Test-Driven Prompting: An Empirical Evaluation of Methods Reconstruction based on Unit Tests

Guilherme Mota Bromonschenkel Lima
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais
guilhermemota@dcc.ufmg.br

João Eduardo Montandon
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil
joao@dcc.ufmg.br

Abstract

Unit tests are a fundamental component of modern software development. Besides their role in verifying the correctness of code, developers also rely on them to document and specify the expected behavior of methods and functions. We argue that Large Language Models (LLMs) can also benefit from them to generate correct source code. This paper investigates to what extent unit tests can improve method generation when they are provided as context to LLMs. For this, we designed a prompting approach that iteratively provides relevant unit tests to the model, requesting it to reconstruct the implementation of a given method based on the provided tests. We generated new implementations for 35 methods from two popular open-source JavaScript/TypeScript repositories, and evaluated their functional equivalence against their original implementations. Our results show that providing unit tests as context increases the number of passing tests by up to 95 p.p, showing that unit tests are a valuable source of information for LLMs when generating code.

1 Introduction

Automated tests are a cornerstone for modern software development. Developers rely on them to verify the correctness of new features under implementation [3], detect regressions during software maintenance [11], and ensure the shipped product meets the expected quality standards. Automated tests come in many flavors, such as unit, integration, and system tests, each fulfilling different purposes and levels of granularity [3, 5, 25].

Unit tests play a central role by verifying individual methods and functions in isolation, thus making them the most common type of automated test in software projects [3]. Beyond their verification purpose, unit tests also define the interface and intended behavior of the methods under test, providing concrete examples about how they should be called and which results are expected given certain inputs [12, 26]. This dual role has enabled development approaches such as Test-Driven Development (TDD), where tests describing the expected behavior and interface for given a method are written before its implementation, thereby satisfying method's correctness and specification simultaneously.

Since they first emerged, Large Language Models (LLMs) have shown remarkable capabilities at source code generation [4, 22]. This potential has been explored in the literature, with studies investigating the use of LLMs for generating code from natural language descriptions [19], tests from code [22], and even migrating code between programming languages and components [1, 20]. Yet, generating source code with quality and precision remains a challenge [17].

In the same way developers rely on unit tests as a behavior and structure reference to implement new features from tests, we advocate LLMs can take advantage on the contextual information provided by unit tests to generate methods that are more likely to satisfy the expected syntax and behavior. To put this hypothesis to the test, **we evaluate to what extent unit tests can improve method generation by LLMs when provided as context**. We designed a prompting approach, called Test-Driven Prompting, that iteratively requests the model to reconstruct the implementation of a given method based on a set of relevant tests. This experiment was conducted with two models from the Gemini family, regenerating the source code of 35 methods from two popular open-source JavaScript and TypeScript repositories. We then measured the quality of these generated methods by executing the full test suite of their original implementation.

After executing 516 experiments across the two repositories, we observed that providing pre-existing tests is most beneficial for methods with non-trivial logic and low baseline pass rate, where gains reach up to 95 p.p., while well-covered methods show marginal or no improvement. We also found that the number of test assertions consistently influenced the method generation quality, while the size of the test negatively affected it.

The contributions of this paper are twofold. First, we present Test-Driven Prompting, a prompting approach that iteratively provides pre-existing unit tests as context to LLMs for method generation. Second, we provide empirical evidence on to what extent unit tests can improve method generation quality when provided as context, and which structural characteristics of unit tests are more informative for the model. The remainder of this paper is organized as follows. Section 2 presents the research questions guiding the study. Section 3 describes the study design, covering project selection, method and test selection, the models evaluated, and the Test-Driven Prompting workflow. Section 4 reports the results for each research question. Section 5 discusses threats to validity. Section 6 reviews related work. Section 7 concludes the paper and outlines directions for future work.

2 Research Questions

The goal of this work is to evaluate the influence of unit tests on the task of method generation using LLMs. Inspired by the Goal-Question-Metric (GQM) approach [2], we propose three research questions to guide the evaluation.

RQ1. How relevant are unit tests for method generation? To answer this question, we designed Test-Driven Prompting, an approach that repeatedly requests the model to generate a given method from a set of relevant tests (Section 3.5). For each iteration, the number of tests provided in the prompt increases by one, starting

with a prompt without tests and ending with a prompt containing all relevant tests. We execute the generated method against the original test suite and calculate the test case pass rate for each iteration. We then analyze how the pass rate changes as more tests are provided in the prompt. This strategy allows us to quantify the impact throughout the progressive addition of tests, identifying saturation points where additional tests no longer yield significant improvements, or even detect negative effects due to prompt overload.

RQ2. Which types of tests contribute most? In this question, we analyze which structural characteristics of typical unit tests are more informative for the model when provided as context. For each relevant test, we collect (a) number of lines; (b) number of assertions; and (c) number of variables, and correlate them with their individual contribution to the test case pass rate. This way, we can observe which of these characteristics are more effective as specifications for method generation.

RQ3. How does model choice affect method generation? The goal of this question is to measure the influence of the model type on the ability to generate methods with our proposed approach. Specifically, we executed the full experiment for each model separately, but with the same set of relevant tests. We then compare the pass rate difference between each test configuration, and analyze which model performs better.

3 Study Design

Our experiment is designed in a four-step workflow: (a) project selection; (b) selection of methods under test; (c) identification of relevant tests; and (d) method generation. Figure 1 depicts these steps, which are detailed below.

3.1 Project Selection

For this empirical study, we limited our analysis to JavaScript and TypeScript repositories, as these languages are widely used in the industry and have many popular projects with comprehensive test suites, essential for our evaluation. On October 1, 2025, we manually assembled an initial set of 22 open-source JavaScript and TypeScript repositories based on our perception of relevance and recognition within their respective ecosystems. Each candidate was then evaluated against the following criteria, applied in sequence:

- **Popularity:** having a minimum of 30,000 GitHub stars, as projects with greater adoption and maturity tend to be used in real-world scenarios, favoring the applicability of the results;
- **Domain diversity:** falling into one of the following domains: platform, library, framework, or back-end service, as this ensures that results are not biased by characteristics specific to a single type of system;
- **Executability:** having a test suite that is executable in a local environment with all tests passing, ensuring our experiment can be executed successfully;
- **Coverage:** having at least 200 methods with 100% line, statement, and branch coverage, as this threshold ensures a sufficiently large set to support the subsequent method selection for our experiment.

Among the 22 repositories evaluated, only two met all criteria and were selected for the experiments. Table 1 presents their characteristics.

Table 1: Repositories selected for the experiments

Repository	Domain	Stars	Eligible methods*
date-fns [7]	Library	36,152	234
Directus [9]	Back-end service	32,874	309

*Methods with 100% line, statement, and branch coverage.

3.2 Method Under Test Selection

Since we depend on the pre-existing test suite to properly generate methods equivalent to their original implementation, we must ensure that the methods included in our experiment are well-covered by their respective test suites. To this end, we selected methods that met the following criteria:

- **Suite uniqueness:** being referenced by only one test suite file, avoiding methods whose tests belong to suites not directly related to their behavior;
- **Test count:** having more than five tests in the suite, as this is considered the minimum number required to ensure variability across experiments;
- **Test coverage:** achieving 100% line, statement, and branch coverage, ensuring that all execution paths of the method are exercised by the tests.

3.3 Relevant Test Selection

As our approach heavily relies on the tests provided as context, we must select tests that are relevant to the method being generated. Irrevelant or poorly written tests can introduce noise into the prompt, thus negatively impacting the model's performance [17]. To ensure we are feeding the model with proper tests, we apply Mutation Testing [8] to identify which tests do effectively the behavior of the method under test.

For this, we executed Mutation Testing using the StrykerJS tool [24] for each method, and recorded which tests were able to detect at least one mutant in the method body. Afterwards, we filtered the methods based on the number of relevant tests, retaining only those with at least three relevant tests; this threshold ensures that our Test-Driven Prompting approach can be executed at least three times for the same method. Table 2 presents the final set of methods and their relevant tests per repository, which will be used in the experiments.

Table 2: Final set of methods and their relevant tests per repository

Repository	Selected methods	Relevant tests
date-fns	23	124
Directus	12	76
Total	**35**	**200**

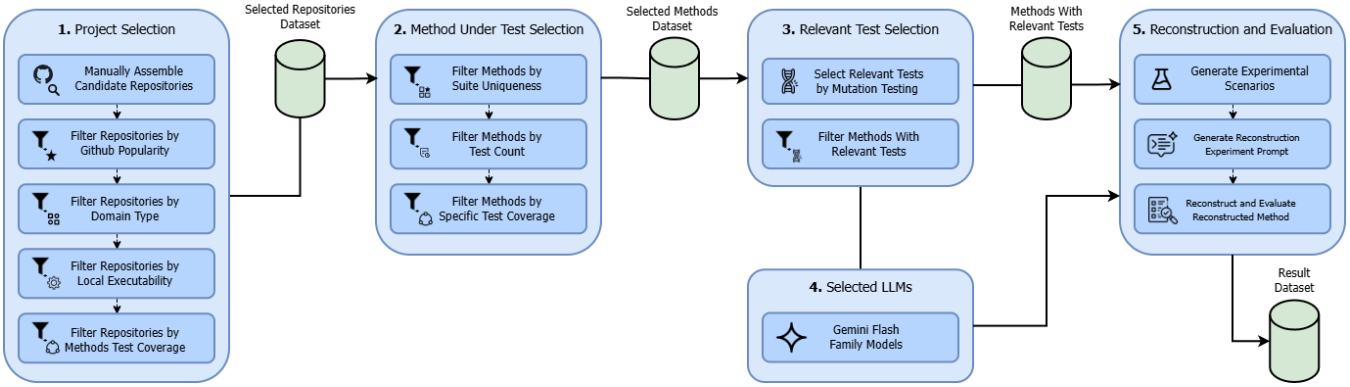


Figure 1: Overview of the study design.

3.4 Selected LLMs

The experiments were conducted using two models from Google's Gemini family: `gemini-2.5-flash` [6, 13] and `gemini-3-flash-preview` [14]. The choice of this model family was guided by three main characteristics: (a) a long context window (1 million tokens), allowing multiple relevant tests to be provided in the method generation prompt without truncation; (b) accessibility, as both models are available via API with a free usage tier [15]; and (c) they are considered state-of-the-art models provided as a service, with consistent results across software development tasks.

We opted for two models from the same family to isolate the effect of reasoning capabilities, while keeping other characteristics such as context window and API the same across models. The `gemini-2.5-flash` model can be configured without thinking support in the free tier, whereas `gemini-3-flash-preview` must have thinking enabled to be used [16]. All experiments were conducted on the same day—May 28, 2026—to ensure that both models were evaluated under the same conditions, as they are accessed through closed APIs and lack explicit version information.

3.5 Test-Driven Prompting Workflow

Here we describe in detail the Test-Driven Prompting workflow, a prompting strategy designed to evaluate the influence of unit tests on method generation by LLMs. In essence, this approach mimics the first step of the red-green-refactor cycle of TDD where, given a set of tests previously written for a method, a developer would implement the method in order to make those tests pass. In our case, the LLM takes the role of the developer and generates the method body based on the context provided by the relevant tests.

Workflow Overview. Considering a tuple $\langle m_i, T_i \rangle$ where m_i is a method selected for the experiment (Section 3.2) and T_i is the set of relevant tests for that method (Section 3.3), the workflow applies the following steps.

❶ ***Permutation Generation.*** Let $n = |T_i|$, i.e., the number of relevant tests for method m_i. We generate $n + 1$ experimental scenarios by varying the subset of tests provided as context:

- **No relevant tests (baseline):** no tests are provided as context. We use this scenario as a baseline to measure the model's performance without any relevant tests, allowing us to quantify the impact of providing tests as context;
- **Permuted relevant tests:** We leverage $n - 1$ different configurations of relevant tests to be passed as context to the model. Each configuration is a permutation of k tests randomly sampled without replacement from T_i, with k ranging from 1 to $n - 1$, each adding one more test to the context;
- **All relevant tests:** the complete set T_i (n tests) is provided as context.

❷ ***Prompt Construction.*** We assemble a two-part prompt following the template shown in Figure 2. The system prompt is fixed across all scenarios. The user prompt contains three sections: the name of the method to be reconstructed (`{{method_name}}`), the full content of the implementation file with the method body removed (`{{implementation_file}}`), and the subset of relevant tests provided as context (`{{relevant_tests_subset}}`).

Prompt Template

```
You are an expert software developer specialized in
reconstructing method implementations in a strict TDD workflow.
Reconstruct only the method body using exclusively the provided
inputs. Constraints: (i) preserve the original signature; (ii)
use no external knowledge; (iii) use only symbols present in the
provided file; (iv) output only the method body, with no
Markdown fences or explanatory text.

# Method Name
{{method_name}}

# Method File Content Without Method Body
{{implementation_file}}

# Context
{{relevant_tests_subset}}
```

Figure 2: Prompt template used in each LLM request.

❸ ***Reconstruction and Evaluation.*** We submit the prompt to the model, which returns the reconstructed method body. We dynamically injected the generated body of m_i into the source code file via the AST processor (*ts-morph* [23]), and replaced its original

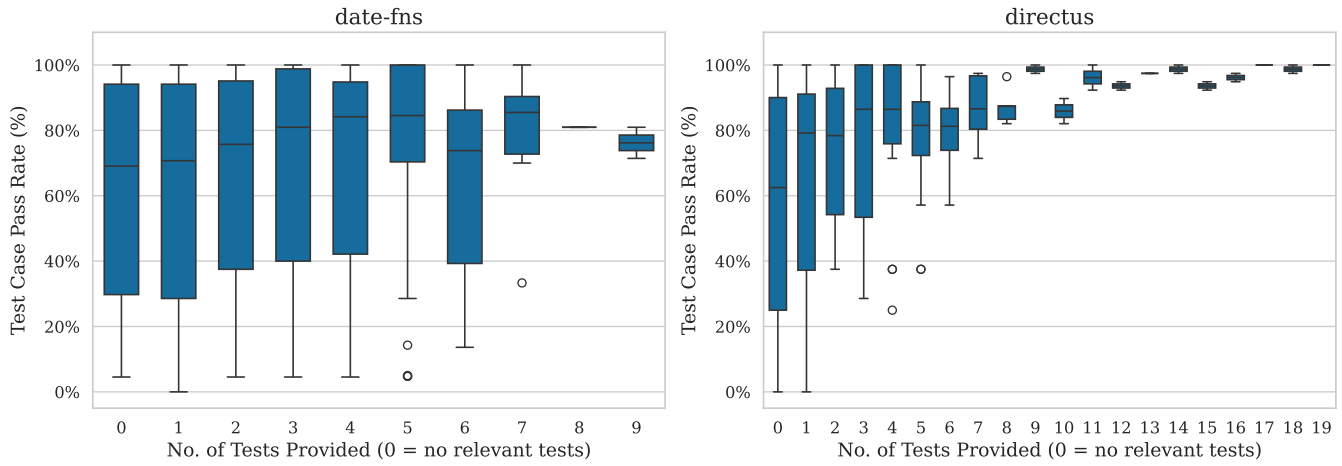


Figure 3: Distribution of suite approval and test case pass rate × Number of tests in context.

implementation. The full test suite associated with m_i is then executed on the new generated method; cases where the body can not be parsed, or where suite execution yields a syntax error, are flagged as *compilation failures*.

We rely on the test suite execution results to evaluate the quality of the generated method. Specifically, we recorded the number of passing and failing test cases and calculated the **test case pass rate** for each experimental scenario. The generated method is considered equivalent to its original implementation if it passes all test cases in the suite; note that the original method is guaranteed to pass all tests (Section 3.2).

4 Results

Table 3 presents the consolidated results of the experiments, organized by repository and context composition strategy. Overall, the models achieved a test case pass rate of 69.7%, a compilability rate of 75.2%, and a fully passing suite rate of 22.3%, across the 35 evaluated methods. `Directus` achieved higher performance over all metrics, with 45.8% of its experiments passing all test cases, a test case pass rate of 83.4% on average, and a compilability rate of 83.3%; `date-fns` showed lower and less consistent performance, with 23.9% of its experiments passing all test cases, a test case pass rate of 66.0% on average, and a compilability rate of 71.7%.

Table 3: Overall results by repository.

	date-fns	Directus
Fully passing suite rate	23.9%	**45.8%**
Test case pass rate	66.0%	**83.4%**
Compilability rate	71.7%	**83.3%**

4.1 RQ1. How relevant are unit tests for method generation?

Figure 3 presents the distribution of the test case pass rate for each project, separated by the number of relevant tests provided as context. The maximum number of relevant tests per method is 9 in `date-fns` and 19 in `Directus`, with the median being five for both.

date-fns. The test case pass rate shows a slight but consistent improvement in the beginning, reaching its peak at $N = 5$ (84.5%, median). The effectiveness then reduces for scenarios with higher number of tests; the median drops to 73.8% at $N = 6$ and oscillates between 76.2% and 85.5% for further configurations.

Directus. The test case pass rate shows a more regular and consistent improvement as more tests are provided. Starting from 62.5% ($N = 0$), the median of test case pass rate suddenly increases to 86.4% at $N = 4$, and remains above 80% for all subsequent configurations, reaching 100% at $N = 17$ and $N = 19$. We observe an even clear improvement for the lowest quartile—boxplot whiskers—which starts at 25% ($N = 0$) and reaches 100% ($N = 19$).

4.2 RQ2. Which types of tests contribute most?

We analyzed structural characteristics of the relevant tests and mapped them to the test case pass rate obtained previously. We considered three characteristics: (a) number of lines, (b) number of assertions, and (c) number of variables. We also calculated the Pearson correlation coefficient between each characteristic and the test case pass rate, to quantify the strength of the relationship. Figure 4 presents this distribution for each repository.

Method Size. We observe a negative correlation between the size of the method and the test case pass rate in both repositories, with `date-fns` presenting a weak correlation ($r = -0.08$) and `Directus` a moderate one ($r = -0.16$). This reduction in performance is especially noticeable for test cases exceeding 10 lines in both repositories: in `date-fns`, the pass rate drops visibly from the 11–15 group onward, while in `Directus`, a similar decline is observed in the same range, becoming more pronounced for test cases with more than 15 lines.

Number of Assertions. The number of assertions presents distinct behaviors across repositories. In `date-fns`, the range is narrow (1 to 3 assertions) and the vast majority of test cases have only one

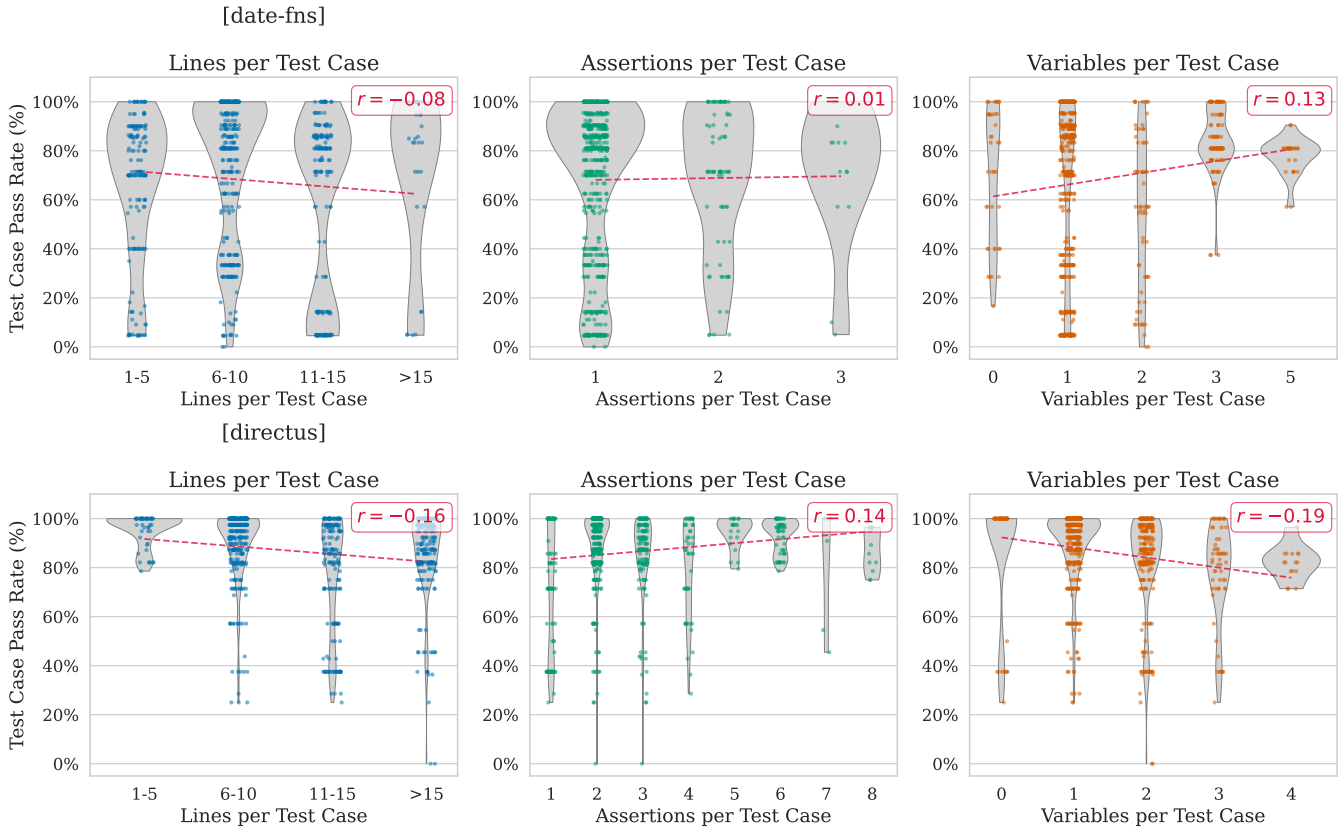


Figure 4: Test pass rate for each permutation by the structural characteristics of the tests.

assertion, making the trend line practically flat (r = 0.01). Differently, `Directus` presents a varied number of assertions, ranging from one to eight. For this repository, we observe a positive correlation between the number of assertions and the test case pass rate (r = 0.14), suggesting that test cases with more assertions provide richer behavioral information to the model, thereby improving its ability to generate methods that satisfy the expected behavior.

Number of Variables. Surprisingly, the number of variables indicate opposite behaviors across repositories. While `date-fns` shows a positive correlation (r = 0.13) between the number of variables and the test case pass rate, `Directus` presents a negative one (r = −0.19). At first glance, the absence of a consistent direction indicates that this characteristic does not influence models performance.

4.3 RQ3. How does model choice affect method generation?

To answer this question, we analyze the performance divergence between the two models across all permutations of relevant tests for each method. Such comparison is possible since the same permutations—hence, the same Test-Driven Prompting scenarios—are executed for both models, thus allowing a direct comparison of their performance in identical scenarios. Figure 5 presents the performance difference—measured as the delta in test case pass rate—between `gemini-3-flash-preview` and `gemini-2.5-flash` for each permutation. For each repository, we ordered the permutations by the delta value, keeping most favorable cases for `gemini-2.5-flash` in blue on the left, and the most favorable ones for `gemini-3-flash-preview` in yellow on the right.

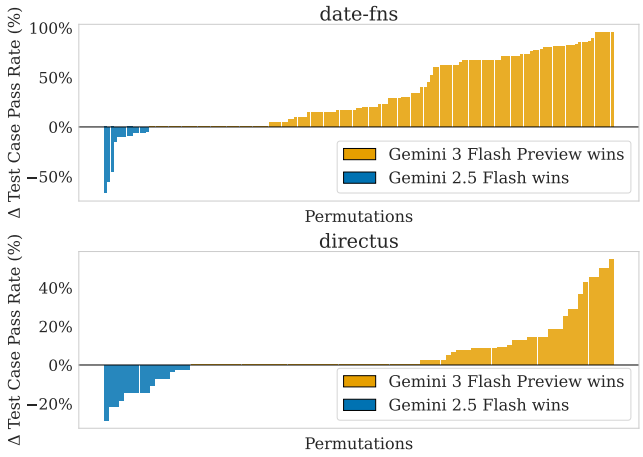


Figure 5: Performance divergence between models per permutation.

date-fns. 90 out of the 135 (67%) permutations show a positive delta, indicating `gemini-3-flash-preview` performs better. This superiority is also notable in magnitude, as the delta towards `gemini-3-flash-preview` is higher than 60% for 48 permutations, achieving 95.5% in the most favorable scenarios. By contrast, `gemini-2.5-flash` performs better in only 14 permutations, with a maximum delta of 66.7% in the most favorable scenario. Both models achieved equivalent performance in 31 cases.

Directus. From all 88 permutations, `gemini-3-flash-preview` performs better in 34 cases, `gemini-2.5-flash` in 15, and both models achieve same performance in 39; a more balanced distribution than in `date-fns`. However, the delta magnitude substantially favors `gemini-3-flash-preview`. While it achieves a maximum delta of 54.5% in its most favorable scenarios, `gemini-2.5-flash` reaches only 28.6%, i.e., almost half the advantage of its competitor.

5 Threats to Validity

Test Suite Quality. The test suites used in this experiment were implemented by project maintainers, and their quality and coverage may vary. To mitigate this risk, we developed a strategy based on mutation testing to identify and select tests that actually evaluate the behavior of the methods under analysis, thus ensuring the tests provided as context are meaningful.

Limitations of the Permutation Strategy. Our permutation strategy generates set of relevant tests based on random sampling, i.e., the selected tests at first context sizes may include tests with lower representativeness. In this case, the observed differences across permutations can be influenced by the composition of the selected tests. Another threat concerns the LLMs selected for this experiment; we analyzed two models from the same family (`gemini-2.5-flash` and `gemini-3-flash-preview`), thus the results may not generalize to others.

Repository Selection Bias. We manually assembled the initial pool of 22 JavaScript/TypeScript repositories based on our perception of relevance and recognition within this ecosystem and, from this pool, we selected two repositories that met the eligibility criteria for the experiment. This selection process may introduce a bias toward well-known, well-maintained projects. Furthermore, we selected only methods that are fully covered by the test suite, which may introduce a bias toward well-tested methods. Therefore, our experiment results may not generalize to other repositories, programming languages, or methods with lower test coverage. We intend to extend the study to include more repositories and different method types in future work to enhance the generalizability of our findings.

6 Related Work

Mathews and Nagappan [21] investigate whether providing LLMs with test cases alongside problem statements improves code generation outcomes. Their results show that including test cases leads to higher success rates in solving programming challenges, supporting TDD as a way to ensure that LLM-generated code captures the intended requirements. The present work extends the investigation to a real-world case study: rather than synthetic benchmarks, we evaluate method generation against pre-existing test suites from production repositories, using tests as the sole behavioral specification provided to the model.

Fakhoury et al. [10] propose TiCoder, an interactive workflow that uses tests to clarify user intent and improve code generation accuracy. Through a user study with programmers, they show that iterative test-based feedback leads to significant accuracy improvements across multiple LLMs, achieved within just a few interactions. The present work differs from TiCoder in that it relies on an iterative feedback loop, whereas our study evaluates single-shot generation, measuring the model's ability to interpret pre-existing tests as specification without refinement cycles.

Jia and Harman [18] provide a comprehensive survey of mutation testing as a fault-based testing technique. The mutation adequacy score measures how well a test set can detect faults, based on the idea that the faults used in mutation testing reflect common programming mistakes. Unlike structural coverage criteria, which only verify that code paths are executed, mutation testing assesses whether tests can distinguish between different program behaviors. The present work leverages this property in the method selection process: we apply StrykerJS to filter tests that effectively discriminate the behavior of each method, ensuring that the tests provided to the model carry meaningful behavioral information.

7 Conclusion

This work investigated how, and to what extent, unit tests improve method generation by LLMs when provided as context in the prompt. To this end, we designed an iterative prompting strategy—called Test-Driven Prompting—that feeds the model with an increasing number of relevant unit tests for the method to be generated, and evaluates whether the generated method is functionally equivalent to its original implementation. We performed a controlled experiment with two state-of-the-art LLMs and two open-source repositories. After analyzing 35 methods with pre-existing test suites and 516 executions, we found that providing unit tests as context consistently improves the quality of the method being generated. The highest gains are observed in methods which presented low pass rates without tests.

As future work, we intend to (a) extend this experiment in other ecosystems; (b) add more, open-weight, models to the study; and (c) improve our strategy to include a feedback loop with test failures, enabling the model to refine its implementation from this feedback.

Replication Package. The replication package for this study is available at https://doi.org/10.5281/zenodo.21183486, including the scripts for data collection and analysis, as well as the raw data and results of the experiments.

Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. It was also supported by grants from CNPq (403304/2025-3) and by FAPEMIG (APQ-02419-23).

References

[1] Altino Alves, João Eduardo Montandon, and Andre Hora. 2026. Testing Framework Migration with Large Language Models. In *7th ACM/IEEE International*

Conference on Automation of Software Test (AST). 1–11.

[2] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. The Goal Question Metric Approach. *Encyclopedia of Software Engineering* (1994).

[3] Kent Beck. 2003. *Test Driven Development: By Example*. Addison-Wesley Professional, Boston.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374 [cs]* (July 2021).

[5] Mike Cohn. 2010. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, Upper Saddle River, NJ.

[6] G. Comanici et al. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[7] date-fns. 2026. date-fns: Modern JavaScript Date Utility Library. https://github.com/date-fns/date-fns. Accessed June 2026.

[8] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41.

[9] Directus. 2026. Directus: The Modern Data Stack. https://github.com/directus/directus. Accessed June 2026.

[10] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2254–2268. doi:10.1109/TSE.2024.3428972

[11] Michael Feathers. 2004. *Working Effectively with Legacy Code* (1st ed.). Pearson.

[12] Mohammad Ghafari, Carlo Ghezzi, Andrea Mocci, and Giordano Tamburrelli. 2014. Mining unit tests for code recommendation. In *Proceedings of the 22nd International Conference on Program Comprehension*. 142–145.

[13] Google. 2025. Gemini 2.5 Flash. https://ai.google.dev/gemini-api/docs/models/gemini-2.5-flash. Accessed June 2026.

[14] Google. 2025. Gemini 3 Developer Guide. https://ai.google.dev/gemini-api/docs/gemini-3. Accessed June 2026.

[15] Google. 2025. Gemini API Pricing. https://ai.google.dev/pricing. Accessed June 2026.

[16] Google. 2025. Thinking — Gemini API. https://ai.google.dev/gemini-api/docs/thinking. Accessed June 2026.

[17] Cristina Improta, Rosalia Tufano, Pietro Liguori, Domenico Cotroneo, and Gabriele Bavota. 2025. Quality In, Quality Out: Investigating Training Data's Role in AI Code Generation. In *33rd International Conference on Program Comprehension (ICPC)*. 454–465.

[18] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[19] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology* 35, 2 (2026), 58:1–58:72.

[20] Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R. Cogo, and Bram Adams. 2025. InterTrans: Leveraging Transitive Intermediate Translations to Enhance LLM-based Code Translation. In *47th International Conference on Software Engineering (ICSE)*. 1153–1164.

[21] Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-Driven Development and LLM-based Code Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. 1583–1594. doi:10.1145/3691620.3695527

[22] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024).

[23] David Sherret. 2026. ts-morph: TypeScript Compiler API Wrapper for Static Analysis and Programmatic Code Changes. https://github.com/dsherret/ts-morph. Accessed June 2026.

[24] Stryker Mutator. 2026. StrykerJS: Mutation Testing for JavaScript and TypeScript. https://stryker-mutator.io. Accessed June 2026.

[25] Matt Wynne, Aslak Hellesoy, and Steve Tooke. 2017. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, Raleigh, North Carolina.

[26] Zixiao Zhu, Yanzhen Zou, Bing Xie, Yong Jin, Zeqi Lin, and Lu Zhang. 2014. Mining API Usage Examples from Test Code. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 301–310.