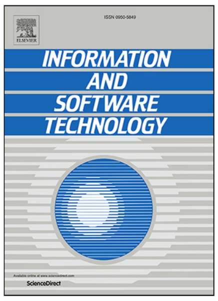
Usage patterns of software product metrics in assessing developers' output: A comprehensive study

Wentao Chen, Huiqun Yu, Guisheng Fan, Zijie Huang, Yuguo Liang



PII: S0950-5849(25)00274-5
DOI: https://doi.org/10.1016/j.infsof.2025.107935
Reference: INFSOF 107935

To appear in: *Information and Software Technology*

Received date : 5 June 2025
Revised date : 23 September 2025
Accepted date : 15 October 2025

Highlights

- We investigate the impact of three LLMs on distorting the LOC metric in software development.
- We analyze the practicality of complex vs. simple software metrics across 9 repositories.
- We evaluate effectiveness and cost of SonarQube and PMD in detecting quality-changing commits.
- We conduct a case study in a dominant company in financial sector to validate our findings in industry, including a rapid review and a software metrics migration plan.

Usage Patterns of Software Product Metrics in Assessing Developers' Output: A Comprehensive Study

Wentao Chen[a], Huiqun Yu[a,*], Guisheng Fan[a,b,*], Zijie Huang[c], Yuguo Liang[a]

[a] *Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, 200237, China*
[b] *Shanghai Engineering Research Center of Smart Energy, Shanghai, 201103, China*
[c] *Shanghai Key Laboratory of Computer Software Testing and Evaluating, Shanghai Development Center of Computer Software Technology, Shanghai, 201112, China*

Abstract

Context: Accurate assessment of developers' output is crucial for both software engineering research and industrial practice. This assessment often relies on software product metrics such as lines of code (LOC) and quality indicators from static analysis tools. However, existing research lacks a comprehensive understanding of the usage patterns of product metrics, and a single metric is increasingly vulnerable to manipulation, particularly with the emergence of large language models (LLMs).
Objectives: This study aims to investigate (1) how developers can intentionally manipulate commonly used metrics like LOC by using LLMs, (2) whether complex efficiency metrics provide consistent advantages over simpler metrics, and (3) the reliability and cost-effectiveness of quality metrics derived from tools such as SonarQube.
Methods: We conduct empirical analyses involving three LLMs to achieve metric manipulation and evaluate product metric reliability across nine open-source projects. We further validate our findings through a collaboration with a large financial institution facing fairness concerns in developers' output due to inappropriate metric usage.

*Corresponding author
Email addresses: wentaochen@mail.ecust.edu.cn (Wentao Chen), yhq@ecust.edu.cn (Huiqun Yu), gsfan@ecust.edu.cn (Guisheng Fan), huangzj@sscenter.sh.cn (Zijie Huang), ygliang@mail.ecust.edu.cn (Yuguo Liang)

Results: We observe that developers can inflate LOC by an average of 60.86% using LLMs, leading to anomalous assessments. Complex efficiency metrics do not yield consistent performance improvements relative to their computational costs. Furthermore, quality metrics from SonarQube and PMD often fail to capture real quality changes and are expensive to compute. The software metric migration plan based on our findings effectively reduces evaluation anomalies in the industry and standardizes developers' commits, confirming our conclusions' practical validity.
Conclusion: Our findings highlight critical limitations in current metric practices and demonstrate how thoughtful usage patterns of product metrics can improve fairness in developer evaluation. This work bridges the gap between academic insights and industrial needs, offering practical guidance for more reliable developers' output assessment.

Keywords: developers' output assessment, software metrics manipulation, industrial practice, software efficiency metrics, software quality metrics

1. Introduction

Software is profoundly reshaping our world. As DevOps continues to gain prominence, its philosophical underpinnings [1, 2, 3] intensify the focus on software efficiency within the software engineering discourse. This leads to a rigorous examination of how developers' output is measured impartially. Developers' output [4, 5, 6] is defined as a composite score derived from various software metrics collected throughout the software development process, providing a comprehensive reflection of both efficiency and work quality. Accurately assessing developers' output is critical for software managers. In open source settings, they enable the identification of potential long-term maintainers, which is essential for the sustainable development of software projects [7]. In industrial settings, they supports fair and reasonable allocation of compensation and salaries, thereby improving overall team efficiency and productivity [8].

Managers' understanding of software metric usage patterns directly affects the precision of the comprehensive evaluation and the quantitative evaluation of the developers' output [9, 10, 11]. Software metrics are generally categorized into two types [12]: process metrics, which reflect aspects of the software development process, and product metrics, which capture characteristics of the software artifact. Product metrics can be further divided into efficiency metrics (e.g., lines of code (LOC), code complexity) and quality metrics (e.g., number of defect). Recent research [4, 5, 9] increasingly focuses on optimizing the usage patterns of software metrics, particularly product metrics. The usage patterns of a software metric in assessing developers' output hinges on the unique aspects it measures and its reliability and time costs compared with similar metrics. With the growing popularity of large language models (LLMs) assisted software development and the introduction of more complex developers' output assessment metrics and models, there remains a gap in our understanding of how LLMs influence output assessment and the usage patterns of software product metrics.

As LLMs become increasingly adopted in software development, developers are able to effectively leverage LLMs to complete assigned coding tasks, thereby enhancing their delivery efficiency [13]. Some studies [9, 12] point out that developers can manipulate automated measurement tools, causing software product metrics to misrepresent their actual contributions. However, there remains an unclear understanding of the extent and methods by which developers may manipulate software product metrics using LLMs, posing potential ethical risks. To assess the potential impact of LLMs on the usage patterns of software product metrics, we aim to address the following research question.

RQ1: Can developers intentionally increase the lines of code metric by utilizing large language models? We construct function code, with one version written by humans for Leetcode problems and the other rewritten by LLMs. We find that developers can intentionally increase the LOC metric using LLMs, with an average increase of 60.86%, which potentially leads to the LOC anomalies. Different LLMs exhibit divergent strategies to increase LOC, which fundamentally differ from the approaches used by traditional developers to manipulate these indicators.

The LOC metric is one of the most commonly used indicators for evaluating developers' output in terms of software efficiency. But LOC is often considered to reflect only a partial view of a developer's output. Some studies [5, 7] propose more complex software efficiency metrics by combining multiple simple efficiency metrics, aiming to better reflect developers' output. However, these complex metrics often incur higher costs and are rarely compared against simple efficiency metrics. To provide insights into the usage patterns of both complex and simple software efficiency metrics, we seek to answer the following research question.

RQ2: Are complex efficiency metrics completely superior to

simple ones? We compare complex and simple metrics across 9 open source repositories and conduct case studies to explain observed differences. We find that complex metrics do not consistently outperform simpler ones across all projects, and their costs are often much higher, indicating that their practicality is not as good as initially expected.

Software quality metrics obtained from static analysis tools (SATs) [14] are widely used in the evaluation of developers' output, while prior studies [15, 16, 17] show that these tools often produce false positives. Previous research often focuses on their capability to identify issues in bug-introducing commits. In the assessments of developers' output, however, quality metrics need not only to indicate the degradation of software quality but also to reflect improvements in software quality. It remains unclear whether changes in these quality metrics can accurately capture developers' contributions to software quality. To provide insights into the usage patterns of quality metrics derived from SATs, we address the following research question.

RQ3: Can quality metrics derived from static analysis tools reliably reflect developers' output? We monitor the changes of quality metrics during commit processes, focusing on instances of defects being introduced or fixed, which highlights key concerns for developers. We discover that the quality metrics obtained from SonarQube and PMD fail to accurately capture quality changes in developers' commits. On average, getting the software quality metrics takes SonarQube 86.82 seconds per commit and PMD 15.70 seconds per commit, showing a high time cost.

To validate the practical relevance of our findings, we collaborate with a large financial institution to address cases of inaccurate developers' output assessments caused by incorrect usage patterns of software product metrics. The company in our study employs over 8,000 professionals and uses the LOC metric as an objective reference for assessing developers' output. However, issues with the software product metrics, such as unusually high LOC changes caused by code cloning, lead to unfair assessments and compensation problems. The company in our study notices an increasing number of these issues and raise concerns about the practicality and reliability of the current metrics, highlighting the need for a more reliable and fair approach to assessing developers' output. Recognizing that relying solely on LOC is not fair, the company develops a plan to revise the software product metrics. This plan aims to introduce new metrics while addressing and correcting the problems with the current ones, and the new metrics will be able to effectively measure developers' output while offering a lower-cost method of evaluating all past commits in the software repositories. Additionally, the company finds that existing software development practices are severely disrupted by the improper use of current product metrics, and aims to complete this migration plan in a relatively short time.

To address the company's concerns, we conduct a rapid review of the use of software metrics in past developers' output assessment schemes in academic research to provide the company with a wider range of software metrics for consideration. We find that software product metrics are widely used in academic research for developers' output assessment, particularly LOC-based metrics and quality metrics from SATs. Next, we examine the practicality of the software metrics that the company plans to use. Based on our findings, we provide the company with four recommendations for the migration plan and gather feedback from three experienced developers' output assessment managers through a semi-structured interview of one hour. Finally, we continuously monitor the effectiveness of the final plan.

As a result, the company adopt the following four points as part of the software metrics migration plan: (1) Introducing a model for detecting LLM-rewritten code in the future, (2) Using SonarQube only to measure the quality of the latest software repository as a reference for developers' quality behavior, without including its quality metrics in the developers' output assessment, (3) Introducing the ELOC metric to measure different aspects of code efficiency while keeping the original LOC-based metrics, especially NALOC, as a supplementary reference, and (4) Using a combination of software metrics and process metrics, and apply mathematical models with process metrics to reduce anomalies in cases of software metric issues. After 5 months of continual observation, we find that the new developers' output assessment way greatly reduce the previous output score anomalies. This indicates that our research findings play a role in shaping the usage patterns of software metrics in developers' output assessment, bridging the gap between industrial practices and academic research.

This paper makes the following contributions:

- We explore how developers can intentionally manipulate the LOC metric using LLMs, leading to significant anomalies that affect the fairness and effectiveness of developers' output assessments.
- We provide a thorough evaluation of existing product metrics, with a particular focus on efficiency metrics and quality metrics from SATs, assessing their practicality and cost-effectiveness.

- We conduct a rapid review of quantitative metrics used in past developers' output research to help the company select relevant software metrics, providing guidance for future quantitative assessments of developers' output.
- We establish a connection between academic research on software product metrics and practical applications in the industry, demonstrating how academic insights can influence real-world developers' output assessment practices.

The rest of the paper is organized as follows: Section 2 provides an overview of our background and related work. Section 3 explains the research questions and the approaches used in our research. Section 4 presents the experiment and results, followed by answers to the research questions. Section 5 covers a practical study. Section 6 discusses implications and the threats to the validity of our study. Finally, Section 7 concludes the paper and suggests directions for future wor7k.

2. Background And Related Work

This section describes research related to developers' output assessment and the software efficiency and quality metrics.

2.1. Assessing Developers' Output

Developers' output [5, 6, 8] is defined as the contributions and efforts made by developers in delivering code during the software development process. Open-source communities, in their efforts to encourage developers to maintain open-source software, require an understanding of the ranking of these contributions [7]. Similarly, commercial companies, in order to fairly compensate developers, need to quantify individual outputs [8]. Toward these ends, the assessment of developers' output is often measured by quantifiable metrics, providing a specific score or rating.

The software development process provides a rich set of metrics for assessing developers' output. Depending on their source, these metrics can be categorized into process metrics and product metrics [12]. Process metrics refer to the measurable parameters generated by developers during the software development process, such as story points and commit frequency, which are often influenced significantly by subjective human factors. Product metrics, on the other hand, involve the measurement of the code submitted by developers, such as LOC and cyclomatic complexity, reflecting the state of the code itself more objectively.

Furthermore, software product metrics can be subdivided into software efficiency metrics and software quality metrics based on the aspects they measure. Software efficiency metrics indicate the quantity and complexity of the code evaluated from the developer, for example, the number of new lines of code changed and complexity. Software quality metrics reflect the quality of the code evaluated from the developer, such as the introduction of new bugs or code smells. The two types of metrics provide different kinds of value for assessing developers' output.

Existing research [18, 19, 20] quantitatively determines the final output score by integrating multiple software process and product metrics based on the dimension of developers' output reflected by software metrics. For instance, Lima et al. [21] evaluate contributions using repository mining metrics, incorporating not only LOC but also quality metrics related to bugs and fixes. Similarly, Diamantopoulos et al. [22] examine 19 repository metrics, proposing quantifiable measures of software development flow. Young et al. [23] introduce the 'All Contributors' model to recognize diverse contributions in open source projects, such as communication and community management. Furthermore, a prominent ICT company develop a human-centric quality assurance framework to quantitatively assess software engineers' output [6, 24]. This framework synthesizes various quantitative metrics and employs techniques like TOPSIS and linear programming for team evaluations, alongside iForest for individual assessments. Li et al. [25] design an explainable measure for quantifying the contributions of industry developers and collected software metrics as well as process metrics by interviewing 18 experienced software engineers. Collectively, these approaches emphasize the necessity of incorporating multiple dimensions of developer activity to achieve a holistic understanding of developer contributions.

2.2. Complex Software Efficiency Metrics

Metrics derived from source code, such as LOC, are proposed as indicators of developer contributions [26, 27]. However, these metrics are limited in that they do not capture the true value or quality of the output. Ren et al. [5] define the developmental value of code as a combination of structural value and non-structural value. They introduce the DevRank algorithm, which leverages call graphs to quantify developer contributions through a Learning-to-Rank (L2R) method. Nonetheless, they do not address how to determine weights for different modification types or whether L2R parameters are consistent across projects. In response to these limitations, Sun et al. [7] propose CVALUE, a method that measures developer contributions by integrating syntactic and semantic information across four dimensions: modification volume, understandability, inter-function impact, and intra-function impact.

2.3. Software Quality Metrics of Static Analysis Tools

SATs are essential for early identification and remediation of software vulnerabilities, as highlighted by Tahaei et al. [28]. However, concerns regarding the reliability of their outputs persist. Nachtigall et al. [29] conduct a systematic usability evaluation of 46 SATs, revealing significant usability issues: over half of the tools provide poor warning messages, and about 75% lack effective support for fixes. This raises questions about their practicality in real-world applications. Building on this, Lenarduzzi et al. [16] assess the agreement and precision of six widely used SATs for Java across 47 projects, finding minimal consensus and low precision in their recommendations. Trautsch et al. [14] further explore SATs' defect detection capabilities, indicating that while they could theoretically detect up to 76% of defects identified in code reviews, actual coverage varies, with style checkers and AST pattern checkers identifying only 25% of defects. Additionally, Mehrpour and LaToza [30] examine the influence of automated SATs on software quality, using PMD for Java. Their findings suggest that defect-prone files receive fewer warnings, correlating with a decline in warning density.

2.4. Motivation

Existing research on developer productivity frequently focuses on constructing improved metrics or models, lacking analysis of the impact of LLM-assisted development and the application patterns of specific metric types. Specifically, current studies exhibit gaps in the following three areas:

The potential implications of LLM-assisted development on metrics. Prior studies [9, 12] indicate that some software metrics can be fooled by developers due to improper usage. Understanding the advantages and usage of specific metrics for assessing developers' output over similar ones can reduce output deviations. Our research bridges the gap in the usage patterns of specific software product metrics for developers' output assessment. Moreover, existing research also neglects the impact of LLMs on software metrics in developers' output, despite their growing use in software development.

The selection and application of software efficiency metrics. While these advanced software efficiency metrics mentioned in Section 2.2 offer a more nuanced understanding of contributions, they often involve significant time costs [7], and prior studies do not sufficiently explore the utility of simpler software efficiency metrics. Besides, these studies claim that complex metrics can completely replace simple ones. This paper investigates the practicality of these simpler metrics in comparison to more complex alternatives.

The selection and application of software quality metrics obtained via SATs. Notably, while these studies mentioned in Section 2.3 often focus on quality deterioration, they typically do not verify the reliability of quality improvements. In developers' output assessments, enhancements in quality are rewarded, while declines lead to penalties. This study aims to address this gap by examining the verification of quality improvements.

3. Research Method

This section introduces our research framework, the motivations and context behind the research questions, as well as the method.

3.1. Overall Framework

Figure 1 outlines a structured approach to exploring the usage patterns of product metrics to assess developers' output in both industrial and academic contexts. In the academic setting, we explore the specific usage patterns of software product metrics, focusing on LLMs' impact on these metrics and the reliability of specific types. We then apply the identified reliability in industry where software metrics are misused, as detailed in Section 5.

3.2. Research Questions

To explore the usage patterns of software product metrics in assessing developers' output, we design three research questions. First, we investigate whether LLMs can further fool existing software product metrics. Then, we compare the reliability and time costs of current software efficiency and quality metrics. The research questions are described below.

RQ1: Can developers intentionally increase the lines of code metric by utilizing large language models? With the growing popularity of LLMs in code generation, existing research ignores the impact of LLMs on software product metrics in developers' output. In this context, we aim to investigate whether developers can leverage LLMs at minimal cost

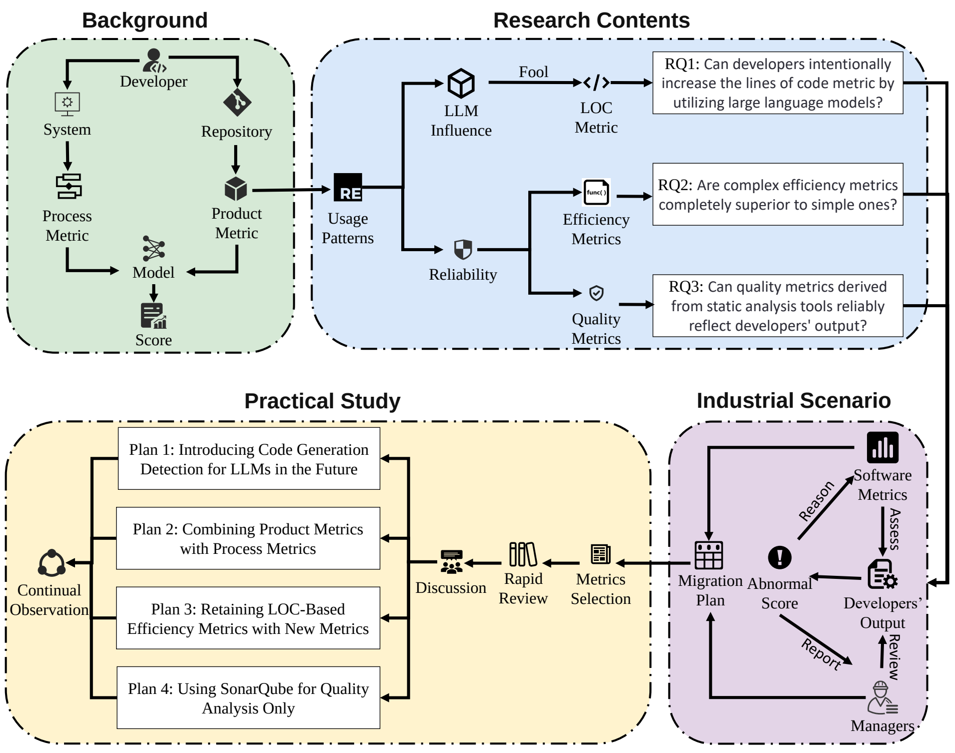

Background
Developer
System
Repository
Process Metric
Product Metric
Model
Score
Research Contents
LLM Influence
Fool
LOC Metric
RQ1: Can developers intentionally increase the lines of code metric by utilizing large language models?
Usage Patterns
Reliability
Efficiency Metrics
RQ2: Are complex efficiency metrics completely superior to simple ones?
Quality Metrics
RQ3: Can quality metrics derived from static analysis tools reliably reflect developers' output?
Practical Study
Plan 1: Introducing Code Generation Detection for LLMs in the Future
Plan 2: Combining Product Metrics with Process Metrics
Plan 3: Retaining LOC-Based Efficiency Metrics with New Metrics
Plan 4: Using SonarQube for Quality Analysis Only
Continual Observation
Discussion
Rapid Review
Metrics Selection
Industrial Scenario
Software Metrics
Reason
Assess
Migration Plan
Abnormal Score
Developers' Output
Report
Review
Managers

Figure 1: Overall Framework of Our Research

to artificially increase the LOC score, thus providing a potential explanation for anomalies in software metrics.

RQ2: Are complex efficiency metrics completely superior to simple ones? An increasing number of complex software efficiency metrics are being developed by integrating multiple simple efficiency metrics and specialized algorithms to avoid the negative impacts that single simple efficiency metrics may have on the assessment of developers' output. However, prior research does not provide evidence that these complex efficiency metrics are more practical than most simple metrics. To investigate whether these complex efficiency metrics yield more practical outcomes across most projects compared to simple metrics, we introduce several widely applied simple metrics from the field of software engineering.

RQ3: Can quality metrics derived from static analysis tools reliably reflect developers' output? Quality metrics obtained from SATs are widely used to represent changes in software quality. Previous research focuses on their capability to identify issues in bug-introducing commits. In the assessments of developers' output, however, quality metrics need not only to indicate the degradation of software quality but also to reflect improvements in software quality. Therefore, we explore whether commonly used SATs can accurately reflect the impact developers have on code quality.

For each research question, we propose the following hypotheses to better articulate our position.

Hypothesis 1: With the assistance of large language models, developers can more easily manipulate the LOC metric, and such manipulations are difficult to detect.

Hypothesis 2: When using a single software efficiency metric to evaluate output, complex metrics do not provide a more accurate reflection of developers' output than simple metrics.

Hypothesis 3: Quality metrics obtained from static analysis tools do not reliably reflect the actual contributions developers make to software quality.

3.3. Evaluation Method for LLM Influence

To investigate the remarkable capabilities of LLMs in code generation, we conduct a study examining the effects of LLMs on product metrics at different levels of code structure. We follow [31] to use LeetCode problems, and select their solutions as original code functions. LeetCode is a platform that offers a collection of programming problems.

Then, we use GPT-4o-mini, Copilot, and Gemini-2.5-flash to generate rewritten versions of these function codes. Our selection of these three models is based on three principles tailored to the specific needs of developer-driven metric manipulation.

First, we address the developers' usage cost. Since developers manipulate LOC primarily to achieve higher ratings or corresponding rewards, the use of paid models (e.g., Claude-Pro) or models requiring self-deployment (e.g., CodeLlama) may introduce significant economic costs or technical barriers. This does not align with their practical motivations. Therefore, we prioritize models that are publicly available or do not incur additional costs.

Second, we consider both the performance and timeliness of the models. We favor models that not only demonstrate superior performance but are also relatively new, as these models provide better generation results and better reflect the current technical environment developers are likely to adopt. Based on this principle, we ultimately select GPT-4o-mini and Gemini-2.5-flash as representative models.

Finally, we integrate the developers' actual application contexts. We assume that developers may either use readily accessible online models or integrate models into commonly used development tools. Given that Copilot is a widely studied and commonly used development tool in the software engineering field, we introduce Copilot as a typical representative for such scenarios.

All models are accessed via their respective official chat interfaces, maintaining the default parameter configurations as set on their official websites. Specifically, for Copilot, we integrate the GPT-4.1 model. These generated codes are submitted to LeetCode to verify their accuracy. If the generated code fails the tests, we regenerate new code until it passes.

We choose the LOC metric as our manipulation target for two main reasons. First, LOC is the most widely used product metric to assess development output [32]. Second, previous studies [9, 12] explicitly highlight the methods of manipulating LOC, providing a useful comparison to LLM-based manipulations.

Figure 2 illustrates the specific process and strategies we use to construct the prompt. Our prompt design strictly follows and operationalizes the framework proposed in [33], systematically applying three strategies—Persona, Context Conveyor, and Meta Language Creation—around the core goal of "LOC Manipulation".

First, we apply the Persona strategy to adjust the model's output perspec-

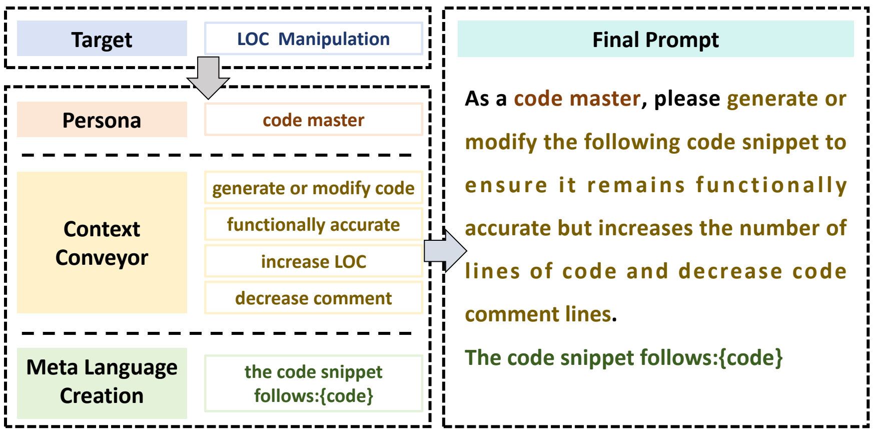

Target
LOC Manipulation
Persona
code master
Context Conveyor
generate or modify code
functionally accurate
increase LOC
decrease comment
Meta Language Creation
the code snippet follows:{code}
Final Prompt
As a code master, please generate or modify the following code snippet to ensure it remains functionally accurate but increases the number of lines of code and decrease code comment lines.
The code snippet follows:{code}

Figure 2: Overall Framework of Our Prompt Design

tive and style. We instruct the model to assume the role of a "code master", aiming to guide it to generate more professional code from the perspective of a code expert, thereby aligning with software engineering practices.

Next, we use the Context Conveyor strategy to precisely control the LLM's contextual environment, strictly limiting its output to a specific task scope. We achieve this by clearly defining the following four constraints:

Task Definition: The model is instructed to "generate or modify" code snippets.

Functional Constraint: The model must "ensure the correctness of the code functionality", which is the most crucial prerequisite.

Goal Instruction: We explicitly ask the model to "increase LOC".

Exclusion Instruction: The model is instructed to "reduce the number of comment lines" to prevent it from achieving the goal by simply adding comments, which would be ineffective.

Finally, we adopt the Meta Language Creation strategy to establish a formal structure for the data format the model processes. We formalize the input using a fixed template: "The code snippet follows: code", ensuring that the model accurately identifies the code content to be processed and distinguishes it from the instruction part.

We comply a dataset comprising 50 instances where LLMs have successfully rewritten code to pass tests on LeetCode. These cases can be accessed at `https://zenodo.org/records/14532880`. We use CLOC [34] to measure the LOC metrics between the original and rewritten versions of the code.

3.4. Evaluation Methods for Efficiency Metrics

3.4.1. Software Efficiency Metrics Choice

We ultimately select five efficiency metrics derived from prior related research in assessing developers' output. Specifically, we select three simple efficiency metrics (which reflect only one aspect of code information): New Added Lines of Code, Unit_Complexity, and Lines of Code, alongside two more complex ones (which reflect multiple facets of code and are often synthesized from a combination of several simple metrics): ELOC and CVALUE. Detailed introductions to each follow.

Lines of Code (LOC): This is a rudimentary approach that gauges developers' output by tallying the number of lines changed in code revisions tracked by version control systems. It counts both added and deleted lines to assess the effort per commit. Though straightforward, it might not accurately reflect the intricacy of code changes or the true contribution of the developer, as it overlooks semantic context.

New Added Lines of Code (NALOC): This method quantifies a developers' output by counting the number of new lines added in code modifications submitted through version control systems. It posits that the addition of new lines of code primarily represents the bulk of a developer's work, while deleted lines make only a minor contribution.

Unit_Complexity: This metric measures the complexity of unit source code using McCabe's cyclomatic complexity [35]. A 'unit' refers to the smallest executable part of the source code, such as a method or function. Originally applied in software maintainability engineering models [36], higher complexity in unit source code in a single commit suggests greater developers' output by the developer. Because developers require more time to complete more complex code [37, 38].

ELOC: ELOC [5] is a more sophisticated technique that considers not only the quantity but also the type and complexity of code changes. By analyzing changes in the Abstract Syntax Tree (AST), including various types of node modifications (additions, deletions, or alterations) and edit operations (such as moves or renames), ELOC aims to provide a more nuanced assessment of developers' output.

CValue: CValue [7] is a multi-dimensional method for assessing developers' output, incorporating information on the volume of code changes, code comprehensibility, inter-function and intra-function impacts across four dimensions. Through syntactic and semantic analysis of source code modifications, CValue quantifies code alterations using AST differences, measures code readability with complexity metrics, analyzes inter-function interactions via call graphs, and evaluates intra-function influence through program dependency graphs. Its objective is to offer a comprehensive and precise scoring system for developers' output.

3.4.2. Method

We assess the relationship between the rank order of manually assigned labels and the rank order of efficiency labels by calculating the Spearman's rank correlation coefficient to determine whether the efficiency labels reflect the actual output. The following is a detailed explanation.

Human Labels: We adopt the human-labeled scores from the dataset introduced in [7], which serve as indicators of the contribution level of each code commit to the respective project. These labels are meticulously assigned by at least two experienced programmers, who evaluate the significance of the changes made in each commit. Contributions that do not affect the code's semantics, such as code formatting adjustments, are given lower scores, reflecting minimal impact on the project.

Evaluation Metrics: The comparative method follows the approach outlined in [7], utilizing Spearman's rank correlation coefficient as a statistical measure to evaluate the effectiveness of methods assessing developers' output. Spearman's coefficient quantifies the strength of association between two datasets, specifically the consistency of rankings. When evaluating methods for assessing developers' output, Spearman's coefficient can compare the correlation between ranks derived from manual labeling and those obtained from automated assessment methods. Ranging from -1 to 1, values closer to 1 or -1 indicate greater agreement between the assessment method's outcomes and manually labeled results. The higher this metric, the more it suggests that the evaluative efficiency metrics are in line with the standards for assessing developers' output. By calculating Spearman's coefficient, the accuracy and reliability of different developers' output assessment methods in software efficiency aspect can be assessed.

3.5. Evaluation Methods for Quality Metrics

3.5.1. Software Quality Metrics Choice

We choose the SonarQube Community 9.9 LTA and PMD-7.15.0 to obtain the corresponding quality metrics. This is because these tools are widely used in the software development processes of related enterprises [39, 40]. We use the official default configurations of SonarQube and PMD to obtain the software quality metrics after each developer's commit.

For **SonarQube**, we select the following three quality metrics.

new_bugs (Bug): This metric indicates the total number of reliability issues first identified in new or recently changed code. Reliability measures the software's ability to perform its intended functions consistently over time and under specified conditions. Tracking new_bugs helps ensure that the software maintains high performance and stability.

new_code_smells (Smell): This metric represents the total number of maintainability issues identified for the first time in new or recently modified code. Maintainability is the ease with which software can be repaired, improved, and understood. High levels of new_code_smells can indicate potential problems that may increase the cost and complexity of future maintenance and development efforts.

new_violations (Violation): This metric indicates the total number of issues first identified in new or recently modified code. An issue is a problem that prevents the code from adhering to Clean Code standards [41], which are defined by a set of rules specific to the programming language used. When a coding rule is violated, an issue is raised, affecting one or more aspects of software quality.

For **PMD**, we select the following three metrics based on the guidelines from its official website [1].

Code Style: The number of violations of rules that enforce a specific coding style before and after each commit. This metric reflects changes in the consistency of the code's format.

Best Practice: The number of violations of rules that enforce generally accepted best practices before and after each commit. This metric reflects potential improvements or regressions in the code's robustness, maintainability, and security.

ALL: The number of violations of relevant Java rules before and after

[1]`https://pmd.github.io/pmd/pmd_rules_java.html`

each commit. This metric provides a global view of the code quality across all detectable dimensions in PMD.

3.5.2. Method

We observe the dynamic changes in the indicators provided by SonarQube when examining bug-fixing commits and bug-introducing commits, to evaluate the effectiveness of SonarQube. The specific details are as follows.

Quality-changing commits: Our investigation focuses on bug-fixing and bug-introducing commits, which reflect genuine code quality concerns encountered by developers during the software development process. We consider both bug-fixing and bug-introducing commits as quality-changing commits. The SZZ algorithm [42, 43, 44] is commonly used to identify bug-introducing commits based on bug-fixing commits. In our approach, we first use keyword matching on commit messages to identify bug-fixing commits. Subsequently, we apply the SZZ algorithm to detect bug-introducing commits that have captured developers' attention. The goal is to assess whether static quality analysis tools accurately report quality changes when developers make these commits.

Evaluation Metrics: Recall and False Positive Rate (FPR) are critical metrics used to assess the reliability of SonarQube in identifying code issues. True Positive (TP) refers to instances where SonarQube correctly identifies changes in quality metrics that correspond to either bug-fixing or bug-introducing commits. Conversely, False Positive (FP) occurs when SonarQube flags changes in quality metrics as problematic, even though the commit does not actually involve fix bugs or introduce new bugs. True Negative (TN) indicates situations where SonarQube accurately identifies that no significant changes in quality metrics have occurred, meaning the commit neither fixes nor introduces bugs. False Negative (FN) refers to cases where SonarQube fails to identify changes in quality metrics that should have been flagged as either bug-fixing or bug-introducing commits. The specific calculation formulas for Recall and FPR are as follows:

$$Recall = \frac{TP}{TP + FN} \tag{1}$$

$$FPR = \frac{FP}{FP + TN} \tag{2}$$

The evaluation of these metrics provides insights into the precision and recall of SonarQube's issue detection capabilities, which are essential for maintaining software quality and efficiency in the development process. High recall ensures that most actual bugs are identified, while a low FPR minimizes unnecessary alerts, thus improving developer trust in the tool.

4. Experiment and Result

4.1. Experiment Projects Selection

We select projects based on the following principles: 1. Only single-language Java projects are chosen; 2. The software metrics we compare are obtainable from the projects.

The decision to focus on single-language Java projects is based on several considerations. Firstly, the assessments of developers' output necessitates a differentiated approach when dealing with diverse programming languages. Secondly, Java projects are frequently incorporated within established evaluation frameworks. It is noteworthy that the state-of-the-art model, CValue, provides its findings exclusively on Java-centric projects. Consequently, these factors have led to the selection of Java projects.

Due to the lack of detailed implementation tools and critical information such as Abstract Syntax Tree node weights provided by CValue, we follow the ten datasets previously evaluated by CValue. After manually verifying the accuracy of these datasets, nine of them are selected for our empirical research (the alibaba/fastjson dataset is excluded due to missing and inaccurate data).

Table 1 presents an overview of the nine chosen projects. The human labels depict the developers' output ratings manually assigned by Sun et al. [7], which focus solely on the software efficiency aspect. The first five projects are sourced from the OSS-Fuzz list of ongoing fuzz testing initiatives. These projects have histories that include both feature additions and maintenance activities on existing code. The remaining four projects are smaller in scale, with simpler functionalities, and their modification histories largely consist of maintenance rather than new function introductions.

4.2. RQ1: Can developers intentionally increase the lines of code metric by utilizing large language models?

4.2.1. Result Analysis

Figure 3 presents a box plot of the LOC growth rate after the functions were rewritten by LLMs. We observe that all the replicated functions show an upward trend in LOC, with an average increase of 60.86%. Among these rewritten functions, the LOC growth varies depending on the LLM used. The

Table 1: Project Information

Project	Developers	LOC	Files	Commits	Human Lables
apache/commons-release-plugin	17	3,639	43	895	148
apache/commons-exec	31	4,684	72	1,170	150
apache/commons-cli	69	10,372	86	1,490	150
apache/commons-ognl	19	22,418	316	991	150
apache/rocketmq	653	232,936	2,030	8,419	135
apache/httpcomponents-core	97	84,428	959	3,827	145
apache/httpcomponents-client	127	76,198	762	3,641	158
google/guice	119	74,696	647	2,110	124
google/gson	164	37,558	261	1,993	84

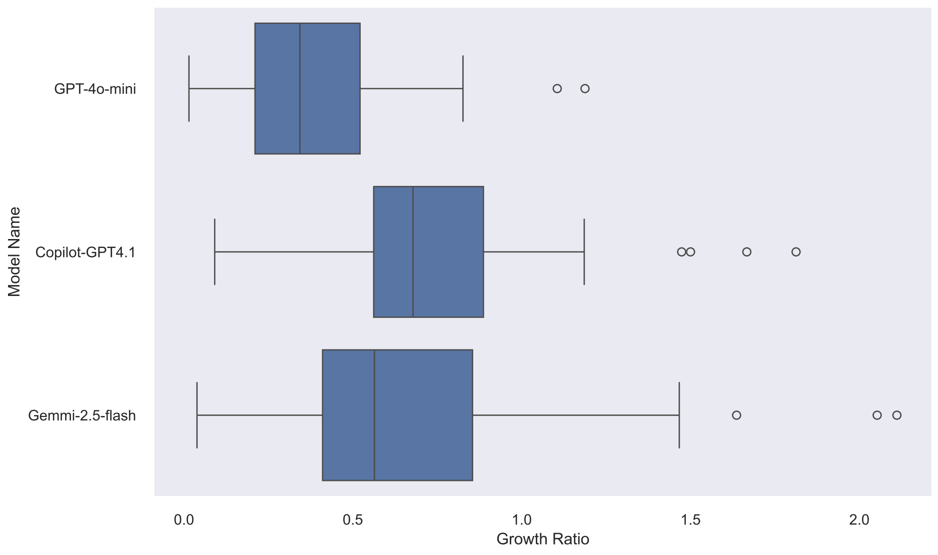


Figure 3: Boxplot of Growth Ratio between Rewritten Code and Original Code

Copilot-GPT4.1 model resulted in the highest average growth rate of 73.86%, while the GPT-4o-mini model exhibited the lowest average growth rate at 39.56%. The GPT-4o-mini model showed the weakest ability to increase LOC among the three models, with the smallest function increasing by only 1.47%. Although the Gemmi-2.5-flash model demonstrated an average growth rate of 69.18%, similar to Copilot-GPT4.1, its high standard deviation of 0.4553 and maximum value of 211.11% revealed the instability of the Gemmi-2.5-flash model in metric manipulation. This empirical result warns that large language models may be used to systematically interfere with developers' output assessments based on lines of code, and developers can achieve more than a 50% increase in LOC ratings through LLMs.

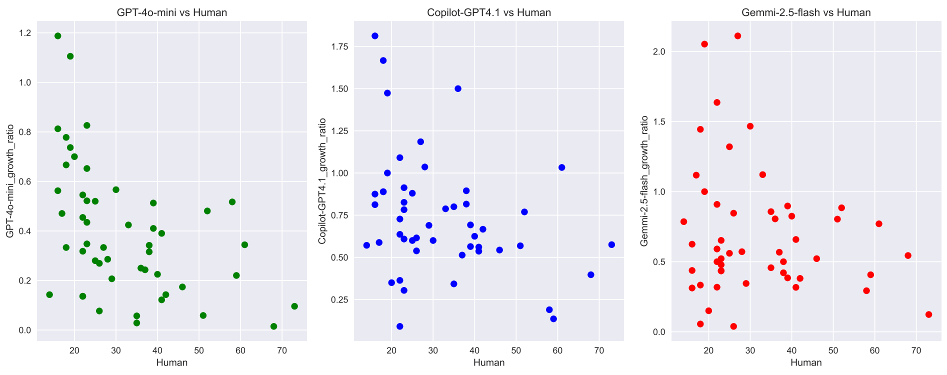
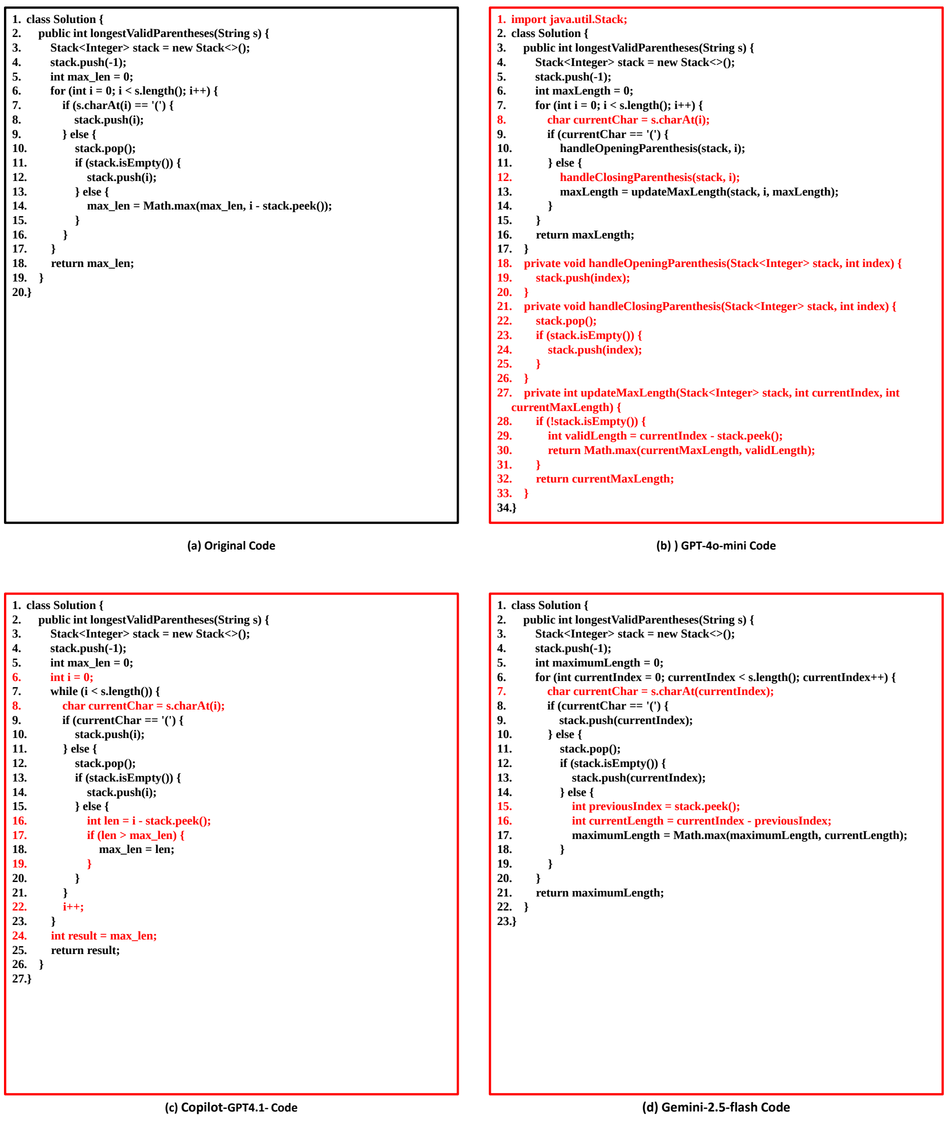


Figure 4: Scatter Plot of Original Code vs. Growth Ratio

Figure 4 also presents three scatter plots comparing the growth rate of code rewritten by different LLMs with the original LOC. Although the growth rate varies across different LLMs for the same code, it is generally observed that functions with smaller original LOC tend to exhibit higher growth rates. As the original LOC increases, the growth efficiency produced by all LLMs gradually decreases. As observed during the dataset construction, functions with larger LOC are more challenging for LLMs to correctly refactor and often fail to pass Leetcode tests, thus compromising the accuracy of the function.

4.2.2. Case Study

To further analyze the approaches different LLMs take when increasing the LOC metric, we manually analyze all samples to summarize their pat-



```
class Solution {
    public int longestValidParentheses(String s) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
        int max_len = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                stack.push(i);
            } else {
                stack.pop();
                if (stack.isEmpty()) {
                    stack.push(i);
                } else {
                    max_len = Math.max(max_len, i - stack.peek());
                }
            }
        }
        return max_len;
    }
}
```

(a) Original Code

```
import java.util.Stack;
class Solution {
    public int longestValidParentheses(String s) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
        int maxLength = 0;
        for (int i = 0; i < s.length(); i++) {
            char currentChar = s.charAt(i);
            if (currentChar == '(') {
                handleOpeningParenthesis(stack, i);
            } else {
                handleClosingParenthesis(stack, i);
                maxLength = updateMaxLength(stack, i, maxLength);
            }
        }
        return maxLength;
    }
    private void handleOpeningParenthesis(Stack<Integer> stack, int index) {
        stack.push(index);
    }
    private void handleClosingParenthesis(Stack<Integer> stack, int index) {
        stack.pop();
        if (stack.isEmpty()) {
            stack.push(index);
        }
    }
    private int updateMaxLength(Stack<Integer> stack, int currentIndex, int
  currentMaxLength) {
        if (!stack.isEmpty()) {
            int validLength = currentIndex - stack.peek();
            return Math.max(currentMaxLength, validLength);
        }
        return currentMaxLength;
    }
}
```

(b)) GPT-4o-mini Code

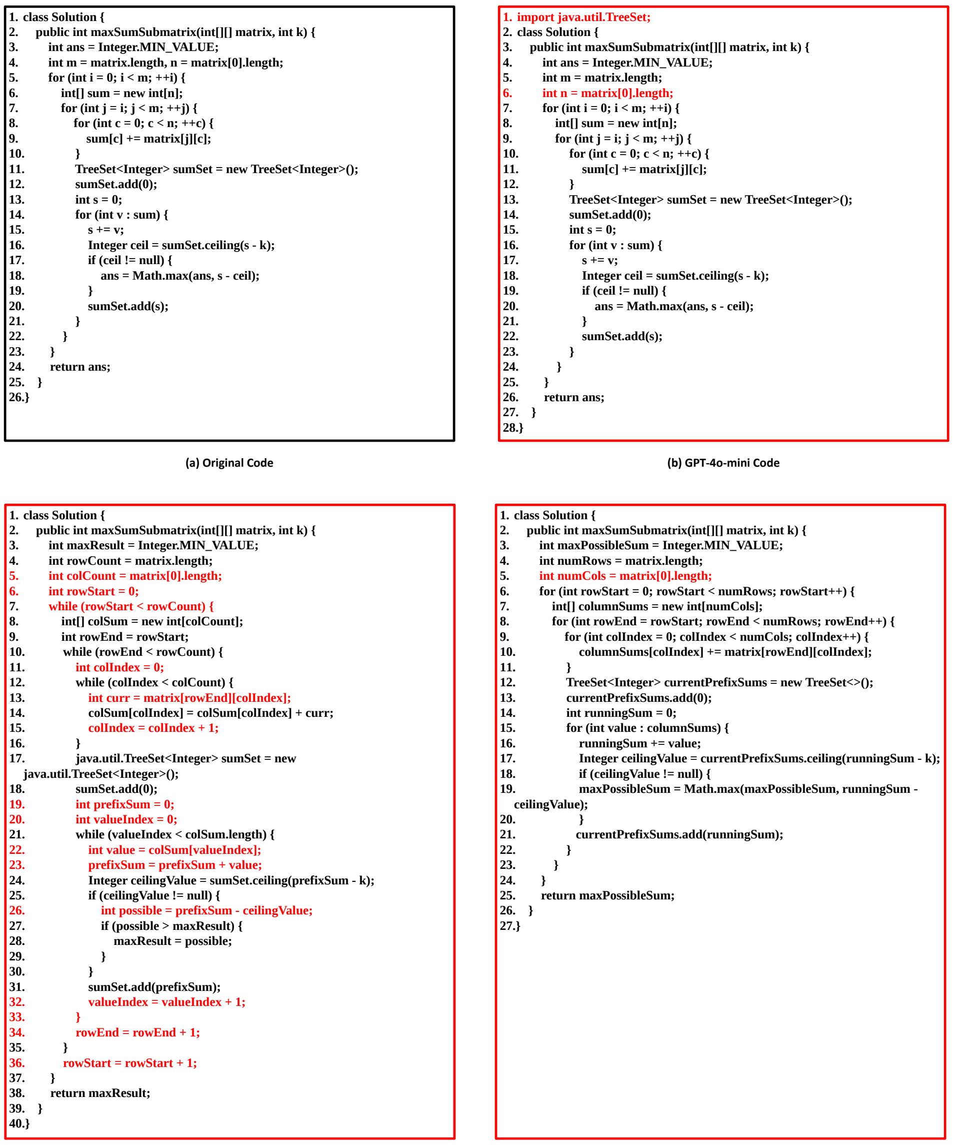
```
class Solution {
    public int longestValidParentheses(String s) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
        int max_len = 0;
        int i = 0;
        while (i < s.length()) {
            char currentChar = s.charAt(i);
            if (currentChar == '(') {
                stack.push(i);
            } else {
                stack.pop();
                if (stack.isEmpty()) {
                    stack.push(i);
                } else {
                    int len = i - stack.peek();
                    if (len > max_len) {
                        max_len = len;
                    }
                }
            }
            i++;
        }
        int result = max_len;
        return result;
    }
}
```

(c) Copilot-GPT4.1- Code

```
class Solution {
    public int longestValidParentheses(String s) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
        int maximumLength = 0;
        for (int currentIndex = 0; currentIndex < s.length(); currentIndex++) {
            char currentChar = s.charAt(currentIndex);
            if (currentChar == '(') {
                stack.push(currentIndex);
            } else {
                stack.pop();
                if (stack.isEmpty()) {
                    stack.push(currentIndex);
                } else {
                    int previousIndex = stack.peek();
                    int currentLength = currentIndex - previousIndex;
                    maximumLength = Math.max(maximumLength, currentLength);
                }
            }
        }
        return maximumLength;
    }
}
```

(d) Gemini-2.5-flash Code

Figure 5: A Case of Original and Rewritten Code Snippets



```
class Solution {
    public int maxSumSubmatrix(int[][] matrix, int k) {
        int ans = Integer.MIN_VALUE;
        int m = matrix.length, n = matrix[0].length;
        for (int i = 0; i < m; ++i) {
            int[] sum = new int[n];
            for (int j = i; j < m; ++j) {
                for (int c = 0; c < n; ++c) {
                    sum[c] += matrix[j][c];
                }
                TreeSet<Integer> sumSet = new TreeSet<Integer>();
                sumSet.add(0);
                int s = 0;
                for (int v : sum) {
                    s += v;
                    Integer ceil = sumSet.ceiling(s - k);
                    if (ceil != null) {
                        ans = Math.max(ans, s - ceil);
                    }
                    sumSet.add(s);
                }
            }
        }
        return ans;
    }
}
```

(a) Original Code

```
import java.util.TreeSet;
class Solution {
    public int maxSumSubmatrix(int[][] matrix, int k) {
        int ans = Integer.MIN_VALUE;
        int m = matrix.length;
        int n = matrix[0].length;
        for (int i = 0; i < m; ++i) {
            int[] sum = new int[n];
            for (int j = i; j < m; ++j) {
                for (int c = 0; c < n; ++c) {
                    sum[c] += matrix[j][c];
                }
                TreeSet<Integer> sumSet = new TreeSet<Integer>();
                sumSet.add(0);
                int s = 0;
                for (int v : sum) {
                    s += v;
                    Integer ceil = sumSet.ceiling(s - k);
                    if (ceil != null) {
                        ans = Math.max(ans, s - ceil);
                    }
                    sumSet.add(s);
                }
            }
        }
        return ans;
    }
}
```

(b) GPT-4o-mini Code

```
class Solution {
    public int maxSumSubmatrix(int[][] matrix, int k) {
        int maxResult = Integer.MIN_VALUE;
        int rowCount = matrix.length;
        int colCount = matrix[0].length;
        int rowStart = 0;
        while (rowStart < rowCount) {
            int[] colSum = new int[colCount];
            int rowEnd = rowStart;
            while (rowEnd < rowCount) {
                int colIndex = 0;
                while (colIndex < colCount) {
                    int curr = matrix[rowEnd][colIndex];
                    colSum[colIndex] = colSum[colIndex] + curr;
                    colIndex = colIndex + 1;
                }
                java.util.TreeSet<Integer> sumSet = new
    java.util.TreeSet<Integer>();
                sumSet.add(0);
                int prefixSum = 0;
                int valueIndex = 0;
                while (valueIndex < colSum.length) {
                    int value = colSum[valueIndex];
                    prefixSum = prefixSum + value;
                    Integer ceilingValue = sumSet.ceiling(prefixSum - k);
                    if (ceilingValue != null) {
                        int possible = prefixSum - ceilingValue;
                        if (possible > maxResult) {
                            maxResult = possible;
                        }
                    }
                    sumSet.add(prefixSum);
                    valueIndex = valueIndex + 1;
                }
                rowEnd = rowEnd + 1;
            }
            rowStart = rowStart + 1;
        }
        return maxResult;
    }
}
```

(c) Copilot-GPT4.1- Code

```
class Solution {
    public int maxSumSubmatrix(int[][] matrix, int k) {
        int maxPossibleSum = Integer.MIN_VALUE;
        int numRows = matrix.length;
        int numCols = matrix[0].length;
        for (int rowStart = 0; rowStart < numRows; rowStart++) {
            int[] columnSums = new int[numCols];
            for (int rowEnd = rowStart; rowEnd < numRows; rowEnd++) {
                for (int colIndex = 0; colIndex < numCols; colIndex++) {
                    columnSums[colIndex] += matrix[rowEnd][colIndex];
                }
                TreeSet<Integer> currentPrefixSums = new TreeSet<>();
                currentPrefixSums.add(0);
                int runningSum = 0;
                for (int value : columnSums) {
                    runningSum += value;
                    Integer ceilingValue = currentPrefixSums.ceiling(runningSum - k);
                    if (ceilingValue != null) {
                        maxPossibleSum = Math.max(maxPossibleSum, runningSum -
    ceilingValue);
                    }
                    currentPrefixSums.add(runningSum);
                }
            }
        }
        return maxPossibleSum;
    }
}
```

(d) Gemini-2.5-flash Code

Figure 6: A Low LOC Growth Rate Case of Original and Rewritten Code Snippets

terns. Figures 5 and 6 showcase two examples we selected to better illustrate the conclusions of our summarized patterns. The problem of Figure 5 requires the user to solve the task of finding the length of the longest valid (well-formed and contiguous) parentheses sub-string in a given string that contains only the characters '(' and ')'. The problem of Figure 6 asks to find and return the maximum sum of submatrix elements that does not exceed a given integer k, for a matrix of size $m \times n$.

Different LLMs exhibit varying preferences in how they increase the LOC metric. Compared to traditional developers who add dead code to manipulate metrics, LLMs often change the code itself. These changes are hard for managers to spot. For all LLMs, they introduce new intermediate variables to store intermediate values. In the original code (a), *s.charAt(i)* is directly used for comparison with the corresponding character. In the rewritten code (b), this value is stored in a newly declared variable *current-Char*.

For the Gemmi-2.5-flash model, its preferred method is to introduce a large number of new intermediate variables to store intermediate values. In Figure 5, it defines three additional intermediate variables `currentChar`, `previousIndex`, and `currentLength` to expand the function. This preference leads to difficulty in expanding human-written code when there are fewer intermediate variables that can be extended. In Figure 6, we observe that due to the limited availability of expandable intermediate variables in human-written code, it is unable to effectively expand the LOC.

For GPT-4o-mini, additional LOC growth is achieved by introducing extra helper functions and external libraries. For introducing extra helper functions,the original code (a) concentrates all logic within the main loop, whereas the rewritten code (b) decomposes distinct responsibilities into separate functions, such as *handleOpeningParenthesis()* and *handleClosingParenthesis()*. For introducing external libraries, the rewritten code (b) adds the *import java.until.Stack* statement to declare the data structure used, specifically the stack. We attribute the relatively low growth rate in Figure 6 to the simplicity of the code's logical structure, as well as the use of existing Java library functions for encapsulation.

For Copilot-GPT-4.1, it further increases LOC by refactoring the code logic. In Figure 5, it rewrites the `for (int i = 0; i < s.length(); i++)` loop as `while (i < s.length())`, resulting in additional LOC. Therefore, even in the sample in Figure 6, where Gemmi-2.5-flash and GPT-4o-mini struggle to achieve LOC expansion, Copilot-GPT-4.1 successfully manipulates LOC.

Finding 1. LLMs can manipulate the LOC metric through the preferred methods they employ, significantly increasing the LOC score via rewriting. However, due to the limitations of the LLMs' code expansion preferences, this results in their inability to significantly expand the LOC score when dealing with certain types of human-written code.

4.3. RQ2: Are complex efficiency metrics completely superior to simple ones?

4.3.1. Result Analysis

Table 2 displays the Spearman correlation coefficients between three simple metrics (NALOC, Unit_Complexity and LOC) and two complex metrics (ELOC and CValue) compared against manually assigned labels, with the better-performing metrics highlighted in bold.

Table 2: Spearman Correlation of Metrics vs. Human Labels

Project	**Simple Metric**			**Complex Metric**	
	NALOC	**Unit_Complexity**	**LOC**	**ELOC**	**Cvalue**
apache/commons-release-plugin	**0.7233**	0.5099	0.7062	0.5337	0.7084
apache/commons-exec	0.5907	**0.6887**	0.4947	0.6018	0.6458
apache/commons-cli	0.4598	**0.6614**	0.3685	0.5718	0.5737
apache/commons-ognl	**0.6214**	0.5703	0.5319	0.5491	0.6210
apache/rocketmq	0.8978	0.4885	0.8650	0.7550	**0.9010**
apache/httpcomponents-core	**0.8230**	0.5000	0.7526	0.7158	0.7935
apache/httpcomponents-client	**0.5623**	0.3885	0.4741	0.5161	0.5351
google/guice	**0.8430**	0.5578	0.5842	0.5481	0.6935
google/gson	0.7472	0.4276	0.5778	**0.8224**	0.7783

Performance of Simple Metrics vs. Complex Metrics: In five out of the analyzed projects, NALOC proves to be a more effective metric for assessing developers' output than either ELOC or CValue. On average, NALOC surpasses the best-performing complexity metric by 1.80%. Notably, in the `google/guice` project, NALOC outperforms CValue, the top complexity metric, by approximately 14.95%. Similarly, in the `apache/commons-exec` and `apache/commons-cli` projects, the Unit_Complexity measure is more effective than CValue, exceeding it by 4.29% and 8.77%, respectively.

Conversely, in two projects, `apache/rocketmq` and `google/gson`, the complex metrics demonstrate a slight advantage over the simple metrics. In the `apache/rocketmq` project, CValue marginally outperforms NALOC by about 0.32%, indicating nearly equivalent effectiveness. In the `google/gson` project, ELOC and CValue show more significant improvements over the simple metrics, with ELOC surpassing the best simple metric by 7.52% and

CValue by 3.11%. However, due to inaccuracies in a portion of the ELOC data, only 22 out of 84 commits in the `google/gson` project have valid ELOC results, which do not yield statistically significant findings. The lack of specific parameter values for ELOC and CValue's implementation logic prevents a deeper investigation into the discrepancies observed.

Comparison of Simple Metrics: Among the simple metrics, NALOC consistently surpasses LOC in measuring code changes, as adding code generally requires more effort than removing them. However, Unit_Complexity exhibits significant variability, influenced by its calculation per commit, focusing on changes involving constructs such as if statements and for loops, which are not uniformly distributed.

Time Expenditure for Metric Acquisition: Table 3 details the time required to acquire both simple metrics and the CValue metric across 9 projects. The study reveals that, in most cases, complex metrics like CValue and ELOC offer little to no advantage over simple metrics. In fact, they often perform worse. The cost discrepancy between adopting CValue, considered the most advanced evaluation method, and using simple metrics is also examined.

Table 3: Time Costs of Acquiring Simple & CValue Metrics

Project	**Simple Metric**		**CValue Metric**	
	Detected	**Time Cost(s)**	**Detected**	**Time Cost(s)**
apache/commons-release-plugin	891	0.73	653	17.91
apache/commons-exec	1,170	0.63	755	38.21
apache/commons-cli	1,490	0.54	1,176	91.23
apache/commons-ognl	991	0.98	809	72.83
apache/rocketmq	8,419	1.34	/	/
apache/httpcomponents-core	3,827	1.46	3,674 (200)	99.17
apache/httpcomponents-client	3,641	1.00	3,376 (200)	93.82
google/guice	2,110	1.51	2,017 (200)	105.64
google/gson	1,993	0.92	1,706 (200)	89.81

For the nine projects where CValue can be evaluated, the time required to obtain CValue for a single commit is significantly higher than for simple metrics. On average, it takes 95 times longer to obtain CValue, with the maximum being 170 times longer. The minimum ratio is 24 times longer, observed in the `apache/commons-release-plugin` project, which has a relatively smaller codebase. Larger projects exhibit a disproportionately greater increase in the time required to obtain CValue metrics compared to simple metrics.

4.3.2. Case Study

Case 1: Simple Metrics vs. Complex Metrics

In the `google/guice` project, we select the commit with the hash value `bb59fbfe998dc741162fdee658ba8f45b7069c53` as a case study. The human label for this commit in the project is the highest within the repository, reflecting its significant contribution to the project. After manually reviewing the commit, we find that it developed one of the core features of the library and provided related test code. We observe that the ELOC measure for this commit is 0.02, and the CVALUE measure is 3.07, both of which significantly underestimated the value of this commit. Meanwhile, the simpler metrics, such as LOC (1100) and Unit_complexity (1), better reflected the true value of the commit. We believe the primary reason for this discrepancy is that both CVALUE and ELOC are calculated based on the AST tree, while the main contribution of this commit is in Kotlin files, which could not be parsed by these tools as AST, leading to the underestimation of its contribution. This case illustrates that although complex metrics capture multiple factors, they can sometimes overlook contributions that are well-measured by simpler metrics, leading to misinterpretations due to the limitations of the tools.

Case 2: NALOC vs. LOC

In the `apache/httpcomponents-client` project, we select the commit with the hash value `de5c6a237a7af88d7f3e127f8c9e41e8e38db7f6` as a case. The human label for this commit in the baseline dataset is 0, as the commit only removed sample code from the project, as described in its commit message: "Deleted sample code moved to HttpComponents Website project." When we measure this commit using LOC and NALOC, we observe that since only three Java files were deleted, the NALOC value is 0, while the LOC value is 269. LOC misestimated the contribution of the deleted code, whereas NALOC accurately captured the contribution of this change. Therefore, when a commit removes a large amount of irrelevant code, LOC measurement can become skewed, but NALOC remains unaffected.

Finding 2. Complex code metrics do not consistently offer advantages over simple ones across various projects, and their high costs raise questions about their practical applicability in real-world scenarios. It is not advisable to completely replace simple metrics with complex ones. In contrast, NALOC is posited as a superior, streamlined metric that enhances the assessment of developers' output with increased efficiency.

4.4. *RQ3: Can quality metrics derived from static analysis tools reliably reflect developers' output?*

Table 4 and Table 5 present metrics related to false positive occurrences, the precision of detecting genuine issues, and the average time investment associated with SATs. The evaluation is contextualized within the framework of each commit that introduces or fixes bugs across nine projects.

Table 4: Performance & Efficiency Metrics for SonarQube

Project	**Time Cost(s)**	**Bug**		**Smell**		**Violation**	
		Recall	**FPR**	**Recall**	**FPR**	**Recall**	**FPR**
apache/commons-release-plugin	32.72	0.0184	0.0081	0.0982	0.0478	0.0982	0.0533
apache/commons-exec	18.33	0.0424	0.0230	0.2783	0.1117	0.2736	0.1148
apache/commons-cli	47.34	0.0782	0.0494	0.3691	0.1612	0.3758	0.1603
apache/commons-ognl	45.23	0.0547	0.0232	0.3593	0.2167	0.3672	0.2202
apache/rocketmq	121.07	0.1930	0.1255	0.3729	0.2535	0.3779	0.2600
apache/httpcomponents-core	117.26	0.1122	0.1153	0.3281	0.3530	0.3381	0.3696
apache/httpcomponents-client	132.21	0.0972	0.0894	0.3628	0.3745	0.3861	0.3902
google/guice	223.62	0.0980	0.1409	0.3303	0.3212	0.3321	0.3263
google/gson	43.65	0.0717	0.0529	0.4435	0.3511	0.4435	0.3587
Avg.	86.82	0.0851	0.0697	0.3269	0.2434	0.3325	0.2504

Table 5: Performance & Efficiency Metrics for PMD

Project	**Time Cost(s)**	**Code Style**		**Best Practice**		**ALL**	
		Recall	**FPR**	**Recall**	**FPR**	**Recall**	**FPR**
apache/commons-release-plugin	16.67	0.9571	0.9672	0.3252	0.2131	0.1043	0.0383
apache/commons-exec	16.90	0.0963	0.0667	0.0667	0.1024	0.1037	0.0802
apache/commons-cli	15.66	0.1477	0.0927	0.2685	0.2738	0.1924	0.1170
apache/commons-ognl	19.87	0.1526	0.0612	0.0684	0.0362	0.1842	0.0886
apache/rocketmq	10.39	0.2851	0.2097	0.2010	0.1508	0.3500	0.2573
apache/httpcomponents-core	13.58	0.0753	0.1243	0.0284	0.0438	0.1449	0.2042
apache/httpcomponents-client	14.03	0.0900	0.1309	0.0726	0.0751	0.1727	0.2187
google/guice	17.43	0.1906	0.1804	0.0980	0.0797	0.2396	0.2263
google/gson	16.76	0.1326	0.1189	0.0435	0.0441	0.1848	0.1649
Avg.	15.70	0.2364	0.2169	0.1303	0.1132	0.1863	0.1551

Perception of Developers' Key Quality Behaviors: The accuracy of all metrics from SonarQube and PMD across the nine projects averages below 40%, with both tools exhibiting a false positive rate below 30%. Furthermore, the tools' ability to perceive developers' key quality behaviors is strongly correlated with the project characteristics. For SonarQube, the average recall rate for the Bug metric, which detects critical defects, is only 8.51%, while maintaining the lowest average false positive rate of 6.97%. In terms of code smell detection, the Smell category achieves an average recall rate of 32.69%, but the false positive rate increases to 24.34%. The accuracy of the rule violation metrics closely resembles that of the Smell metric, with an average recall rate of 33.25% and an average false positive rate of 25.04%. Cross-project analysis reveals that `apache/rocketmq` achieves the highest recall rate for the Bug metric at 19.30%, while `google/gson` obtains a recall rate of 44.35% for the Smell metric, confirming that tool performance is significantly affected by project characteristics. For PMD, the Code Style metric achieves an average recall rate of 23.64%, but it also generates an average false positive rate of 21.69%. Of particular note is the `apache/commons-release-plugin` project, which achieves a 95.71% recall rate for code style but with an alarmingly high false positive rate of 96.72%, reflecting significant issues with the credibility of the detection results. In contrast, the Best Practices metric demonstrates more conservative behavior, with the average recall rate dropping to 13.03% and the false positive rate reducing to 11.32%, indicating that while this metric has a better ability to control false positives, it suffers from a higher rate of missed detections. Overall, the combined rule metrics show a fluctuation in the average recall rate at 18.63% with an average false positive rate of 15.51%.

Time Consumption of Static Analysis Tools: The two tools exhibit significant differences in time efficiency. SonarQube's average time cost is 86.82 seconds, indicating its high computational expense. The execution times across projects show considerable variability, ranging from 18.33 seconds to 223.62 seconds. Larger projects demonstrate a clear time burden, with the `google/guice` project taking up to 223.62 seconds, and two projects exceeding 100 seconds. This time consumption strongly correlates with project size and complexity, suggesting that the tool's computational resource requirements increase non-linearly with code volume. Although smaller projects, such as `apache/commons-exec`, remain within an acceptable time frame of 18.33 seconds, the overall time expense could challenge the responsiveness of continuous integration pipelines. In contrast, PMD offers better operational efficiency, with the average analysis time across all measured projects remaining steady at 15.70 seconds, and execution times ranging from 10.39 seconds to 19.87 seconds, reflecting a reasonable distribution. This slight variability is positively correlated with project size and complexity, as the `apache/rocketmq` requires only 10.39 seconds, while the `apache/commons-ognl` takes 19.87 seconds.

Finding 3. The metrics provided by both SonarQube and PMD fail to accurately detect key software quality changes in developers' output. Additionally, both tools incur substantial time costs, particularly SonarQube. These limitations restrict the utility of static analysis tools in providing appropriate and valuable assessments of code quality in developers' output assessments.

5. Practical Study

5.1. Industrial Scenario for Assessing Developers' Output

We collaborate with a comprehensive securities company that employs over 8,000 people. This company is a listed securities financial holding group that offers a full range of professional integrated financial services, including securities, futures, asset management, wealth management, investment banking, investment consulting, and securities research. The company in our study uses a combination of process metrics and product metrics to comprehensively assess the developers' output, assigning each developer a score at the end of every month to reflect their output for that period. Besides, the company in our study only uses LOC as the metric for software efficiency and does not adopt software quality metrics.

Recently, personnel responsible for assessing developers' output at the company reports anomalies in the LOC software metric, leading to incorrect evaluations of several developers' output. This raises concerns about the fairness of the company's compensation practices. One example is the LOC changes made by developers each month, which is a commonly used efficiency metric for evaluating developers' output. We track the LOC changes for a software development department consisting of eight members. Several members of this department record monthly LOC changes that far exceeds the expected upper limit. Upon manual inspection of their code commits, we discover that many of these commits are cloned from their previous repositories. This phenomenon raises concerns within the company about the reliability of the software metrics used. Given that the evaluation of certain software metrics in some projects may produce anomalies, the relevant evaluators within the company seek a clearer understanding of the current status of these anomalies in software metric occurrences. The company agrees that even if only a portion of project outputs are incorrectly assessed, it can impact the fairness of developers' output assessments. The company in our study is particularly interested in understanding the use of software metrics in academic research. Moreover, the company seeks to investigate the usage patterns of both software efficiency and software quality metrics. To achieve these objectives, we provide both theoretical support and practical observations for software metrics migration plan.

When selecting new software metrics, the company aims to focus on both effectiveness and cost. **Effectiveness** refers to the company's desire for the new metrics to accurately reflect developers' output and deliver good results across most projects. **Cost** refers to the company's goal of ensuring that the new metrics can be applied to every commit in past repositories within a reasonable computational cost. Given that developers tend to submit their code at the end of the month, collecting relevant developer metrics at this time could require more than three days to process. To prevent the existing flawed developer output evaluation models from further damaging developers' trust in managers [45, 46], we avoid directly involving developers in specific software output evaluation schemes, such as through surveys

5.2. Rapid Review

Since the company's current system for assessing developers' output relies solely on LOC as a metric, there is limited familiarity with alternative software metrics. Therefore, we aim to review past academic research on software metrics, focusing on commonly used software quality and efficiency metrics. This review will provide alternative options for migration plan. In addition, the company required the completion of the migration plan in a short time frame.

5.2.1. Previous Researches Metrics Collection

Due to the significant impact of abnormal software metrics on the company's current software projects, as discussed in Section 5.1, the company aims to quickly develop a software metrics migration plan within a limited time. Considering that our literature review is based on practical needs and has a greater sensitivity to time, we adopt the Rapid Review method. Rapid Review [47, 48] is a secondary research strategy that accelerates the traditional Systematic Review process. The goal is to significantly shorten the time required to collect, analyze, interpret, review, and publish evidence that may benefit practitioners. Although some concessions are made in the methodology to provide evidence to practitioners more promptly, Rapid Review still follows systematic protocols. Specifically, we follow the guidelines for Rapid Review [49] and integrate the approach by Rico et al. [50], selecting a rapid review to examine the quantitative metrics used for developer output assessment in previous academic literature.

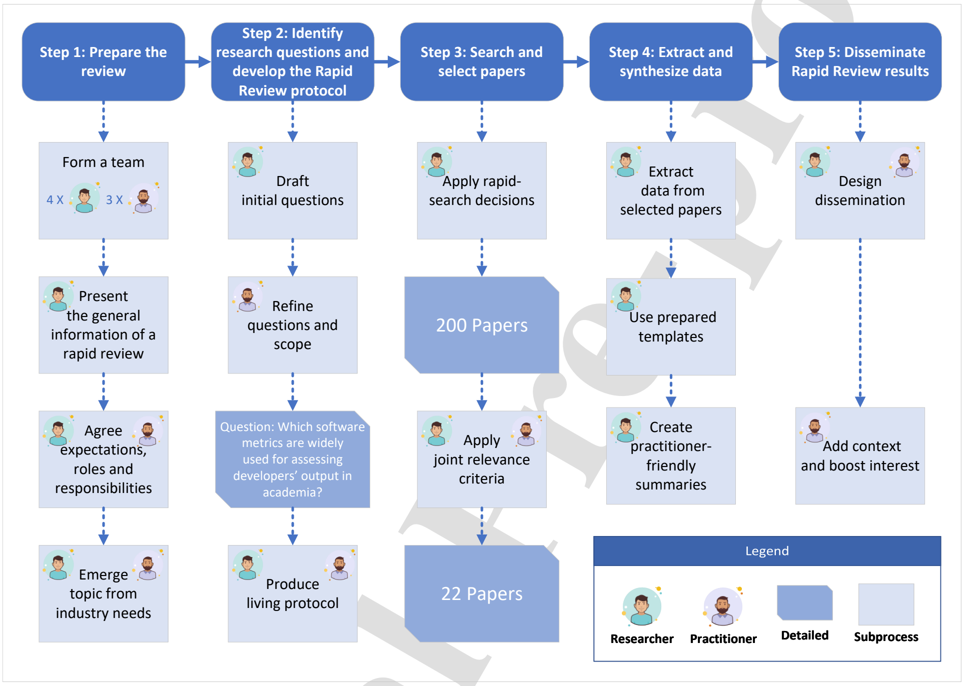

Step 1: Prepare the review
Step 2: Identify research questions and develop the Rapid Review protocol
Step 3: Search and select papers
Step 4: Extract and synthesize data
Step 5: Disseminate Rapid Review results
Form a team
4 X
3 X
Present the general information of a rapid review
Agree expectations, roles and responsibilities
Emerge topic from industry needs
Draft initial questions
Refine questions and scope
Question: Which software metrics are widely used for assessing developers' output in academia?
Produce living protocol
Apply rapid-search decisions
200 Papers
Apply joint relevance criteria
22 Papers
Extract data from selected papers
Use prepared templates
Create practitioner-friendly summaries
Design dissemination
Add context and boost interest
Legend
Researcher
Practitioner
Detailed
Subprocess

Figure 7: Overall Framework of Rapid Review

Figure 7 illustrates our Rapid Review framework. First, a team is formed consisting of four software engineering researchers and three experienced industry developers with expertise in output assessment management to understand the specific needs, as described in Section 5.1. The team reaches a consensus on the actual industry needs, as well as the objectives and roles for the rapid review process.

In the second step, the research team, based on discussions, formulates the research question: Which software metrics are widely used for assessing developers' output in academia? The scope of the search is agreed to focus on quantitative metrics for assessing developers' output. Subsequently, the team establishes the criteria for literature selection and requirements for the Rapid Review, while considering the timeliness demands.

In the third step, we conduct searches across several databases, including Google Scholar, IEEE Xplore, and the ACM Digital Library. We adhere to the principle [49, 51, 52] of searching relevant content within a limited number of databases to accelerate the Rapid Review process. For constructing the search strings, we refer to the literature [50] and initially design the strings based on our expertise, testing their effectiveness across different search ranges. Based on the search results, we assess the relevance of the returned papers to our research questions. Prior to conducting the search, we present the potential search strings to practitioners and finalize the search string based on their feedback. The search string "(software OR developer) AND (performance OR output OR contribution)" is used to identify relevant studies in the title. The search is limited to publications between January 2021 and March 2024, without restrictions on publication types. Initially, 200 papers are collected, and after manual screening, papers not related to quantitative assessment are excluded, leaving 18 papers. Specifically, the team members first examine the titles and abstracts to filter out irrelevant papers, then proceed to a deeper review to determine their relevance to software metrics for developers' output assessment. In addition, using a snowballing technique, we identify additional papers by examining the references of the selected studies. The team aims to compensate for any papers potentially overlooked due to the time constraints of the Rapid Review process. By carefully applying this systematic approach, we obtain a final set of 22 papers.

In the fourth step, we analyze the use of process and product metrics, with particular attention to efficiency and quality metrics within product metrics. Finally, the team shares the results and selects the software metrics that the company plans to adopt.

5.2.2. Result

We present a coarse-grained overview of the factors to which the metrics belong in various developers' output assessment schemes. Specifically, we focus on software efficiency metrics related to LOC and software quality metrics associated with SATs.

Table 6 presents the factors utilized for developers' output assessment, as derived from the papers we have collected.

In the entirety of the collected literature, only 22.7% (5 out of 22) of the documents comprehensively took into account all categories of metric factors. software efficiency metrics and process metrics are prominently considered as primary factors in developers' output, with approximately 65% (14 out of

Table 6: Developers' Output Evaluation Factors in Reviewed Papers

Metric Factor	**Mentioned Paper**
Process Metric	[4] [6] [22] [23] [27] [39] [53] [54] [55] [56] [57] [58] [59] [60]
Software Efficiency Metric	[4] [5] [6] [7] [19] [21] [22] [23] [26] [27] [40] [54] [61] [62]
Software Quality Metric	[6] [19] [21] [22] [23] [26] [27] [39] [40] [54] [61]

22) of the articles referencing the use of indicators related to these aspects. In contrast, the least considered software quality metric factor category is included in just 50.0% (11 out of 22) of the reviewed papers as part of their evaluation criteria.

Table 7 exhibits the application of software efficiency metrics associated with LOC and software quality metrics acquired through the employment of SATs, as presented in the reviewed papers.

Table 7: LOC Related Efficiency & SATs Quality Metrics in Reviewed Papers

Metric	**Mentioned Paper**
LOC Related	[4] [6] [19] [21] [22] [26] [27] [40] [54] [61] [62]
Quality Metric in SATs	[6] [22] [39] [40] [54]

Within the gamut of methodologies employing source code attributes, a notable 78.6% incorporate metrics that are intimately associated with LOC, highlighting its prevalent consideration. Conversely, in the context of strategies that hinge upon quality attributes, a substantial 45.5% of the methodologies derive their metrics from the application of automated SATs, underscoring the centrality of these tools in contemporary quality assessment practices.

Finding 4. Current output assessment methods primarily rely on a limited set of factors, with LOC related metrics remaining the dominant approach for measuring software efficiency metrics. Additionally, automated SATs are becoming increasingly essential for effectively assessing software quality metrics.

5.3. Difficulty in Replication Study

We don't choose industrial projects to replicate our study. The primary reason is that developers make abnormal commits to fool software metrics, lowering development quality in software repositories. Besides, some research tools can't be used commercially (e.g., ELOC [63]), and data is restricted by company confidentiality.

To further highlight the anomalies in the current software repository, we selected a typical repository, T1. Table 8 shows the LOC metrics for developer submissions over the 10 months before the company's software metrics migration plan. It is clear that developers rarely submitted code with fewer than 1,000 LOC, instead preferring to submit commits with more than 10,000 LOC per submission. Specifically, from April 2023 to July 2024, all submissions exceeded 10,000 LOC. This shows that developers tend to make tangled commits [64], including a large amount of functional code in a single commit. Previous research [64, 65] shows that tangled commits increase maintenance costs and further increase defect proneness. These tangled commits far exceed the expected workload for developers, indicating poor software development and maintenance practices within these repositories.

Month	[0,100)	[100,1000)	[1000,5000)	[5000,10000)	[10000,+∞)
2023-02	1	0	0	0	5
2023-03	0	0	0	1	7
2023-04	0	0	0	0	4
2023-05	0	0	0	0	3
2023-06	0	0	0	0	5
2023-07	0	0	0	0	5
2023-08	0	0	1	0	3
2023-09	0	0	6	0	5
2023-10	2	1	1	1	2
2023-11	1	1	0	0	2

Table 8: Repository T1 LOC Range Distribution of Commits by Month

Figure 8 shows the change in the total number of commits per month for the T1 project during the same 10-month period. We observe that in most months, the number of commits is fewer than 10, with the lowest number of commits being only 3 in one month. This raises further concerns about the quality of industry projects.

Overall, the submissions in these repositories, which do not meet the company's development standards, contribute to abnormal software metrics and lead to developers' output assessments that do not reflect their actual

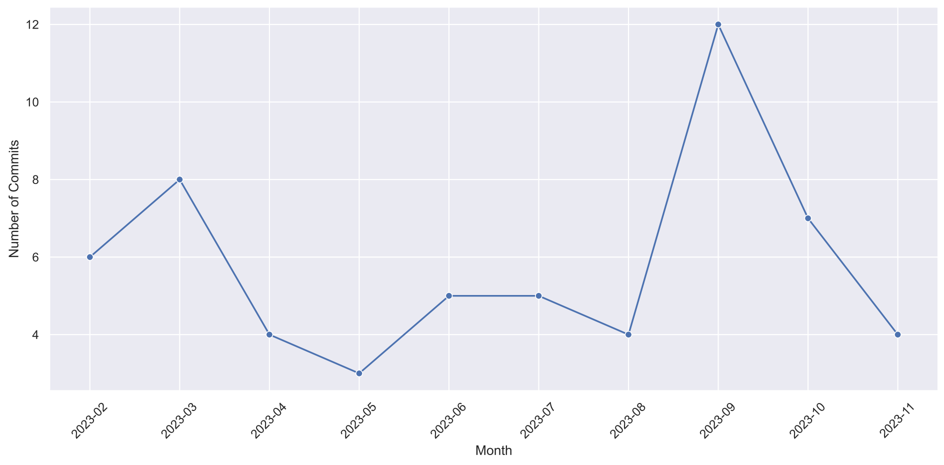


Figure 8: Monthly Commit Count of T1

workload. In most of the company's projects, the development practices are poor and do not follow the expected standards. As a result, both the research team and company management agree that these repositories do not accurately reflect the role of software metrics in assessing developers' output.

5.4. Industry Feedback Methods

To ensure that the company's software metrics migration plan address its needs, we begin preparing the plan in December 2023, working closely with the company. The plan is officially launched in July 2024. Throughout this process, we held weekly one-hour unstructured meetings with three experienced developers' output managers to report on developer commit behaviors that caused software metric anomalies. These meetings ensure that both the company and our team shared a common understanding of the issues and supported the development of the migration plan. Once the plan is nearing finalization, we conduct a one-hour semi-structured interview with the management team, during which we present our research methods and findings, offering recommendations tailored to the company's situation. We then collect feedback on our conclusions from the developers' output managers. By integrating the feedback with our research results, the company is able to finalize the migration plan. To track its effectiveness, we continue to hold weekly one-hour meetings to monitor any new anomalies in the revised software metric assessment model over the following five months.

5.5. Research Recommendations

Based on the four findings in Section 4 and Section 5.2.2, we provide four practical recommendations for the company's software metrics migration plan with the expectation that our results will have a positive impact on the migration plan. We offer the following four recommendations.

For Finding 1: We recommend introducing an LLM-generated code detection model, such as GPTSniff [66] or DetectGPT [67], to identify code generated by LLMs in developer commits. However, these models currently focus on code directly generated by LLMs at the function level. In future improvements, we aim to develop a commit-level detection model that determine if the code is edited by an LLM to change its output score.

For Finding 2: We recommend that the company retain simple metrics alongside complex ones for reference. While simple metrics may only reflect one aspect of developer efficiency, they can highlight details that more complex metrics may overlook.

For Finding 3: We recommend that the company avoid using SonarQube to remeasure every commit in existing repositories, as this would be time-consuming. Given the low accuracy of SonarQube's quality metrics, they should not be included in developers' output assessments. However, they can still serve as a reference for monitoring changes in software quality.

For Finding 4: We recommend the company use both process metrics and product metrics to balance the subjectivity and objectivity of developers' output. While product metrics may show anomalies, process metrics, with their more subjective nature, can help mitigate the impact of these anomalies on output assessment. This combined approach can enhance fairness and reliability in evaluating developers' performance.

5.6. Software Metrics Migration Plan

Based on the findings from our study, the company has developed a software metrics migration plan to improve the fairness and accuracy of evaluating developers' output. This plan addresses the challenges caused by non-standard code submissions, such as those generated by LLMs, and aims to balance both objective and subjective metrics. The following strategies are part of the migration plan.

5.6.1. Introducing Code Generation Detection for LLMs in the Future

Due to concerns that LLMs could be used by developers to cause anomalies in LOC, the company hopes to introduce a detection mechanism for code rewritten by LLMs in the future. The main reasons the company does not plan to introduce this model in the current initiative are threefold. First, existing LLM code generation detection tools focus primarily on detecting code directly generated by LLMs, which does not meet the company's needs. Second, after manual review by both the company and our team, it is found that developers can use less costly code cloning techniques to increase LOC. Third, LLM-rewritten code tends to show better performance in increasing LOC in functions with fewer LOC.

5.6.2. Retaining LOC-Based Efficiency Metrics with New Metrics

The company will continue using the traditional LOC-based metrics to evaluate developer efficiency, but will also introduce new metrics such as ELOC. Although complex software metrics incur high costs, as observed in our subsequent review where the company only migrated the metric measurements for a small portion of its libraries, they can reflect more aspects of software efficiency. By combining LOC-based metrics and ELOC, the company will get a better understanding of developer efficiency across different repositories. This will allow for a more complete evaluation of productivity and ensure that software metrics are not overly focused on one aspect of development.

5.6.3. Using SonarQube for Quality Analysis Only

SonarQube will be used solely for quality analysis, not for performance evaluation. While SonarQube can help identify potential quality issues in code, our study found that its quality metrics are not always accurate in reflecting changes in developers' work, with an accuracy rate below 50%. Therefore, SonarQube will not be used to evaluate developer performance directly, but will still be helpful for understanding the overall quality of the repository.

5.6.4. Combining Product Metrics with Process Metrics

The company will combine objective product metrics (e.g., LOC) with subjective process metrics to better assess developers' output. Product metrics alone can be influenced by anomalies, such as non-standard code submissions. Process metrics, such as the time spent on tasks or collaboration efforts, will help provide additional context. This approach allows the company to consider both the efficiency and the quality of a developer's work. By using both types of metrics, the company can improve the accuracy of developer evaluations and reduce the impact of metric distortions.

5.6.5. Selecting Appropriate Software Product Metrics

The selection of software product metrics fundamentally depends on the specific needs of the organization. This process is inherently contextual and involves critical trade-offs, rather than seeking a universal optimal solution. The primary task in selecting software product metrics is to clearly define the core objectives of quantitative assessment. Different goal orientations drive distinctly different combinations of metrics.

A thorough evaluation of the inherent characteristics of candidate metrics is essential, including their effectiveness in reflecting the target dimensions, robustness against manipulation, the comprehensive cost of data collection, computation, and maintenance, as well as the interpretability of the results for stakeholders. Decisions must be firmly grounded in the specific context of the project and organization, considering factors such as project size, the availability of historical data, and the feasibility cost of re-calculating metrics. The existing toolchain also forms the technical foundation for the automation of metric collection and integration.

At the strategy level, the company adopt a combination of core and auxiliary metrics. For example, basic metrics such as NALOC, which are low-cost, easily collected, and relatively robust, are used as core metrics, while enhanced metrics such as the ELOC, which provide deeper insights, are used as auxiliary metrics in key scenarios or when resources allow. To mitigate the potential bias of using a single product metric, multidimensional cross-validation should be performed by integrating process metrics such as task complexity and collaboration records.

Finally, the construction of a metric system is a dynamic process. It is essential to establish continuous monitoring and iterative optimization mechanisms, regularly evaluating the actual effectiveness and cost-efficiency of the metric combination. Adjustments should be made based on project evolution and feedback to ensure that the metrics consistently serve the established business objectives. Due to confidentiality requirements, the final metric combination cannot be disclosed in detail. However, the aforementioned systematic framework ensures that the metric selection decisions closely align with organizational needs.

5.7. Continual Observation

We select three industry software repositories to represent how repositories starting at different times adapted to the new evaluation system. As mentioned in Section 5.3, T1 is a long-term, ongoing software repository. T2 is a repository that started development during the preparation of the software metrics migration plan, starting in April 2024. T3 is a repository that began development after the official software metrics migration plan is launched, starting in July 2024. This is because the company started preparing the software metrics migration plan in December 2023 and informed the relevant development managers about the existing anomalies in the software metrics. In July 2024, the company officially informed developers about the new factors in the developers' output assessment system.

Figure 9 shows the change in the total number of commits over time for these three software repositories during this period. We find that after the software metrics migration plan is implemented, developers initially increased the number of code commits only in small amounts during the first two to three months. After becoming more familiar with the new software metrics system, developers begin to make more commits. This suggests that after adopting the new software metrics system, developers gradually standardize their commit processes. Developers are now less inclined to submit tangled commits and instead break them down. The change in the developers' output assessment model encouraged developers to submit more organized code, which in turn helped improve software quality [65, 68, 69].

Lessons Learned 1. Developers need some time to adjust to the new developers' output assessment system. In the long run, this adaptation will benefit the quality of the software delivered by developers.

Table 9 shows the LOC metrics for developer commits in these three software repositories, from the start of the software migration plan preparation or the beginning of the project until November 2024.

During the preparation period for the migration plan, we observe that in T1, after the company informed developers about the software metric anomalies, developers reduced the number of commits with more than 10,000 LOC from January to May. They split these large commits into smaller ones. However, in June, when the migration plan had not yet been implemented, developers submitted five commits with more than 10,000 LOC. In T2, since this project is newly started, developers tried to avoid large commits, but

Table 9: LOC Range Distribution of Three Industrial Repositories by Month

	Month	[0,100)	[100,1000)	[1000,5000)	[5000,10000)	[10000,+∞)
T1	2023-12	1	4	4	0	5
	2024-01	0	4	5	0	1
	2024-02	0	1	2	1	0
	2024-03	0	0	2	1	1
	2024-04	0	0	4	2	3
	2024-05	0	0	2	2	2
	2024-06	0	2	3	1	5
	2024-07	1	11	16	0	0
	2024-08	8	52	6	4	0
	2024-09	22	41	11	0	0
	2024-10	31	65	11	0	0
	2024-11	26	83	6	1	1
T2	2024-04	0	1	0	1	1
	2024-05	0	0	1	2	1
	2024-06	0	0	2	1	1
	2024-07	0	0	4	0	0
	2024-08	8	9	3	0	0
	2024-09	16	9	4	0	0
	2024-10	20	8	3	0	0
	2024-11	17	14	2	0	0
T3	2024-07	1	2	5	0	0
	2024-08	6	6	2	0	0
	2024-09	7	8	1	0	0
	2024-10	1	6	7	1	0
	2024-11	7	24	3	0	0

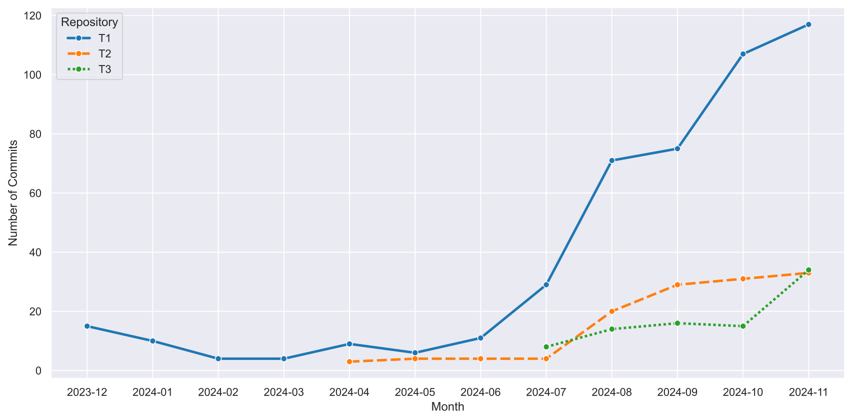


Figure 9: Monthly Commit Count of Three Industrial Repositories

they still did not submit commits with fewer than 1,000 LOC. This shows that after being informed of the software metric anomalies, developers tried to avoid making large LOC commits. However, since the software metrics used for evaluation had not changed, after a few months of adjusting their commit patterns, developers returned to making large LOC commits.

Lessons Learned 2. When the software metrics used in the developers' output assessment system remain unchanged and developers are simply informed about existing anomalies, developers try to change their commit patterns to avoid extreme software metric scores. However, this does not fully fix the anomalies, and after a period, developers submit commits causing extreme anomalies in the software metrics again.

After the new software metrics system is implemented, developers in all three projects stop submitting commits with more than 10,000 LOC and mostly avoid commits larger than 5,000 LOC. We find that under the new software metrics system, developers mainly submit commits with LOC between 100 and 1,000, which aligns with the small and micro commits recommended in the literature [70, 71]. This supports the idea that the old software metrics system led to unfair and unreasonable developers' output assessment.

Lessons Learned 3. The new software metrics system significantly reduces the occurrence of metric anomalies and helps developers change their commit patterns, improving the quality of software delivery.

6. Discussions

6.1. Ethical Risks of LLM-assisted development

This study reveals the phenomenon where developers may manipulate assessment metrics through LLMs, and suggests the future development of detection models for code generated by LLMs. While this finding is crucial for ensuring the fairness of performance evaluations, it is important to recognize the potential ethical risks associated with overly restricting LLM tools, as such limitations could hinder the revolutionary improvement in software development efficiency.

Firstly, the core demand of the software industry is to maximize productivity and innovation efficiency. The use of LLM-assisted development has been proven to accelerate code delivery. If such tools are completely prohibited due to flaws in the evaluation system, it would amount to "throwing the baby out with the bathwater." Secondly, excessive restrictions on LLM usage could lead to counterproductive incentives. Developers may be forced to revert to inefficient manual coding or turn to more covert forms of metric manipulation, such as code cloning, which would further exacerbate evaluation distortion.

More importantly, the root of the problem lies in the inadequate adaptability of the current developers' output assessment system, rather than the tools themselves. Therefore, we propose that the development of detection models for LLM-generated code should aim to distinguish between manually written code and code developed with LLM assistance, providing a basis for evaluating developers' outputs according to different scenarios, rather than expecting the prohibition of LLM tools. Furthermore, in Section 6.2.2, we anticipate new evaluation models for developers' outputs under LLM-assisted development.

6.2. Implications

Our study provides practical guidance on the usage patterns of software product metrics in the context of developers' output assessment. These insights are particularly valuable for practitioners, researchers, educators, and tool builders responsible for evaluating developers' output.

6.2.1. *Implications for Practitioners*

Implication 1: Goal-Oriented Usage Patterns.

Study Results: Finding 1 indicates that LLMs can significantly manipulate simple metric LOC through their preferred methods. Finding 3 reveals that existing software quality metrics derived from SATs do not align with developers' true quality contributions.

Analysis: This confirms that when managers overly emphasize metric-driven evaluation models in developers' output assessments, developers tend to focus on improving metric scores rather than enhancing the actual quality, potentially undermining software quality and team collaboration. On the other hand, this further suggests that when managers adopt overly metric-driven usage patterns, they are often influenced by the inherent evaluation biases of software product metrics, leading to a skewed assessment of developers' contributions. Therefore, we recommend that practitioners adapt the principle of goal-oriented software product metrics usage.

Actionable Recommendation: Managers should focus primarily on business and technical objectives such as timely delivery of value, reducing production environment defects, and improving code maintainability. Software metrics should then be used as supplementary tools to support, measure, and standardize high-quality development processes, rather than as ultimate goals. As noted in [72], metrics must be aligned with stakeholder goals from the outset.

Implication 2: Multi-Dimensional Usage Patterns.

Study Results: Findings 2 and 3 indicate that single-dimensional software product metrics often suffer from inherent flaws due to their calculation methods, such as inaccurate evaluations or excessive evaluation time.

Analysis: When only narrow aspects of software metrics are selected for evaluation, these flaws are further amplified, leading to the emergence of entangled commits in our practical scenarios, thereby damaging software quality. Additionally, an excessive and broad measurement process results in the accumulation of costs, leading to prohibitively high expenses. Furthermore, Research [73] highlights that such an over-measurement approach does not provide significant value. As our practical study demonstrated, the software metrics usage pattern we constructed considers the multi-dimensional aspects of the metrics and balances their specific measurement value. Our findings suggest that this new evaluation model better motivates developers to improve delivery quality.

Actionable Recommendation: Managers should avoid over-relying on single dimensions and instead expand the evaluation dimensions to include code efficiency, quality, maintainability, and other factors, weighing the costs of collection against the actual value of the metrics to more comprehensively and efficiently reflect developers' contributions.

Implication 3: Usage Patterns Consistent with Scenario Requirements.

Study Results: The lessons learned from our practical study show that developers adjust their submission strategies based on the evaluation system's model, and the failure of the original evaluation system highlights the need for updates based on the situation.

Analysis: If a particular metric exhibits persistent anomalies during the evaluation process, managers should consider whether to replace that metric and analyze its specific impact. Otherwise, the continued use of that metric within the evaluation system may lead to long-term negative consequences for the company, as seen in our practical scenario.

Actionable Recommendation: Managers should gradually promote the iterative optimization of the evaluation system and metric framework, guiding developers to adapt to new evaluation standards, thereby improving software delivery quality over the long term. As stated in [74], the usage patterns of software product metrics must evolve according to the needs of the development scenario.

6.2.2. *Implications for Researchers*

Implication 4: Building Developers' Output Evaluation Systems in LLM-Assisted Development Scenarios.

Study Results: Finding 1 indicates that the LOC metric is susceptible to manipulation by LLMs, so traditional software product metrics need to be adapted.

Analysis: The susceptibility of LOC to manipulation suggests that traditional software metrics might not accurately reflect developer effort or contributions when LLMs are involved. This challenges the validity of existing metrics as proxies for productivity and underscores the need to reassess their applicability in LLM-assisted contexts. Furthermore, the value of LOC as a meaningful measure of output has significantly diminished in such scenarios.

Actionable Recommendation: Future research should first clarify the validity of traditional software product metrics in LLM-assisted development scenarios, as well as the extent to which they are influenced. It aims to investigate whether existing software metrics remain a valid proxy for developer effort and contributions in the context of LLM-assisted software development. Additionally, future research should further build on the value dimensions for measuring developers' output in LLM-assisted development scenarios, as the value of the LOC metric has been significantly reduced. Furthermore, future studies should explore developers' output assessment metrics based on LLM as a Judge, which has already been proven to have unique value in the software engineering field [75, 76, 77].

Implication 5: Optimizing the Cost-Effectiveness of Complex Efficiency Metrics.

Study Results: Finding 2 indicates that current complex software efficiency metrics incur high costs but do not comprehensively outperform simpler software efficiency metrics.

Analysis: This suggests that the additional computational and resource investments required for complex metrics do not yield proportionally greater insights compared to simpler alternatives. The narrow focus on limited comparison dimensions may lead to biased or incomplete conclusions about their effectiveness.

Actionable Recommendation: Future research needs to optimize the calculation efficiency of complex software efficiency metrics to support more efficient developers' output assessment. Moreover, when measuring the effectiveness of complex software efficiency metrics, future research should incorporate a broader range of comparison dimensions to avoid misunderstandings that may arise from focusing on a single aspect.

Implication 6: Improving Quality Verification Methods.

Study Results: Finding 3 indicates that SAT tools have demonstrated poor performance in assessing developers' quality contributions.

Analysis: This poor performance stems from the fact that current SAT tools lack tailored evaluation mechanisms for capturing real quality changes in developers' output. The generic design of these tools fails to accommodate configurable and context-aware criteria necessary for accurate contribution assessment.

Actionable Recommendation: Future work should build evaluation frameworks that can verify real quality changes. Existing SAT tools are not specifically designed for measuring developers' output contributions, often neglecting the unique requirements within those contributions. Future work should develop a quality analysis tool specifically for configurable evaluation criteria within the developers' output assessment process, as mentioned in [78].

6.2.3. Implications for Educators

Implication 7: Incorporating Professional Ethics in Developers' Output Evaluation in Software Engineering Curriculum.

Study Results: Finding 1 reveals that developers can significantly manipulate LOC metrics by rewriting code to artificially inflate the LOC score, leading to evaluation anomalies.

Analysis: This phenomenon highlights the real-world risks of metric manipulation and its impact on evaluation fairness and software quality. Additionally, as reflected in the Lessons Learned, the developers' behavior in manipulating metrics and the adaptation period to new evaluation systems further emphasize this issue.

Actionable Recommendation: Educators should stress the importance of professional ethics in their curriculum, explicitly warning students that any attempts to manipulate metrics through non-technical means are not only ineffective but also undermine team trust, software quality, and personal professional reputation. By incorporating the results of this study into teaching cases, educators can provide students with more realistic learning experiences and promote advancements in software engineering education, particularly in evaluation ethics and technical practices.

6.2.4. Implications for Tool Builders

Implication 8: Introducing More Detection Features Related to Developer Activities.

Study Results: As explicitly revealed in Finding 1, LLMs can manipulate traditional metrics like LOC through their preferred code rewriting methods, significantly undermining the credibility of developer output evaluations based on such metrics.

Analysis: The ability of LLMs to artificially inflate or distort conventional metrics like LOC indicates that using simple code metrics alone is no longer sufficient to accurately reflect genuine human developer contributions. This manipulation poses a growing threat to fair and trustworthy performance assessment.

Actionable Recommendation: Tool builders should incorporate detection and identification features for LLM-generated code into developer output evaluation tools. This will provide managers with an essential additional dimension to identify and filter out code potentially aimed at exploiting metric systems, helping ensure evaluations better reflect true developer effort and are based on more reliable data.

Implication 9: Enhancing Metric Visualization.

Study Results: Based on the lessons learned of our practical study, developers often adjust their submission behavior due to the metrics set in the evaluation system, but such adjustments do not always improve code quality and may even lead to new anomalies. Findings 2 and 3 highlight the discrepancies between multiple software product metrics and actual evaluations, as well as the high costs of some metrics.

Analysis: The behavioral changes triggered by metric-driven evaluation systems do not consistently align with genuine quality improvement goals, and may inadvertently encourage counterproductive actions. The divergence between metric readings and real-world outcomes, combined with substantial measurement costs, suggests that an over-reliance on such metrics without clarifying their purpose and constraints can misrepresent developer performance and introduce operational risks.

Actionable Recommendation: Output evaluation tools should provide more intuitive and interpretable metric visualization interfaces to help both managers and developers understand the actual meanings and limitations of the metrics. Moreover, the tools should support traceable analysis of metric data, helping users understand the code change behaviors behind metric fluctuations, thereby preventing new risks from arising due to blindly pursuing specific metrics.

Implication 10: Supporting Multi-Dimensional Integration.

Study Results: Finding 2 suggests that complex metrics are not always superior to simple metrics and may incur high costs. Validation in industrial migration contexts in Section 5.6.4 confirms that a combined "process + product" metric approach can be effective.

Analysis: This implies that hard-coding a fixed set of metrics into development tools is inefficient, as no single metric or combination optimally serves all evaluation scenarios. Instead, metric utility depends heavily on project-specific factors such as type, team maturity, and quality objectives.

Actionable Recommendation: The tools should aim to provide a flexible, configurable multi-dimensional metric integration framework. This means that tools should not only offer a wide range of software metric measurements but, more importantly, provide powerful interfaces and configuration options that allow managers to easily select, weight, and integrate different metrics based on specific project contexts (e.g., project type, team maturity, quality goals).

6.3. Threats to Validity

Several factors may limit the validity of our findings, which we outline below. These potential threats encompass both internal and external validity, as well as construct and conclusion validity.

6.3.1. Construct Validity

Our experiment faces several potential construct validity threats. When investigating the impact of LLMs on the LOC metric, we used the cloc tool to measure LOC. However, this tool itself may introduce bias. To reduce the effect of the tool, we used the same tool the company uses. During the process of collecting software metrics, we applied the Rapid Review method, which could have caused us to overlook some relevant literature and metrics. However, the results from our final migration plan show that the existing solutions helped the company reduce the issues in software metrics, which reduces this threat. Another issue is that we used tools like SonarQube to collect software metrics, and these metrics could be affected by the tools themselves. To assess the reliability of static analysis, we used bug-fixing commits and the SZZ algorithm to detect bug-introducing commits. However, this algorithm has limitations, such as false positives and false negatives, which can introduce noise and affect the accuracy of our conclusions about the effectiveness of static analysis.

6.3.2. Conclusion Validity

In our case study, we only selected a small number of examples that were most relevant to the research questions, which may mean we missed other important cases. Also, when studying the impact of LLMs on LOC, we collected only 50 samples, which could affect the validity of our conclusions. However, this sample size is large enough to show that developers can intentionally increase their LOC scores by using LLMs. Due to our use of the Rapid Review method to collect relevant literature, we do not include databases such as Scopus and restricted our search to specific sources. This limitation may have some impact on the completeness of our conclusions. To mitigate this effect, we employ the snowballing technique to supplement additional literature.

6.3.3. Internal Validity

The assessment of simple and complex code metrics' performance relies on an open-source dataset sourced from the primary authors. While reproduction efforts confirm initial findings, the basis for variations among metrics remains unclear, posing a substantial internal validity challenge due to unresolved causality.

6.3.4. External Validity

A possible issue is that we focused only on single-language Java projects. While this approach ensures consistency, it may limit how well our results apply to other programming languages. Additionally, our conclusions were based on data from a single company, which means our recommendations for software metric migration might not be applicable to other companies. Furthermore, due to the lower quality of industry repositories, we only selected nine open-source projects for the experiment, which might not apply to other types of projects. In future research, we plan to explore how software metrics are used in repositories that have returned to normal development modes within the company.

7. Conclusion

As software product metrics play an increasingly central role in assessing developers' output, understanding their usage patterns becomes critical for ensuring fair, efficient, and practical assessments. Through a series of empirical studies, this work reveals key challenges in the current usage patterns of software product metrics. First, we demonstrate that developers can intentionally manipulate the LOC metric with the assistance of LLMs, resulting in inflated output scores. Second, we show that complex efficiency metrics do not consistently outperform simpler ones across all projects and often incur substantial costs, calling their practicality into question. Third, we find that widely used quality metrics derived from static analysis tools, such as those from SonarQube and PMD, fail to accurately reflect developers' contributions to software quality, especially during defect-fixing activities.

To validate and extend these findings, we collaborate with a large financial institution to examine how existing software product metrics influence developers' output assessments in practice. Our work helps the company redesign its assessment scheme by introducing cost-effective and more reliable metrics while limiting the inappropriate use of existing tools. The adopted migration plan significantly reduces output score anomalies within five months, demonstrating the practical value of our empirical insights.

Future research will focus on: (1) finding ways to measure developers' output in LLM-assisted development, (2) identifying developer behavior patterns that cause metric anomalies using machine learning techniques and (3) validating metrics and conclusions in industry settings with normal commit behaviors.

CRediT authorship contribution statement

Wentao Chen: Conceptualization, Methodology, Writing – original draft, Visualization. Huiqun Yu: Validation, Writing – review & editing, Funding acquisition. Guisheng Fan: Validation, Writing – review & editing. Zijie Huang: Validation, Writing – review & editing, Funding acquisition. Yuguo Liang: Validation, Writing – review & editing.

Data Availability Statement

The code for prediction and the dataset are available at `https://zenodo.org/records/14532880`

Conflict of Interest

The authors declare no potential conflict of interest.

Funding

This work was supported by the National Natural Science Foundation of China [grant 62372174]; the Computational Biology Program of Shanghai Science and Technology Commission [grant 23JS1400600]; the Capacity Building Project of Local Universities Science and Technology Commission of Shanghai Municipality [grant 22010504100]; the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies [grant 2021-GYHLW-01007]; the China Postdoctoral Science Foundation [grant 2024M761927]; and the Shanghai 2024 Science and Technology Innovation Action Plan Star Cultivation (Sailing Program) [grant 24YF2720000].

References

[1] M. A. Akbar, A. A. Khan, N. Islam, S. Mahmood, Devops project management success factors: A decision-making framework, Software: Practice and Experience 54 (2) (2024) 257–280.

[2] D. López-Fernández, J. Díaz, J. García, J. Pérez, Á. González-Prieto, Devops team structures: Characterization and implications, IEEE transactions on software engineering 48 (10) (2021) 3716–3736.

[3] N. Patrikeos, Y. Jiang, F. Rabhi, Bridging the gap between academic curricula and industrial practice with devops education, in: 2023 30th Asia-Pacific Software Engineering Conference, 30th Asia-Pacific Software Engineering Conference, 2023, pp. 549–553.

[4] V. Orrei, M. Raglianti, C. Nagy, M. Lanza, Contribution-based firing of developers?, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, p. 2062–2066.

[5] J. Ren, H. Yin, Q. Hu, A. Fox, W. Koszek, Towards quantifying the development value of code contributions, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 775–779.

[6] J. Zhang, H. Zhang, L. Yang, L. Dong, Y. Li, An experience report on assessing software engineer's outputs in practice, in: 2023 IEEE/ACM International Conference on Software and System Processes, IEEE/ACM International Conference on Software and System Processes, 2023, pp. 13–24.

[7] Y. Sun, Z. Xu, C. Liu, Y. Zhang, Y. Liu, Who is the real hero? measuring developer contribution via multi-dimensional data integration, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, 38th IEEE/ACM International Conference on Automated Software Engineering, 2023, pp. 825–836.

[8] A. Mockus, P. C. Rigby, R. Abreu, P. Suresh, Y. Chen, N. Nagappan, Modeling the centrality of developer output with software supply chains, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 1809–1819.

[9] G. Asproni, Hans dockter on developer productivity, IEEE Software 42 (01) (2025) 129–132.

[10] C. Jaspan, C. Green, Measuring productivity: All models are wrong, but some are useful, IEEE Software 42 (2) (2025) 13–17.

[11] E. Oliveira, E. Fernandes, I. Steinmacher, M. Cristo, T. Conte, A. Garcia, Code and commit metrics of developer productivity: a study on team leaders perceptions, Empirical Software Engineering 25 (2020) 2519–2549.

[12] M. Beller, V. Orgovan, S. Buja, T. Zimmermann, Mind the gap: on the relationship between automatically measured and self-reported productivity, IEEE Software 38 (5) (2020) 24–31.

[13] T. Weber, M. Brandmaier, A. Schmidt, S. Mayer, Significant productivity gains through programming with large language models, Proceedings of the ACM on Human-Computer Interaction 8 (EICS) (2024) 1–29.

[14] A. Trautsch, S. Herbold, J. Grabowski, Are automated static analysis tools worth it? An investigation into relative warning density and external software quality on the example of Apache open source projects, Empirical Software Engineering 28 (3) (2023) 66.

[15] H. J. Kang, K. L. Aw, D. Lo, Detecting false alarms from automatic static analysis tools: How far are we?, in: Proceedings of the 44th International Conference on Software Engineering, 44th International Conference on Software Engineering, 2022, p. 698–709.

[16] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, F. Palomba, A critical comparison on six static analysis tools: Detection, agreement, and precision, Journal of Systems and Software 198 (C) (apr 2023).

[17] P. Yu, Y. Wu, X. Peng, J. Peng, J. Zhang, P. Xie, W. Zhao, Violationtracker: Building precise histories for static analysis violations, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, 45th International Conference on Software Engineering, 2023, pp. 2022–2034.

[18] P. Alliez, R. D. Cosmo, B. Guedj, A. Girault, M.-S. Hacid, A. Legrand, N. Rougier, Attributing and referencing (research) software: Best practices and outlook from Inria, Computing in Science & Engineering 22 (1) (2020) 39–52.

[19] G. Gousios, E. Kalliamvakou, D. Spinellis, Measuring developer contribution from software repository data, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, International Working Conference on Mining Software Repositories, 2008, pp. 129–132.

[20] F. Ramin, C. Matthies, R. Teusner, More than code: Contributions in Scrum software engineering teams, in: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, 42nd International Conference on Software Engineering Workshops, 2020, pp. 137–140.

[21] J. Lima, C. Treude, F. F. Filho, U. Kulesza, Assessing developer contribution with repository mining-based metrics, in: 2015 IEEE International Conference on Software Maintenance and Evolution, IEEE International Conference on Software Maintenance and Evolution, 2015, pp. 536–540.

[22] T. Diamantopoulos, M. D. Papamichail, T. Karanikiotis, K. C. Chatzidimitriou, A. L. Symeonidis, Employing contribution and quality metrics for quantifying the software development process, in: 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, 17th International Conference on Mining Software Repositories, 2020, pp. 558–562.

[23] J.-G. Young, A. Casari, K. McLaughlin, M. Z. Trujillo, L. Hébert-Dufresne, J. P. Bagrow, Which contributions count? Analysis of attribution in open source, in: 2021 IEEE/ACM 18th International Con-

ference on Mining Software Repositories, 18th International Conference on Mining Software Repositories, 2021, pp. 242–253.

[24] Y. Li, H. Zhang, L. Yang, L. Dong, J. Zhang, B. Liu, Measuring software engineer's contribution in practice: An industrial experience report, Journal of Software: Evolution and Process 37 (1) e2722.

[25] Y. Li, H. Zhang, Y. Jin, Z. Ren, L. Dong, J. Lyu, L. Yang, D. Lo, D. Shao, An Explainable Automated Model for Measuring Software Engineer Contribution, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 783–794.

[26] A. MacCormack, C. Kemerer, M. Cusumano, B. Crandall, Trade-offs between productivity and quality in selecting software development practices, IEEE Software 20 (5) (2003) 78–85.

[27] N. Ramasubbu, M. Cataldo, R. K. Balan, J. D. Herbsleb, Configuring global software teams: A multi-company analysis of project productivity, quality, and profits, in: 2011 33rd International Conference on Software Engineering, 33rd International Conference on Software Engineering, 2011, pp. 261–270.

[28] M. Tahaei, K. Vaniea, K. K. Beznosov, M. K. Wolters, Security notifications in static analysis tools: Developers' attitudes, comprehension, and ability to act on them, in: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, CHI Conference on Human Factors in Computing Systems, 2021.

[29] M. Nachtigall, M. Schlichtig, E. Bodden, A large-scale study of usability criteria addressed by static analysis tools, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, p. 532–543.

[30] S. Mehrpour, T. D. LaToza, Can static analysis tools find more defects?, Empirical Software Engineering 28 (1) (2022) 5.

[31] Z. Liu, Y. Tang, X. Luo, Y. Zhou, L. F. Zhang, No need to lift a finger anymore? assessing the quality of code generation by ChatGPT, IEEE Transactions on Software Engineering (2024) 1–35.

[32] K. Petersen, Measuring and predicting software productivity: A systematic map and review, Information and Software Technology 53 (4) (2011) 317–343.

[33] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, D. C. Schmidt, A prompt pattern catalog to enhance prompt engineering with chatgpt, in: Proceedings of the 30th Conference on Pattern Languages of Programs, The Hillside Group, USA, 2023.

[34] AlDanial, Cloc tool, `https://github.com/AlDanial/cloc`, accessed 2024-12-17.

[35] T. McCabe, A complexity measure, IEEE Transactions on Software Engineering SE-2 (4) (1976) 308–320.

[36] M. di Biase, A. Rastogi, M. Bruntink, A. van Deursen, The delta maintainability model: Measuring maintainability of fine-grained code changes, in: 2019 IEEE/ACM International Conference on Technical Debt, International Conference on Technical Debt, 2019, pp. 113–122.

[37] G. K. Gill, C. F. Kemerer, Productivity impacts of software complexity and developer experience (1990).

[38] N. Nan, D. E. Harter, Impact of budget and schedule pressure on software development cycle time and effort, IEEE Transactions on Software Engineering 35 (5) (2009) 624–637.

[39] H.-M. Chen, B.-A. Nguyen, C.-R. Dow, Code-quality evaluation scheme for assessment of student contributions to programming projects, Journal of Systems and Software 188 (2022) 111273.

[40] K. Kopec-Harding, S. Eraslan, B. Cai, S. M. Embury, C. Jay, The impact of unequal contributions in student software engineering team projects, Journal of Systems and Software 206 (2023) 111839.

[41] SonarQube, Clean as you code, `https://docs.sonarsource.com/sonarqube/9.9/user-guide/clean-as-you-code`, accessed 2025-05-11.

[42] M. Borg, O. Svensson, K. Berg, D. Hansson, SZZ unleashed: An open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project, in: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, 2019, p. 7–12.

[43] S. Guo, D. Li, L. Huang, S. Lv, R. Chen, H. Li, X. Li, H. Jiang, Estimating uncertainty in labeled changes by SZZ tools on just-in-time defect prediction, ACM Transactions on Software Engineering and Methodology 33 (4) (apr 2024).

[44] C. Williams, J. Spacco, SZZ revisited: Verifying when changes induce fixes, in: Proceedings of the 2008 Workshop on Defects in Large Software Systems, Workshop on Defects in Large Software Systems, 2008, p. 32–36.

[45] M. Javidmehr, M. Ebrahimpour, Performance appraisal bias and errors: The influences and consequences, International Journal of Organizational Leadership 4 (2015) 286–302.

[46] S. Evans, D. Tourish, Agency theory and performance appraisal: How bad theory damages learning and contributes to bad management practice, Management Learning 48 (3) (2017) 271–291.

[47] B. Cartaxo, G. Pinto, B. Fonseca, M. Ribeiro, P. Pinheiro, M. T. Baldassarre, S. Soares, Software engineering research community viewpoints on rapid reviews, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, International Symposium on Empirical Software Engineering and Measurement, 2019, pp. 1–12.

[48] B. Cartaxo, G. Pinto, S. Soares, The role of rapid reviews in supporting decision-making in software engineering practice, in: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, 22nd International Conference on Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA, 2018, p. 24–34.

[49] B. Cartaxo, G. Pinto, S. Soares, Rapid Reviews in Software Engineering, Springer International Publishing, Cham, 2020, pp. 357–384.

[50] S. Rico, N. B. Ali, E. Engström, M. Höst, Experiences from conducting rapid reviews in collaboration with practitioners — two industrial cases, Information and Software Technology 167 (2024) 107364.

[51] S. Matalonga, D. Amalfitano, A. Doreste, A. R. Fasolino, G. H. Travassos, Alternatives for testing of context-aware software systems in non-academic settings: results from a rapid review, Inf. Softw. Technol. 149 (C) (Sep. 2022). doi:10.1016/j.infsof.2022.106937.

[52] F. Ponce, G. Márquez, H. Astudillo, Migrating from monolithic architecture to microservices: A rapid review, in: 2019 38th International Conference of the Chilean Computer Science Society, 2019, pp. 1–7. doi:10.1109/SCCC49216.2019.8966423.

[53] A. Mockus, P. C. Rigby, R. Abreu, P. Suresh, Y. Chen, N. Nagappan, Modeling the centrality of developer output with software supply chains, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, p. 1809–1819.

[54] L. Cheng, E. Murphy-Hill, M. Canning, C. Jaspan, C. Green, A. Knight, N. Zhang, E. Kammer, What improves developer productivity at google? code quality, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, p. 1302–1313.

[55] M. Daun, J. Brings, P. A. Obe, V. Stenkova, Reliability of self-rated experience and confidence as predictors for students' performance in software engineering: Results from multiple controlled experiments on model comprehension with graduate and undergraduate students, Empirical Software Engineering 26 (4) (jul 2021).

[56] S. Hamer, C. Quesada-López, A. Martínez, M. Jenkins, Using git metrics to measure students' and teams' code contributions in software development projects, CLEI Electronic Journal 24 (2) (2021) 8:1–29.

[57] S. A. Licorish, D. A. da Costa, E. Zolduoarrati, N. Grattan, Relating team atmosphere and group dynamics to student software development teams' performance, Information and Software Technology 167 (2024) 107377.

[58] V. Piantadosi, S. Scalabrino, A. Serebrenik, N. Novielli, R. Oliveto, Do attention and memory explain the performance of software developers?, Empirical Software Engineering 28 (5) (aug 2023).

[59] O. H. Plant, J. van Hillegersberg, A. Aldea, Towards a contextual model of software engineering capabilities: Factors affecting team performance, in: 2023 IEEE 25th Conference on Business Informatics (CBI), 25th Conference on Business Informatics, 2023, pp. 1–10.

[60] H. Shinbori, M. Tsunoda, How does grit affect the performance of software developers?, in: 2022 29th Asia-Pacific Software Engineering Conference, 29th Asia-Pacific Software Engineering Conference, 2022, pp. 568–569.

[61] P. R. de Bassi, G. M. P. Wanderley, P. H. Banali, E. C. Paraiso, Measuring developers' contribution in source code using quality metrics, in: 2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design, 22nd International Conference on Computer Supported Cooperative Work in Design, 2018, pp. 39–44.

[62] V. K. Sihombing, M. E. Simaremare, D. J. Samosir, V. O. Limbong, Improving prometer, a measure for programmer performance, in: 2022 IEEE International Conference of Computer Science and Information Technology, IEEE International Conference of Computer Science and Information Technology (ICOSNIKOM), 2022, pp. 01–06.

[63] Merico, Eloc, `https://www.merico.cn/`, accessed 2025-05-11.

[64] H. Kirinuki, Y. Higo, K. Hotta, S. Kusumoto, Hey! are you committing tangled changes?, in: Proceedings of the 22nd International Conference on Program Comprehension, 2014, pp. 262–265.

[65] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi, et al., A fine-grained data set and analysis of tangling in bug fixing commits, Empirical Software Engineering 27 (6) (2022) 125.

[66] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, M. Di Penta, Gptsniffer: A codebert-based classifier to detect source code written by chatgpt, Journal of Systems and Software 214 (C) (Jul. 2024).

[67] Y. Shi, H. Zhang, C. Wan, X. Gu, Between lines of code: Unraveling the distinct patterns of machine and human programmers, in: 2025 IEEE/ACM 47th International Conference on Software Engineering, 2025, pp. 51–62.

[68] K. Herzig, S. Just, A. Zeller, The impact of tangled code changes on defect prediction models, Empirical Software Engineering 21 (2016) 303–336.

[69] H. Kirinuki, Y. Higo, K. Hotta, S. Kusumoto, Splitting commits via past code changes, in: 2016 23rd Asia-Pacific Software Engineering Conference, IEEE, 2016, pp. 129–136.

[70] K. Agrawal, S. Amreen, A. Mockus, Commit quality in five high performance computing projects, in: 2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science, IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science, IEEE, 2015, pp. 24–29.

[71] M. Kondo, D. M. German, Y. Kamei, N. Ubayashi, O. Mizuno, An empirical study of token-based micro commits, Empirical Software Engineering 29 (6) (2024) 148.

[72] N. B. Lindström, D. Koutsikouri, M. Staron, W. Meding, O. Söder, Understanding metrics team-stakeholder communication in agile metrics service delivery, in: 2021 28th Asia-Pacific Software Engineering Conference, IEEE, 2021, pp. 401–409.

[73] M. Staron, Critical role of measures in decision processes: Managerial and technical measures in the context of large software development organizations, Information and Software Technology 54 (8) (2012) 887–899.

[74] W. Meding, M. Staron, O. Söder, Meteam—a method for characterizing mature software metrics teams, Journal of Systems and Software 180 (2021) 111006.

[75] W. Tong, T. Zhang, CodeJudge: Evaluating code generation with large language models, in: Y. Al-Onaizan, M. Bansal, Y.-N. Chen (Eds.), Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, 2024, pp. 20032–20051.

[76] G. Crupi, R. Tufano, A. Velasco, A. Mastropaolo, D. Poshyvanyk, G. Bavota, On the effectiveness of llm-as-a-judge for code generation and summarization, IEEE Transactions on Software Engineering (2025).

[77] R. Wang, J. Guo, C. Gao, G. Fan, C. Y. Chong, X. Xia, Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering, Proceedings of the ACM on Software Engineering 2 (ISSTA) (2025) 1955–1977.

[78] M. Staron, W. Meding, C. Nilsson, A framework for developing measurement systems and its industrial evaluation, Information and Software Technology 51 (4) (2009) 721–737.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: