

Evaluating LLM-Generated and Human-Written Unit Tests for Iterative Code Refinement

Heitor Fernandes Vicente Machado
Universidade Federal da Fronteira Sul
Chapecó – SC, Brazil
heitor.machado@estudante.uffs.edu.br

Prof. Dr. Samuel da Silva Feitosa
Universidade Federal da Fronteira Sul
Chapecó – SC, Brazil
samuel.feitosa@uffs.edu.br

Abstract

Large Language Models have shown promising results in automatic code generation, but challenges still exist related to the quality of the produced implementations and the effective use of testing during the generation process. This work investigates the use of unit tests automatically generated by LLMs in contrast with manually crafted tests as an iterative code refinement mechanism for programming problems. The proposed approach consists of a tool capable of generating unit tests from a task specification, validating implementations produced by the models, and performing successive correction cycles guided by the test results. The methodology was evaluated using the HumanEval benchmark in conjunction with the EvalPlus framework, considering scenarios of direct generation, self-validation with automatically generated tests, and validation using the benchmark’s original tests. The evaluation considered the Pass@1 and Pass@10 metrics to measure problem-solving performance, as well as the analysis of the generated test suites, classified as valid, invalid, or not produced. The results demonstrate that synthetic tests are capable of improving the performance of all evaluated models, although they still present lower performance compared to that obtained with manually crafted tests. The findings indicate that automatic test generation by LLMs represents a viable strategy to support code generation and refinement processes.

Keywords

Large Language Models, code generation, unit tests, HumanEval, iterative refinement, software engineering, test generation.

1 Introduction

In recent years, *Large Language Models* (LLMs) have stood out as promising tools for automatic code generation and for supporting programming problem resolution. Traditional evaluations of these models generally rely on benchmarks such as HumanEval and MBPP, in which the proposed solutions are verified against previously defined test suites, where the model’s performance is usually measured by metrics such as *pass@k*, which evaluates the success rate considering different generation attempts.[4, 8]

One aspect already explored in the literature is the influence that additional information provided to the model can have on the quality of the generated solutions for programming problems. When only the textual description of the requirement is used, the model must deduce the input and output conditions on its own, which can limit the completeness of the solutions. Recent studies also suggest that providing test cases can contribute to greater accuracy by making explicit, through unambiguous language, the requirements and constraints that are not always clear in the problem statement [9].

The elaboration of test cases before the development of a requirement is already a consolidated practice in the software engineering field, being formally explored through the *Test-Driven Development* (TDD) paradigm. In this paradigm, development is driven by the prior writing of tests, which act as executable specifications and guide code implementation [1]. Similarly, when considering the use of LLMs for code generation, test cases can assume the role of formal specifications, guiding the model in producing solutions aligned with the desired requirements.

However, although several studies demonstrate that tests can improve the performance of automatic code generation, it is not yet clear to what extent tests automatically generated by LLMs are capable of replacing or approximating the effectiveness of manually crafted test suites. Since test quality directly influences the ability to identify defects and guide corrections, incorrect or incomplete tests can limit the benefits obtained during the iterative refinement of implementations.

In this work, we investigate the impact of using unit tests automatically generated by LLMs as an iterative code refinement mechanism. To this end, an approach consisting of two main stages was developed: the automatic generation of unit tests and the iterative generation of implementations guided by these tests.

To validate the approach, experiments were conducted using the HumanEval benchmark in conjunction with the EvalPlus framework. The evaluation compares three distinct scenarios: direct generation without intermediate validation, refinement guided by automatically generated tests, and refinement guided by manually written tests extracted from the benchmark. The main objective is to analyze whether synthetic tests can provide benefits similar to those obtained with human-crafted tests, considering both the quality of the produced test suites and the impact of these tests on the final performance of the code generation models.

2 Theoretical Background

This section presents the theoretical foundations that support the present study. The principles of *Test-Driven Development* (TDD) and its role in software development are discussed, as well as the concepts related to *Large Language Models* (LLMs), highlighting the *Transformer* architecture and its capability for automatic code generation. Furthermore, the main aspects of the benchmarks used in the evaluation of these models are presented, emphasizing HumanEval as a reference to measure their performance in programming tasks.

2.1 Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development practice that seeks to incrementally integrate the test and feature development phases in a project [2]. In this method, unit tests are developed before the implementation of the code that must satisfy these tests.

The main objective of this practice is that, by writing tests that describe the expected behavior of a system, the developer is naturally guided to produce more decoupled, modular, and maintainable code [2], in addition to quickly identifying code snippets that do not produce the expected result. Thus, TDD tends to favor behavior-driven design, promoting a cleaner and more testable software structure compared to traditional approaches, in which tests are developed after the implementation of features [2].

2.2 Large Language Models (LLMs)

Large Language Models (LLMs) represent one of the fields in the area of Natural Language Processing (NLP) that enable machines to interpret and produce text in a coherent and contextualized manner, with their emergence linked to the development of the *Transformer* architecture, the increase in computational capacity, and the vast availability of training data [10]. LLMs differ from traditional language models by being trained on large textual corpora in a self-supervised manner, where the model learns general linguistic representations that can be applied to different tasks. This characteristic also allows the models emergent abilities such as reasoning, decision-making, and in-context learning [10].

2.2.1 Transformer Architecture. The *Transformer* architecture, introduced by Vaswani et al. [12], serves as the foundation for most modern language models. In this architecture, an attention mechanism is used to model the relationship between tokens given an input sequence. The architecture consists of two main components: the *Encoder*, which is responsible for transforming the input sequence into numerical representations, and the *Decoder*, which generates the output token by token.

Each Encoder layer has two blocks: *multi-head self-attention* and *feed-forward*. The *multi-head* mechanism allows different “heads” to learn different types of relationships between various tokens, enabling the model to learn syntactic and semantic patterns more effectively than previously used techniques [12].

2.2.2 Code Generation with Large Language Models. LLMs have become one of the primary methods for the task of automatic code generation. In this context, models are used to transform textual instructions, which may include functional specifications and/or usage examples, into code implementations [6].

Thus, a natural language description of a given problem can be represented as a textual sequence x , from which the model, parameterized by θ , produces a code y by modeling the conditional distribution $P_\theta(y | x)$ [6]. After generation, the code can be evaluated through execution in test environments, compilers, or specific validators, resulting in a performance metric. This cycle allows for verifying whether the generated code meets the requirements specified by the user.

2.3 Benchmarks for Language Models

A *benchmark*, in the context of language models, is a practice aimed at the systematic evaluation of an LLM’s capabilities performance through the use of standardized datasets and metrics [11]. In this process, the model is subjected to specific tasks, such as reading comprehension, logical reasoning, or response generation, allowing for an objective measurement of its generalization ability, coherence, and accuracy.

The main purpose of this practice is to ensure that the evaluation of a model occurs in a consistent and comparable manner across different architectures and approaches, contributing to the continuous improvement of models and training techniques [11]. Consequently, *benchmarks* foster a deeper understanding of the quality and robustness of LLMs [11].

2.3.1 HumanEval. *HumanEval* is a dataset of 164 handwritten programming problems created by OpenAI with the aim of evaluating the performance of code-trained models on programming tasks, where each problem contains a function signature with a *docstring*, an empty body, a set of associated unit tests, and a natural language description of the problem to be solved [4]. The problem set was designed to cover language comprehension, algorithmic reasoning, and basic mathematics tasks.

The *benchmark* uses unit tests to evaluate the solution proposed by the model, considering a solution correct when it passes all tests [4]. It also uses the *pass@k* metric, which estimates the probability that at least one of k generated samples solves a given problem. To reduce variance in the estimate, rather than sampling exactly k solutions, $n \geq k$ solutions are generated and the following unbiased estimator is applied [4]:

$$\text{pass@}k = \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

where n is the total number of generated samples, $c \leq n$ is the number of correct samples, and k is the number of attempts considered. The original HumanEval evaluation reports results for $k \in \{1, 10, 100\}$ [4]. The metric has since been widely adopted as a standard measure of code generation performance [3, 5, 7–9].

3 Related Work

Several recent studies have explored the application of *Large Language Models* (LLMs) in automatic code and test generation tasks, also proposing approaches that use test cases to guide, validate, or improve the solutions produced by the models.

Mathews and Nagappan [9] present the *TGen* framework, which integrates the *Test-Driven Development* (TDD) cycle into code generation by LLMs, providing the model with the problem statement along with public test cases and employing an iterative cycle of generation, execution, and correction based on failure feedback identified by *PyTest*. Evaluated with the *GPT-4 Turbo* and *Llama 3* models, the framework achieved gains of 12% on the *MBPP* benchmark and 9% on *HumanEval*, in addition to further improvements of 3% to 5% provided by the remediation process, producing more robust solutions capable of generalizing to initially unprovided tests. Unlike the approach proposed in this work, TGen only uses previously elaborated test cases, whereas the present research also

investigates the effectiveness of employing unit tests automatically generated by LLMs as an iterative refinement mechanism for implementations.

Chen et al. [3] proposed the *CODET (Code Generation with Generated Tests)* framework, in which solutions and test suites are simultaneously generated for self-evaluation, utilizing the concept of *Dual Execution Agreement* to select the best response based on the consistency among multiple solutions and the number of passed tests. The approach was motivated by the significant difference between the *pass@100* and *pass@1* metrics and demonstrated substantial improvements in benchmarks such as *HumanEval* and *MBPP*, achieving an (18.8%) increase in the *pass@1* metric compared to the *code-davinci-002* model without using the framework. While CODET uses the generated tests primarily to select and rank multiple candidate implementations, the present research employs these tests as an iterative code refinement mechanism, comparing their effectiveness with that of manually crafted tests.

Lahiri et al. [7] propose *TICODER*, a test-driven code generation tool that uses automatically generated test cases to formalize the user’s intent expressed in natural language. The method combines code suggestions and candidate tests, allowing the user to iteratively validate which tests correctly represent the desired behavior, while the system ranks both the tests and the generated solutions based on their discriminative power and the number of passed tests. Evaluated on the *MBPP* and *HumanEval* benchmarks, the workflow showed gains exceeding (22%) and (24%), respectively, with just one user interaction, achieving improvements above (37%) and (53%) after up to five interactions. In contrast, the present work adopts a fully automated process, in which tests generated by LLMs are used directly for the iterative refinement of implementations, without relying on user validation or intervention, in addition to comparing their effectiveness with that of manually crafted tests.

Lin Yang et al. [13] conducted an empirical study on the use of LLMs for automatic unit test generation, comparing *open-source* models, such as *CodeLlama* and *DeepSeek-Code*, and *closed-source* models, such as *GPT-4*, as well as the traditional *EvoSuite* tool. The work also investigated the impact of *prompt* design and techniques such as *Chain-of-Thought (CoT)* and *Retrieval-Augmented Generation (RAG)* on the quality of the generated tests. The results indicated that the *prompt* format exerts the greatest influence on the models’ performance, providing coverage gains especially for *CodeLlama*, while RAG-based approaches did not produce significant improvements and the use of CoT showed limited benefits, potentially even degrading performance in models with reduced context windows. In contrast, the present work does not focus on the isolated evaluation of the generated tests’ quality, but rather on investigating how their use as an iterative refinement mechanism influences the performance of code generation models, directly comparing them with manually crafted tests.

Cui [5] presents *WebApp1K*, a benchmark aimed at evaluating LLMs in test-driven development scenarios, in which the test cases themselves act as the specification of the expected behavior, replacing natural language descriptions. The benchmark encompasses web development tasks implemented in *React*, covering different domains, such as e-commerce, blogs, and social networks, and was used to evaluate 19 models, including *GPT-4o*, *o1-preview*, *Claude 3.5 Sonnet*, *DeepSeek-R1*, *Gemini 2.0*, and *Llama 3-8b*. The results

demonstrated better performance from models with advanced reasoning capabilities, such as *o1-preview*, while *open-source* models obtained lower results in the *pass@k* metric; furthermore, it was found that increasing the number of tests or the complexity of the tasks reduces the models’ performance due to the difficulty in handling extensive instructions and denser contexts. Unlike this approach, the present work uses tests as a mechanism for validation and iterative refinement of the generated implementations, further investigating whether test suites automatically produced by LLMs can play a role similar to that of manually crafted tests.

In general, the analyzed works highlight the growing interest in utilizing tests in conjunction with *Large Language Models* to improve automatic code generation tasks. The proposed approaches include the use of pre-existing test cases to guide the generation of solutions, the automatic generation of tests for self-evaluation or iterative refinement, the employment of tests as a specification of expected behavior (*tests as prompt*), and studies aimed at evaluating the capability of the models themselves to generate unit tests. In contrast, the present work aims to conduct a comparison of the impact of two distinct sources of tests (human-crafted and LLM-generated) as a guiding mechanism for code generation, seeking to evaluate whether synthetic tests can provide benefits similar to those obtained with traditional test suites.

4 Methodology

This work investigates the impact of using unit tests automatically generated by Large Language Models (LLMs) in solving programming problems. To this end, an approach consisting of two main stages was proposed: (i) automatic generation of unit tests and (ii) iterative generation of test-guided implementations. To enable this approach, an experimental tool was developed, responsible for orchestrating all stages of the process, including test generation, implementation validation, iterative refinement, and final model evaluation.

The experiments were conducted using the *HumanEval* benchmark in conjunction with the *EvalPlus* framework, which is responsible for expanding the original set of tests provided by the benchmark. The evaluation was carried out using the *Pass@1* and *Pass@10* metrics. For each task, the unit test suite is generated once and reused across all evaluation samples. Under *Pass@1*, a single implementation sample is produced per task, and the model has up to ten refinement iterations to produce a solution that passes all tests. Under *Pass@10*, ten independent implementation samples are generated per task, all using the same pre-generated test suite. Each sample independently undergoes the same iterative refinement process, with up to ten correction cycles.

4.1 Experimental Setup

The evaluated models were Qwen 3.5 2B, Qwen 3.5 4B, Gemma 4 E2B, and Gemma 4 E4B. These models were selected due to the competitive performance reported in recent code generation benchmarks, combined with their reduced computational requirements.

All experiments were executed using the following inference parameters: temperature = 0.2, top-p = 0.9, maximum number of tokens = 5000.

4.2 Evaluated Scenarios

The evaluation was conducted in three distinct scenarios:

- **Self-validation:** the unit tests used during the validation of the implementations are automatically generated by the model itself.
- **Oracle Validation:** the implementations are validated using the tests originally provided by the HumanEval benchmark.
- **Single Attempt:** the implementations are produced directly from the task statement, without any intermediate validation or test-based refinement step.

The comparison among these scenarios allows evaluating the effect of using automatically generated tests when compared both to manually written tests and to the absence of intermediate validation.

4.3 Automatic Test Generation

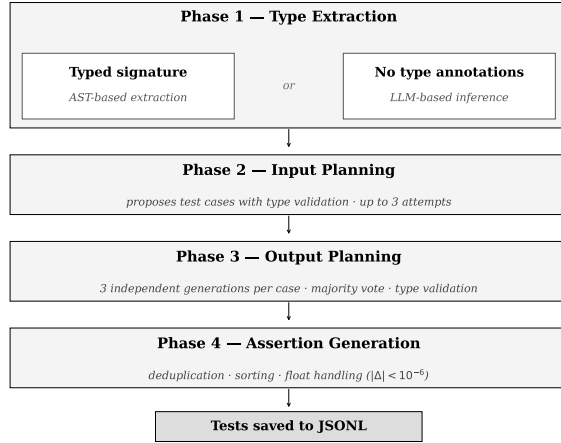


Figure 1: Overview of the automatic test generation process.

Figure 1 provides an overview of the test generation process, organized into four phases and performed individually for each HumanEval task.

Phase 1 — Type Extraction. The process begins with the identification of input parameter types and the return type of the target function. When type annotations are present in the function signature, extraction is performed directly via AST-based analysis. Otherwise, the model is instructed to infer the types from the textual description of the task. Listings 1 and 2 present the prompts used for this inference.

The obtained types are used as structural constraints throughout the subsequent stages.

Phase 2 — Input Planning. The model plans the test cases, producing a set of candidate inputs represented in JSON format. Each case contains the proposed input values and a textual description of the behavior or scenario to be validated. The prompt used in this stage is presented in Listing 3.

```

"""
Given the following Python function specification, what is
↪ the type of each parameter?

{state['prompt']}

RULES:
* Do NOT mention the return type
* Include the parameter variable name and its type
* If a parameter's type cannot be determined, write: null
* Output one JSON object per line, nothing else

Examples:
[{"name": "a", "type": "float"}]
[{"name": "b", "type": "List[int]"}]
[{"name": "c", "type": null}]
"""
  
```

Listing 1: Input type extraction.

```

"""
Given the following Python function specification, what
↪ should be the return type?

{state['prompt']}

RULES:
* Reply with ONLY the return type as a single Python type
↪ name
* If the return type cannot be determined, reply with: None
"""
  
```

Listing 2: Output type extraction.

The generated inputs are subjected to a type validation process. Cases incompatible with the previously defined signature are discarded, and the generation process is repeated, limited to three consecutive attempts.

Phase 3 — Output Planning. After obtaining valid inputs, the model is used to infer the respective expected outputs. To increase the reliability of the results, a majority voting mechanism based on multiple independent generations is employed. For each test case, three distinct inferences are produced, and the output that achieves majority agreement is selected, as illustrated in Figure 2. The prompt used for the majority voting inference is presented in Listing 4.

Cases for which there is no consensus among the inferences are discarded. The outputs approved by the voting mechanism are then subjected to a new consistency check, ensuring compatibility with the expected return type of the function.

Phase 4 — Assertion Generation. The approved cases are converted into Python assertion statements, forming the final set of unit tests associated with the task. A process of duplicate removal and sorting is then applied to the generated tests.

Additionally, comparisons involving float type values receive specific treatment. Due to the inherent limitations of floating-point

```

"""
You are an expert Python tester.

TASK:
{state['prompt']}{param_types_hint}

Write exactly 7 test input scenarios as a JSON list.

Each scenario is an object with:
- "description": short string explaining what this tests
- "inputs": an object mapping each parameter name to a
  ↳ value of the correct type

Example of ONE scenario (do not copy these values):
```json
[
 {example_scenario}
]
```

Rules:
- Output ONLY the JSON list inside ```json``` fences.
- No code. No explanation. No expected outputs.
- Cover distinct behaviors, not minor variations.
- All values must match the parameter types.
"""

```

Listing 3: Input planner.

```

"""
You are an expert Python tester reasoning about function
↳ behavior.

TASK:
{state['prompt']}

INPUT:
{inputs_str}

RETURN TYPE:
{state['return_type']}

Your job is to reason carefully about what this function
↳ should return for this specific input.

Guidelines:
- Think step by step.
- If the output is a simple deterministic value, reason
  ↳ towards that exact value.
- Be precise and concrete in your conclusion.
- Do not write code.

End your response with a clear conclusion: "CONCLUSION:
↳ <what you expect the output to be>
"""

```

Listing 4: Output planner.

number representation, exact comparisons can produce false negatives even when the implementation is correct. To mitigate this

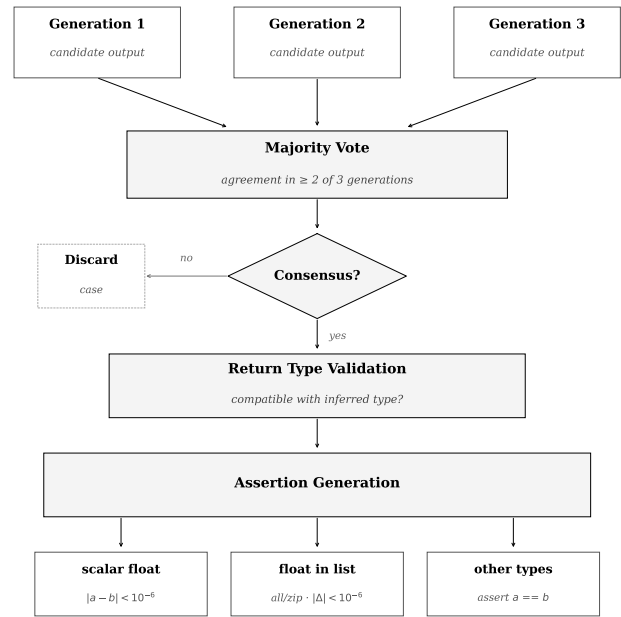


Figure 2: Detailed steps of the test generation process.

issue, the comparisons are rewritten using numerical tolerance based on absolute difference, considering values equivalent if their difference is less than 10^{-6} .

At the end of the process, each task has an automatically generated test suite saved in a JSONL file, with the possibility of empty suites due to the potential lack of agreement during the voting stage, or all tests being discarded during the type validation process.

4.4 Iterative Generation of Implementations

The generation of the implementations follows an iterative strategy based on initial generation, test validation, and feedback-guided refinement, as shown in Figure 3.

```

"""
You are a Python code generator.

RULES:
- Output ONLY valid, executable Python code
- Include necessary imports
- Define the function as described
- No comments, explanations, or markdown

TASK:
{state['prompt']}

TESTS:
{state['test_code']}

Output ONLY Python code.
"""

```

Listing 5: Generate implementation with tests.

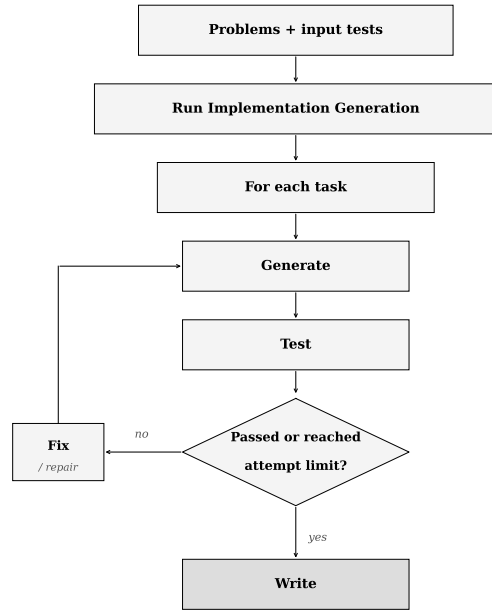


Figure 3: Iterative process of test-guided implementation generation.

```

"""
You are an expert Python programmer. Given the following:

TASK:
{state['prompt']}

IMPLEMENTATION:
{state['implementation']}

TESTS:
{state['test_code']}

RESULT:
{state['error']}

RULES:
- YOU CANNOT ALTER THE TESTS

Provide a concise and brief instruction on how to fix the
↔ implementation in natural language.
"""

```

Listing 6: Generate instructions for correction.

Initially, the model receives the task description and produces a candidate implementation for the required function, as shown in Listing 5. Subsequently, this implementation is executed against the test suite associated with the considered experimental scenario.

When failures are identified, a diagnostic stage begins. In this phase, the model receives information about the failed tests and is instructed to produce solely a textual analysis of the probable cause

```

"""
You are a Python programmer.

TASK:
Fix the implementation below according to the remediation
↔ message.

IMPLEMENTATION TO FIX:
{state['implementation']}

REMEDIATION:
{state['remediation_message']}

Rules:
- Output ONLY valid, executable Python code
- Include any necessary imports
- Define the function as required
- Do NOT include comments, explanations, or markdown
- Output ONLY the corrected code
"""

```

Listing 7: Generate correction

of the error, without directly modifying the existing implementation, as shown in Listing 6. The objective is to obtain a structured description of the necessary corrections prior to generating a new solution.

Based on this diagnosis, the model produces a revised version of the implementation incorporating the suggested corrections (Listing 7). The new implementation replaces the previous one and is once again subjected to test execution.

The cycle of execution, diagnosis, and regeneration is repeated until all tests pass or a maximum limit of ten refinement iterations is reached. This limit was set to control the computational cost of the refinement process and to prevent infinite correction loops.

If the maximum limit is reached without the complete approval of the test suite, the last produced implementation is considered the final result of the task.

4.5 Generation of Implementations Without Tests

In the scenario without the use of tests, no intermediate validation step is performed. In this case, the model generates only an initial implementation from the task statement, undergoing solely the necessary post-processing to extract the source code when it is returned encapsulated in Markdown code blocks.

5 Results

This section presents the results obtained from the conducted experiments. Initially, the quality of the unit tests generated by the language models is evaluated. Subsequently, the code generation results across the different experimental scenarios are presented, comparing the use of automatically generated tests, tests provided by the HumanEval benchmark, and the absence of intermediate validation.

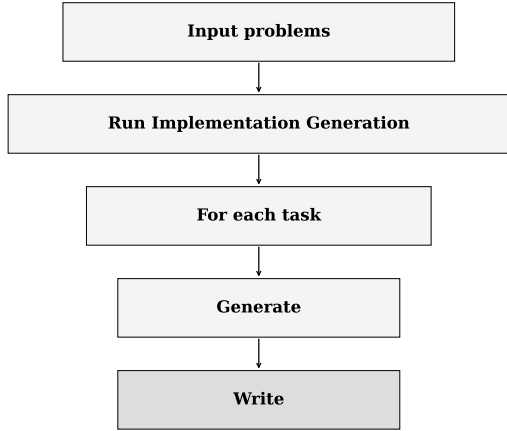


Figure 4: Implementation generation process without test-based validation.

5.1 Unit Test Generation

For this evaluation, each generated test suite was executed against the canonical solution provided by the HumanEval benchmark. A test suite was considered *valid* when the canonical solution passed all the generated tests. When at least one test failed the correct solution, the suite was classified as *invalid*. Cases in which no test could be produced after the generation process were also recorded.

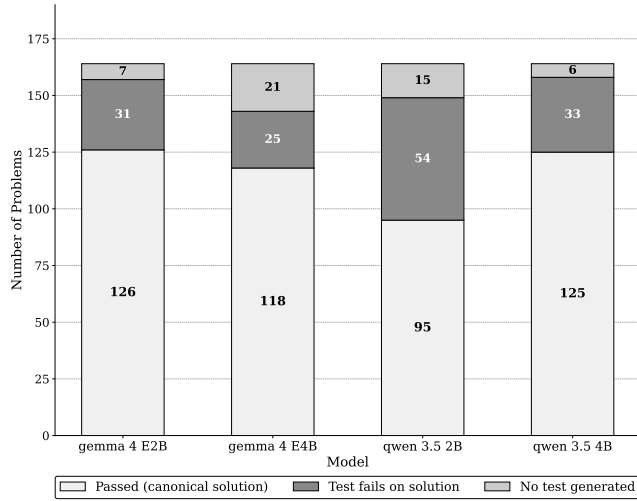


Figure 5: Distribution of generated test suites for HumanEval tasks.

Figure 5 presents the distribution of the results obtained for the four evaluated models. It is observed that all models were able to generate valid test suites for a significant portion of the benchmark tasks.

The best results were obtained by Gemma 4 E2B and Qwen 3.5 4B, which produced valid suites for 126 and 125 tasks, respectively.

Regarding invalid test suites, Qwen 3.5 2B presented the highest value, with 54 tasks in this category, followed by the Qwen 3.5 4B and Gemma 4 E2B models, with 33 and 31 cases, respectively.

When comparing the Gemma 4 E2B and Gemma 4 E4B models, it is observed that the E2B model produced more valid suites (126 versus 118), but also presented a higher number of invalid suites (31 versus 25). On the other hand, the E4B model recorded more cases in which no test suite could be produced (21 versus 7).

5.2 Code Generation Performance (Pass@1)

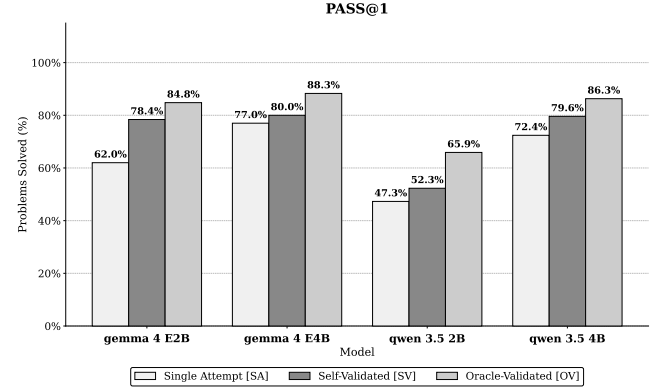


Figure 6: Results obtained using the Pass@1 metric.

Figure 6 presents the results obtained using the Pass@1 metric in the three evaluated scenarios: single attempt (SA), in which the implementation is generated without any validation step; self-validation (SV), in which the model uses the tests it generated itself during the iterative correction process; and oracle validation (OV), which uses the original tests from the HumanEval benchmark as a mechanism for validation and refinement of the implementations.

All models showed an increase in the problem-solving rate when subjected to some test-based validation mechanism.

Across all evaluated models, the oracle validation (OV) scenario presented the best results.

The highest absolute gain between the SA and SV scenarios was observed in the Gemma 4 E2B model, whose performance increased from 62.0% to 78.4%. Gains were also observed for the Gemma 4 E4B, Qwen 3.5 2B, and Qwen 3.5 4B models.

5.3 Code Generation Performance (Pass@10)

Figure 7 presents the results obtained using the Pass@10 metric. Unlike Pass@1, this metric considers up to ten independent samples for each problem.

The results show that all models presented superior performance in the self-validation scenario when compared to the single attempt scenario. Likewise, the oracle validation scenario presented the best results for all evaluated models.

The highest absolute gains between the SA and SV scenarios were observed in the Gemma 4 E2B and Qwen 3.5 2B models, which showed increases of 12.2 and 11.0 percentage points, respectively. The overall best result was obtained by Qwen 3.5 4B in the OV scenario, achieving 93.3% of solved problems.

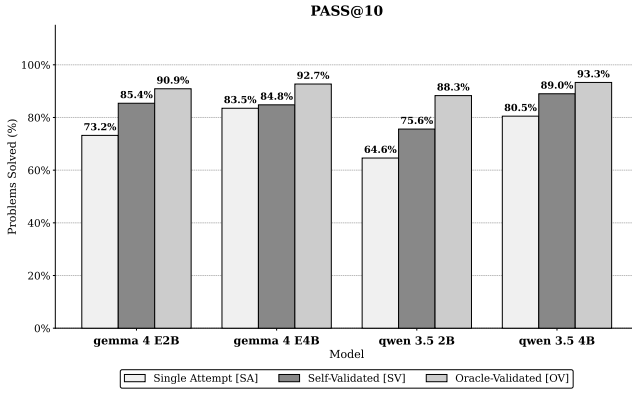


Figure 7: Results obtained using the Pass@10 metric.

6 Discussion

The obtained results demonstrate that the evaluated models are capable of generating valid test suites for a significant portion of the HumanEval tasks. However, substantial differences were observed in the quality of these suites, especially regarding the number of tests incompatible with the canonical solution. In general, models that showed better overall performance also produced more reliable test suites, suggesting a greater capacity to correctly infer the expected behavior of the evaluated functions.

When comparing the Gemma 4 E2B and Gemma 4 E4B models, it is noted that, despite belonging to the same family of models, the E2B variant produced a slightly higher number of valid test suites (126 versus 118). However, the E2B model also presented a higher number of invalid suites (31 versus 25), while the E4B model registered a substantially higher number of tasks in which no test suite could be produced (21 versus 7).

This behavior suggests that the E2B model tends to produce consensus more frequently, which increases both the number of valid suites and invalid suites. In contrast, the E4B model seems to present greater divergence among the multiple generations performed, resulting in a higher number of discarded cases.

Regarding code generation, the results indicate that the use of tests during the refinement process can contribute significantly to the quality of the produced implementations. Both in Pass@1 and Pass@10, the self-validation strategy outperformed direct generation without validation for all evaluated models. These results suggest that automatically generated tests are capable of providing useful information to identify and correct errors present in the initial implementations.

The results also show that the quality of the tests directly influences the potential benefit of the proposed strategy. Since the generated tests are used during the iterative correction process, invalid suites can induce the models to make unnecessary or incorrect modifications. On the other hand, valid suites tend to serve as a refinement mechanism for the generated solutions.

Finally, it is observed that, although all models showed gains in the self-validation scenario compared to single attempt, the magnitude of these gains varied considerably across models.

7 Limitations

The present study is subject to a number of limitations that should be considered when interpreting the results. First, the evaluation was conducted exclusively on the HumanEval benchmark, which consists of relatively simple algorithmic programming problems in Python. It is therefore unclear whether the findings generalize to other benchmarks, programming languages, or problem domains closer to real-world software development.

Second, several design decisions in the test generation pipeline were not individually evaluated on their impact on the final results. These include the majority voting mechanism used to infer expected outputs, the tolerance-based rewriting of floating-point assertions, and the type validation process applied to the generated test inputs and outputs.

Third, the maximum number of refinement iterations was fixed at ten and was not evaluated through an ablation study. It remains unclear how varying this limit would affect performance, particularly whether fewer iterations would be sufficient for simpler problems or whether additional iterations would yield further gains.

Fourth, in the Pass@10 evaluation, the same test suite generated for a given task is shared across all ten independent samples. Consequently, any systematic errors present in the generated tests affect all samples equally, which may influence the observed results for this metric.

Finally, the evaluation was restricted to compact, locally executable models. Frontier models such as the GPT-5 series, Claude Opus 4.8, or Gemini 3 Pro and Gemini 3 flash were not included, and it remains unclear whether the observed trends hold for models with substantially higher capacity.

8 Conclusion

This work investigated the impact of using unit tests as a validation mechanism during code generation by language models. To this end, three distinct scenarios were compared: direct generation of implementations without intermediate validation, validation using tests automatically generated by the models themselves, and validation using the tests provided by the HumanEval benchmark.

The obtained results demonstrated that the evaluated models are capable of producing valid test suites for a significant portion of the benchmark tasks. Although relevant differences were observed among the models, all presented the ability to infer part of the expected behavior of the evaluated functions and generate usable test cases during the development process.

The analysis of the code generation results indicated that the use of tests during the refinement process tends to consistently produce performance improvements when compared to direct generation without validation, as both the self-validation and oracle-validation scenarios presented gains in Pass@1 and Pass@10 for all evaluated models, evidencing that the feedback produced by the tests can assist in identifying and correcting errors present in the generated implementations.

As expected, the best results were obtained when the original HumanEval tests were used as a validation mechanism, evidencing that tests automatically generated by the models do not have the same effectiveness as manually written tests by humans in highlighting the expected behavior in each function of the benchmark.

It was also observed that, although all models benefited from test-based validation, the magnitude of these gains varied considerably across the evaluated models.

For future work, we intend to investigate more robust strategies for test generation, including additional validation mechanisms for the produced cases, more sophisticated consensus techniques, and hybrid approaches that combine automatically generated tests with partial reference tests. It would also be interesting to evaluate the proposal on additional benchmarks and on programming tasks closer to real software development scenarios.

References

- [1] Kent Beck. 2003. *Test-Driven Development: By Example*. Addison-Wesley Professional.
- [2] Pedro Calais and Lissa Franzini. 2022. Test-Driven Development Benefits Beyond Design: Quality, Flow State and Developer Experience. In *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*. Rio de Janeiro, Brazil.
- [3] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT5: Code Generation with Generated Tests. arXiv:2207.10397 [cs.CL] <https://arxiv.org/abs/2207.10397>
- [4] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] <https://arxiv.org/abs/2107.03374>
- [5] Yi Cui. 2025. Tests as Prompt: A Test-Driven-Development Benchmark for LLM Code Generation. arXiv:2505.09027 [cs.SE] <https://arxiv.org/abs/2505.09027>
- [6] Zilong Jiang et al. 2024. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology* (2024). <https://arxiv.org/abs/2406.00515>
- [7] Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2023. Interactive Code Generation via Test-Driven User-Intent Formalization. arXiv:2208.05950 [cs.SE] <https://arxiv.org/abs/2208.05950>
- [8] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE] <https://arxiv.org/abs/2305.01210>
- [9] 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)*. ACM, 1583–1594. doi:10.1145/3691620.3695527
- [10] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2024. A Comprehensive Overview of Large Language Models. arXiv:2307.06435 [cs.CL] <https://arxiv.org/abs/2307.06435>
- [11] Seyed Mahmoud Sajjadi Mohammadabadi, Burak Cem Kara, Can Eyupoglu, Can Uzay, Mehmet Serkan Tosun, and Oktay Karakuş. 2025. A Survey of Large Language Models: Evolution, Architectures, Adaptation, Benchmarking, Applications, Challenges, and Societal Implications. *Electronics* 14, 18 (2025). doi:10.3390/electronics14183580
- [12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] <https://arxiv.org/abs/1706.03762>
- [13] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. arXiv:2406.18181 [cs.SE] <https://arxiv.org/abs/2406.18181>