code, Technical Debt, Code Maintainability, Refactoring Patterns, Code Smells, SonarQube, PyRef, Generative AI Tools

I. Introduction

Generative Artificial Intelligence (GenAI) tools are becoming increasingly integrated into modern software development workflows. Tools like ChatGPT and GitHub Copilot promise to accelerate programming tasks and improve developer productivity by offering suggestions or even entire code snippets to their users. However, their broader implications for software engineering (SE) practices, particularly those regarding long-term code quality, remain under-explored. While developers appreciate the immediate advantages, growing concerns in the software engineering community focus on whether AI-generated code may subtly increase technical debt—through design compromises, persistent code smells, or other issues that hinder future modifications.

Technical debt, defined as the future cost incurred when choosing expedient over sustainable design solutions, is critical for evaluating the evolution of software quality. Most studies on AI code generation have concentrated on immediate outcomes such as functional correctness or performance on benchmark datasets like HumanEval and MBPP. Although these evaluations show that AI can produce working code, they tend to overlook how code evolves through bug fixes, refactoring, and feature updates. Early indications suggest that AI-generated code may harbor maintainability issues: for instance, Li et al. [7] identified inefficient logic and redundant loop structures in Copilot-generated code, while Liu et al. [8] observed deep nesting and excessive parameterization in ChatGPT-produced code. Additionally, Yetistiren et al. [12] have reported quality variations and latent security vulnerabilities through static analysis using SonarQube. However, these studies often offer only a snapshot of code quality or analyze isolated snippets rather than the full lifecycle effects.

This study bridges that gap by systematically comparing AI-assisted and human-written code in real-world software repositories. Our main objective is to investigate whether generative AI involvement affects long-term maintainability, particularly by altering technical debt accumulation and refactoring practices. We focus on public repositories—ensuring that our analysis reflects authentic development scenarios rather than artificially constructed examples—so that the true impact of AI assistance on code evolution can be captured.

The remainder of this paper is structured as follows: Section II introduces the research questions that guide our investigation. Section III reviews the related work and highlights the novelty of our study. In Section IV, we outline the dataset and explain the data collection process. Sections V and VI provide a detailed description of the methodology used and the results obtained for each research question, respectively. Section VII offers an interpretation of our findings, while Section VIII discusses potential threats to the validity of our study. Finally, Section IX summarizes our conclusions and outlines directions for future work.

II. Research Questions

To comprehensively evaluate the long-term maintainability of AI-assisted code, we pose three interrelated research questions:

RQ1: *Does code co-authored by GenAI exhibit higher or lower levels of technical debt compared to purely human-written code?*

As AI-assisted coding tools become more prevalent in professional development environments, it is essential to un-

derstand whether GenAI contributions introduce subtle maintainability challenges. One potential concern is that GenAI-generated code may lead to higher technical debt (manifesting as code smells, duplications, or even vulnerabilities) due to the model's limited contextual awareness or tendency to prioritize syntactic completion over design quality. These concerns are supported by Patel et al. (2024) [10], who observed that AI-generated solutions often feature higher cyclomatic complexity and recurring issues such as dead local stores and convoluted logic.

On the other hand, GenAI tools are trained on large code corpora that include established design patterns and best practices, which may enable them to produce cleaner, more maintainable code in some cases.

By empirically assessing technical debt in AI-assisted versus human-only codebases, this research question aims to provide quantitative evidence regarding the long-term impact of GenAI on code maintainability. To measure these indicators we used SonarQube across both AI-assisted and human-written repositories. SonarQube's built-in metrics, including code smells, bugs, and vulnerabilities, allow us to compare overall maintainability scores and highlight structural differences in quality between the two groups.

RQ2: *How do software developers refactor AI-assisted code in software repositories compared to human-written code?*

Even if GenAI code begins with issues, regular refactoring can mitigate this long-term. Conversely, if AI-assisted code commits consistently require extensive restructuring, then it will introduce hidden overhead for teams. Investigating real-world refactoring behavior helps illuminate whether AI-generated contributions create a manageable technical burden or introduce persistent maintenance challenges.

Prior work by Idrisov and Schlippe (2024) [5] highlights this concern, showing that while many AI-generated solutions are functionally correct, they frequently require manual refactoring to meet acceptable quality standards. Additionally, research by Zabardast et al. (2020) [14] found that unresolved technical debt, such as code smells or security vulnerabilities, often persists over time if not addressed early, reinforcing the importance of timely maintenance.

By analyzing refactoring patterns in AI-assisted versus human-written code, this research question aims to uncover how developers respond to potential quality issues and whether AI-generated code demands more frequent or intensive intervention. These findings can help shape practical guidelines for integrating GenAI tools into development workflows without sacrificing maintainability.

We used Pyref, a static analysis tool designed for Python, to detect and categorize refactoring operations across project histories. This enabled us to compare how frequently and in what ways developers refactor AI-assisted versus human-written code over time.

RQ3: *What types of code smells are most prevalent in code co-authored by GenAI compared to human-written code?*

Beyond evaluating overall technical debt levels, it is important to understand the specific quality issues that are more commonly introduced by AI-assisted code. This research question focuses on identifying the types of code smells that appear most frequently in GenAI co-authored code compared to purely human-written code. This detailed investigation can reveal the "signature" quality patterns of GenAI-generated code and pinpoint areas where technical debt may accumulate more easily. If certain types of code smells prove to be more prevalent in GenAI code, developers can be more vigilant in reviewing these constructs. Recognizing patterns allows teams to proactively manage maintainability risks in a timely manner. Prior research supports the need for this kind of analysis. Zhang et al. (2024) [15] found that 14.8% of Copilot-generated Python files exhibited code smells. Their findings also highlight how certain smells are more likely to arise in AI-generated code and may require targeted mitigation strategies. Additionally, Zabardast et al. (2020) [14] showed that technical debt, such as code smells, tends to persist in software projects if not addressed early. By mapping the smell landscape of AI-assisted code, this research question contributes to a deeper understanding of where and how AI-generated contributions may deviate from quality best practices, helping developers make informed decisions about reviewing and integrating such code.

To identify and categorize code smells, we used SonarQube's smell tagging system, which assigns descriptive labels (e.g., "bad practice", "redundant", "unused") to flagged code smells. This allowed us to systematically compare the distribution and the frequency of specific smell types between the two repository groups.

III. Previous Work

Recently, researchers have increasingly focused on the capabilities and limitations of GenAI code assistants, particularly in controlled settings such as competitive programming tasks. Several studies have used LeetCode problems as a benchmark to evaluate AI-generated code. For example, Coignion et al. (2024) [3] and Nguyen and Nadi (2022) [9] focused on code generation for small-scale LeetCode problems, evaluating the code in terms of correctness, complexity, and understandability.

Building on this, Baskhad Idrisov and Tim Schlippe (2024) [5] also examined AI-generated code using LeetCode problems, evaluating correctness and maintainability across various coding assistants. They found that while some tools produced correct solutions, the generated code often required manual refactoring. Their findings further emphasize the trade-offs between AI-generated and human-written code, particularly in terms of efficiency and long-term maintainability, aligning with the limitations observed in other studies. While these studies provide valuable insights into code correctness and quality aspects, their scope remains limited to isolated code snippets in artificial contexts.

In a similar vein, Patel et al. (2024) [10] conducted a comparative analysis of AI-generated and human-written Java code, focusing on software metrics and bug patterns in LeetCode solutions. Their study revealed that AI-generated code

exhibited higher cyclomatic complexity and recurring issues, such as dead local stores. Our approach draws from theirs in performing a comparative study between AI-assisted and human-written code, but we extend their work by focusing on real-world software development projects with extended lifespans and collaborative authorship, rather than short, well-defined problems.

Recent studies have also investigated other important aspects of AI-generated code, such as performance, maintainability, and security. For instance, Li et al. (2024) [7] analyzed Copilot-generated code across various datasets, including HumanEval, AixBench, and MBPP. Their study found that while the code often passes correctness checks, it can suffer from performance regressions, highlighting a gap in the performance stability of AI-generated code. Similarly, Yetiştiren et al. (2023) [13] compared various coding assistants in terms of quality metrics such as code validity, maintainability, and technical debt. Using the HumanEval dataset, their study uncovered variations in quality across these tools, revealing average code smells and approximate remediation times. While these studies investigate various aspects of code quality, they too are confined to benchmark datasets and do not fully capture the dynamics of real-world software projects.

Further investigating code quality, Zhang et al. (2024) [15] examined the prevalence of code smells in Copilot-generated Python code using Pysmell and evaluated the effectiveness of Copilot Chat in addressing these issues. Their findings revealed that 14.8% of analyzed files contained code smells, with Multiply-Nested Container (MNC) being the most common. These results underscore the need for comprehensive evaluation frameworks to assess AI-generated code quality and mitigate potential issues. Their study also emphasizes the importance of detecting and addressing common code quality problems to ensure maintainability over time.

Expanding on these concerns, Fu et al. (2025) [4] focused on identifying security vulnerabilities in Copilot-generated code snippets sourced from GitHub repositories. Their research uncovered widespread weaknesses in Python and JavaScript code, moving beyond controlled environments into real-world development settings. Although Fu et al. primarily concentrated on security concerns, their work highlights the importance of evaluating AI-generated code within actual open-source projects, where the dynamics of real-world collaboration and project evolution present unique challenges and risks.

Our study builds on these foundations by shifting the focus in several important ways. First, instead of analyzing isolated AI-generated snippets or benchmark problems, we investigate entire real-world repositories that are either AI-assisted or purely human-written. This dataset allows for a direct comparison, making it possible to assess not only the performance of GenAI tools but also their impact on overall code quality relative to traditional human development practices. Second, we emphasize technical debt, refactoring patterns, and code smells, essential yet under-explored dimensions of long-term code maintainability. By doing so, we aim to characterize the consequences of AI assistance beyond mere correctness or performance, focusing on the structural and evolutionary quality of software projects. Finally, we examine AI-assisted code as it appears organically in self-reported repositories, where developers acknowledge AI involvement. This offers a more realistic view of how these tools are being used in the wild, beyond controlled prompt-completion scenarios.

IV. Dataset

This study analyzes publicly available GitHub repositories, specifically Python projects developed with and without the assistance of generative AI tools. We curated a balanced dataset of 20 repositories: 10 purely human-written projects and 10 co-authored by generative AI tools such as GitHub Copilot and ChatGPT. To ensure a clear distinction between the two categories, human-written repositories were selected based on their last update dates being prior to 2021, before generative AI tools gained widespread traction in development workflows. In contrast, AI-assisted repositories were chosen from projects created after 2021 that explicitly mention the use of generative AI tools. Note that we only included projects where such mentions clearly apply to the entirety of the repository, rather than to individual commits, files, or one-off contributions.

Our entire data collection process was carried out between February and March 2025.

A. Data Collection Process

We first identified AI-assisted repositories and then selected comparable human-written repositories to enable a fair comparison.

In addition to the time-based separation outlined above, we applied several filtering criteria to ensure the quality and comparability of the dataset.

For AI-assisted repositories, the following requirements were enforced:

1) The repository must be publicly available on GitHub.
2) The primary programming language must be Python 3.
3) The repository must contain a minimum of 200 lines of Python code.
4) AI-assisted repositories must explicitly acknowledge generative AI co-authorship at the project level.
5) The repository must represent a complete and functional project, rather than a collection of isolated code snippets, toy examples, or partial implementations.

For human-written repositories, we applied the following criteria:

1) The repository must be publicly available on GitHub.
2) The primary programming language must be Python 3.
3) The repository's code must have been written and last edited before 2021.
4) The repository must be comparable to AI-assisted repositories in terms of size, number of collaborators, and commit history.
5) The repository must represent a complete and functional project, rather than a collection of isolated code snippets, toy examples, or partial implementations.

These filtering decisions were informed by our own domain expertise. Our goal was to create a dataset that balances relevance and quality, ensuring that selected repositories were substantial enough to support meaningful technical debt analysis. The minimum code size threshold helped exclude trivial projects, while the requirement for complete and functional projects ensured that they reflected realistic software development practices.

For AI-assisted projects, we required explicit acknowledgement of AI involvement at the project level to avoid ambiguity and reduce the risk of missclassification. Similarly, for the human-written group, the temporal cutoff ensured that the code predated the widespread adoption of AI coding assistance, increasing our confidence in its human authorship.

Finally, matching repositories by size, collaborator count, and commit activity allowed us to control for potential confounding variables, facilitating a more controlled and meaningful comparison between the two groups.

We adopted a hybrid approach to identify suitable AI-assisted repositories, combining multiple complementary strategies:

- **GitHub Advanced Search**: We leveraged GitHub's search functionality using keyword matching queries such as "co-authored by GitHub Copilot" OR "ChatGPT generated" OR "AI generated", to locate repositories that explicitly mentioned GenAI in their documentation. We restricted search results to repositories created after June 2021 (the release date of GitHub Copilot) as a temporal representation of GenAI availability. This keyword-based filtering aligns with methods used in prior studies [15] and [4].
- **Community Sources**: We monitored discussion forums such as Reddit and Stack Overflow to identify repositories openly shared by developers referencing the use of Ai tools in their workflow.
- **Literature Review**: We incorporated repositories identified in recent studies on AI-assisted code development, particularly those used by Fu et al. in [4].

Despite using multiple sources, it was still challenging to identify repositories that met all our criteria. In many cases, AI usage was mentioned at the file or commit level or only reported by a single contributor, making it difficult to attribute co-authorship to the entire project. To ensure the reliability of our dataset, we prioritized explicit documentation of AI usage over scale of the project and scale of the dataset, resulting in a smaller but more consistent set of AI-assisted repositories.

To reduce false positives and ensure accurate selection, two authors independently reviewed each candidate repository. They flagged and excluded projects where AI was mentioned in non-coding contexts (e.g., generating README files, images, or documentation) or where the mention of AI tools did not imply co-authorship. The reviewers compared their final selections, discussed discrepancies, and reached consensus on a final list of 10 AI-assisted repositories that met all inclusion criteria.

After finalizing the list of 10 AI-assisted repositories, we used the Software Engineering Archive Research Toolkit GitHub Search Engine (SEART GHS) to identify comparable human-written repositories. We ensured that these repositories followed a similar distribution in terms of size, number of commits, and number of contributors while also adhering to the previously established selection guidelines to mitigate the impact of confounding variables.

B. Dataset Composition

Our final dataset consists of 20 Python repositories, evenly distributed between AI-assisted and human-written categories. This balanced selection enables a direct comparison of technical debt indicators while controlling for key repository characteristics. To ensure fairness in comparison, we selected repositories with similar levels of contributor activity, commit frequency, and project maturity. These factors help mitigate potential biases that could arise from differences in repository scale or development stage. Table I summarizes descriptive statistics for the dataset.

TABLE I
DESCRIPTIVE STATISTICS OF THE REPOSITORY DATASET

Metric	Value	Description
Total Repositories	20	Total number of repositories
Total Commits	370	Sum of commits across repositories
Avg. Commits per Repository	18.5	Mean commits per repository
Total Contributors	32	Sum of contributor counts per repository
Avg. Contributors per Repository	1.6	Mean number of contributors per repository
Total Stars	4475	Sum of stars
Avg. Stars per Repository	223.8	Mean stars per repository
Repositories with Test Files	4	Number of repositories with test files

Due to ethical considerations and privacy concerns, we have pseudonymized repository names and links in our published materials. This approach safeguards repository owners' privacy while maintaining scientific rigour through transparent selection criteria. Despite this pseudonymization, we have retained and analyzed all relevant metadata—including commit histories, contributor counts, and code quality metrics—ensuring a robust evaluation of technical debt in both AI-assisted and human-written code.

Our dataset is publicly available to facilitate reproducibility and future research: dataset.

V. METHODOLOGY

For each research question, we outline the specific methods and tools that were be used to analyze the dataset and extract relevant insights.

RQ1: *Does code co-authored by GenAI exhibit higher or lower levels of technical debt compared to purely human-written code?*

A. Technical Debt Measurement

We identified a dataset containing 10 purely human-written repositories and 10 AI-assisted repositories, as detailed in Section IV. To measure technical debt we used SonarQube reports. We analyzed only the most recent version of each

repository available on GitHub as of March 31, 2025. This allowed us to assess the current maintainability state of the code, independent of its development history.

To ensure consistency, we manually identified projects that included test files and excluded them from the analysis when running the SonarQube analysis from its API. This step was taken to avoid inflating the technical debt metrics with auxiliary code not directly related to core functionality.

To quantify technical debt, we leveraged multiple metrics extracted from the resulting reports, each representing different aspects of maintainability, security, and code quality. The extracted initial key metrics are shown in Table II.

TABLE II
INITIALLY SELECTED SONARQUBE TECHNICAL DEBT METRICS

Metric Name	Metric ID	Definition
Lines of Code	$ncloc$	Number of lines of code
Bugs	$bugs$	Number of bugs detected
Code Smells	$code\_smells$	Number of code smells
Technical Debt	$sqale\_index$	Technical debt in minutes
Maintainability	$sqale\_rating$	Maintainability rating (A to E)
Vulnerabilities	$vulnerabilities$	Number of security vulnerabilities
Blocker Issues	$blocker\_violations$	Number of blocker severity issues
Critical Issues	$critical\_violations$	Number of critical severity issues
Major Issues	$major\_violations$	Number of major severity issues
Minor Issues	$minor\_violations$	Number of minor severity issues
Informational Issues	$info\_violations$	Number of informational severity issues

We first conducted a near-zero variance analysis to discard uninformative features. The metrics *vulnerabilities*, *blocker_violations*, *info_violations*, *bugs*, and *sqale_rating* were dropped due to minimal variance across repositories.

To ensure a fair comparison between repositories of different sizes, we normalized the remaining metrics by the number of non-comment lines of code ($ncloc$), following the methodology in Jin et al. (2023) [6].

We then conducted a correlation analysis using Pearson correlation coefficients to identify potential redundancy among variables. This analysis revealed a strong correlation between *sqale_index* and *code_smells*, indicating multicollinearity. Since *sqale_index* provides a direct estimate of technical debt, we chose to retain it and discard *code_smells*.

The final set of metrics used in the study is mentioned in Table III.

TABLE III
FINAL SET OF TECHNICAL DEBT METRICS FOR RQ1

Metric Name	Computation
Technical Debt Density	$bug/ncloc$
Critical Issue Density	$critical\_violations/ncloc$
Major Issue Density	$major\_violations/ncloc$
Minor Issue Density	$minor\_violations/ncloc$

B. Statistical Analysis

Given the small dataset size, we assumed a non-normal distribution of the metrics. We first created box plots to visually evaluate the median and distribution differences between AI-assisted and purely human-written code.

To statistically assess whether significant differences exist, we conducted the Mann-Whitney U test for each metric. Since multiple comparisons were made on the same dataset, we applied a Benjamini-Hochberg (BH) correction to the p-values to control the false discovery rate (FDR). This correction was chosen over the Bonferroni correction to balance Type I and Type II errors, given the exploratory nature of our study.

Statistical significance was determined at a threshold of $\alpha = 0.05$, meaning that only metrics with BH-corrected p-values below this threshold were considered significantly different between human-written and AI-assisted projects.

RQ2: *How do software developers refactor AI-assisted code in software repositories compared to human-written code?*

A. Refactoring Detection with PyRef

PyRef[1] is a detection tool designed to automatically find refactoring operations in Python repositories. It is specifically made for Python, the language of our chosen repositories, and it helps to objectively identify refactoring actions across the dataset by analyzing the commit history and matching code changes to predefined refactoring patterns. However, like any automated tool, PyRef might not detect all refactorings (especially complex or manually performed ones) and may occasionally misidentify changes. Despite this, it provides a consistent approach for detecting common refactorings across all repositories in our study. The authors manually reviewed a sample of detected refactorings as part of the qualitative analysis to contextualize PyRef's output.

B. Refactoring Categorization Scheme

To understand the *types* of refactoring carried out beyond simple counts, we developed a categorization scheme to group Pyref's detected operations into meaningful categories based on their primary impact on the code. This scheme was developed through discussion and consensus among the authors. We logically grouped the specific method-level refactorings identified by PyRef based on their primary impact, aligning with common refactoring goals such as improving code understanding (Naming), managing interfaces (Parameters), controlling internal complexity (Composition), and organizing code structure (Movement). The rationale was to create conceptually distinct groups representing common developer intents during maintenance. A qualitative review examined the dataset on examples within these categories, providing preliminary insights into their real-world application and conceptual coherence. The resulting categories used in our analysis are:

1) **Naming Improvements:** Enhance code clarity through better naming.
   - Includes PyRef's: *Rename Method*, *Rename Class*, *Rename Variable*.
2) **Parameter Modifications:** Change method signatures for clarity or functionality.
   - Includes PyRef's: *Add Parameter*, *Remove Parameter*, *Change Parameter*, *Rename Parameter*.

3) **Method Composition:** Alter method structure by extracting or merging code.
   - Includes PyRef's: *Extract Method*, *Inline Method*.
4) **Method Movement:** Relocate methods within the class hierarchy or to other classes.
   - Includes PyRef's: *Move Method*, *Pull Up Method*, *Push Down Method*.
5) **Other:** A catch-all for any refactoring type detected by PyRef that did not map to these four primary categories was implicitly grouped as 'Other' during data aggregation. Due to its low frequency in our dataset, this 'Other' category was not included in the primary Chi-Squared analysis comparing distributions.

C. Quantitative Data Analysis

To compare refactoring quantitatively, a Python script `aggregate_metrics.py` was executed. This script cloned each repository, ran PyRef, extracted commit metadata using GitPython, and calculated several metrics per repository. These metrics include:

- **Total Refactorings:** The sum of all refactoring events identified by PyRef.
- **Refactoring Category Counts:** The number of refactorings falling into each of the categories defined above (Naming Improvements, Parameter Modifications, Method Composition, Method Movement).
- **Refactoring Commits Percentage:** The percentage of total commits containing at least one refactoring operation detected by PyRef. Calculated as: $(Num.of Refactoring Commits/Total Commits) * 100\%$.
- **Average Time-to-Refactor (sec):** The average time interval between consecutive refactoring commits (calculated only for repositories with ≥ 2 refactoring commits).
- **Refactoring Timestamp Difference (days):** The difference between the average timestamp of refactoring commits and the average timestamp of all commits, indicating whether refactorings tend to occur earlier (negative value) or later (positive value) in the project timeline. Calculated as: $(AvgTimestamp_{RefactoringCommits} - AvgTimestamp_{AllCommits})/(86400)$.
- **Number of Refactoring Contributors:** The count of unique developers (identified by email) who authored commits containing refactorings detected by PyRef.

Statistical tests were then applied using the `statistical_analysis.py` script to compare these measurements between the AI-assisted and human-written repository groups:

- **Chi-Squared Test:** Used to compare the distribution of refactoring counts across the four main defined categories (Naming Improvements, Parameter Modifications, Method Composition, Method Movement) between the two repository types. This determines if the relative proportions of different refactoring types differ significantly. Thus, the test assesses the null hypothesis that the proportional distribution of refactoring counts across categories is the same for both AI-assisted and human-written groups.
- **Mann-Whitney U Tests:** Used to compare the distributions of the numerical metrics (Total Refactorings, Refactoring Commits Percentage, Refactoring Timestamp Difference, Number of Refactoring Contributors) between the two groups. This non-parametric test was chosen due to the small sample size and potential non-normality of the data, checking for significant differences in the overall intensity or frequency of refactoring. Thus, the tests assess the null hypothesis that the underlying distribution of each intensity metric is the same for both groups.

D. Qualitative Code Review

To supplement the quantitative findings, an initial qualitative code review was conducted by one author. This involved manually examining code changes and commit messages for a subset of refactoring examples identified by PyRef in both AI-assisted and human-written repositories. This subset contained repositories that displayed the highest counts of refactoring within our predefined categories. The focus was primarily on examples falling into the "Naming Improvements" and "Parameter Modifications" categories, as these showed potential differences in the quantitative analysis. The goals were to:

- **Understand the reasons for refactoring:** Infer the developers' intent behind specific refactorings.
- **Analyze the nature of refactoring changes:** Observe the specific code modifications made.
- **Identify potential differences:** Look for qualitative distinctions in refactoring motivations or styles between the AI-assisted and human-written groups.

This exploratory review aimed to provide context for the statistical results. We acknowledge that involving only a single author in this qualitative step is a limitation that introduces potential subjectivity and restricts the ability to assess inter-rater reliability. Consequently, the qualitative insights should be considered exploratory, and future work would benefit significantly from employing multiple independent reviewers to validate these observations.

RQ3: *What types of code smells are most prevalent in GenAI co-authored code compared to human-written code*

A. Technical Debt Measurement

To examine the prevalence and types of code smells in AI-assisted versus human-written code, we leveraged SonarQube as our primary static analysis tool. SonarQube systematically scans source code and detects maintainability issues it classifies as code smells. Each code smell is tagged based on its nature, e.g., Bad Practice, Performance, or Convention, which helps in organizing them into meaningful categories for comparison.

For consistency, SonarQube was configured uniformly across all 20 repositories. This ensured that all detected smells were generated using the same version and configuration,

minimizing tool-based bias in the detection process. By isolating SonarQube's tag-based smell types, we were able to focus our analysis on the distribution of code smell categories, enabling a more fine-grained comparison between AI-assisted and human-written codebases.

B. Statistical Analysis

To determine whether the observed differences in technical debt between AI-assisted and human-written repositories are statistically significant, we used a two-step approach :

- **Normalization:** For each repository, we calculated the proportion of each smell tag relative to the total number of smells identified.
- **Mann-Whitney U Test:** We used Mann-Whitney U Test to each smell tag category to assess whether the distribution of normalized code smells differed significantly between AI-assisted and human-written codebases.

Given the multiple comparisons (one for each smell tag), we applied a Benjamini-Hochberg correction to the resulting p-values to control for the false discovery rate. This correction was particularly important to reduce the risk of Type I errors in our exploratory analysis.

VI. Results

A. *Results for RQ1*

1) Comparison of Technical Debt Metrics: Figure 1 presents the distribution of technical debt metrics for AI-assisted and purely human-written projects. Each box plot visualizes the median, inter-quartile range, and potential outliers for the normalized metrics. This comparison provides an initial exploratory view of whether AI-assisted code exhibits higher or lower levels of technical debt compared to human-written code.

To formally assess whether the differences observed in the metrics are statistically significant, we applied the Mann-Whitney U test to each technical debt metric. Since multiple tests were performed, we applied the Benjamini-Hochberg (BH) correction to control the false discovery rate.

2) Statistical Significance of Differences: Table IV reports the raw p-values obtained from the Mann-Whitney U test for each metric, alongside the adjusted p-values after applying the BH correction.

TABLE IV
Mann-Whitney U Test Results Technical Debt Metrics

Metric	p-value	Corrected p-value	Sign. ($\alpha = 0.05$)
Technical Debt Density	0.677585	0.677585	No
Critical Violation Density	0.405326	0.677585	No
Major Violation Density	0.676435	0.677585	No
Minor Violation Density	0.383957	0.677585	No

The results indicate no statistically significant difference in any of the technical debt metrics between AI-assisted and human-written projects.

B. *Results for RQ2*

The results were obtained from running a descriptive and statistical analysis of refactoring data on the dataset containing 10 human-written and 10 AI-assisted Github software repositories.

1) Distribution of Refactoring Categories: Chi-Squared Test: To determine if the visually observed differences in category usage reflect a statistically significant pattern overall, we performed a Chi-Squared test. The Chi-squared test compared the distribution of refactoring counts across the four main categories (Naming Improvements, Parameter Modifications, Method Composition, Method Movement) between gen-AI and human-written repositories.

The test did not reveal a statistically significant difference in the overall distribution across all four categories between the two groups ($\chi^2(3) \approx 6.55$, p ≈ 0.088) in our sample. Therefore, we cannot reject the null hypothesis that the distribution of refactoring types is the same ($\alpha = 0.05$).

Table V presents the observed counts used in the test, and Figure 2 visualizes these counts. Although statistical significance was not reached for the overall distribution, Figure 2 shows a noticeably higher absolute count of 'Naming Improvements' in the Human Written group compared to the AI-assisted group (21 vs. 2).

TABLE V
Observed Refactoring Counts per Category (Chi-Squared Input)

Refactoring Category	AI-Assisted (Obs.)	Human Written (Obs.)
Naming Improvements	2	21
Parameter Modifications	8	12
Method Composition	1	1
Method Movement	1	4

2) Overall Refactoring Activity: Mann-Whitney U Tests: Beyond the types of refactoring, we investigated whether the overall amount or intensity of refactoring differed between the groups. Mann-Whitney U tests were conducted on several numerical metrics. As shown in Table VI, these tests revealed no statistically significant differences between the AI-assisted and human-written groups for any of the measured intensity metrics (all p-values greater than 0.05). These results indicate that, based on these metrics, the overall frequency, proportion of commits involving refactoring, timing relative to project history, and number of contributors involved in refactoring were statistically similar between the two groups in our dataset.

TABLE VI
Mann-Whitney U Test Results for Numerical Metrics: Comparing Overall Refactoring Intensity

Metric	Mann-Whitney U Stat.	p-value	Sign. ($\alpha = 0.05$)
Total Refactorings	38.50	0.3475	No
Refactoring Commits %	35.00	0.2158	No
Refactoring Timestamp Difference (days)	42.50	0.5503	No
Number of Refactoring Contributors	37.00	0.2777	No

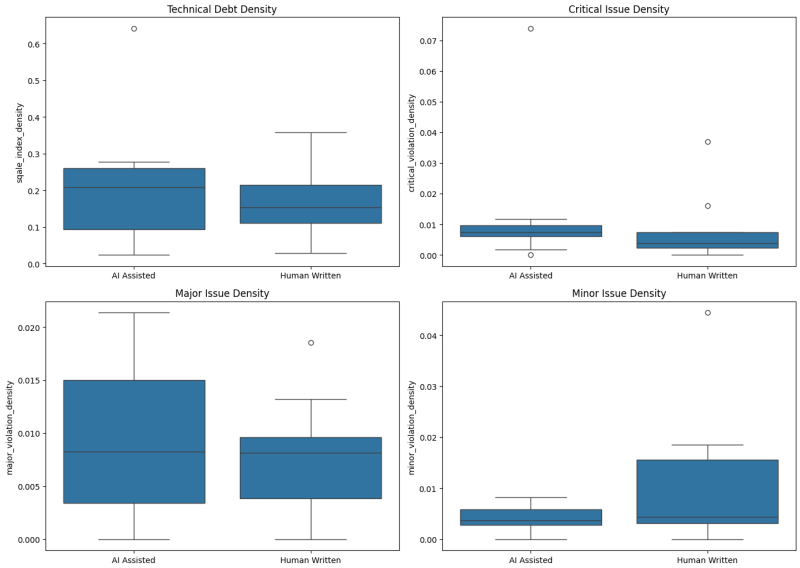

Fig. 1. Distribution of technical debt metrics for Human-Written and AI-Assisted Projects for RQ1.

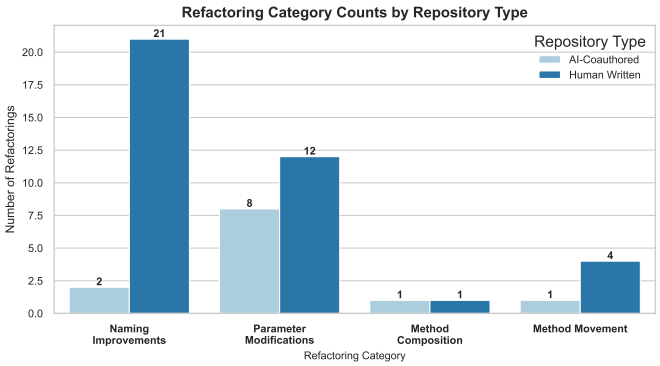

Fig. 2. Refactoring category counts by repository type

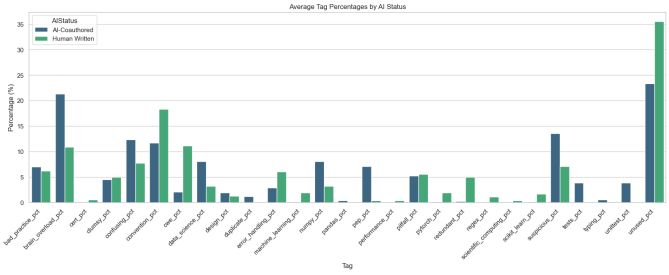

Fig. 3. Average Tag Percentage by AI Status

3) Qualitative Insights from Code Review: A preliminary manual review of a sample of refactoring instances provided context for the quantitative results. In human-written code examples categorized as 'Naming Improvements', commit messages often explicitly stated the goal of improving clarity (for example, "clarify naming"). In AI-assisted code examples categorized as 'Parameter Modifications', instances were observed where parameters were added to methods seemingly generated by AI, potentially for adapting them to specific project needs or adding controls. These qualitative observations are based on a limited sample reviewed by a single author and serve primarily as context rather than generalizable findings; further interpretation is reserved for the Discussion section.

C. *Results for RQ3*

1) *Code Smell Distribution Comparison:* Our initial analysis included 27 distinct SonarQube smell tags, ranging from high-level categories like Bad Practice and Performance to more domain-specific ones like Pandas, Regex, and Machine Learning. The relative frequency of each tag was calculated and compared between both groups (Figure 3).
2) *Statistical Findings:*The results from the Mann-Whitney U tests indicated no statistically significant differences between AI-assisted and human-written repositories for any individual smell category. Even after applying the Benjamini-Hochberg correction, all adjusted p-values were above the significance threshold ($\alpha = 0.05$) as shown in Table VII.

VII. Discussion

A. *Discussion for RQ1*

Our findings for RQ1 suggest no statistically significant variation in technical debt between codebases with and without AI assistance. This suggests that, based on our dataset, AI-assisted development does not inherently produce code that is more or less maintainable than human-written code, at least as measured by the selected SonarQube metrics. The distributions

TABLE VII
MANN-WHITNEY U TEST RESULTS FOR CODE SMELL CATEGORIES

Smell Tag	p-value	Corrected p-value
Bad Practice	0.81	0.91
Brain Overload	0.07	0.70
Cert	0.37	0.70
Clumsy	0.39	0.70
Confusing	0.56	0.80
Convention	0.69	0.85
Cwe	0.26	0.70
Data Science	0.23	0.70
Design	1.00	1.00
Duplicate	0.37	0.70
Error Handling	0.69	0.85
Machine Learning	0.50	0.80
Numpy	0.23	0.70
Pandas	0.37	0.70
Pep	0.28	0.70
Performance	0.37	0.70
Pitfall	0.76	0.89
Pytorch	0.50	0.80
Redundant	1.00	1.00
Regex	0.17	0.70
Scientific Computing	0.17	0.70
Scikit Learn	1.00	1.00
Suspicious	0.59	0.80
Tests	0.17	0.70
Typing	0.58	0.80
Unittest	0.17	0.70
Unused	0.29	0.70

shown in Figure 1 supports the findings from the statistical tests, indicating that both sets of projects exhibit a similar spread and central tendency across technical debt indicators.

One possible interpretation is that developers are effectively reviewing and integrating AI-generated code in ways that maintain consistent quality with their human-written counterparts. An important consideration is that AI-assisted code in our sample might have benefited from post-editing by humans, potentially neutralizing any initial quality differences. Alternatively, the capabilities of GenAI tools may have improved to the point that the maintainability of AI-generated code is comparable to that of experienced human developers.

However, it is also important to acknowledge potential limitations. The lack of significant findings may be partially attributed to the relatively small sample size (20 repositories), which reduces statistical power and makes it harder to detect subtle effects. Additionally, technical debt is just one aspect of software quality. It is possible that other dimensions, such as readability or architectural decisions, could reveal different patterns.

Overall, our findings challenge the assumption that AI assistance necessarily introduces greater technical debt. At the same time, they underscore the need for broader studies to fully understand how GenAI tools influence code quality in real-world projects.

B. Discussion for RQ2

Looking at how developers refactor or clean up code, our quantitative results were quite consistent: we found no statistically significant differences between the AI-assisted and human-written projects. This applied both to the types of refactoring done (four predefined categories, Chi-Squared $p \approx 0.088$) and the overall amount or frequency of these refactoring operations (Mann-Whitney U tests, all p-values greater than 0.05).

This lack of a significant difference in overall refactoring intensity contrasts with studies like Idrisov and Schlippe [5] which suggested AI code often requires manual refactoring. Our results, based on PyRef, did not quantify a higher frequency of these specific method-level changes, perhaps indicating that the 'manual refactoring' needed for AI code involves changed PyRef doesn't detect (such as bigger structural changes), or maybe developers in our sample reviewed the AI code well enough initially that frequent, detectable refactoring wasn't needed later.

The large, though not statistically significant for the overall distribution, difference in 'Naming Improvements' (2 counts in AI-assisted vs. 21 counts in human-written (Figure 2)) warrants analysis. It's likely because other refactoring types (such as Method composition and Movement)happened too infrequently in both groups in our small sample. This low activity in some categories reduces the power of the statistical test to confirm if the overall pattern is truly different, even if one category stands out visually. This specific observation about naming raises questions: Do developers simply focus more on clarity in code they write themselves? Or do current AI tools already generate reasonably clear names, reducing the need for this specific refactoring? It's hard to say from our numbers alone, and this highlights where talking directly to developers or conducting a more thorough qualitative investigation could provide valuable insight.

For developers, these findings tentatively suggest that adopting AI assistance, at least within the context studied, might not drastically increase the number of common refactoring tasks. However, developers might need to be extra vigilant during code reviews, perhaps focusing less on basic naming (if AI does that well) and more on ensuring the AI-generated code integrates correctly with the rest of the system, for example, by checking its inputs and outputs (parameters).

It's important to remember the limitations. Our sample size (20 projects) is small, potentially limiting our ability to detect real differences. Additionally, PyRef only focuses on very specific method-level refactorings, and could be missing other types of code changes or architectural adjustments where refactoring differences might exist. Furthermore, confounding factors like project age and specific AI tool versions were not fully controlled. Therefore, while our quantitative results show similarity for these specific metrics, further research is definitely needed. This should involve larger, more diverse datasets, perhaps different analysis tools (including qualitative ones), and better control over project characteristics to get a complete picture of how AI impacts code maintenance.

C. Discussion for RQ3

The results for RQ3 revealed that there were no statistically significant differences in the distribution of code smell categories between AI-assisted and purely human-written reposi-

tories. These findings suggest that, in the context of real-world repositories, GenAI code may not be inherently more prone to introducing specific types of maintainability issues.

One possible explanation is that AI-generated code often mirrors common coding patterns found in its training data, including a blend of good and bad practices, resulting in smell distributions similar to traditional code. Alternatively, developers working with AI-assisted code might refactor or review these contributions, mitigating any initial quality gaps before they become persistent smells.

VIII. Threats to Validity

The validity of this study is discussed in terms of internal, external, construct, and conclusion validity threats.

Internal Validity One potential threat to internal validity arises from the temporal gap between the two groups of repositories. The human-written repositories were last updated before 2021, while the AI-assisted repositories were selected from more recent projects. As a result, any observed differences in technical debt levels, refactoring patterns, or types of code smells—even if not statistically significant—may reflect changes in coding standards or best practices over time, rather than the direct impact of AI assistance. This temporal discrepancy could have introduced confounding factors beyond our control. This decision, however, was driven by the challenge of confidently identifying truly human-written projects in the post-AI era. Choosing current projects without explicit mentions of AI use would not guarantee the absence of AI-generated code, it would merely reflect a lack of disclosure.

Another limitation lies in the heterogeneity of AI tools used in the development of the AI-assisted repositories. The performance and behaviour of AI coding assistants vary significantly across tools and even across versions. For instance, early versions of GitHub Copilot likely provided very different suggestions compared to more advanced releases. This variations was not controlled in our study and may have introduced inconsistencies in the quality or nature of AI-generated code.

Additionally, human-AI collaboration styles may have impacted the results. Developers interact with AI in diverse ways, some might accept suggestions exactly as they come, while others might heavily modify them. Consequently, the lack of evidence to reject the null hypotheses may reflect the variability in how developers leverage AI assistance, rather than the absence of impact from AI itself. This means that the effect of AI-generated code on technical debt may have been diluted by the degree of human intervention during the development process. Lastly, AI involvement was inferred primarily through documentation such as README files or comments. However, this approach has limitations: developers may not accurately report their use of AI tools, leading to a gap between documentation and reality. In some cases, projects may be labeled as AI-assisted based on broad claims, even if AI was only used minimally. This can result in an overestimation of AI-involvement, blurring the distinction between AI-assisted and human-written code.

External Validity Several factors may limit the generalizability of our findings. First, the dataset includes only 20 repositories, many of which are relatively small in terms of codebase size and commit history. This limited scope may not reflect the broader trends or practices found in larger, production-level software projects. Second, a significant portion of the repositories appears to be maintained by individual contributors or small teams, which can lead to development dynamics, coding practices, and technical debt profiles that differ markedly from those found in larger-scale projects or those developed by a wider pool of developers.

Third, our analysis focused exclusively in Python projects. While Python is a widely-used language, the effectiveness of AI coding assistants can vary across programming languages [2]. As such, the conclusions drawn here may not readily apply to AI-assisted development in other languages like Java, JavaScript, or C++.

Lastly, our study takes a snapshot in time, examining the most recent state of the repositories as of March 2025. This approach does not account for the evolution of technical debt over time, which might be different for AI-generated and human-written code. Our findings may not capture these potential divergencies in long-term maintainability.

Construct Validity The way technical debt is measured in this study, such as through static analysis, code smells, and maintainability index, may not fully capture all aspects of technical debt in real-world scenarios. If these metrics overlook certain types of debt or fail to represent it accurately, the study's conclusions could be limited or skewed. Additionally, the classification of code smells based on SonarQube's predefined tags may not fully capture the complexities of different code quality issues. While SonarQube's categorization provides a useful and standardized approach to identifying code smells, it could potentially overlook some smells or misclassify others, leading to an incomplete representation of the code quality landscape. However, we believe that despite these limitations, SonarQube's predefined tags offer a valuable starting point for analyzing code quality, and the results should still provide meaningful insights into the differences between AI-assisted and human-written code.

Conclusion Validity One of the key limitations of our study is the limited statistical power due to the relatively small sample size of 20 repositories. Statistical tests with a small dataset may not have enough power to detect meaningful differences, and as a result, our ability to make definitive conclusions about broader trends is limited. Larger datasets would be needed to enhance the robustness and confidence in our conclusions, as they would allow for more reliable statistical testing and potentially uncover subtle differences in the data.

IX. Conclusion

The increasing integration of Generative AI tools into software development necessitates a deeper understanding of their impact on long-term code quality and maintainability.

To understand the impact of Generative AI tools on long-term code quality, this study empirically compared AI-assisted and purely human-written Python repositories across three key dimensions: technical debt levels (RQ1), refactoring patterns (RQ2), and code smell prevalence (RQ3). Using a curated dataset of 20 repositories (10 AI-assisted, 10 human-written) and employing established analysis tools (SonarQube for technical debt and smells, PyRef for refactoring), our quantitative analysis yielded a consistent finding: we found *no statistically significant differences* between the two groups across any of the three aspects studied.

Specifically, our results indicate that, based on the selected SonarQube metrics normalized by code size, AI-assisted code did not exhibit significantly higher or lower technical debt density or specific issue densities compared to human-written code (RQ1). Similarly, analysis of method-level refactoring activities detected by PyRef showed no significant difference in either the proportional distribution of refactoring types (RQ2, Categories) or the overall frequency and intensity of refactoring actions (RQ2, Intensity). Furthermore, the distribution of common code smell types, identified via SonarQube tags, did not differ significantly between the groups (RQ3).

These findings challenge the initial hypothesis that AI-assisted code might inherently possess poorer maintainability characteristics as measured by these common quantitative indicators. They suggest that, within the limitations of our study and the chosen measurement techniques, the readily quantifiable aspects of technical debt, refactoring behavior, and common code smells in AI-assisted projects may be more similar to those in human-written projects than often assumed. This could reflect effective developer oversight and integration practices, the improving capabilities of GenAI tools, or simply that the differences manifest in ways not captured by these specific tools and metrics (such as at the architectural level or in qualitative aspects).

However, the lack of statistically significant findings, particularly given the small sample size (N=20), underscores the need for caution and further investigation. Our study represents an initial step, highlighting that common quantitative tools may not reveal dramatic differences. Future work is essential to explore these questions with larger, more diverse datasets, employing a wider range of methodologies including qualitative analysis and architectural-level assessments, and controlling more carefully for contextual variables like distinct AI tools and models, their versions, and specific project characteristics. Understanding the nuanced, long-term impact of GenAI on software maintainability remains a critical area for future software engineering research.

Replication Package: We have set up a replication package including the dataset and the scripts used for the data extraction processes and the statistical analyses per RQ [11].

X. Authorship

Individual contributions are summarized as follows:

- **Laura Quiroga-Sánchez:** One of the two authors involved in the data collection process for the AI-assisted repositories. Created a bash script to automate the data extraction and execution of the SonarQube analysis for the RQ1. Performed the technical debt analysis over the initially selected metrics, finalized the set of metrics used for RQ1 and performed the respective statistical analysis.
- **Mariya Putwa:** One of the two authors involved in the data collection process for the AI-assisted repositories. Led the RQ2 analysis by developing Python scripts for the end-to-end pipeline (`aggregate_metrics.py`, `statistical_analysis.py`, `descriptive.py`, `visualizations.py`), encompassing data extraction (cloning, commit metrics, PyRef execution), refactoring categorization, metric aggregation, statistical testing, and visualization. Executed the pipeline on all repositories, performed the RQ2 quantitative analysis, interpreted the results, and conducted the initial qualitative review of refactoring examples.
- **Prateek Balani:** One of the authors who set up SonarQube initially and tested in a local environment to confirm proper functionality. The tool was subsequently applied in RQ3 to analyze code quality and identify key issues.
- **Richard Pillaca Burga:** Conducted an exploratory analysis of SonarQube's capabilities. Collected human-written repositories based on the established criteria. Performed a comprehensive literature review to support the methodology design and highlight the novelty of our study.

References

[1] Hassan Atwi et al. "PYREF: Refactoring Detection in Python Projects". In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2021. DOI: 10.1109/SCAM52516.2021.00025.

[2] Alessio Buscemi. *A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages*. 2023. arXiv: 2308.04477 [cs.SE]. URL: https://arxiv.org/abs/2308.04477.

[3] Tristan Coignion, Clément Quinton, and Romain Rouvoy. "A Performance Study of LLM-Generated Code on Leetcode". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE 2024. ACM, June 2024, pp. 79–89. DOI: 10.1145/3661167.3661221. URL: http://dx.doi.org/10.1145/3661167.3661221.

[4] Yujia Fu et al. *Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study*. 2025. arXiv: 2310.02059 [cs.SE]. URL: https://arxiv.org/abs/2310.02059.

[5] Baskhad Idrisov and Tim Schlippe. "Program Code Generation with Generative AIs". In: *Algorithms* 17.2 (2024), p. 62. DOI: 10.3390/a17020062.

[6] Siyuan Jin et al. "Software code quality measurement: Implications from metric distributions". In: *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE. 2023, pp. 488–496.

[7] Shuang Li et al. “Assessing the Performance of AI-Generated Code: A Case Study on GitHub Copilot”. In: *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2024, pp. 216–227.

[8] Yue Liu et al. “Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues”. In: *ACM Transactions on Software Engineering and Methodology* 33.5 (2024), 116:1–116:24.

[9] Nhan Nguyen and Sarah Nadi. “An empirical evaluation of GitHub copilot’s code suggestions”. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 1–5.

[10] Abhi Patel, Kazi Zakia Sultana, and B. Samanthula. “A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study”. In: *2024 IEEE International Conference on Big Data (BigData)*. 2024, pp. 7521–7529. DOI: 10.1109/BigData62323.2024.10825958.

[11] Mariya Putwa. *COSC-419O-Mining-Software-Repositories*. Accessed: 2025-04-08. 2023. URL: https://github.com/mariyaputwa/COSC-419O-Mining-Software-Repositories/tree/main.

[12] Burak Yetistiren et al. “Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT”. In: *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*. 2023, pp. 93–104.

[13] Burak Yetiştiren et al. “Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt”. In: *arXiv preprint arXiv:2304.10778* (2023).

[14] Ehsan Zabardast, Kwabena Ebo Bennin, and Javier Gonzalez-Huerta. “Further Investigation of the Survivability of Code Technical Debt Items”. In: *arXiv preprint arXiv:2010.05178* (2020).

[15] Beiqi Zhang et al. “Copilot-in-the-Loop: Fixing Code Smells in Copilot-Generated Python Code using Copilot”. In: *arXiv preprint arXiv:2401.14176* (2024). https://arxiv.org/abs/2401.14176.