Towards a Multidimensional Diagnostic Framework for Automated Test Quality Assessment

Victor Alves
University of São Paulo (USP)
São Carlos, São Paulo, Brazil
victorpa@usp.br

Carla Bezerra
Federal University of Ceará (UFC)
Quixadá, Ceará, Brazil
carlailane@ufc.br

Marcio Delamaro
University of São Paulo (USP)
São Carlos, São Paulo, Brazil
delamaro@icmc.usp.br

Abstract

Automatic generation of software tests faces challenges related to the limitations of traditional metrics in accurately predicting the actual effectiveness of tests in detecting defects. As a step toward overcoming this challenge, this work introduces the foundations of the Test Quality Score (TQS), a multidimensional metric that quantifies the quality of synthetic tests by balancing a Potential Vector and a Noise Vector. The fundamental premise of the metric is to quantify the multidimensional nature of testing by consolidating quality attributes already established in the literature. To assess the practical applicability of the proposed metric, we conducted a preliminary empirical study by implementing the TQS in a Python environment and applying it to unit tests generated by Large Language Models (LLMs) under different prompting strategies. For empirical validation, the Mutation Score (MS) was established as the external comparison variable to measure the actual defect detection capability of the tests. The results demonstrated a positive correlation between TQS and MS (Pearson r=0.476,p<0.01), while the ROC/AUC analysis (AUC=0.666) indicated partial alignment with mutation-based effectiveness. The observed divergence between TQS and MS revealed that the framework captures additional structural quality dimensions not addressed by mutation analysis, including deficiencies identified in tests with high MS. These findings provide preliminary evidence that TQS can complement mutation-based assessment through a multidimensional diagnostic perspective. We conclude that TQS establishes a promising and extensible foundational heuristic for future research on scalable automated test quality evaluation.

Keywords

Automated Test Generation, Test Quality Assessment, Software Metrics

1 Introduction

The automatic generation of unit tests has proven to be a disruptive innovation in software development, offering the potential to reduce costs and accelerate development cycles [6, 36, 41]. Thus, this process addresses a major challenge in the software testing landscape, as writing tests is often time-consuming, labor-intensive, and expensive [22]. Consequently, test generation based on Artificial Intelligence (AI) has become increasingly prevalent in the daily work of professionals [8, 35, 50]. The context of large-scale test generation, especially through AI, has been extensively explored in the current literature, demonstrating that this approach is effective and promising for modern software engineering [7, 30, 38].

However, given the large volume of automatically generated test samples and data, one of the main challenges identified is measuring the quality of these tests. Several studies delve into the investigation of the structure of tests generated by LLMs [2, 39], while others analyze the quality of tests produced by traditional tools [21, 44]. Given the growing use of tools to automate unit test generation, it is essential to develop methods and approaches to evaluate the quality of the generated code.

Current research has explored the automatic generation of tests from a quality perspective, addressing different dimensions in isolation, such as the detection of test smells [3, 44], coverage metrics [37], indicators such as Code Correctness (CC) and Mutation Score (MS) [13], as well as the analysis of edge cases [9, 49]. Although these studies acknowledge the importance of quality in automatically generated tests, most offer a fragmented view of the effectiveness of the artifacts by focusing on only one dimension. However, the literature indicates that test quality is a multidimensional concept, involving aspects such as maintainability, robustness, scenario diversity, and functional coverage [16, 24]. In this context, and given the growing adoption of AI-based tools, it is essential to integrate these multiple criteria into a unified evaluation approach to bridge the gap in more systemic analyses of test quality in large scale.

This study proposes a first step toward a multidimensional and automated assessment of test quality. In this context, we present Test Quality Score (TQS), a composite metric designed to assess the quality of unit tests by balancing a Potential Vector (contributions from positive attributes) and a Noise Vector (penalties for negative attributes). To empirically validate the TQS, we implemented the metric calculation in a Python project and conducted an experimental study evaluating test suites generated by different Large Language Models (LLMs) under multiple prompting strategies. By adopting the Mutation Score as a reference variable for fault detection capability, our analysis demonstrates a statistically significant positive correlation between TQS and MS, as well as TQS's ability to effectively distinguish quality attributes that are not evident in the MS analysis, thereby serving as a complementary metric.

2 Background And Related Work

2.1 Unit Testing & Test Code Quality

Unit testing is a technique designed to validate the functionality of the smallest testable parts of software (modules, objects, and classes), which can be examined in isolation [15]. This practice aids in preventing programming errors and identifying issues early in the development cycle [20, 32].

The quality of test code has been assessed through various metrics in the literature. One common approach involves the analysis of test coverage, which serves as a key indicator for identifying the extent to which a test suite exercises the relevant portions of the

codebase [10, 14]. Complementing this perspective, other studies focus on assessing the structural quality of test code by detecting test smells [28]. Test smells typically originate from suboptimal design decisions made during test implementation and can significantly hinder the effectiveness of testing activities [31, 41, 43].

Another test quality concept frequently addressed in the literature is Edge Cases, situations in which tests validate the limits of a given scenario. These cases are commonly considered in research that uses AI tools for automatic test generation to analyze the quality of assertions and the coverage of expectations [9, 49]. Finally, several studies focus on evaluating the test in the context of mutation testing, using Mutation Scores as a metric. Mutational analysis evaluates the effectiveness of a suite of tests by assessing their ability to detect intentional syntactic changes (mutations) injected into the source code [4, 5, 19].

2.2 Automatic Test Generation

The automated generation of unit tests streamlines the creation and execution of many inputs aimed at thoroughly exercising software components [47]. Empirical studies on open-source projects have shown that such tools can effectively uncover software defects [1]. Tools like EvoSuite[1] [11] and Randoop[2] are well-established in the Java ecosystem, producing test cases in the JUnit framework. In other programming environments, alternatives such as PYNGUIN[3] [27], designed for Python, and JSEFT[4] [29], which applies function coverage and abstraction-based algorithms for JavaScript, have gained visibility. Despite their potential, the adoption of automated unit testing tools in industry remains limited, primarily because interpreting and validating the generated test cases requires significant effort [12].

LLMs have assumed a significant role in test generation driven by Natural Language Processing (NLP) [48]. LLMs can generate code snippets, documentation, and even fix bugs by training on human-language inputs. To achieve this, one must strategically design task-specific instructions, known as prompts, guiding the model's output without changing the settings [25]. The literature has extensively examined prompts in LLMs-driven tasks, emphasizing that the output quality is closely tied to the clarity and specificity of the instructions provided [23, 26].

2.3 Related Work

Prior research on automated test generation has assessed artifact quality through isolated indicators, such as code coverage, MS, correctness, and test smell detection [2, 13, 37]. Studies involving traditional generators, such as EvoSuite, as well as recent evaluations of LLM-generated tests, consistently analyze these dimensions separately [33, 44], often emphasizing coverage or MS as proxies for effectiveness. However, these isolated perspectives cannot automatically capture structural degradation, semantic robustness, and technical debt all at once within a single evaluation task. The literature recognizes that test quality is inherently multidimensional, encompassing maintainability, assertion expressiveness, scenario diversity, and fault revelation capability [16, 24].

Composite indicators have been explored in software engineering, particularly for maintainability and technical debt assessment. Prior studies report the use of aggregated models, such as the Maintainability Index, and comparative frameworks such as SIG and SQALE, which combine multiple software attributes into unified quality indicators [17, 40]. However, similar multidimensional approaches remain largely unexplored for assessing the quality of automatically generated unit tests. Thus, no previous study has proposed a diagnostic framework or an indicator that explicitly breaks down the quality of the tests generated into constructive and detrimental dimensions, integrating both verifiability and structural degradation into a single evaluation model.

3 Test Quality Score

Assessing the quality of unit tests is a multifaceted problem that requires integrating multiple dimensions to accurately reflect test effectiveness. Grounded in the Design Science Research (DSR) paradigm [46], we propose the Test Quality Score (TQS) as a diagnostic artifact rather than a mere composite metric. TQS models test quality as an interpretable balance between constructive and detrimental attributes, using two inversely related vectors.

3.1 Mathematical Formalization

TQS is defined as the difference between two vectors (see Equation 1), allowing the metric to identify the source of low scores.

$$TQS = V_P - V_N \tag{1}$$

Where:

V_P **Potential Vector:** Represents the weighted sum of attributes that add value to the test. Based on this premise, V_P quantifies the positive constructs of the artifact, rewarding characteristics that demonstrate analytical rigor. Given P as the set of n observable positive attributes p_i, and ω_i as the weight or scaling factor associated with each attribute, the vector is formalized in Equation 2.

$$V_P = \sum_{i=1}^{n} (w_i \cdot p_i) \tag{2}$$

V_N **Noise Vector:** Represents the weighted sum of penalties for failures and negative attributes in the test. V_N applies penalties proportional to the incidence of harmful attributes that inflate the technical debt and compromise reliability. Equation 3 presents the calculation of V_N, given N be the set of m negative attributes n_j, each weighted by a penalty γ_j. The vector is defined by the magnitude of this degradation.

$$V_N = \sum_{j=1}^{m} \gamma_j \cdot n_j \tag{3}$$

TQS was designed to be extensible, allowing researchers and engineers to calibrate the vector weights (ω, γ) or incorporate new quality attributes (p, n) according to the specific needs of the domain or the target programming language.

[1] https://www.evosuite.org/
[2] https://randoop.github.io/randoop/
[3] https://www.pynguin.eu/
[4] https://github.com/saltlab/JSeft

4 Empirical Study Design

This study evaluates an initial heuristic instantiation of the TQS architecture for exploratory validation. Mutation Score (MS) was adopted as the control dependent variable because it is one of the most accurate measures of test suite defect-detection capability [5, 19]. Therefore, the study focused on (i) defining the research questions, (ii) constructing the experimental environment, and (iii) running the environment and defining the analysis methods.

4.1 Research Questions (RQs)

RQ_1: ***Is there a statistically significant correlation between TQS and the MS across different models and prompt strategies?*** The purpose of this *RQ* is to assess the statistical stability of the metric, verifying whether the predictive relationship between TQS and MS is consistent.

RQ_2: ***Can TQS complement MS in automated test assessment?*** This *RQ* evaluates whether TQS provides quality information beyond fault-detection effectiveness measured by MS.

RQ_3: ***Can TQS diagnose issues in tests that MS doesn't detect?*** The goal of this question is to demonstrate the diagnostic value of the TQS by investigating its ability to identify and penalize structural deficiencies in tests that the MS fails to detect.

4.2 Experimental Setup And Execution

To address the RQs defined for this study, we implemented the TQS in an automated experimental environment for subsequent data analysis. Figure 1 provides a visual illustration of the experiment's execution flow.

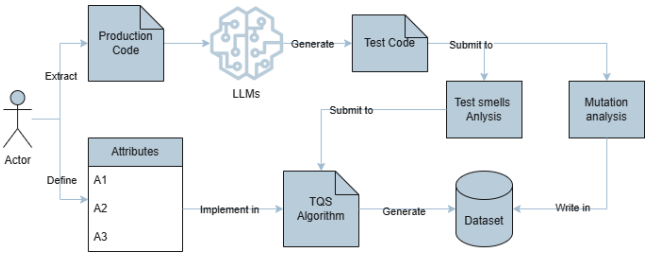

Figure 1: Study Steps

4.2.1 TQS Instantiation For This Study. Before beginning implementation, we selected the attributes (p, n) and their corresponding weights (ω, γ) for each vector in TQS. For this study, the attributes were chosen based on the concepts most commonly found in current research. The weighting configuration adopted in this study should be interpreted as an initial heuristic operationalization intended to instantiate the framework for exploratory validation, rather than as an optimized or universally generalizable calibration. Table 1 presents each attribute, the vector (V_P or V_N) in which it is assigned, and the respective weights assigned to it.

FC represents the function coverage ratio. Scaling rules: Line coverage (A_2) adds +0.25 for each quartile achieved. Assertions (A_4) add between +0.07 and +0.30 depending on the variety of types detected. We explain the rationale behind the weighting of each attribute below:

Table 1: TQS attributes and weights selected for this study

ID	Attribute (p, n)	Vector	Weight (ω, γ)
A_1	Edge Cases	V_P	+1.5
A_2	Code Coverage	V_P	Up to +1.0 (Scaled)*
A_3	Functional Coverage	V_P	$+1.0 \times FC$
A_4	Assertion Diversity	V_P	Up to +1.0 (Scaled)**
A_5	Syntax Error	V_N	-1.0
A_6	Test Smells	V_N	-0.5 (per type)

- **Foundations of Potential Vector (V_P):** Code coverage (A_2) and functional coverage (A_3) are traditionally adopted by the literature and industry as primary metrics for the effectiveness of a test suite. However, the literature warns that high coverage alone does not guarantee the detection of defects if the checks are superficial [18, 34]. For this reason, V_P incorporates semantic attributes, such as assertion diversity (A_4) and the presence of edge cases (A_1).
- **Foundations of Noise Vector (V_N):** The software maintenance studies demonstrates that test smells (A_6) impair the readability and maintainability of the project [31, 42]. Additionally, attributes such as syntactic errors (A_5) represent an immediate breakdown of automation. In the context of automatically generated tests, an artifact that fails prematurely negates the main benefit of generation: the reduction of manual effort.

It is important to note that these foundations were established solely for this study. TQS architecture is scalable and allows for the inclusion of any attribute and weights in the vectors that is relevant to the context of the domain being evaluated.

4.2.2 Configuring Generative Models. To generate the tests to be evaluated by TQS, we utilize LLMs with varying levels of specialization: Meta's LLaMA 3.3 and CodeLLaMA, Google's CodeGemma[5], and Microsoft's WizardCoder[6]. A sample of code to be submitted to the LLMs was selected. To do this, we extracted a sample from HumanEval[7] dataset. The sample code was submitted to the LLMs with three prompting strategies (zero-shot, few-shot, and Chain-of-Thought), which were then used to generate the tests suites.

4.2.3 Environment And Tools. After the LLMs generate the test suites, we developed a Python pipeline to first collect the necessary quality metrics and their values. The first approach operates as a processing algorithm that executes the following sequential steps:

1 **Static Analysis & Parsing**: it uses the *ast*[8] (Abstract Syntax Tree) library to decompose the test suite structure and identify syntax errors (A_5).
2 **Dynamic Execution:** the algorithm integrates with the coverage.py[9] tool to run the test suite on the production code and collect data for V_P, such as row coverage (A_2).

[5] https://ai.google.dev/gemma/docs/codegemma
[6] https://www.microsoft.com/en-us/research/publication/wizardcoder-empowering-code-large-language-models-with-evol-instruct/
[7] https://github.com/openai/human-eval
[8] https://docs.python.org/3/library/ast.html
[9] https://pypi.org/project/coverage/

3 **Attribute Extraction:** an automated rules engine also using *ast* extracts multidimensional attributes, including the diversity of assertions (A_4), such as *AssertEquals, AssertTrue*, etc., and the presence of edge cases (A_1), mapping whether the test inserts extreme values (such as zero, null, empty, positive and negative values). The analysis of test smells (A_6) was performed separately from the execution pipeline using the PYNose [45] tool for each test suite.

4.2.4 Code Processing And Extraction. Each test suite and its corresponding attributes were stored in a dataset. The TQS algorithm then read the attribute values, applied the predefined weights (Table 1), and computed the final score for each test suite.

4.2.5 Mutation Analysis. After running the algorithm and generating the dataset, we began the TQS validation phase. To mitigate the self-contained metric bias, the test verification effectiveness was calibrated against mutant analysis for evaluating test oracles. We used the *pyproteum*[10] library for each test suite that passed the baseline execution (*status = passed*).

4.3 Data Analysis

4.3.1 Data Sample. The corpus used in this study consists of 52 test suites automatically generated by LLMs, totaling 203 test cases distributed across 17 programming problems extracted from the HumanEval. We considered only those tests that executed successfully without errors to be valid outputs (a prerequisite for MS analysis). CodeGemma was the most represented model (16 suites and 71 test cases), while WizardCoder produced the smallest volume (18 cases across 7 suites). CodeLLaMa generated 11 suites with 23 cases. LLaMA3, although present in only 7 suites, had the highest average per file (6.0 test cases). After identifying the quality attributes and calculating TQS and MS, we obtained a distribution of the attributes across the test suites, as shown in Figure 2.

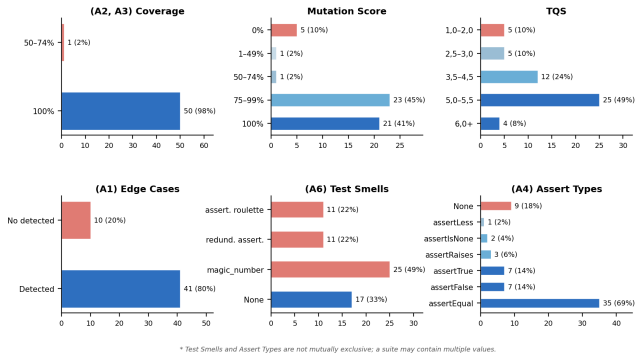

Figure 2: Distribution of the corpus

4.3.2 Evaluated Metrics And Variables. This study investigates TQS as the primary independent variable, along with its constituent components (V_P and V_N) which quantify the structural and semantic characteristics of the generated test suites. To empirically validate the proposed heuristic, MS has been defined as the dependent variable, serving as a standard reference for the actual fault-detection capability of the generated tests. Table 2 presents a detailed description and the role of each variable collected during the experiment. For each evaluation metric used, we define its purpose and the RQ in which it was used.

Table 2: TQS Evaluation Metrics

Metric	Goal	*RQ*
Pearson r	Determine whether an increase in TQS corresponds to a proportional increase in MS	RQ_1
Spearman ρ	It measures monotonic agreement using a nonparametric method and is resistant to asymmetric distributions of MS	RQ_1
Kendall τ	Measures the ratio of concordant pairs to discordant pairs in the joint TQS and MS ranking	RQ_1
AUC-ROC	Evaluates how TQS aligns with MS while preserving sensitivity to additional structural quality dimensions across different thresholds.	RQ_2
Percentage analysis	Calculates the prevalence of each structural defect that TQS detects but MS ignores	RQ_3

5 Results

5.1 RQ_1: Correlation And Statistical Stability (TQS vs. MS)

To answer RQ_1, we calculated the TQS and MS for each valid test suite, and then subjected the data to a statistical correlation analysis. Figure 3 presents a statistical correlation graph between the MS values and the TQS. The maximum values for each variable were 7 for the maximum TQS and 1 for the maximum MS. The figure also shows the number of assertions in each test file using a heat map ranging from 1 to 3 assertions.

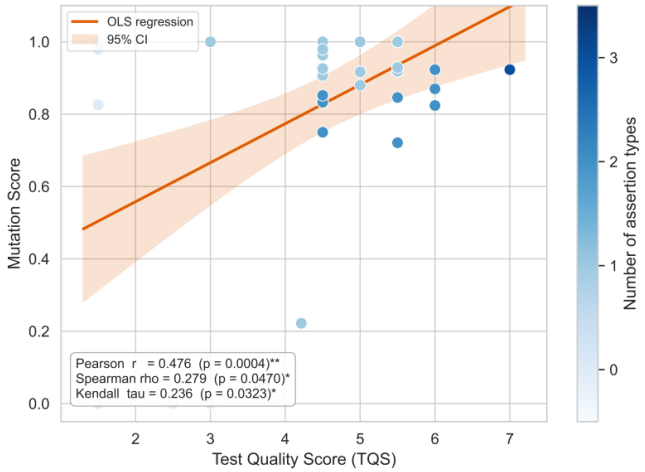

Figure 3: Correlation between TQS and MS

Analysis of the graph in Figure 3 reveals a positive correlation between TQS and MS. OLS regression illustrates the trend that increases in TQS are associated with greater effectiveness in fault detection (higher MS). This relationship is supported by the metrics calculated for this context: Pearson's correlation coefficient

[10] https://pypi.org/project/pyproteum/

indicates a moderate linear correlation (r = 0.476) with high significance (p = 0.0004), while the nonparametric Spearman (ρ = 0.279, p = 0.0470) and Kendall (τ = 0.236, p = 0.0323) tests confirm the consistency of this monotonic association. The 95% confidence interval (CI) band indicates greater predictive accuracy of the model in the central region of the axis (between TQS values of 4 and 6) where the highest density of samples is concentrated.

Answer to RQ_1: Based on the correlation analysis presented, TQS showed a positive and statistically significant correlation with MS. This means that, within the corpus of generated test suites, most tests with high MS also achieved high TQS.

5.2 RQ_2: TQS Complementing MS

RQ_2 investigates whether the TQS provides additional information about quality beyond the mutation score. Rather than treating the mutation score as an absolute measure of test quality, this analysis assesses whether the TQS captures additional dimensions related to structure and maintainability that are not reflected in the mutation analysis. Figure 4 presents the ROC analysis comparing TQS against the binary classification of tests suites with high mutation effectiveness (MS ≥ 0.90).

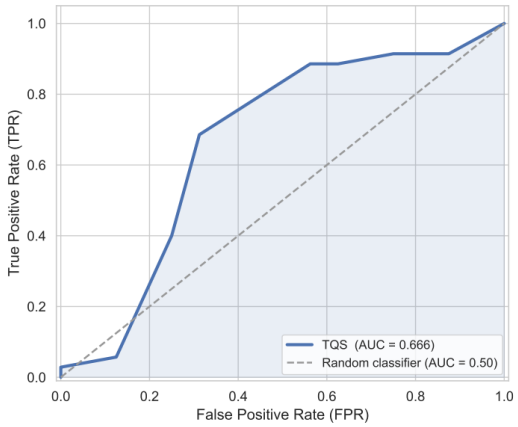

Figure 4: TQS vs. MS ≥ 0.90

Based on an analysis of Figure 4 the moderate AUC (0.666) value is consistent with the multidimensional nature of the TQS. If the model were exclusively aligned with mutation-based efficacy, the ROC curve would be close to the upper-left corner, yielding an AUC closer to 1.0. Instead, the observed overlap between positive and negative classifications indicates that the TQS does not strictly reproduce the behavior of the mutation score. This behavior is expected because, unlike mutation analysis, the TQS also incorporates structural characteristics and those related to ease of maintenance into its assessment. As a result, some tests with high MS received lower TQS values due to deficiencies outside the scope of mutation analysis, causing a partial overlap in the ROC space and consequently reducing the AUC. These discrepancies are investigated further in RQ_3.

Answer to RQ_2: Overall, the results indicate that TQS complements mutation scoring by providing diagnostic information that goes beyond the mere effectiveness of fault detection. This is reflected in the ROC and AUC analyses, in which the moderate overlap between classes suggests that TQS captures additional structural dimensions not addressed by mutation analysis.

5.3 RQ_3: TQS Diagnostic Value

To answer RQ3, we analyzed the test suites that achieved a high MS (≥ 0.90) in order to identify which quality attributes (see Table 2) lowered their TQS scores. Figure 5 shows the percentage analysis results in two views.

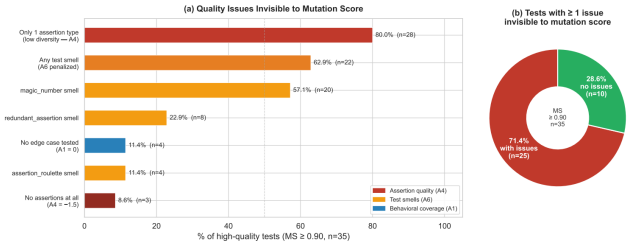

Figure 5: Quality issues detected in tests with MS ≥ 0.90

Based on the analysis in Figure 5, the proportion chart (b) reveals a limitation in the exclusive use of MS: the vast majority of tests with high MS mask structural or design problems. Specifically, 71.4% of tests with high MS exhibit at least one quality issue that is essentially invisible to mutant analysis. The bar chart (a) details the nature and frequency of these deficiencies, categorizing them by each attribute configured in TQS in Table 2. The most prevalent flaw relates to low verification diversity (A_4): 80.0% of the tests in the sample rely on only one type of assertion. Furthermore, the incidence of test smells (A_6) affects 62.9% of cases, with Magic Number being the most common (57.1%), followed by Redundant Assertion (22.9%) and Assertion Roulette (11.4%). Even more critically, the survey indicates that severe omissions in test design, such as the total absence of tests for edge cases (A_1) (11.4%) and even tests lacking any assertions (8.6%).

In this way, it can be inferred that TQS was able to diagnose problems that fall outside the scope of MS, penalizing the cases evaluated for their respective quality flaws. It is worth noting that the low diversity of assertions (A_4) may stem from the very nature of production code, in scenarios where a single type of validation is already sufficient. However, the detection of test smells (A_6) and the lack of coverage of edge cases (A_1) demonstrate that a suite with a high MS score can still harbor structural problems, which compromise its long-term maintainability and reliability. Thus, the findings demonstrate that TQS is effective in accurately reflecting and diagnosing the quality attributes for which it was designed.

Answer to RQ_3: TQS was able to detect and penalize quality issues in tests with high MS, which explains why some tests with high MS received lower TQS scores. Some of the issues detected and penalized included test smells and the absence of edge cases.

6 Threats To Validity

- **Internal Validity:** The study relied on third-party tools (python libraries and test smell detection tools) and custom scripts to orchestrate data extraction and the calculation of

the TQS. Defects or limitations in these tools could introduce noise into the data. To mitigate this risk, we used frameworks established by academia and industry and performed manual validations on a sample of the generated and evaluated tests.

- **External Validity:** The experiment was conducted on a specific set of code and within a particular programming language paradigm (Python); the results observed may not be replicated identically in other languages, application domains, or legacy systems with different architectural characteristics. Future studies will be conducted using a larger data sample and across other domains and languages.
- **Construct Validity:** The use of MS as a reference variable to represent the effectiveness of test cases yielded satisfactory results. However, MS has limitations, such as the problem of equivalent mutants, which can inflate scores. To mitigate this risk, we adopted an effectiveness threshold (MS ≥ 0.90) to classify the tests. Furthermore, the weights assigned to the TQS factors represent heuristic choices grounded in the literature. However, variations in these weights could alter the results. The intentional use of heuristic weights in this initial study enables future research directions involving empirical calibration, and domain-specific optimization strategies.

7 Conclusion And Future Works

This work represents a first step toward a multidimensional and automated assessment of test quality. To this end, we propose a method for constructing the Test Quality Score (TQS), a multidimensional metric designed to evaluate the quality of unit tests generated at scale. We propose a mathematical calculation of the metric based on positive and negative vectors (V_P and V_N) and implement the calculation in a Python algorithm. We conducted an empirical study involving unit tests suites generated by LLMs in different contexts and propose a validation using Mutation Score (MS) as the dependent variable.

The results suggest that TQS has a statistically significant correlation with MS and can support automated assessment by identifying structural deficiencies and non-executable generated tests. The main distinguishing feature of this indicator lies in its diagnostic power: TQS was able to identify and penalize structural deficiencies, such as the presence of test smells and the absence of edge cases, in tests that mutation analysis would classify as ideal.

In future studies, we plan to expand the empirical validation of the metric to new domains, as well as explore the dynamic calibration of model weights (ω, γ) tailored to industry-specific requirements. Furthermore, we intend to integrate TQS as the central reward function in an architecture based on autonomous agents for quality analysis and unit test optimization.

Artifact Availability

The datasets, environment, and algorithms developed for this study are publicly available under an open-source (MIT) license at: https://doi.org/10.5281/zenodo.21498424

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. The authors thank the support of CNPq, under process number 406941/2025-4 and 403304/2025-3.

References

[1] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. In *An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application*. 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[2] Victor Alves, Carla Bezerra, Ivan Machado, Larissa Rocha, Tassio Virgínio, and Publio Silva. 2025. Quality Assessment of Python Tests Generated by Large Language Models. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE '25)*. Association for Computing Machinery, New York, NY, USA, 394–405. https://doi.org/10.1145/3756681.3756964

[3] Victor Alves, Cristiano Santos, Carla Bezerra, and Ivan Machado. 2024. Detecting Test Smells in Python Test Code Generated by LLM: An Empirical Study with GitHub Copilot. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 581–587. https://doi.org/10.5753/sbes.2024.3561

[4] Samuel Amorim, Leo Fernandes, Márcio Ribeiro, Rohit Gheyi, Marcio Delamaro, Marcio Guimarães, and André Santos. 2024. Reducing Manual Efforts in Equivalence Analysis in Mutation Testing. *Journal of Software Engineering Research and Development* 12, 1 (Mar. 2024), 3:1 – 3:17. https://doi.org/10.5753/jserd.2024.3588

[5] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 402–411. https://doi.org/10.1145/1062455.1062530

[6] Luciano Baresi and Matteo Miraz. 2010. TestFul: automatic unit-test generation for Java classes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 281–284. https://doi.org/10.1145/1810295.1810353

[7] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. In *Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation* (, Toronto ON, Canada,) *(SIGCSE 2023)*. Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, 500–506. https://doi.org/10.1145/3545945.3569759

[8] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit Test Generation using Generative AI: A Comparative Performance Analysis of Autogeneration Tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code* (Lisbon, Portugal) *(LLM4Code '24)*. Association for Computing Machinery, New York, NY, USA, 54–61. https://doi.org/10.1145/3643795.3648396

[9] Gabriel Darbord, Anne Etien, Nicolas Anquetil, Benoît Verhaeghe, and Mustapha Derras. 2023. A Unit Test Metamodel for Test Generation. In *CEUR Workshop Proceedings*. Stephane Ducasse and Gordana Rakic, Lyon, France. https://hal.science/hal-04219649

[10] Kaj Dreef, Vijay Krishna Palepu, and James A. Jones. 2023. Exploring granular test coverage and its evolution with matrix visualizations. *Information and Software Technology* 155 (2023), 107085. https://doi.org/10.1016/j.infsof.2022.107085

[11] Gordon Fraser and Andrea Arcuri. 2011. In *EvoSuite: automatic test suite generation for object-oriented software* (Szeged, Hungary) *(ESEC/FSE '11)*. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 416–419. https://doi.org/10.1145/2025113.2025179

[12] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. In *Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study*, Vol. 24. ACM Trans. Softw. Eng. Methodol., Article 23, 49 pages. https://doi.org/10.1145/2699688

[13] Yulong Fu and Yingtong Zhang. 2025. UAgent: Adversarial Co-evolution for Targeted Bug Revelation in Unit Testing. In *Proceedings of the 2025 Workshop on Recent Advances in Resilient and Trustworthy MAchine Learning-DriveN Systems (ARTMAN '25)*. Association for Computing Machinery, New York, NY, USA, 51–56. https://doi.org/10.1145/3733821.3763027

[14] Pouria Golshanrad and Fathiyeh Faghih. 2024. DeepCover: Advancing RNN test coverage and online error prediction using state machine extraction. *Journal of Systems and Software* 211 (2024), 111987. https://doi.org/10.1016/j.jss.2024.111987

[15] D. Graham, R. Black, and E. van Veenendaal. 2021. *Foundations of Software Testing ISTQB Certification, 4th edition*. Cengage Learning. https://books.google.com.br/books?id=mOwxEAAAQBAJ

[16] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. 2020. Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 336–347. https://doi.org/10.1109/ICSME46990.2020.00040

[17] Tjaša Heričko and Boštjan Šumak. 2023. Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems. *Applied Sciences* 13, 5 (2023). https://doi.org/10.3390/app13052972

[18] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[19] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[20] V. Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns: Effective testing styles, patterns, and reliable automation for unit testing, mocking, and integration testing with examples in C#*. Manning. https://books.google.com.br/books?id=CbvZyAEACAAJ

[21] Jeshua S. Kracht, Jacob Z. Petrovic, and Kristen R. Walcott-Justice. 2014. Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites. In *2014 14th International Conference on Quality Software*. 256–265. https://doi.org/10.1109/QSIC.2014.33

[22] Chun Li. 2022. In *Mobile GUI test script generation from natural language descriptions using pre-trained model* (Pittsburgh, Pennsylvania) *(MOBILESoft '22)*. Association for Computing Machinery, New York, NY, USA, 112–113. https://doi.org/10.1145/3524613.3527809

[23] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth* 1, 1 (2024), 9. https://doi.org/10.1007/s44336-024-00009-2

[24] Bin Lin, Csaba Nagy, Gabriele Bavota, Andrian Marcus, and Michele Lanza. 2019. On the Quality of Identifiers in Test Code. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 204–215. https://doi.org/10.1109/SCAM.2019.00031

[25] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (jan 2023), 35 pages. https://doi.org/10.1145/3560815

[26] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE] https://arxiv.org/abs/2102.04664

[27] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: automated unit test generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 168–172. https://doi.org/10.1145/3510454.3516829

[28] Luana Martins, Heitor Costa, and Ivan Machado. 2024. On the diffusion of test smells and their relationship with test code quality of Java projects. *Journal of Software: Evolution and Process* 36, 4 (2024), e2532. https://doi.org/10.1002/smr.2532

[29] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. JSEFT: Automated Javascript Unit Test Generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102595

[30] Partha Sarathi Mohapatra. 2025. Artificial Intelligence-Driven Test Case Generation in Software Development. *Intelligent Assurance: Artificial Intelligence-Powered Software Testing in the Modern Development Lifecycle* 4 (2025), 38.

[31] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic Test Smell Detection Using Information Retrieval Techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 311–322. https://doi.org/10.1109/ICSME.2018.00040

[32] Zedong Peng, Xuanyi Lin, Michelle Simon, and Nan Niu. 2021. Unit and regression tests of scientific software: A study on SWMM. *Journal of Computational Science* 53 (2021), 101347. https://doi.org/10.1016/j.jocs.2021.101347

[33] Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. 2022. An Empirical Comparison of EvoSuite and DSpot for Improving Developer-Written Test Suites with Respect to Mutation Score. In *Search-Based Software Engineering*, Mike Papadakis and Silvia Regina Vergilio (Eds.). Springer International Publishing, Cham, 19–34.

[34] David Schuler and Andreas Zeller. 2011. Assessing Oracle Quality with Checked Coverage. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 90–99. https://doi.org/10.1109/ICST.2011.32

[35] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. In *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation*, Vol. 50. IEEE Transactions on Software Engineering, 85–105. https://doi.org/10.1109/TSE.2023.3334955

[36] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. 2019. On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 121–125. https://doi.org/10.1109/MSR.2019.00028

[37] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 313–322. https://doi.org/10.1145/3661167.3661216

[38] Esdras Silva, Roberta Coelho, and Lyrene Silva. 2025. LLMs as Test Generators: A Comparative Benchmarking Study. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 25–36. https://doi.org/10.5753/sbes.2025.9618

[39] Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2025. Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation. In *2025 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*. 37–44. https://doi.org/10.1109/DeepTest66595.2025.00011

[40] Peter Strečanský, Stanislav Chren, and Bruno Rossi. 2020. Comparing maintainability index, SIG Method, and SQALE for technical debt identification. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 121–124. https://doi.org/10.1145/3341105.3374079

[41] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE]

[42] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 4–15. https://doi.org/10.1145/2970276.2970340

[43] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. In *Refactoring Test Code*, M. Marchesi and G. Succi (Eds.). Proceedings 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001).

[44] Tássio Virgínio, Luana Almeida Martins, Larissa Rocha Soares, Railana Santana, Heitor Costa, and Ivan Machado. 2020. An empirical study of automatically-generated tests from the perspective of test smells. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 92–96. https://doi.org/10.1145/3422392.3422412

[45] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2022. In *PyNose: a test smell detector for python* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 593–605. https://doi.org/10.1109/ASE51524.2021.9678615

[46] Roel Wieringa. 2014. *Design science methodology for information systems and software engineering*. Springer.

[47] Tao Xie and David Notkin. 2006. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal* 13, 3 (July 2006), 345–371.

[48] Shengcheng Yu, Chunrong Fang, Yucheng Ling, Chentian Wu, and Zhenyu Chen. 2023. LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities. In *LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities*. 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), 206–217. https://api.semanticscholar.org/CorpusID:262459296

[49] Dovydas Marius Zapkus and Asta Slotkienė. 2024. Unit Test Generation Using Large Language Models: A Systematic Literature Review. *Vilnius University Open Series* (May 2024), 136–144. https://doi.org/10.15388/LMITT.2024.20

[50] Sintayehu Zekarias Esubalew and Beakal Gizachew Assefa. 2025. Reimagining Unit Test Generation With AI: A Journey From Evolutionary Models to Transformers. *IEEE Access* 13 (2025), 154908–154929. https://doi.org/10.1109/ACCESS.2025.3597049