Code Without Context: Can We Trust LLMs to Test Software from Informal Descriptions?

Amneh Al Abdi[1], **Saad Ezzini**[1,2*]
[1]Information and Computer Science Department,
King Fahd University of Petroleum and Minerals
[2]IRC for Intelligent Manufacturing and Robotics, KFUPM
Dhahran, Saudi Arabia
[*]saad.ezzini@kfupm.edu.sa

Abstract

Recent advances in Large Language Models (LLMs) highlight their potential to automate software engineering tasks. However, LLM-generated outputs raise robustness and reliability concerns, particularly when relying on informal natural language instead of structured inputs. This paper evaluates LLMs for automated test case generation in this weaker-input setting, based solely on unstructured problem descriptions. We use 191 programming problems to assess a general-purpose LLM, GPT-5-mini, and a code-specialized LLM, Qwen2.5-Coder-7B. The generated test cases are executed on reference Python solutions and evaluated for test-level and problem-level pass rates. Results show GPT-5-mini outperforms Qwen2.5-Coder-7B, achieving 63.72% and 59.16% pass rates, respectively, compared to Qwen2.5-Coder-7B's 21.62% and 2.09%. These findings indicate both models struggle to capture the semantics of informal descriptions, offering early insights into the feasibility of LLM-based test generation from natural language. The results also emphasize potential deployment risks, such as undetected misinterpretations in safety-critical applications.

1 Introduction

In the software development life cycle, software testing is one of the main critical tasks where the quality of the produced software is validated before release (Baresi and Pezze, 2006). This validation can be evaluated based on a set of predefined test cases that aim to achieve high coverage, catch bugs early, and support refactoring (Liu et al., 2025). As the manual generation of the test cases is time-consuming, has a high possibility of human errors, and results in incomplete test coverage (Baresi and Pezze, 2006), software engineering communities have increasingly explored the use of Artificial Intelligence (AI) techniques in software engineering tasks, including software testing (Wang et al., 2024; Fan et al., 2023)

Large Language Models (LLMs) are a type of AI system that have been trained on a huge amount of data, making them able to generate human-like language (Naveed et al., 2025). Given the ability to generate outputs based on Natural Language (NL) inputs, LLMs can be utilized to generate test cases for software systems based solely on their NL descriptions (Liu et al., 2025). From a software security perspective, generating reliable test cases from informal requirements is a critical step in identifying software vulnerabilities early. Relying on LLMs introduces potential security risks if the model misinterprets constraints and generates test cases that fail to cover critical security-relevant edge cases, thereby allowing vulnerabilities to propagate into production.

Recent studies have explored and evaluated LLMs for test generation tasks (Jiang et al., 2026). Most of these studies depend on the structured description of the programming problems, such as source code and formal specifications. However, developers can also rely on informal NL descriptions, such as README files; therefore, an evaluation of the LLMs' capabilities for generating the required test cases based on the understanding of informal NL descriptions needs to be further explored.

In this paper, we evaluate two LLMs—the Generative Pretrained Transformer (GPT), specifically GPT-5-mini, a general-purpose LLM, and Qwen-Coder, specifically Qwen2.5-Coder 7B, an LLM trained for code generation, on the task of test case generation based only on their understanding of NL descriptions of programming problems stored in README files. This study is guided by the following research questions:

1. **RQ.1.** To what extent are Large Language Models reliable for generating executable test cases from natural-language problem descriptions?

Proceedings of the Second International Conference on Natural Language Processing and Artificial Intelligence for Cyber Security (NLPAICS'2026)
11–12 June, 2026. ©2026 The Authors

2. **RQ.2.** What are the main runtime error types in LLM-generated test cases affecting their reliability?
3. **RQ.3.** How does problem difficulty relate to the distribution and frequency of observed failure types in LLM-generated test cases?

The contributions of this paper are : (1) providing an execution-driven evaluation to measure the extent to which LLMs can generate test cases based only on natural-language problem descriptions, without access to the source code; (2) providing an analysis of the quality of the generated test cases by identifying and discussing common failure cases.

The rest of the paper will be organized as follows: Section 2 addresses some related works, Section 3 explains the used methodology, including its main stages of data collection, test case generation, execution preparation, and execution and evaluation. The results and the results analysis are provided in Sections 4 and 5, respectively. Section 6 states the threats to validity of this research, and finally, Section 7 provides a conclusion and future work.

2 Related Works

Test case generation is one of the main processes of software testing, where multiple test cases (inputs and expected outputs) are created to cover all possible cases (i.e., different scenarios, including edge cases) of the computer program being checked and validated (Wang et al., 2025).

Despite software testing in general, and test case generation in particular, being time-consuming (Gurcan et al., 2022), the software engineers must carefully consider this step, as performing it insufficiently can lead to serious consequences, including, but not limited to, bad user experiences and low-quality software (Kassab et al., 2021). Given the proven effectiveness of LLMs in tasks related to natural language understanding of coding problems and code generation (Wang et al., 2024), the literature has explored their capabilities in coding-related tasks, including test case generation.

(Walczak et al., 2026) discussed the impact of code context and prompting strategies on the quality of LLM-generated test cases. They investigated whether the use of sequential multi-turn prompting improves quality compared to direct single prompting. For this purpose, they defined two prompting strategies: Simple prompting (S1), which asks for direct test generation, and Sequential multi-turn prompting (S2), which decomposes the task into three sequential messages. The authors targeted six LLMs (GPT-4.5, GPT-o3, GPT-o4-mini-high, Claude 3.7 Sonnet, Gemini 2.5 Pro, and DeepSeek-V3). The results show that the S2 strategy provides a moderate improvement in quality and improves coverage, producing around 20–40% more test cases compared to the S1 strategy.

(Plein et al., 2024) evaluated ChatGPT and CodeGPT models in terms of their ability to generate test cases based on informal bug reports of Java projects. The results show that the fine-tuned GPT model (i.e., CodeGPT) outperformed ChatGPT in generating relevant test cases based on the bug reports provided, with an increase of 15%.

(Yin et al., 2024) evaluated the performance of Llama models in different scales in the task of generating test cases based on the understanding of natural language user stories. The authors tried different prompting techniques, including prompt chaining and few-shot prompting, and agent-based approaches. The results showed that the bigger model was the best in terms of test coverage; prompt chaining with a few examples provided the best performance, while too many examples with too complex prompts yielded the worst performance; sequential prompting helped with complex test case generation; and the best way to add edge test cases is to explicitly specify them in the prompts.

(Roy Chowdhury et al., 2024) proposed an approach to improve the generation of unit tests for focal Java methods using LLMs. Instead of relying on sample usages and/or the entire class of that method, the authors proposed injecting the used prompts with useful information that could be obtained through static program analysis. In addition to GPT-4 for benchmarking, the authors evaluated Llama 7b/70b 2.0 and CodeLlama 34b on commercial and open-source Java projects. The results showed that the proposed approach outperformed a baseline approach, which used the entire Java source file and asked LLMs to fill in the required unit test file.

Although these existing studies have addressed the LLMs' effectiveness in generating test cases from structured artifacts, such as source code and formal specifications, limited attention has been given to investigating their capabilities in generating test cases based solely on informal specifications, such as README-style descriptions. This study aims to address this gap by evaluating the

execution-level correctness of LLM-generated test cases across different LLM families and varying problem difficulty levels.

3 Methodology

In this section, we explain the steps we follow to answer the research questions of this study. The used pipeline consists of four main processes: data collection, test cases generation, execution preparation, and execution and evaluation. Figure 1 visualizes this pipeline, while the next four subsections address them in detail.

3.1 Data Collection

To collect the required dataset, we search for "Competitive Programming problems" in GitHub, using "Repositories" and "Python" as the code and language filters, respectively. From the resulting repositories, we refined them by selecting those that have both a natural-language description of the programming problem stored in a README file and a single Python solution. This selection is to ensure a clear input–output mapping, reduce the risk of having duplicate or near-duplicate solutions, and simplify the problem–solution pairing process. As a result, we select a repository containing a collection of programming problems collected from LeetCode (LeetCode, 2026; Kassa, 2026).

The selected GitHub project contains 217 folders (i.e., programming problems) across forty-nine programming topics, such as Arrays, Hash Table, and Math. After applying the refinement process, 191 folders remained, each containing two files: a ".py" file for the Python solution and a ".md" file for the corresponding README. Each problem is classified by LeetCode into one of three difficulty levels: easy, medium, or hard, with approximately 38%, 57%, and 5%, respectively.

3.2 Test Case Generation

In this stage, the objective is to prompt selected LLMs to generate a specific number of test cases (ten in our setting) for each of the programming problems in the collected dataset based only on the problem's natural language description stored in the README.md file. For this objective, the selected LLMs are prompted to generate a JSON file containing the generated test cases in a predefined structure, such that each JSON file must have a meaningful name of the test case, inputs, and expected_outputs. We fix the number of the generated test cases to ten per problem to balance between test diversity and computational cost. Furthermore, we use JSON as the output format, because it is structured, machine-readable, and can be parsed automatically (Langdale and Lemire, 2019). For model selection, we target in this study two LLMs from two distinct classes: a general-purpose model, GPT-5-mini, and a coding-specialized local model, Qwen2.5-Coder-7B. GPT-5-mini is accessed through the OpenRouter API, whereas Qwen2.5-Coder-7B is served locally using Ollama's chat. The selection of these models is not to have a model-to-model performance comparison, but rather to investigate the capabilities of two distinct, representative, and practically accessible instances from different model classes. Regarding prompting, the same prompt structure is used for both models. The full README content is injected as input to instruct the models to generate the test cases. The used prompt also enforces that the models follow the input format, generate test cases that have unique outputs, follow the constraints, include edge and tricky cases, and avoid extremely large inputs to prevent issues like timeouts and crashes. The prompt used in our experiments is shown below. In this prompt, we use a controlled zero-shot prompt without extensive prompt engineering. Although advanced prompting strategies (e.g., multi-turn prompting and few-shot prompting) may improve performance, our goal is to maintain a controlled evaluation setting while minimizing prompt-engineering effects.

For each of the models, a separate pipeline is used to generate the required test cases; however, both share the same conceptual stages: README understanding, constrained generation, JSON parsing, schema validation, results storing, and a retry or repair stage.

The used prompt

```
You are given a programming problem
    description from README.md. Your task:

Generate EXACTLY ten test cases for this
    problem.
Each test case must contain:
- "name": a short identifier
- "input": the input for the solution,
    formatted exactly as described
- "expected_output": the correct output
    value
IMPORTANT REQUIREMENTS:
1) Base everything ONLY on the problem
    description below.
```

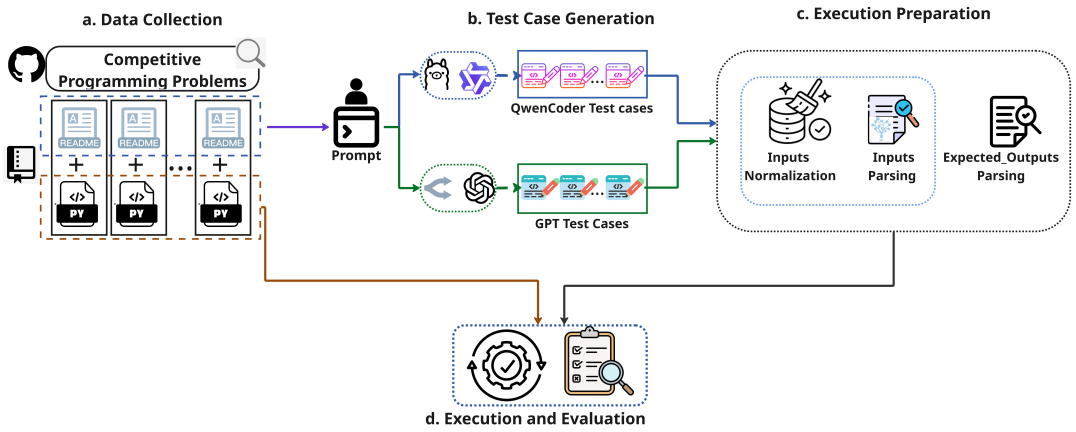

Figure 1: Overview of the proposed methodology pipeline.

```
2) Inputs must strictly follow the stated
    input format.
3) expected_output must be EXACTLY what a
    correct reference solution would
    return.
4) Use ONLY test cases where the output is
     uniquely determined.
5) Respect all constraints mentioned in
    the problem.
6) Avoid extremely large inputs that could
     cause timeouts.
7) Include: 2 edge cases, 2 tricky corner
    cases, and the remaining normal cases.

8) Output VALID JSON ONLY.
9) JSON MUST match this schema exactly:
{ "test_cases": [ { "name": "string", "
    input": "string", "expected_output":
    "string" } ] }
Problem description (README.md): {readme}
```

Starting with the GPT-5-mini pipeline, the model is invoked through OpenRouter's OpenAI-compatible chat completions endpoint, using the proper values of the `max_tokens` parameter to be large enough to generate the required test cases. For the outputs, the pipeline requests JSON-formatted outputs and attempts to parse the responses as valid JSON. The failures of this process are categorized as empty or truncated output, parsing errors (i.e., malformed JSON), or other failures, such as runtime failures. These failures are recovered by increasing `max_tokens` values, instructing the model to repair the malformed outputs into a valid and JSON-compatible schema, and strictly regenerate if the number of the generated test cases is less than the required.

For Qwen2.5-Coder-7B, the same general pipeline is used. The differences are that the model is served locally, and instead of using `max_tokens`, the Ollama generation is controlled with runtime options like `num_predict`, which mitigate truncation and under-generation. For failure handling, the Ollama calls are used with retry logic. Finally, the output parsing and JSON schema validation mirror the GPT pipeline. As a result of this step, each problem folder contains, in addition to a Python solution and the README file, two JSON files representing the ten test cases generated by each model.

3.3 Execution Preparation

In this stage, three main steps are performed: input normalization, input parsing, and expected output parsing. For input parsing, a sub-process first applies two normalization steps. First, comma-separated assignments are converted into Python-compatible assignments; for example, inputs like "a = 1, b = 2" become "a = 1; b = 2". Second, JSON-style literals are converted into Python equivalents, for example: 'null' is converted to 'None', 'true' is converted to 'True', and 'false' is converted to 'False'. These normalizations are needed to ensure that the model-generated inputs are correctly interpreted by the Python parser. After normalization, the input string is parsed using Python's Abstract Syntax Tree (AST) to be analyzed as a sequence of Python statements. Two input formats are supported in this stage: either assignment-based inputs or single-expression inputs. If the input contains multiple expressions or unsupported syntax, the "Unsupported input statement" ValueError is raised. For the expected outputs parsing, ast.literal_eval is used. If the parsing succeeds, the expected output becomes a Python object (e.g., int, list, etc.); otherwise, it is kept as plain text.

3.4 Execution and Evaluation

In this stage, after generating and post-processing the test cases, to evaluate the LLMs' generated test cases, we execute each Problem' Python solution against its corresponding test cases. This evaluation is conducted based on the assumption that the provided Python solution is correct; therefore, if a test case fails, this indicates a possible incorrect problem interpretation that results in incorrect test cases. As each execution may produce failures, each problem is evaluated by an isolated Python sub-process. The input of this sub-process is a JSON payload containing the solution path and the list of test cases. To prevent the evaluation from hanging and to simulate the real programming competitions, we declare a per-problem timeout, such that the sub-process will terminate if the time limit is exceeded. In the final step of this stage, for each problem and each corresponding test case, the outputs after executing the solution (i.e., got) are compared to the parsed expected output (i.e., expected). We check the equality of both "got" and "expected"; If they are equal, the test case is considered as passed; otherwise, it is marked as failed.

3.5 Performance Metrics

For the test case generation, We define a generation as successful only if a JSON file: successfully created on the disk, can be parsed by json.loads() without errors, contains a key "test_cases", the number of the generated test_cases is exactly ten, and each test case has the required fields: "name", "input", and "expected_output". On the other hand, for test case Execution, we use two performance metrics: the success rates on the test case and the programming problem levels. Equations (1) and (2) show the formulas for these metrics, the formulas for these metrics, where N_{TC} is the total number of test cases, t_i denotes the i-th test case, and $\mathbb{1}(t_i)$ is an indicator function that returns 1 if the test case is successfully passed and 0 otherwise. Similarly, N_{P} denotes the total number of problems, p_j represents the j-th problem, and $\mathbb{1}(p_j)$ equals 1 if all test cases for problem j are successfully passed, and 0 otherwise.

$$\text{Success rate}_{\text{test}} = \frac{1}{N_{\mathrm{TC}}} \sum_{i=1}^{N_{\mathrm{TC}}} \mathbb{1}(t_i) \times 100\% \quad (1)$$

$$\text{Success rate}_{\text{problem}} = \frac{1}{N_{\mathrm{P}}} \sum_{j=1}^{N_{\mathrm{P}}} \mathbb{1}(p_j) \times 100\% \quad (2)$$

4 Results

In this section, for each model, we show the results in terms of test case generation (Table 1), error type distribution (Table 2), and error distribution across difficulty levels (Figure 2 and Figure 3).

4.1 Test Cases Generation

4.1.1 GPT-5-Mini

The results show that GPT-5-Mini successfully passes the test case generation criteria for approximately 90% of the problems. The remaining 10% generation errors were related to various failure groups: An incorrect number of the generated test cases, such that the model generated the JSON file successfully but with fewer than the required ten test cases; the API returned an empty content, even with enough token budget; and truncated or incomplete JSON files due to JSON parsing errors. For the recovery process of these 10% failures, we use higher token limits, use a repair prompt to convert the raw text into valid JSON (i.e., matching the required schema), and use a small output prompt for very large outputs to force small inputs and bounded outputs.

4.1.2 Qwen2.5-Coder-7B

The results show that Qwen2.5-Coder-7B successfully generated the required test cases for around 96% of the problems. The 4% failures are related to timeout (i.e., the Ollama server exceeding the configured time limit), an incorrect number of the generated test cases, generating a JSON file that misses at least one of the required fields, and generating invalid syntactic JSON due to parsing errors. Even with the recovery process, the model did not improve the results in terms of the defined generation success metrics

4.2 Test Cases Execution and Evaluation

To assess the functional correctness of the generated test cases, we executed each Python solution against its corresponding set of test cases. For the GPT-5-Mini model, the Success $\text{rate}_{\text{test}} = 63.72\%$ and the Success $\text{rate}_{\text{problem}} = 59.16\%$. On the other hand, for Qwen2.5-Coder-7B, the Success $\text{rate}_{\text{test}} = 21.62\%$ and the Success $\text{rate}_{\text{problem}} = 2.09\%$.

4.3 Error types and difficulty levels

To understand the types of errors that lead to the obtained results, Table 2 shows a quantitative summary of the error types related to GPT-5-mini and Qwen2.5-Coder-7B models. As the collected programming problems are classified based on their difficulty into three levels(i.e., easy, medium, and hard), we also report the quantitative results in terms of error distribution and the difficulty levels. Figure 2 shows these results in terms of the GPT-5-mini model, while Figure 3 shows those related to the Qwen2.5-Coder-7B model.

5 Results Analysis Discussion

In this section, we analyze the results and we provide the answer to each research question.

5.1 RQ.1: Test Case Generation Effectiveness

The results indicate that the GPT-5-mini model achieves a relatively high level of reliability for the test case generation task based solely on its understanding of the natural language description of the programming problem. With an initial success rate of 90%, the model shows its ability to follow the instructions and produce well-formed and structured outputs. The failures, such as truncated outputs, empty responses, or malformed JSON, indicate that the whole process is sensitive to decoding limits and output length constraints. From the execution perspective, with a test-level success rate of 63.72% and a problem-level success rate of 59.16%, this indicates that the model is able not only to generate the test cases but also to generate them in a way that is aligned with the problem's natural language description. However, the results also show some limitations related to this task, with around one-third of the generated test cases not matching the expected outputs. These limitations can be attributed to several factors, such as misinterpretation of the constraints or the required input and output formats. On the other hand, Qwen2.5-Coder-7B shows its reliability in understanding the natural language descriptions of the problems and in producing valid test cases as JSON files, for around 96% of the problems. As the JSON format is a code-like format, this explains its strength in generating such files. However, this high level of success in generating the required test cases does not lead to high execution-based results, as the model achieves only 21.62% and 2.09% for the test-level and problem-level success rates, respectively. These low execution success rates imply that most of the successfully generated test cases contain incorrect inputs, incorrect expected outputs, or both. This behavior also indicates the model's limitations in capturing the problem semantics.

Answer to RQ.1: *The results show that LLMs can generate executable test cases from natural-language problem descriptions to a limited and model-dependent extent.*

5.2 RQ.2: Error Types in LLM-Generated Test Cases

The results shown in Table 1 and Table 2 indicate that most of the errors caused by the mismatches between the generated test cases and the reference solutions are in terms of formatting. For GPT-5-mini, Table 2 shows that around 51% of these errors belong to the AttributeError class, which means that although the model often generates test cases with the correct structure, it frequently assumes the wrong data type, uses the wrong field or method names, or misunderstands the structure of the input or output. For Qwen2.5-Coder-7B, the results show that the majority of the errors are TypeErrors, at around 65%. This result means that the model, in most cases, generates inputs with incorrect data types, such as using strings instead of integers, using a list instead of a single value, mixing types in a list, or using tuples instead of lists. The generated inputs, in most cases, appear correct; however, these mismatches lead to runtime errors such as TypeErrors. The errors raised by each model's generated test cases indicate that the models try to infer the correct data types from ambiguous natural-language descriptions, leading to incorrect assumptions, type-inconsistent inputs, and overgeneralization of common coding patterns. It is worth noting that the ambiguity of README specifications may act as a main source of failures across both models. We also observed that the ability of LLMs to generate executable test cases does not mean they are semantically correct, where the generated test cases followed the required format but contained incorrect expected outputs.

Answer to RQ.2: *The results show that LLM-generated test cases mainly fail due to type mismatches, with AttributeErrors and TypeErrors being the most frequent.*

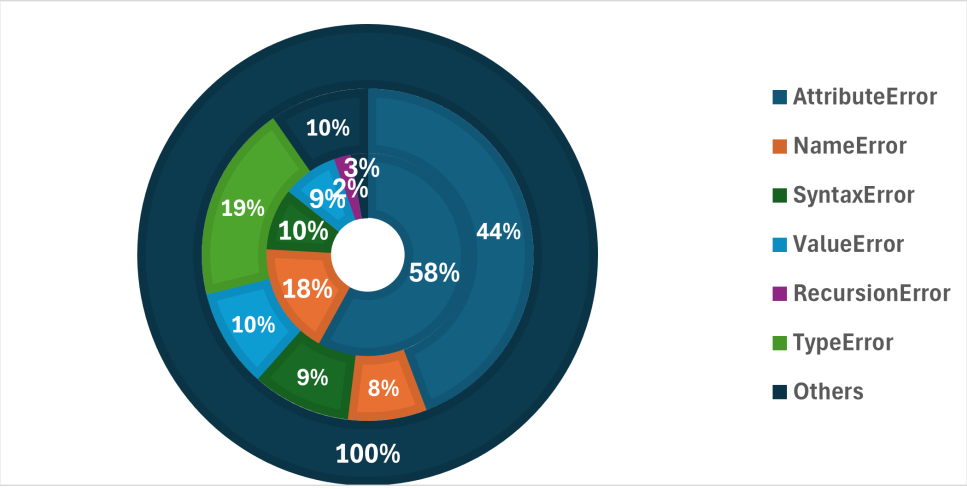

Figure 2: GPT error distribution across difficulty levels (inner ring: Easy; middle ring: Medium; outer ring: Hard).

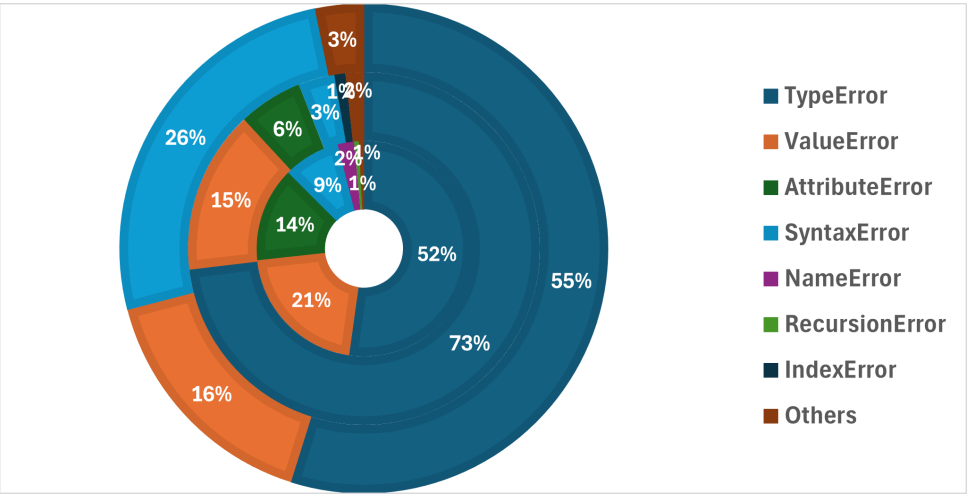

Figure 3: QwenCoder error distribution across difficulty levels (inner ring: Easy; middle ring: Medium; outer ring: Hard).

Model	**Gen. Success (%)**	**Success Rate (Test %)**	**Success Rate (Problem %)**
GPT-5-Mini	90%	63.72%	59.16%
Qwen2.5-Coder-7B	96%	21.62%	2.09%

Table 1: Test case generation and execution results for GPT-5-Mini and Qwen2.5-Coder-7B.

Error Type	**GPT-5-mini**	**Qwen2.5-Coder-7B**
AttributeError	111 (50.9%)	682 (63.9%)
NameError	28 (12.8%)	185 (17.3%)
SyntaxError	21 (9.6%)	93 (8.7%)
ValueError	20 (9.2%)	71 (6.6%)
TypeError	20 (9.2%)	10 (0.9%)
RecursionError	3 (1.4%)	8 (0.7%)
IndexError	-	4 (0.4%)
Other Runtime Errors	15 (6.9%)	15 (1.4%)
Total	**218 (100%)**	**1068 (100%)**

Table 2: Error type distribution for GPT-5-Mini and Qwen2.5-Coder-7B.

5.3 RQ.3: Impact of Problem Difficulty on Failure Patterns

To explore the possible relationship between the execution failures and the problem difficulty, we provide an analysis of the error distribution across easy, medium, and hard problems for both models. Starting with the GPT-5-mini model, for easy problems, the errors are dominated by AttributeError followed by NameError with around 58% and 18% respectively. For the medium problems, the AttributeError decreases to around 14%, while the NameError shows a little increase with around 1%. This shift may indicate that as the difficulty level increases from easy to medium, the errors related to structural mismatches become less concentrated. As the number of errors related to hard problems is very small (i.e., only two errors), we cannot conclude a common trend based only on the error distributions. Moving to the Qwen2.5-Coder-7B model, the error distributions show that TypeError is the dominant error type across all difficulty levels, with 52%, 73%, and 55% for easy, medium, and hard problems, respectively. This error distribution may indicate that the model tends to generate type-inconsistent inputs regardless of the problem difficulty level. For the hard problems, the distribution of the errors appears more heterogeneous with SyntaxError and ValueError. In hard problems, the error distribution appears more diverse, with SyntaxError and ValueError contributing more than in Easy and Medium problems.

Answer to RQ.3: *The results show that as problem difficulty increases, the main errors categories remain the same, while only the percentages of different error types change.*

6 Threats to Validity

This research has several threats to validity. The first is related to the assumption of the correctness of the Python solution for each problem. Although the collected solutions are from accepted LeetCode solutions, no manual verification beyond execution-based evaluation was performed. Second, the dataset has only 191 LeetCode problems collected from a single GitHub repository and is limited to Python, which affects the generalization of the results. Third, the number of the generated test cases is fixed, which may affect the diversity of inputs. Finally, the distribution of the problems is not balanced, with a small percentage of hard problems, which affects the strength of the conclusions related to difficulty-related patterns.

7 Conclusion and Future Work

This study mainly investigates whether LLMs, specifically GPT-5-mini and Qwen2.5-Coder-7B, can generate test cases for programming problems based solely on their natural language description. Although both models have shown their reliability in generating valid test cases, the execution-based evaluation demonstrates a notable gap between structural validity and semantic correctness. GPT-5-mini shows a moderate performance in terms of execution success, while Qwen2.5-Coder-7B provides much less effectiveness. The error analysis shows that the trend of the errors is related to type mismatches and attribute-related inconsistencies, implying the limitations of capturing the input-output semantics by depending only on natural-language descriptions. Furthermore, the analysis across difficulty levels shows that the error categories remain the same, although their proportions are different. Overall, the results of this study suggest the ability of LLMs to generate executable test cases from natural-language descriptions; however, the ability to capture the semantics is limited and model-dependent. This inability of LLMs to capture complex semantics accurately poses a significant security risk in automated workflows and safety-critical applications, as inadequate test cases could leave software vulnerabilities undetected before deployment.

For future work, this study can be expanded in many directions. First, by including a more balanced distribution of difficulty levels. Second, by including results from other programming languages to enhance the generalization of the results. Third, by including more general-purpose and code-specialized LLMs. Finally, by including human-written test cases to compare with LLM-generated ones to measure their quality based on a ground truth.

Limitations

This work evaluates LLMs on a limited dataset of Python competitive programming problems, which may not generalize to real-world industrial software or other programming languages. The evaluation assumes the correctness of the reference solutions without manual verification. Furthermore, using a fixed number of generated test cases might restrict

input diversity.

Ethical Statement

The authors declare that there are no ethical issues regarding the data used in this study, as it relies on publicly available competitive programming datasets. No human subjects or sensitive personal information were involved. The findings aim to improve the reliability of LLM-generated software testing, highlighting current vulnerabilities to prevent their propagation in real-world applications.

Acknowledgments

The authors would like to acknowledge the support provided by King Fahd University of Petroleum and Minerals (KFUPM) for funding this work.

References

Luciano Baresi and Mauro Pezze. 2006. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89–111.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large language models for software engineering: Survey and open problems. In *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53.

Fatih Gurcan, Gonca Gokce Menekse Dalveren, Nergiz Ercil Cagiltay, Dumitru Roman, and Ahmet Soylu. 2022. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. *IEEE Access*, 10:106093–106109.

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):1–72.

Mekdi Kassa. 2026. compitative_programming. `https://github.com/Mekdi-kassa/compitative_programming`. GitHub repository, accessed 23 February 2026.

Mohamad Kassab, Phillip Laplante, Joanna Defranco, Valdemar Vicente Graciano Neto, and Giuseppe Destefanis. 2021. Exploring the profiles of software testing jobs in the united states. *IEEE Access*, 9:68905–68916.

Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of json per second. *The VLDB Journal*, 28(6):941–960.

LeetCode. 2026. Leetcode – the world's leading online programming learning platform. `https://leetcode.com`. Accessed: 2026-02-23.

Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025. Exploring the potential of general purpose llms in automated software refactoring: an empirical study. *Automated Software Engineering*, 32(1):26.

Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2025. A comprehensive overview of large language models. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–72.

Laura Plein, Wendkûuni C Ouédraogo, Jacques Klein, and Tegawendé F Bissyandé. 2024. Automatic generation of test cases based on bug reports: a feasibility study with large language models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 360–361.

Sujoy Roy Chowdhury, Giriprasad Sridhara, AK Raghavan, Joy Bose, Sourav Mazumdar, Hamender Singh, Srinivasan Bajji Sugumaran, and Ricardo Britto. 2024. Static program analysis guided llm based unit test generation. In *Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD)*, pages 279–283.

Jakub Walczak, Piotr Tomalak, and Artur Laskowski. 2026. Impact of code context and prompting strategies on automated unit test generation with modern general-purpose large language models. *Journal of Systems and Software*, page 112834.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. Testeval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562.

Hang Yin, Hamza Mohammed, and Sai Boyapati. 2024. Leveraging pre-trained large language models (llms) for on-premises comprehensive automated test case generation: An empirical study. In *2024 9th International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS)*, volume 9, pages 597–607. IEEE.