Rethinking Cognitive Complexity for Unit Tests: Toward a Readability-Aware Metric Grounded in Developer Perception

1st Wendkûuni C. Ouédraogo
University of Luxembourg
wendkuuni.ouedraogo@uni.lu

2nd Yinghua Li
University of Luxembourg
yinghua.li@uni.lu

3rd Xueqi Dang
University of Luxembourg
xueqi.dang@uni.lu

4th Xin Zhou
Singapore Management University
xinzhou.2020@phdcs.smu.edu.sg

5th Anil Koyuncu
Bilkent University
anil.koyuncu@cs.bilkent.edu.tr

6th Jacques Klein
University of Luxembourg
jacques.klein@uni.lu

7th David Lo
Singapore Management University
davidlo@smu.edu.sg

8th Tegawendé F. Bissyandé
University of Luxembourg
tegawende.bissyande@uni.lu

Abstract—**Automatically generated unit tests—from search-based tools like EvoSuite or LLMs—vary significantly in structure and readability. Yet most evaluations rely on metrics like Cyclomatic Complexity and Cognitive Complexity, designed for functional code rather than test code. Recent studies have shown that SonarSource's Cognitive Complexity metric assigns near-zero scores to LLM-generated tests, yet its behavior on EvoSuite-generated tests and its applicability to test-specific code structures remain unexplored. We introduce CCTR, a *Test-Aware Cognitive Complexity* metric tailored for unit tests. CCTR integrates structural and semantic features like assertion density, annotation roles, and test composition patterns—dimensions ignored by traditional complexity models but critical for understanding test code. We evaluate 15,750 test suites generated by EvoSuite, GPT-4o, and Mistral Large-1024 across 350 classes from Defects4J and SF110. Results show CCTR effectively discriminates between structured and fragmented test suites, producing interpretable scores that better reflect developer-perceived effort. By bridging structural analysis and test readability, CCTR provides a foundation for more reliable evaluation and improvement of generated tests. We publicly release all data, prompts, and evaluation scripts to support replication.**

Index Terms—**Automatic Test Generation, Unit test, Cognitive complexity, Large Language Models, Metrics, Test Code Understandability, Software Quality, Empirical Software Engineering.**

I. INTRODUCTION

As test automation evolves, traditional tools such as Evo-Suite [1] are increasingly complemented by Large Language Model (LLM)-based approaches [2], [3], [4], [5], [6], [7], [8], enabling scalable unit test generation across a wide range of software systems. However, recent studies report that only 40–70% of LLM-generated unit tests compile successfully without manual interventions [7], [3], [2], [8]. Even when syntactically correct, the structural quality and readability of these tests remain inconsistent and difficult to evaluate.

This raises the question of how to meaningfully assess the complexity and maintainability of generated test code—an area where existing metrics offer limited guidance. Traditional static complexity metrics like Cyclomatic Complexity [9] and SonarSource's Cognitive Complexity [10] were originally designed to evaluate functional logic, not test code. While recent studies [11], [12], [3], [6], [8] consistently report very low cognitive complexity scores for LLM-generated tests, none have evaluated this metric on EvoSuite-generated suites or examined its ability to reflect the structural and semantic complexity of test code—raising concerns about their suitability for assessing test quality.

In parallel, research on test readability [13], [14] highlights the importance of assertion clarity, descriptive naming, and structural conventions. The Arrange–Act–Assert (AAA) pattern, in particular, has been widely adopted for improving test readability by clearly separating setup, execution, and verification phases. The machine-learned readability model proposed by Scalabrino et al. [15], and later validated by Sergeyuk et al. [16], approximates human judgments of code readability based on textual features. However, it focuses primarily on surface-level attributes—such as identifier clarity and formatting—and does not account for deeper structural aspects like fragmentation, assertion density, or control-flow complexity. This leaves a critical gap in the quantitative assessment of test structure and maintainability.

We introduce **CCTR**, a *Test-Aware Cognitive Complexity* metric tailored for unit tests. CCTR integrates structural elements—assertion usage, annotation roles, and test composition patterns—into a lightweight, interpretable score that reflects both the syntactic structure and the congitive effort required to understand test suites. To assess its effectiveness, we evaluate 15,750 test suites generated by EvoSuite, GPT-4o, and Mistral Large across Defects4J [17] and SF110 [18]. Results show that CCTR effectively discriminates between fragmented, auto-generated tests and well-structured code, aligning more closely with dimensions valued by developers—such as clarity of

assertions, test structure, and semantic intent—compared to SonarSource's Cognitive Complexity [10], while providing a meaningful proxy for test maintainability.

Contributions. This paper provides:

- A critical analysis of the limitations of Cognitive Complexity when applied to test code;
- A novel metric, CCTR, designed to quantify structural and semantic complexity in unit tests;
- A large-scale empirical evaluation of CCTR across 15,750 test suites;
- Evidence demonstrating that CCTR more accurately reflects test readability aspects compared to existing metrics;
- A publicly available dataset of 15,750 generated test suites across 350 Java classes, including CCTR scores, prompt templates, and evaluation scripts: https://github.com/Cwendkuuni/CCTR

Paper organization. The remainder of this paper is structured as follows: Section II motivates the need for a test-specific metric through concrete examples; Section III presents our experimental setup and dataset; Section IV discusses the shortcomings of existing complexity metrics; Section V introduces and evaluates CCTR; Section VI discusses findings and limitations; Section VII reviews related work; and Section VIII concludes.

II. MOTIVATING EXAMPLE

To motivate the need for a test-specific complexity metric, we present three examples: two real test suites for the `CommandLine` class from the `Cli-1` bug (Defects4J), and one synthetic control-flow case. Full versions are available in our replication package. We use PMD's [19] implementation of Cognitive Complexity, based on SonarSource's specification [10], which scores control-flow constructs and nesting to estimate mental effort.

A. Example 1: Deeply Nested Trivial Code

```
public class ExampleTest {
    public void testExample() {
        if (condition) {
            for (int i = 0; i < 10; i++) {
                while (anotherCondition) {
                    // nested logic
                }
            }
        }
    }
}
```

Listing 1: Manually written nested code

Listing 1 exhibits structural nesting but minimal logic. PMD assigns it a Cognitive Complexity of 12.

B. Example 2: LLM-Generated Test Suite

```
@Test
public void testGetOptionValueWithDefaultValue() {
    assertEquals("default", commandLine.getOptionValue("test", "default"));
}
```

Listing 2: Excerpt from GPT-4o-generated test

Listing 2 shows part of a 12-method suite generated by GPT-4o. Despite meaningful naming, consistent structure, and annotations, comprehension still requires effort due to multiple methods, assertion logic, and parameter variation. Yet PMD assigns it a Cognitive Complexity of 0, missing this moderate structural and semantic load.

C. Example 3: EvoSuite-Generated Test Suite

```
@Test(timeout = 4000)
public void test02() throws Throwable {
    CommandLine commandLine0 = new CommandLine();
    Option option0 = new Option("", "\"Jd", true, "Dl,L");
    option0.addValue("org.apache.commons.cli.CommandLine");
    commandLine0.addOption(option0);
    String string0 = commandLine0.getOptionValue("--", null);
    assertNotNull(string0);
    assertEquals("org.apache.commons.cli.CommandLine", string0);
}
```

Listing 3: Excerpt from EvoSuite-generated test

Listing 3 is part of a 17-method suite generated by EvoSuite. It features cryptic naming and lacks structural clarity or semantic grouping. Yet, its Cognitive Complexity score—according to PMD—is 0.

As these examples illustrate, the control-flow-oriented approach fails to capture key structural and semantic aspects of unit tests. Despite clear differences in readability, structure, and developer intent, all three cases receive nearly identical scores—highlighting a misalignment between the SonarSource assumptions and the actual comprehension effort involved in test code.

D. Summary

Table I summarizes the PMD Cognitive Complexity scores across the three examples. To better illustrate the disconnect between PMD's scoring and intuitive comprehension effort, we include a perceived complexity label based on structural and semantic clarity. While the trivial nested example receives a non-zero score, more cognitively demanding test suites are rated as zero—highlighting a critical misalignment with developer intuition.

TABLE I: Cognitive Complexity (PMD) vs. Perceived Complexity

Example	Origin	PMD Score	Perceived Complexity
Nested Trivial Code	Manual	12	Low
LLM-Generated Test Suite	GPT-4o	0	Medium
EvoSuite Test Suite	EvoSuite	0	High

These results highlight a critical gap: while PMD captures structural nesting, it fails to reflect the human-perceived complexity of real-world test suites—especially those with rich assertion logic, mocking, or ambiguous structure. This motivates the introduction of a test-specific cognitive complexity metric in Section V.

III. EMPIRICAL OBSERVATIONS AT SCALE

A. Dataset and Test Generation Setup

To evaluate CCTR, we conducted a large-scale study using two established Java code corpora: Defects4J [17] and SF110 [18]. These benchmarks provide diverse real-world classes with varying complexity, serving as a robust foundation for evaluating both traditional and LLM-generated test suites. **Dataset composition.** We selected 15 projects from Defects4J (147 classes) and 69 projects from SF110 (203 classes), totaling 350 classes across 84 projects. Table II summarizes the overall characteristics.

Test generation. We generated unit tests for each class using three approaches: EvoSuite [1] with a 3-minute time budget

TABLE II: Dataset Composition and Code Statistics

Dataset	#Projects	#Classes	LOC avg	Token avg	Methods/Class avg
Defects4J	15	147	127.76	1779.67	11.88
SF110	69	203	146.82	1518.86	14.21

and the DynaMOSA search algorithm [18], GPT-4o [20], and Mistral Large-2407[1] with zero-shot prompting. To mitigate randomness in LLM outputs, we set the temperature to 0.1 and kept top-p at its default. Each tool/model was run 15 times per class, resulting in 2,205 test suites per model on Defects4J and 3,045 per model on SF110. In total, we produced 10,500 test suites with LLMs and 5,250 with EvoSuite, yielding 15,750 test suites overall.

Test characteristics. Table III presents average metrics across all generated tests. While EvoSuite produced longer test classes with more methods and higher token counts, LLM-generated tests were more concise yet semantically expressive. Compilation rates were 100% for EvoSuite and lower for LLMs due to occasional syntax or API misuses.

TABLE III: Generated Test Suite Statistics per Dataset and Model

Dataset	Model	#Tests	LOC avg	Token avg	Methods avg	Compilability
Defects4J	EvoSuite	2205	169.40	1958.94	17.38	100%
	GPT-4o	2205	95.04	822.98	14.45	67.58%
	Mistral-L	2205	112.06	939.33	16.20	42.42%
SF110	EvoSuite	3045	294.14	3462.57	27.82	100%
	GPT-4o	3045	96.40	894.36	13.71	49.80%
	Mistral-L	3045	112.06	939.33	16.20	42.42%

B. Code Complexity and Readability Analysis

We present a large-scale comparative analysis of unit test suites generated by LLMs and EvoSuite across three complementary perspectives: Cyclomatic Complexity, Cognitive Complexity, and automated readability scores from Scalabrino et al. [15].

Cyclomatic complexity. Cyclomatic Complexity [9] estimates linearly independent paths through code and approximates control-flow density. As shown in Table IVa, test suites generated by LLMs and EvoSuite exhibit substantial control-flow complexity, particularly in SF110, confirming that generated test code contains non-trivial execution paths and is far from structurally simplistic.

TABLE IV: Cyclomatic Complexity and Readability Scores across Models for Each Dataset.

(a) **Cyclomatic Complexity**

Dataset	Model/Tool	Min	Q1	Median	Q3	Max	Mean
Defects4J	GPT-4o	2	10	13	19	35	14.95
	Mistral-L	2	10	12	18	37	14.89
	EvoSuite	1	6	17	29	131	19.78
SF110	GPT-4o	3	8	14	19	92	15.69
	Mistral-L	3	9	13	19	92	16.91
	EvoSuite	1	8	17	34	230	25.55

(b) **Readability Scores (Scalabrino et al., 0–100)**

Dataset	Model/Tool	Min	Q1	Median	Q3	Max	Mean
Defects4J	GPT-4o	0.0	63.0	69.3	74.2	87.5	67.0
	Mistral-L	19.2	62.3	69.0	74.0	85.2	67.0
	EvoSuite	8.0	41.8	58.5	72.8	95.7	56.3
SF110	GPT-4o	23.8	58.7	66.3	70.8	90.3	64.4
	Mistral-L	24.6	60.2	67.3	72.4	84.4	64.9
	EvoSuite	4.1	–	50.9	–	95.7	51.1

Readability. To approximate perceived readability at scale, we use the model by Scalabrino et al. [15], which combines

TABLE V: Sonar Cognitive Complexity (PMD).

Dataset	Model/Tool	Min	Q1	Median	Q3	Max	Mean
Defects4J	GPT-4o	0	0	0	0	8	0.285
	Mistral-L	0	0	0	0	14	0.761
	EvoSuite	0	1	2	6	30	3.931
SF110	GPT-4o	0	0	0	0	46	0.585
	Mistral-L	0	0	0	0	46	1.203
	EvoSuite	0	1	2	6	102	4.745

104 structural, textual, and visual features trained on annotated Java code, including JUnit tests. As shown in Table IVb, it consistently ranks LLM-generated tests as more readable than those from EvoSuite. Sergeyuk et al. [16] found this model to best align with human ratings among established alternatives [21], [22], [23], [24], [25].

Cognitive complexity. We use PMD's implementation of Cognitive Complexity [10], which penalizes nested control structures, logical operators, and flow-disrupting elements. As shown in Table V, the metric yields extremely low values—often zero—for LLM-generated tests and modest scores for EvoSuite despite highly fragmented test classes. This suggests that Cognitive Complexity overlooks many characteristics that increase cognitive effort in understanding unit tests, including assertion logic, mocking, and test-specific design idioms.

Takeaway. Each perspective captures a specific facet: control-flow density (Cyclomatic), perceived readability (Scalabrino), and nesting-driven logic effort (Cognitive Complexity). Cyclomatic and Scalabrino operate effectively within their scopes, but Sonar's Cognitive Complexity—despite growing use in quality assessments—was designed for functional code, not test code. Our results show it fails to reflect structural and semantic effort in understanding unit tests, overlooking test-specific patterns like assertion logic, mocking constructs, and annotation-based roles. This gap motivates the need for a dedicated cognitive complexity metric that reflects the unique characteristics of test code.

IV. LIMITS OF COGNITIVE COMPLEXITY FOR TESTS

Although SonarSource's Cognitive Complexity metric [10] is widely adopted to assess source code understandability, it was originally designed to measure the mental effort required to comprehend functional logic—not unit tests. In this section, we identify three key limitations that emerge when applying this metric to test code.

A. Not Designed for Unit Test Constructs

The Cognitive Complexity metric penalizes nested control structures (`if`, `for`, `while`), logical operators, and flow interruptions (`break`, `return`)—constructs common in business logic but less prevalent in well-written unit tests. Conversely, constructs heavily influencing test code readability and structure—assertions, mocking invocations, and annotations like `@Test` or `@ParameterizedTest`—are completely ignored. Test methods that heavily rely on test-specific constructs—such as assertions, mocking invocations, and annotations—may still receive zero complexity scores,

undermining the metric's applicability for evaluating generated tests or comparing test suites across tools.

B. Empirical Observations from Generated Tests

We report empirical observations from our analysis of 15,750 test suites generated by EvoSuite, GPT-4o, and Mistral Large, as previously detailed in Sections II and III. Despite significant differences in structural quality and readability, PMD's Cognitive Complexity assigns zero scores to over 99% of LLM-generated test methods and only modest scores to EvoSuite outputs. As illustrated in Section II, even a deeply nested dummy example scored 12, while real test suites containing semantically rich and non-trivial assertions often received a score of zero. This highlights a key limitation: the metric fails to capture the structural and cognitive effort inherent in test code, particularly for tests generated by modern LLMs.

C. Lack of Alignment with Test Readability Dimensions

Recent studies emphasize that test code readability extends beyond control flow. Winkler et al. [13] show developers prioritize clear structure (AAA pattern), meaningful naming, assertion logic, and setup clarity. Similarly, Daka et al. [14] and Biagiola et al. [11] highlight that human-perceived test quality correlates more with naming, structure, and assertion intent than nesting or branches. Yet none of these elements are captured by Cognitive Complexity. The increasing use of this metric for evaluating LLM-generated tests [11], [12] lacks foundational justification. A test-specific complexity metric should reflect cognitive effort from test-specific constructs—not control-flow alone.

These limitations motivate the development of a new metric that accounts for the actual structural and semantic features of test suites.

V. TOWARD A TEST-SPECIFIC COGNITIVE COMPLEXITY

We propose **CCTR** (Cognitive Complexity for Test Readability), a test-specific metric designed to quantify structural and semantic effort in test comprehension. CCTR addresses limitations in existing metrics by capturing constructs that developers actively process when reading tests—assertions, mocking patterns, and annotations—while maintaining control-flow sensitivity.

A. Empirical Motivation

CCTR is grounded in the developer-centric empirical model proposed by Winkler et al. [13], which identifies three main dimensions of test comprehension: **test structure** (e.g., AAA clarity, indentation), **naming and expression semantics** (e.g., assertion intent, variable clarity), and **test logic and purpose** (e.g., relevance of setup, test goal). Each component of CCTR is designed to reflect one or more of these dimensions. In particular, our inclusion of annotations is inspired by Guerra et al. [26], who found that annotations like `@Test` and `@NotNull` improve code readability and convey test roles—but also warned against overuse. CCTR assigns weights

TABLE VI: CCTR (Test-Aware Cognitive Complexity)

Dataset	Model/Tool	Min	Q1	Median	Q3	Max	Mean
Defects4J	GPT-4o	3	13	23	34	127	26.44
	Mistral-L	4	13	23	40	132	27.91
	EvoSuite	0	10	31	53	494	39.47
SF110	GPT-4o	2	16	24	34	206	28.41
	Mistral-L	2	16	23	34	206	30.61
	EvoSuite	0	15	38	75	1557	58.61

accordingly, balancing informativeness with complexity impact.

B. CCTR Definition

We define CCTR as:

$$\text{CCTR} = \alpha \cdot N + \beta \cdot A + \gamma \cdot M + \delta \cdot T \quad (1)$$

Where N denotes the control-flow nesting complexity, following the original SonarSource definition. The term A corresponds to the number of assertion or `fail()` statements present in the test method. The component M captures the number of mocking-related constructs, such as calls to `mock()`, `verify()`, or `when()`. Finally, T represents annotation-based signaling. It is computed by assigning a value of +1 for each occurrence of common test-related annotations such as `@Test`, `@BeforeEach`, and `@AfterEach`, and a value of +2 for the presence of more specialized annotations such as `@ParameterizedTest`.

We adopt initial weights:

$$\alpha = \beta = \gamma = \delta = 1.0$$

This uniform weighting reflects our design goal of simplicity and interpretability, while aligning with empirical studies suggesting these factors contribute comparably to test comprehension. Our treatment of annotations is further informed by Guerra et al. [26], who show that annotations like `@Test` and `@NotNull` enhance structural clarity and intent signaling—especially when defining test roles or validation rules. However, they also caution against overuse or ambiguity. To balance informativeness and score inflation, CCTR assigns moderate, fixed weights to annotation types. Tuning these weights remains future work guided by large-scale developer studies.

C. Evaluation and Observations

To evaluate CCTR, we computed scores across two benchmarks (Defects4J, SF110) and three test generation strategies: GPT-4o, Mistral Large, and EvoSuite.

Results. Several insights emerge from Table VI. **LLM-generated test suites** (GPT-4o, Mistral) show moderate complexity (mean 26–30), consistent with their structured yet concise test style. In contrast, **EvoSuite tests** score significantly higher due to their fragmented structure, high method count, and dense assertions. CCTR also distinguishes between test suites that SonarSource's Cognitive Complexity fails to separate—capturing both generator-level and dataset-level differences. Finally, despite variability in LOC and token count, CCTR scales robustly across tools and datasets. These results support that CCTR better reflects test comprehension

effort, aligning with developer-centric principles and structural variation.

D. Illustration

To illustrate how CCTR captures structural and semantic variation, we revisit the three examples from Section II. Table VII shows both PMD and CCTR scores, highlighting how traditional metrics fail to reflect comprehension differences.

TABLE VII: Comparison of PMD and CCTR scores across example test classes

Example	PMD Score	CCTR Score	Perceived Complexity
Nested Trivial Code	12	12	Low
LLM-Generated Test Suite	0	12	Medium
EvoSuite Test Suite	0	35	High

Each test class represents a distinct style and level of structural clarity. **Nested Trivial Code** (Listing 1), while semantically simple, involves deep control-flow nesting and yields a moderate CCTR score of **12**, reflecting structural complexity. **LLM-Generated Test Suite** (Listing 2) achieves the same score, but for different reasons: concise methods, consistent assertions, clear naming, and informative annotations. In contrast, the **EvoSuite-Generated Test Suite** (Listing 3) contains 17 fragmented methods with synthetic names and redundant assertions, resulting in a much higher CCTR score of **35**, indicative of increased comprehension effort.

These examples show that CCTR produces meaningful, differentiated scores at the class level, reflecting structural variation across test styles. Combined with Table VI, this illustrates its applicability to both individual cases and large-scale benchmarks.

VI. DISCUSSION AND FUTURE WORK

Implications for practice. CCTR offers a test-specific lens for evaluating unit tests where readability matters, aiding refactoring, design, and automated review. By integrating assertions, mocking, and annotations, it provides a cognitively grounded proxy for maintainability. CCTR also scales to auto-generated tests, enabling large-batch evaluation of synthesis quality.

Revisiting prior metrics. Our results reveal that existing structural metrics like Cyclomatic Complexity and Cognitive Complexity fail to reflect semantic and structural dimensions central to unit test design. This explains why prior studies [11], [12] relied on workarounds like CodeBLEU [27] or cosine similarity for refactoring validation despite lacking theoretical grounding. CCTR offers a principled alternative.

Limitations. CCTR, like other static metrics, does not capture all test quality aspects—it misses behavioral correctness, flakiness, and fault detection. Its fixed weights, though empirically motivated, may not generalize across all test styles. Still, CCTR complements dynamic analyses and other metrics to support broader assessment. Cross-domain and cross-language validation remains future work.

Future work. Several extensions are planned. First, we aim to conduct a human study to validate the correlation between CCTR scores and developer-perceived test comprehensibility. Second, we plan to explore CCTR's utility in test generation workflows, particularly as a scoring signal for selecting or prioritizing LLM-generated test candidates. Finally, we envision extending CCTR into a multi-dimensional model that incorporates code coverage and fault-detection ability to offer a more comprehensive view of test effectiveness.

VII. RELATED WORK

Code complexity metrics. McCabe's Cyclomatic Complexity [9] models control-flow density, while Cognitive Complexity [10] penalizes nesting and flow disruptions to approximate mental effort. Yet, empirical studies [28], [29] show limited alignment with human comprehension. Our findings confirm that PMD's implementation fails to distinguish test classes with different structural and semantic clarity. In contrast, CCTR captures test-specific constructs—assertions, mocking, annotations—offering a better proxy for perceived test complexity.

Readability models. Several models estimate code readability using lexical, syntactic, or visual features, including work by Buse and Weimer [21], [22], Dorn et al. [24], and Posnett et al. [23]. Scalabrino et al. [15] proposed a 104-feature model, later validated by Sergeyuk et al. [16]. While effective for general code, these models emphasize surface-level cues and overlook structural or cognitive traits crucial to test code. CCTR addresses this gap with a metric tailored to test-specific comprehension.

Evaluating generated tests. With LLM-based code generation, new methods assess generated test quality. Biagiola et al.[11] and Deljouyi et al.[12] used CodeBLEU and semantic similarity to detect hallucinations or assess refactoring correctness, but these metrics lack formal grounding in test comprehension. Our work introduces a test-specific complexity metric capturing structural signals—assertions, annotations, and test logic—ignored by existing syntactic metrics and embedding-based scores.

VIII. CONCLUSION AND FUTURE WORK

We introduced **CCTR**, a test-aware complexity metric designed to capture structural and semantic elements critical to unit test comprehension—such as assertion density, mocking constructs, and annotation semantics. Grounded in empirical studies on test readability, CCTR extends traditional metrics by aligning more closely with developer intuition. Our evaluation across 15,750 test suites shows that CCTR effectively differentiates between generation styles and produces scores reflective of structural effort. By bridging complexity analysis and test-specific readability, CCTR opens new possibilities for evaluating, refactoring, and improving unit tests. Future work will focus on human-centered validation, adaptive weighting, and integration into LLM-based test generation workflows.

REFERENCES

[1] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[2] M. L. Siddiq, J. C. Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. C. Lopes, "Using large language models to generate junit tests: An empirical study," 2024.

[3] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "Chatgpt vs sbst: A comparative assessment of unit test suite generation," *IEEE Transactions on Software Engineering*, 2024.

[4] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[5] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.

[6] W. C. Ouedraogo, K. Kabore, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyande, "Llms and prompting for unit test generation: A large-scale evaluation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2464–2465.

[7] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang *et al.*, "On the evaluation of large language models in unit test generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1607–1619.

[8] W. C. Ouédraogo, K. Kaboré, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyandé, "Large-scale, independent and comprehensive study of the power of llms for test case generation," *arXiv preprint arXiv:2407.00225*, 2024.

[9] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[10] G. A. Campbell, "Cognitive complexity: A new way of measuring understandability," https://www.sonarsource.com/docs/CognitiveComplexity.pdf, 2023, version 1.7.

[11] M. Biagiola, G. Ghislotti, and P. Tonella, "Improving the readability of automatically generated tests using large language models," *arXiv preprint arXiv:2412.18843*, 2024.

[12] A. Deljouyi, R. Koohestani, M. Izadi, and A. Zaidman, "Leveraging large language models for enhancing the understandability of generated unit tests," *arXiv preprint arXiv:2408.11710*, 2024.

[13] D. Winkler, P. Urbanke, and R. Ramler, "Investigating the readability of test code," *Empirical Software Engineering*, vol. 29, no. 2, p. 53, 2024.

[14] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 107–118.

[15] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018.

[16] A. Sergeyuk, O. Lvova, S. Titov, A. Serova, F. Bagirov, E. Kirillova, and T. Bryksin, "Reassessing java code readability models with a human-centered approach," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 225–235.

[17] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[18] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[19] PMD, "PMD An extensible cross-language static code analyzer." https://pmd.github.io/, 2012, [Online; accessed 06-Jun-2024].

[20] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[21] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 121–130.

[22] ——, "Learning a metric for code readability," *IEEE Transactions on software engineering*, vol. 36, no. 4, pp. 546–558, 2009.

[23] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th working conference on mining software repositories*, 2011, pp. 73–82.

[24] J. Dorn, "A general software readability model," *MCS Thesis available from (http://www. cs. virginia. edu/weimer/students/dorn-mcs-paper. pdf)*, vol. 5, pp. 11–14, 2012.

[25] Q. Mi, Y. Hao, L. Ou, and W. Ma, "Towards using visual, semantic and structural features to improve code readability classification," *Journal of Systems and Software*, vol. 193, p. 111454, 2022.

[26] E. Guerra, E. Gomes, J. Ferreira, I. Wiese, P. Lima, M. Gerosa, and P. Meirelles, "How do annotations affect java code readability?" *Empirical Software Engineering*, vol. 29, no. 3, p. 62, 2024.

[27] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[28] M. Muñoz Barón, M. Wyrich, and S. Wagner, "An empirical validation of cognitive complexity as a measure of source code understandability," in *Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, 2020, pp. 1–12.

[29] T. Sharma and D. Spinellis, "Do we need improved code quality metrics?" *arXiv preprint arXiv:2012.12324*, 2020.