Evaluating the Quality of GPT-4 Generated Unit Tests for JavaScript

João Vítor Santana Depollo
Federal University of Minas Gerais
Belo Horizonte, Brazil
Email: jvsd@ufmg.br

Advisor:
Eduardo Figueiredo

Co-advisor:
Pedro Saint Clair Garcia

***Abstract*—This research presents a focused evaluation of the quality of unit tests generated by the OpenAI GPT-4 model for the JavaScript programming language. The study is conducted using Unit Cloud Gen, a platform developed to automate the generation and analysis of tests. The work concentrates on a corpus of problems from LeetCode, specifically chosen to assess the LLM's performance on code of varying complexity, with the objective of identifying where the model succeeds and where it fails. The evaluation goes beyond traditional code coverage, incorporating metrics such as fault detection capability, readability, and maintainability. The tests are executed in isolated Docker containers, and the results are stored for analysis and benchmarking. This study aims to identify the strengths and limitations of GPT-4 in generating tests for algorithmic problems, providing a foundation for improving the reliability and efficiency of AI-assisted test generation. generation.**

***Index Terms*—large language models, automated test generation, unit testing, software quality, test evaluation**

1 Introduction

Unit tests are a fundamental component in modern software development, used to verify that individual code units, such as functions or methods, work as intended. This practice is crucial not only for maintaining correct component behavior as a system evolves but also for promoting more reliable development by enabling early failure detection and encouraging modular architectures [8].

The emergence of Large Language Models (LLMs) has sparked a significant transformation in code generation automation. These models have shown a remarkable ability to interpret natural language instructions and produce functional code, including unit tests. However, the effectiveness of these automatically generated tests has not yet been subjected to systematic and comprehensive evaluation.

This thesis proposes an investigation into methodologies for evaluating the adequacy and effectiveness of unit tests generated by the **OpenAI GPT-4** model for the **JavaScript** language. This analysis is essential to ensure that AI-automated tests can be safely integrated into software development workflows in professional environments. The study will use a controlled corpus of **LeetCode problems** to systematically test the core hypothesis: that LLMs can generate effective unit tests for code with low to moderate complexity.

Our work aims to move beyond simple code coverage measurements to evaluate test quality more holistically. We will include metrics related to readability, maintainability, fault detection capability, and adherence to best coding practices. By focusing on a single, powerful LLM and a representative set of programming problems, we intend to provide a detailed analysis of its strengths and limitations in a specific, real-world context.

LLMs represent a new paradigm in software engineering, with the potential to revolutionize developer productivity by generating unit tests from functional descriptions or source code itself. To fully leverage this potential, it is critical to develop robust evaluation methods that can ensure the quality and reliability of these automatically generated tests. This research contributes to that effort by providing a structured framework and empirical data on the performance of a leading LLM in a focused domain.

2 Background

2.1 Unit Testing

Unit tests are a fundamental software engineering practice that consists of individually verifying isolated code units, such as functions, methods, or classes, to ensure they work as expected. According to Meszaros [8], this approach not only identifies failures early but also promotes modular design and documents the expected code behavior.

Manual creation of unit tests, although valuable, represents a process that is often time-consuming and laborious. Developers frequently need to balance test quality with time and resource constraints, which can result in incomplete coverage or low-quality tests [6].

2.2 Large Language Models (LLMs)

Large Language Models, such as GPT, Claude, LLaMA, and CodeLlama, represent significant advances in natural language and code processing and generation capabilities. Trained on vast corpora of text and source code, these models demonstrate remarkable ability to generate functional code from natural language instructions [4], [5].

The use of LLMs for code generation has attracted growing interest in the software engineering community, with studies exploring their applications in various aspects of development, including automated generation of unit tests [14].

2.3 TestGen-LLM: Meta's Approach

In the article "Automated Unit Test Improvement using Large Language Models at Meta," Alshahwan et al. [1] present TestGen-LLM, a tool that uses LLMs to automatically improve existing unit tests. The approach adopts the concept of "Assured Offline LLMSE" (Large Language Model Software Engineering with Offline Guarantees), where language models are incorporated into a broader workflow that provides verifiable code improvement recommendations.

The system applies a series of progressively demanding semantic filters to candidate solutions generated by the models:

- Compilation verification
- Test execution to eliminate those that fail
- Identification and elimination of "flaky" (unstable) tests
- Coverage measurement to ensure improvement

A significant limitation identified in the study is the exclusive reliance on line coverage as an evaluation metric, neglecting important qualitative aspects of the generated tests. In an evaluation of Instagram's Reels and Stories products, 75% of test cases generated by TestGen-LLM compiled correctly, 57% passed reliably, and 25% increased coverage.

2.4 Quality Metrics for Unit Tests

The quality of unit tests goes beyond simple code coverage. Studies such as those by Panichella et al. [10] and Palomba et al. [9] highlight the importance of evaluating aspects such as:

- Readability and maintainability: tests should be understandable and easily modifiable [7]
- Adequacy of assertions: tests should verify significant behaviors, not just execute code
- Fault detection: test capability to identify real problems [12]
- Robustness against changes: resistance to refactoring in the tested code
- Absence of "code smells": problematic patterns specific to test code [3]

These metrics provide a more holistic basis for evaluating automatically generated tests, complementing traditional coverage metrics.

2.5 LeetCode Problems

LeetCode is a popular online platform that provides a vast collection of algorithmic and data structure problems. It is widely used by software developers to practice coding skills, prepare for technical interviews, and benchmark their problem-solving abilities. The platform's problems are categorized by topic and difficulty, making it an ideal resource for creating a structured and diverse dataset for evaluating LLM performance.

The problems on LeetCode are categorized into various types, including but not limited to:

- Algorithms: Sorting, Searching, Dynamic Programming, Greedy, Recursion.
- Data Structures: Arrays, Strings, Linked Lists, Trees, Graphs, Hash Tables.
- Concepts: Two Pointers, Sliding Window, Backtracking.

Problems are also ranked by difficulty, providing a clear scale for comparison:

- Easy: Straightforward problems that test fundamental concepts.
- Medium: More complex problems that require a solid understanding of algorithms and data structures.
- Hard: Challenging problems that often combine multiple concepts and require advanced problem-solving skills.

3 Methodology

This study will adopt an experimental methodology to evaluate the quality and reliability of unit tests generated by the **GPT-4** model. The approach centers on the development and use of **Unit Cloud Gen**, a platform designed to streamline the process of test generation, execution, and analysis. The evaluation process is structured to assess not only how the LLM generates unit tests from a curated corpus of problems but also how effectively those tests function. Ultimately, the goal is to identify systematic limitations and potential for improvement in LLM-assisted test generation.

3.1 Development Corpus

A curated set of source code samples will be used as input for test generation. These samples will be implemented in **JavaScript** and will be selected from **LeetCode**. LeetCode is a popular platform that provides a vast collection of algorithmic and data structure problems.

The corpus will be chosen to represent a range of algorithmic and data structure problems, with a focus on those that vary in complexity. Problems are ranked by difficulty, providing a clear scale for comparison:

- **Easy:** Straightforward problems that test fundamental concepts.
- **Medium:** More complex problems that require a solid understanding of algorithms and data structures.
- **Hard:** Challenging problems that often combine multiple concepts and require advanced problem-solving skills.

All code will be manually verified and structured to ensure consistency in the test generation and evaluation phases.

3.2 Test Generation Workflow

Unit tests will be generated automatically by the **OpenAI GPT-4** model. For each selected code sample, prompt templates will be defined and adapted to achieve a state-of-the-art level of test generation. The model will receive the source code as input and return the generated test cases. No human-written reference tests will be provided during generation, to ensure that the tests reflect the model's reasoning based solely on the code context.

3.3 Dual Evaluation Process

Each generated test will go through two evaluation stages:

1) **LLM Self-Evaluation:** The same or a different LLM will be asked to assess the quality of the generated test, using a prompt structured to produce a score or explanation regarding its adequacy, coverage, and clarity.
2) **Empirical Evaluation:** The test will be executed in isolated Docker containers, where coverage metrics (e.g., line and branch coverage) will be collected using tools such as `Istanbul` for JavaScript. In addition, custom scripts will analyze assertion usage, naming conventions, edge case coverage, and fault injection effectiveness.

This dual process allows for cross-validation between what the LLM *believes* is a good test and what is empirically observed, helping to reveal where the model overestimates or misinterprets test quality.

3.4 Gap Analysis and Failure Pattern Identification

After evaluating multiple test samples, the study will focus on identifying recurring gaps and weaknesses in the tests generated by GPT-4. These include:

- Incomplete coverage or missing edge cases.
- Misunderstanding of algorithmic logic.
- Overconfident or incorrect self-evaluations.
- Performance bottlenecks in local model inference.

Qualitative patterns will be cataloged to guide future prompt engineering strategies and model improvements.

3.5 Data Collection and Benchmarking

All results—generated tests, coverage metrics, self-evaluations, and execution logs—will be stored in a structured relational database. This will support future benchmarking efforts and enable comparative studies across different problem types, difficulties, and prompt strategies. The data will also be used to analyze the computational cost associated with each test generation request, supporting cost-benefit assessments for this focused use case.

4 Practical Implementation

The study will include the development of an open-source tool that integrates the insights obtained in the research, called **Unit Cloud Gen**, offering a unified interface for generating and evaluating unit tests using OpenAi GPT4 Model.

4.1 Architecture

The architecture of **Unit Cloud Gen** is structured in **four logical layers**, promoting separation of concerns and following principles from Domain-Driven Design (DDD) and Clean Architecture. Each layer encapsulates specific responsibilities and interacts only with adjacent layers via well-defined interfaces.

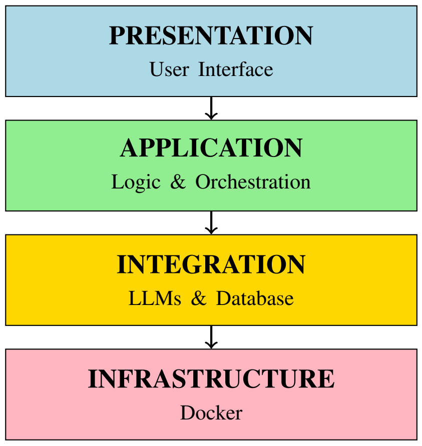

Figure 1. Four-Layer Architecture of Unit Cloud Gen

4.2 Layer Responsibilities

Presentation Layer The user interface is built with React and TypeScript, centered around the Monaco Editor to allow seamless, in-browser code editing. As developers type, syntax highlighting and linting provide immediate feedback, while responsive layouts ensure the experience runs smoothly across devices. For the supported language, JavaScript, the UI includes a starter test template that demonstrates key patterns and conventions for typical LeetCode-style problems. Below is a representative JavaScript example of a common algorithmic problem that the platform provides:

```javascript
/**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function(strs) {
    if (!strs || strs.length === 0) {
        return "";
    }

    let prefix = strs[0];
    for (let i = 1; i < strs.length; i++) {
        while (strs[i].indexOf(prefix) !== 0) {
            prefix = prefix.substring(0, prefix.length - 1);
            if (prefix === "") {
                return "";
            }
        }
    }

    return prefix;
};
```

Application Layer The Application Layer provides FastAPI-based RESTful endpoints with asynchronous support, acting as the main entry point for client interactions. It manages HTTP request handling, input validation, rate limiting, and standardized error responses to ensure a robust and user-friendly API experience. The core endpoints include:

- `/generate-tests`: Receives source code and language input, invokes the appropriate prompt template, and delegates test generation to the Integration Layer.
- `/evaluate-test-quality`: Accepts test cases and corresponding source code, triggers execution in the isolated environment, and returns a quality score with coverage metrics.

- `/providers`: Returns a list of available AI providers (e.g., OpenAI) and their configurations.
- `/models`: Lists the supported models for each provider, along with metadata such as context size, pricing tier, and status (cloud or local).

Beyond the API interface, the application layer is also responsible for preparing the execution context for each test. It dynamically selects language-specific prompt templates, transforms source code into standardized formats expected by the testing environments, and packages test requests for asynchronous processing. It coordinates container startup and monitoring, and delegates responsibilities to the business layer while handling response lifecycle and error fallback strategies.

Integration Layer The Integration Layer manages communication with external services through a unified LLM client interface. It supports commercial models such as OpenAI's GPT-4. This layer abstracts prompt execution, implements model fallback strategies, and handles provider-specific constraints (rate limits, token size, pricing). It also interfaces with the test execution system, launching Docker containers tailored for JavaScript (using Jest), capturing execution logs, code coverage reports, and error messages. Test results are parsed and structured for feedback and stored in a relational database, enabling traceability and future analytics.

Infrastructure Layer The Infrastructure Layer is responsible for provisioning and managing the underlying execution environment for the entire platform. It adopts the Infrastructure as Code (IaC) paradigm through the use of Terraform—a declarative tool that allows developers to define infrastructure components in version-controlled configuration files. In the context of Unit Cloud Gen, Docker containerization is applied to isolate execution environments for each supported programming language, ensuring consistency and security during test evaluation. Additionally, each host environment has an associated cost profile. These configurations are tracked and integrated into the test execution pipeline, allowing the platform to calculate and report the estimated cost of generating and evaluating each unit test.

Application Layer The Application Layer provides FastAPI-based RESTful endpoints with asynchronous support, acting as the main entry point for client interactions. It manages HTTP request handling, input validation, rate limiting, and standardized error responses to ensure a robust and user-friendly API experience.

The core endpoints include:

- `/generate-tests`: Receives source code and language input, invokes the appropriate prompt template, and delegates test generation to the Integration Layer.
- `/evaluate-test-quality`: Accepts test cases and corresponding source code, triggers execution in the isolated environment, and returns a quality score with coverage metrics.

- `/providers`: Returns a list of available AI providers (e.g., OpenAI) and their configurations.
- `/models`: Lists the supported models for each provider, along with metadata such as context size, pricing tier, and status (cloud or local).

Beyond the API interface, the application layer is also responsible for preparing the execution context for each test. It dynamically selects language-specific prompt templates, transforms source code into standardized formats expected by the testing environments, and packages test requests for asynchronous processing. It coordinates container startup and monitoring, queues jobs to be evaluated, and delegates responsibilities to the business layer while handling response lifecycle and error fallback strategies.

4.2.1 Inter-Layer Communication

The four layers of the architecture communicate through well-defined interfaces, adhering to dependency inversion principles to ensure decoupling and maintainability:

1) **Presentation** $\rightarrow$ **Application**: Sends HTTP requests with validated input for test generation, evaluation, or model selection.
2) **Application** $\rightarrow$ **Integration**: Delegates tasks such as LLM invocation, database operations, and test execution coordination through service abstractions.
3) **Integration** $\rightarrow$ **Infrastructure**: Interacts with cloud resources, Docker containers, and storage systems to perform the actual execution and persistence operations.

5 Analysis and Discussion of Empirical Results

The evaluation focuses on assessing the quality and reliability of unit tests generated by the GPT-4 model. By contrasting traditional code coverage metrics with the empirical error rate observed during test execution, we identify a clear paradox: high coverage does not imply high reliability.

5.1 Code Coverage Analysis

The initial analysis suggests that GPT-4 is highly effective at traversing the code structure. As shown in Figure 2, the model achieves perfect line coverage for Easy and Medium problems, with only a slight variation observed for Hard problems.

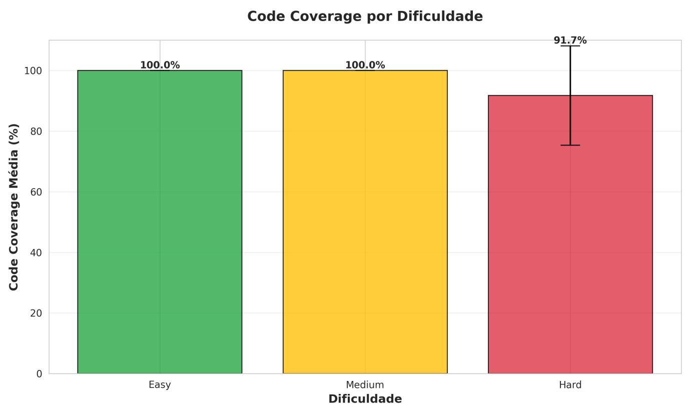

Figure 2. Mean Code Coverage by Difficulty. The model maintains high structural coverage across all levels.

However, this metric can be misleading. While the coverage remains stable and high, it masks the model's inability to generate semantically correct assertions for complex logic.

5.2 Error Rate and Reliability

When analyzing the error rate (tests that failed execution), a direct correlation with problem complexity becomes evident. Figure 3 illustrates that while Easy problems maintain a low failure rate (7.7%), Hard problems suffer from a severe drop in reliability, reaching a 36.9% error rate.

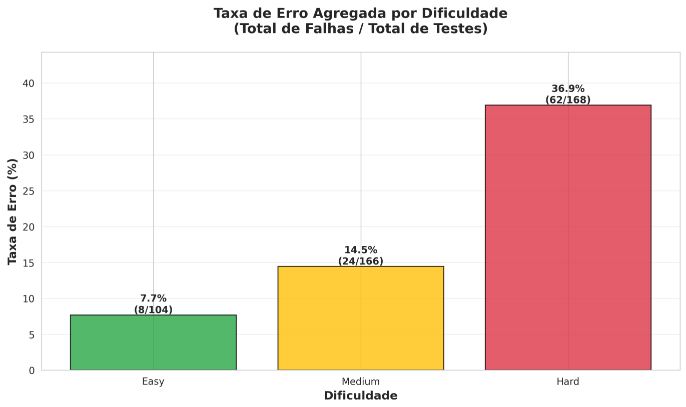

Figure 3. Aggregated Error Rate by Difficulty. Reliability drops sharply as algorithmic complexity increases.

This confirms the hypothesis that high cyclomatic complexity challenges the LLM's reasoning capabilities, leading to "hallucinated" tests that cover lines but fail to assert correct behavior.

5.3 Domain-Specific Analysis (Heatmaps)

To understand precisely where these failures occur, we contrast the Code Coverage Heatmap with the Error Rate Heatmap.

As seen in Figure 4, coverage is generally green (high) across most categories. However, Figure 5 reveals the underlying reality: specific domains like **Binary Search** and **Arrays** (Hard) present critical failure points. For instance, Binary Search problems maintained ≈89% coverage but suffered from a ≈59% error rate.

5.4 Cost Efficiency

Finally, the economic viability was analyzed. Figure 6 demonstrates a linear increase in cost associated with difficulty.

The model requires the highest financial investment ($0.0735) to generate tests for Hard problems, which paradoxically yield the lowest reliability. This results in a poor cost-benefit ratio for complex algorithmic generation.

6 Limitations and Future Work

This study provided a focused evaluation of GPT-4's capabilities; however, several limitations and avenues for future research were identified.

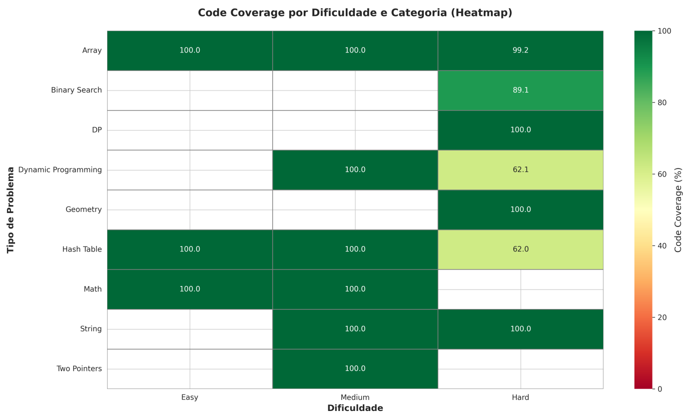

Figure 4. Code Coverage Heatmap.

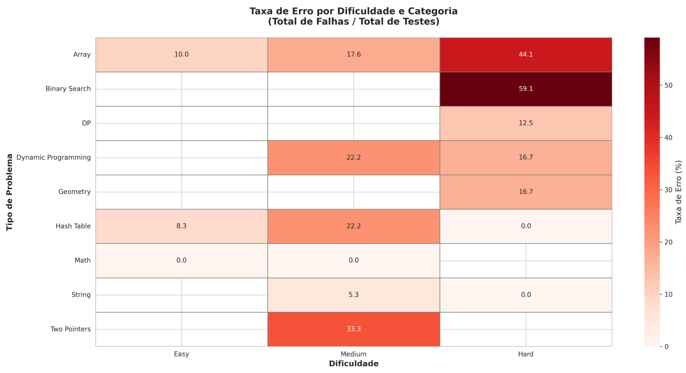

Figure 5. Error Rate Heatmap.

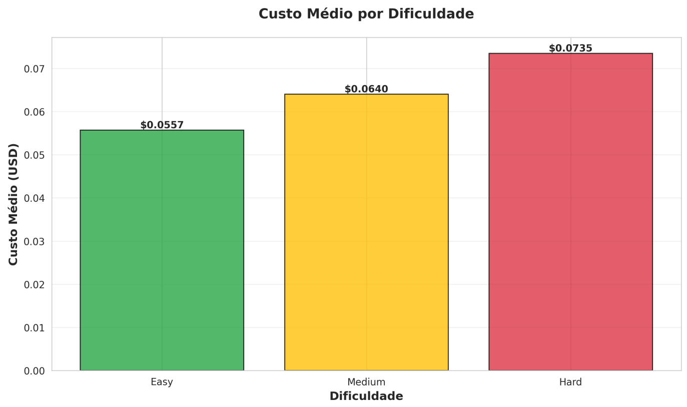

Figure 6. Average Cost by Difficulty (USD). Harder problems are more expensive due to verbose solutions and longer contexts.

6.1 Limitations

The primary limitation of this study was the **financial cost associated with API usage**. As analyzed in the results and reinforced here, the computational cost per test generation increases linearly with problem complexity due to longer context windows and more verbose outputs.

As evidenced in Figure 6, the model requires significantly higher investment for "Hard" problems compared to "Easy" ones. Conducting a large-scale analysis with thousands of problems or performing multiple iterations (retries) per problem would require a significantly larger budget. This financial constraint directly limited the size of our dataset and the ability to perform extensive regression testing.

Additionally, the reliance on LeetCode problems constitutes a scope limitation. While excellent for algorithmic benchmarking, isolated LeetCode functions may not fully represent the messy, interdependent nature of enterprise software development, which often involves mocking database connections, API calls, and complex class hierarchies.

6.2 Future Work

To address these limitations and expand the body of knowledge, the following steps are proposed:

- **Evaluation of Open Source Projects:** Future studies should apply this methodology to real-world JavaScript repositories (e.g., from GitHub). This would assess how LLMs handle dependencies, mocking, and architectural complexity rather than just algorithmic logic.
- **Comparison of Models:** Expanding the benchmark to include other state-of-the-art models, such as Claude 3.5 Sonnet, Llama 3, or distinct GPT-4 versions, to determine if the observed limitations are intrinsic to LLMs or specific to the model architecture used.
- **Automated Test Repair:** Investigating "Self-Healing" workflows where the LLM is fed back the error message from the failed test execution to see if it can iteratively correct its own mistakes, potentially mitigating the high error rates in complex scenarios.

Availability of Data and Materials

The source code for the *Unit Cloud Gen* platform, developed and used in this study, is open-source and publicly available at the following GitHub repository: https://github.com/JoaoVitorSD/unit-cloud-gen.

Declaration of Generative AI Use

The author acknowledges the use of generative artificial intelligence tools to assist in the translation of specific sections of this manuscript from Portuguese to English. The conceptualization, data analysis, and final validation of the text remain the sole responsibility of the author, in accordance with the academic guidelines.

References

[1] N. Alshahwan et al., "Automated Unit Test Improvement using Large Language Models at Meta," in *Proceedings of the 32nd ACM Symposium on the Foundations of Software Engineering (FSE '24)*, 2024.

[2] P. Bareis et al., "Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code," arXiv preprint arXiv:2206.01335, 2022.

[3] G. Bavota et al., "Test smells in practice: A study on 120 Java projects," in *Proc. of the Working Conference on Reverse Engineering*, 2015.

[4] T. B. Brown et al., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, 2020.

[5] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.

[6] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *25th International Symposium on Software Reliability Engineering*, 2014.

[7] E. Daka et al., "Modeling readability to improve unit tests," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[8] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[9] F. Palomba et al., "On the diffusion of test smells in automatically generated test code," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, 2016.

[10] A. Panichella et al., "Revisiting test smells in automatically generated tests," in *IEEE International Conference on Software Maintenance and Evolution*, 2020.

[11] M. Schäfer et al., "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," arXiv preprint arXiv:2302.06527, 2023.

[12] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011.

[13] M. Tufano et al., "Unit test case generation with transformers and focal context," arXiv preprint, 2021.

[14] Z. Yuan et al., "No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation," arXiv preprint arXiv:2305.04207, 2023.