Beyond Functionality: Studying Non-Functional-Requirement-Aware Code Generation Using Large Language Models

Anonymous ACL submission

Abstract

Recently, developers have increasingly utilized Large Language Models (LLMs) to assist with their coding. Apart from functional correctness, Non-Functional Requirements (NFRs), such as code performance, play a crucial role in ensuring software quality. However, the capability of LLMs in addressing NFRs has yet to be systematically investigated. In this paper, we propose *NFRGen*, an automated framework aimed at investigating how can LLMs better perform in NFR-aware coding. Our evaluation reveals that incorporating NFRs in the prompts considerably improves the effectiveness in addressing them. In the meantime, incorporating NFRs results in a decrease in Pass@1 by up to 26%. However, such impact can be mitigated when NFRs are initially specified in the same prompt. Our study highlights the implications for balancing both functional correctness and addressing NFRs in various coding workflows.

1 Introduction

Large Language Models (LLMs) have become an integrated part of modern software development with their growing popularity and advanced capabilities (Zhang et al., 2024). LLM-based services such as ChatGPT (OpenAI, 2023), GitHub Copilot (Copilot, 2024a) and Cursor (Cursor, 2024) have gained widespread adoption for their ability to simplify and accelerate the coding process by generating source code following the requirements provided by developers.

Prior studies focus on evaluating the functional correctness of the generated code, such as examining if the code passes all the provided test cases (Austin et al., 2021; Chen et al., 2021). However, Non-Functional Requirements (NFRs), such as reliability and performance, are also crucial to code quality and are underexplored. Figure 1 shows an example of generated code with and without considering performance. Without performance requirements, the generated code relies on exhaustive iteration, which results in a timeout when the loop size is large. When performance is considered, the generated code is optimized using a mathematical approach that efficiently handles such cases. This motivates the need for systematically evaluating how LLMs perform considering NFRs.

In this paper, we propose *NFRGen*, an automated framework that studies the ability of LLMs in code generation when NFRs are incorporated. *NFRGen* examines two primary NFR-aware workflows: (1) *NFR-Integrated*, where functional and non-functional requirements are provided in a single prompt, and (2) *NFR-Enhanced*, where existing code is refined to meet additional non-functional requirements. For the dimensions of NFRs, *NFRGen* particularly focuses on metrics related to code design, reliability, readability, and performance. They are among the most commonly targeted when using LLMs to refine code (Rasheed et al., 2024).

Through comprehensive experiments on various LLMs and benchmarks, we find that incorporating NFRs can considerably improve metrics related to NFRs (e.g., reducing code smells by up to 66.7%), but also negatively impacts the functional correctness of the generated code (lower Pass@1 by up to 26%). However, such impact is smaller when the NFRs are specified in a single prompt (i.e., *NFR-Integrated*) compared to the two-step workflow (i.e., *NFR-Enhanced*). We also find that these two workflows exhibit complementary strengths: *NFR-Integrated* excels in addressing runtime-related aspects such as exception handling and performance, while *NFR-Enhanced* is more effective in improving code structure-related dimensions like code design and readability.

We summarize our contributions as follows:

- We propose *NFRGen*, a novel framework for systematically evaluating the effectiveness and robustness of LLMs in addressing NFRs during code generation.
- We study two distinct workflows (i.e., *NFR-*

{Task} Model: gpt-3.5-turbo-0125 Data: HumanEval-83

```
def starts_one_ends(n):
    """ Given a positive integer n, return the count of the numbers of n-digit positive integers that start or end with 1.
    >>> starts_one_ends(1)
    1
    >>> starts_one_ends(2)
    18
    """
```

Function-Only	**Performance**
Finish the following code generation task: {Task}	*Keep performance in mind while solving the following code generation task: {Task}*

Function-Only

```python
def starts_one_ends(n):
    count = 0
    for i in range(10**(n-1), 10**n):
        if str(i)[0] == '1' or str(i)[-1] == '1':
            count += 1
    return count
```

Test (TimeOut)

```python
assert starts_one_ends(2) == 18
assert starts_one_ends(18) == 180000000000000000
```

Performance

```python
def starts_one_ends(n):
    if n == 1:
        return 1
    else:
        count = 2 * 9 * 10 ** (n - 2)
        return count
```

Test (Pass)

```python
assert starts_one_ends(2) == 18
assert starts_one_ends(18) == 180000000000000000
```

Figure 1: Example of generated code with and without the consideration of performance.

Integrated and *NFR-Enhanced*) and analyze their trade-offs in balancing functional correctness and meeting NFRs.

- Our comprehensive experiments across various LLMs and benchmarks provide actionable insights and highlight opportunities for improving NFR-aware code generation. We also share the replication package for future studies (Anonymous, 2024).

2 Related Work

Non-Functional Requirements (NFRs) in Coding. Prior studies proposed approaches for examining or refining existing source code to meet the NFRs. Pereira dos Reis et al. (2022) summarized a series of studies on detecting and visualizing code smells that negatively impact code design. Vitale et al. (2023) trained models to improve the readability of given code snippets and Li et al. (2023) identified readability issues from logging code. Zhang et al. (2020) proposed an automated approach to generate exception handling code based on existing source code to improve the overall software reliability. Gao et al. (2024) leveraged LLMs to improve the execution efficiency of source code. Han et al. (2024) proposed a framework that incorporate software requirements from textual descriptions for code generation. Unlike previous studies focusing on detecting or refining NFR issues in existing code, we examine the quality of NFR-aware code generated using different practical coding workflows and the associated robustness issues.

Studying the Robustness of LLMs in Code Generation. Wang et al. (2022), Chen et al. (2024), and Shirafuji et al. (2023) explored robustness by perturbating different components in the prompts (e.g., problem descriptions, docstrings) with diverse patterns. Chen et al. (2023) and Lin et al. (2024) reported that ChatGPT's performance on code generation can change substantially between different versions of the same model. Mishra et al. (2024) examined how robustness varies across various models and model sizes. These studies primarily focused on the functional correctness of the generated code. Given the critical role of NFRs in software development, our study addresses the importance of exploring the impact of incorporating NFR considerations into various coding workflows for LLM-based code generation. We also study the stability on the functional and NFR code quality across semantically equivalent prompts and model versions.

3 Methodology

In this section, we introduce an automated framework, *NFRGen*, to study the capability of LLMs considering various non-functional requirements in code generation. We refer to such code generation as *NFR-aware code generation*.

3.1 Studied Dimensions of NFRs

NFRs such as maintainability and readability, are critical aspects of code quality. However, existing studies on code generation often overlook NFRs and only focus on functional correctness metrics. For example, Pass@1 (Chen et al., 2021, 2023) is commonly used to assess whether the code generated by the LLM passes all test cases on its first attempt. Without considering NFRs, the generated code might be only functionally correct but lack reliability, readability, or efficiency. Such neglect can lead to significant maintenance challenges and

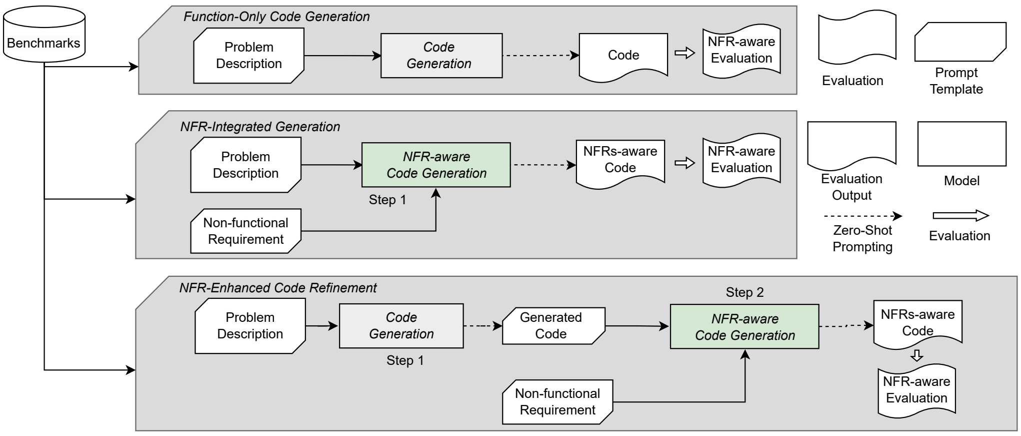

Figure 2: *NFRGen* consists of Function-Only code generation, NFR-Integrated code generation, and NFR-Enhanced code refinement. We compare the functional and non-functional quality of the code across the three workflows.

impact software quality (Chung and do Prado Leite, 2009). Hence, *NFRGen* is designed to study the capability and robustness of LLMs in addressing NFRs during the code generation process.

Specifically, we examine four non-functional requirements dimensions that contribute to the code's maintainability, reliability, and efficiency:

Code design refers to the structural and architectural quality of code, where bad designs can significantly hinder maintainability and scalability (Walter and Alkhaeir, 2016).

Reliability is the code's ability to handle unexpected inputs and ensure stable execution under various scenarios (e.g., exception handling) (Zhang et al., 2020).

Readability is how easily code can be understood and modified. Readable code should follow coding style guidelines and conventions to ease understanding and collaboration (Piantadosi et al., 2020).

Performance assesses the efficiency of code, where performance issues (e.g., slower execution) can cause higher operational costs and reduce user satisfaction (Malik et al., 2013).

3.2 NFR-Aware Coding Workflows

In addition to *Function-Only* code generation, modern code generation tools, such as Cursor (Cursor, 2024) and GitHub Copilot (Copilot, 2024a), provide two typical workflows for NFR-aware code generation (Copilot, 2024b). 1) ***NFR-integrated code generation*** involves developers providing both the functional and non-functional requirements in one prompt to generate the complete code in one shot. 2) ***NFR-enhanced code refinement*** is when developers utilize the LLM to refine existing code for improved code quality or to better align with specific requirements (White et al., 2024).

➢Function-Only Code Generation
Complete the following code.
Input: '{Problem Description}'
Response: '{Code}'

➢NFR-Integrated Code Generation
Given the problem description, generate code by considering {NFR}."
Input: '{Problem Description}'
Response: '{NFR-aware Code}'

➢NFR-Enhanced Code Refinement
Step 1 - Existing code to be refined -> '{Code}'
Step 2 - Refine the code with NFR
Given the following code, your goal is to improve its {NFR}."
Input: '{Code}'
Response: '{NFR-aware Code}'

Figure 3: An example of prompt templates for different coding workflows.

While both workflows provide the instruction to generate code that satisfies specific requirements, the final output may be different, as the ways of interacting with LLMs may significantly affect the generated results (Lee et al., 2024). Therefore, in our framework, we consider both of these two workflows to incorporate the four NFRs into the code. In the remainder of the paper, we denote NFR-integrated code generation as *NFR-Integrated* and NFR-aware code enhancement as *NFR-Enhanced* for conciseness.

Figure 2 provides an overview of *NFRGen*. *NFR-Gen* contains one baseline workflow that considers only the functional requirement (i.e., *Functional-Only Code Generation*, denoted as *Functional*, and two NFR-aware workflows (i.e., *NFR-Integrated* and *NFR-Enhanced*). To analyze the results, we compare the functional and non-functional code quality metrics across the code generated by three distinct workflows, examining the impact of NFR-aware code generation on overall code quality.

3.3 Prompt Construction

Workflow-specific Prompt Templates. Figure 3 shows the prompt templates for each workflow. *Functional* only contains the functional requirement in the prompt. *NFR-Integrated* incorporates NFRs directly into the prompt template. For example, when considering reliability, the prompt asks the LLM to generate code that meets the functional requirements and optimize reliability in a single prompt. *NFR-Enhanced* adopts a two-step process. It leverages the code generated by *Functional*, and it sends a separate prompt asking the LLM to enhance the code by addressing a specific NFR.

Constructing Diverse NFR-Aware Prompts. Prior research (Chen et al., 2024; Wang et al., 2022; Shirafuji et al., 2023) suggests that variations in prompt templates, even when preserving semantic context, can generate significantly different code. Hence, we repeat the code generation process using different but semantically equivalent prompts. To mitigate potential biases introduced by manually altering the prompts, we leverage GPT-4o to generate various prompts for each dimension of NFRs while preserving the same semantics. This approach allows us to study the result's stability by measuring the variations across the prompt templates.

Table 3 in Appendix A shows the prompts generated for each dimension of NFRs. Initially, we manually crafted a seed prompt with the structure: "*Consider [NFR] and complete the following code*", where "*[NFR]*" corresponds to specific non-functional requirements, such as code design or readability. We then provided the seed prompt to ChatGPT to generate 10 semantically equivalent prompts for the experiment. Both *NFR-Integrated* and *NFR-Enhanced* use the same NFR-aware prompt templates outlined in Table 3. We incorporate all 10 prompt variants to assess the robustness of the workflow against semantic preserving changes in the prompt. In total, we execute NFR-aware code generation 40 times (10 variations per NFR) for each workflow and each LLM version. Although we conduct our experiment on existing code generation benchmarks (i.e., HumanEval and MBPP), *NFRGen* is highly adaptable, and future studies using *NFRGen* can tailor the *NFR-Integrated* and *NFR-Enhanced* process to new non-functional requirements.

3.4 Functional and NFR Metrics

We use different metrics to examine the functional correctness and NFRs.

Metric of Functional Correctness. We use Pass@1 (Chen et al., 2021) to check if the generated code passes all test cases on its first attempt.

Metrics of NFRs. We consider a diverse number of NFRs, where each NFR has its own unique aspect. Hence, we use different metrics for each NFR. ①**Code Design:** We focus on the presence of code smell, which serves as a proxy for the quality of code design. The term "code smell" refers to code that negatively impacts maintainability (Fowler, 2018), such as overly complex functions or excessive duplication. We use the *Refactor* checker of Pylint (PyCQA, 2024a) to detect code smells. It includes predefined static code checkers to detect various code smells. We calculate and report the code smell density as the number of detected smells per 10 Lines Of Code (LOC) since the generated code may have different lengths. ②**Reliability:** We calculate exception density as the number of exception-handling statements per 10 LOC. This metric highlights the extent of error-handling logic (De Padua and Shang, 2017). ③**Readability:** Similar to code design, we use Pylint to detect issues like inconsistent naming, incorrect indentation, and missing comments. We also report the density of readability issues per 10 LOC. ④**Performance:** We measure the execution time in milliseconds for all tests associated with each coding problem. To minimize measurement fluctuations, we run each test case five times and calculate the mean.

To study the robustness and sensitivity of the NFR-aware code generation workflows, we compute the mean and standard deviation (abbreviated as STDEV) for the evaluation metrics across the semantically equivalent prompts for each LLM version (i.e., 10 prompts for each NFR per model version). A high STDEV indicates greater sensitivity of the LLM to the variations.

4 Evaluation

Studied LLMs. We conducted the study using GPT-3.5 and GPT-4o from OpenAI, and Claude-3.5 from Anthropic. Specifically, we used *gpt-3.5-turbo-1106* and *gpt-3.5-turbo-0125* for GPT-3.5, *gpt-4o-2024-05-13* and *gpt-4o-2024-08-06* for GPT-4o, and *claude-3-5-sonnet-20240620* and *claude-3-5-haiku-20241022* for Claude. We interacted with the models through the APIs provided by vendors. To reduce variances in LLM's outputs, we set the temperature value to 0.

Benchmark Datasets. We selected four datasets: HumanEval, HumanEval-ET, MBPP, and MBPP-ET, which are commonly used in code generation research (Huang et al., 2023; Lin et al., 2024) and provide test cases to evaluate the correctness of the generated code. HumanEval (Chen et al., 2021) comprises 164 programming problems, while MBPP (Austin et al., 2021) includes 427 programming problems (we used the sanitized version provided by the original authors). Furthermore, HumanEval-ET and MBPP-ET, published by Dong et al. (2023), use the same problems as HumanEval and MBPP but offer more test cases with approximately 100 test cases for each problem.

Environment. Our experiments were conducted on a Mac Mini (Apple M4, 10 cores, 16GB RAM), using Python 3.9.19 to implement *NFRGen* and the evaluation scripts. The OpenAI API library used was version 1.14.3, and the Claude API library used was version 0.39.0. For detecting code smells and readability issues, we used Pylint version 3.2.5.

RQ1: How Do NFR-Aware Workflows Affect Functional Correctness?

Motivation. Non-Functional Requirements (NFRs) play a critical role in software quality assurance. This RQ examines the functional correctness of the generated code when using *NFR-Integrated* and *NFR-Enhanced* to generate NFR-aware code.

Approach. We compute the Pass@1 when generating each of four types of NFR-aware code (i.e., design, reliability, readability, and performance) using *NFR-Integrated* and *NFR-Enhanced*. We also compare the Pass@1 with our baseline (*Function-Only*), where we generate the code by only considering the functional requirements. We conduct the study across various model versions and four benchmarks as discussed in Section 4.

Results. ***Incorporating NFRs results in lower Pass@1 across all benchmarks by up to 26%.*** Table 1 shows the Pass@1 results on *Function-Only*, *NFR-Integrated*, and *NFR-Enhanced* for all four NFRs across all benchmarks. Overall, adding NFRs lowers the Pass@1. For example, in HumanEval, the average Pass@1 across all LLMs (OpenAI and Anthropic) for *Function-Only* is 84.61%, whereas the Pass@1 decreases by 1.3% to 8.15% for *NFR-Integrated*, and 8.97% to 14.40% for *NFR-Enhanced*. Compared to *Function-Only*, the average decrease is from 1.2% and up to 26%.

NFR-Integrated almost always achieves better Pass@1 than NFR-Enhanced. Our finding shows that a two-step approach has a negative impact on Pass@1, and the difference can be over 20% (e.g., between *NFR-Integrated* and *NFR-Enhanced* for Code Design in MBPP), depending on the specific NFR and dataset. For code design and readability, the decrease is even more notable in *NFR-Enhanced* (10% to over 20% compared to *Function-Only*) compared to *NFR-Integrated* (1.3% to 3.91% over *Function-Only*). In contrast, even though exception handling (i.e., Reliability) has the largest decrease in *NFR-Integrated*, the difference with *NFR-Enhanced* is smaller. Performance has relatively more stable results between *NFR-Integrated* and *NFR-Enhanced*. Our findings show that the one-step approach may allow the LLM to balance the objectives better, and ***generative models may perform worse at Pass@1 on a two-step code enhancement, especially if the NFR is more related to re-structuring the code (i.e., code design and readability)***.

Incorporating NFRs reduces the capability of LLMs in stably generating functionally correct code, resulting in more variable Pass@1, especially in earlier versions of the LLMs. *NFR-Integrated* and *NFR-Enhanced* consistently exhibit higher standard deviations (STDEV) of Pass@1 across all benchmarks compared to *Function-Only*. For example, in HumanEval, the STDEV for Pass@1 ranges from 1.84 to 2.64 for *NFR-Integrated* and 2.70 to 7.54 for *NFR-Enhanced*, both much higher than the STDEV of 0.70 for *Function-Only*. Moreover, we find that *NFR-Enhanced* exhibits higher variability in Pass@1 than *NFR-Integrated*, ***which aligns with our earlier finding that LLMs are better at generating functionally correct code in one-step approach***.

Earlier versions of the LLMs also experience a much larger STDEV of Pass@1 after incorporating *NFR-Integrated* or *NFR-Enhanced*. For example, comparing the results of Code Design in HumanEval using *GPT-3.5-1106* vs. *GPT-4o-0513*, the STDEV for Pass@1 decreases from 2.71 to 1.49 for *NFR-Integrated*, and from to 18.72 to 1.23 for *NFR-Enhanced*. These findings suggest that ***some model versions may struggle to balance functional and non-functional requirements effectively***, especially in two-step enhancements, highlighting the

Task	Approach	Model	HumanEval Pass@1	HumanEval Δ(%)	HumanEval Average	HumanEval-ET Pass@1	HumanEval-ET Δ(%)	HumanEval-ET Average	MBPP Pass@1	MBPP Δ(%)	MBPP Average	MBPP-ET Pass@1	MBPP-ET Δ(%)	MBPP-ET Average
Function-Only (*Functional*)	Raw	**GPT3.5-1106**	76.46±0.77	-		66.83±0.51	-		63.47±0.55	-		44.75±0.64	-	
		GPT3.5-0125	72.50±0.73	-		64.33±1.05	-		67.82±0.48	-		47.21±0.39	-	
		GPT4o-0513	92.56±0.85	-	**84.61±0.70**	81.52±1.00	-	**74.50±0.74**	75.34±0.58	-	**71.57±0.41**	53.91±0.49	-	**51.19±0.33**
		GPT4o-0806	90.55±1.16	-		80.18±0.96	-		74.43±0.55	-		53.37±0.34	-	
		Claude3.5-0620	89.39±0.33	-		78.54±0.51	-		75.97±0.27	-		54.94±0.13	-	
		Claude3.5-1022	86.22±0.33	-		75.61±0.43	-		72.37±0.00	-		52.93±0.00	-	
Code Design (*NFR-Aware*)	NFR Integrated	**GPT3.5-1106**	72.44±2.71	-5.26		64.63±2.80	-3.29		66.53±1.05	4.82		46.49±0.85	3.89	
		GPT3.5-0125	72.87±1.82	0.51		64.51±2.15	0.28		67.68±1.40	-0.21		47.61±1.08	0.85	
		GPT4o-0513	90.73±1.49	-1.98	**81.69±1.84**	80.12±1.93	-1.72	**71.93±2.09**	73.79±0.74	-2.06	**69.50±2.06**	52.95±1.04	-1.78	**49.18±1.71**
		GPT4o-0806	89.33±0.77	-1.35	↓ 3.46%	79.63±1.08	-0.69	↓ 3.45%	73.37±1.12	-1.42	↓ 2.89%	52.58±1.10	-1.48	↓ 3.91%
		Claude3.5-0620	84.02±1.85	-6.01		72.56±1.93	-7.61		70.87±2.58	-6.71		49.79±2.23	-9.37	
		Claude3.5-1022	80.73±2.42	-6.37		70.12±2.62	-7.26		64.73±5.47	-10.56		45.67±3.93	-13.72	
	NFR Enhanced	**GPT3.5-1106**	52.62±18.72	-31.18		47.62±17.28	-28.74		40.49±18.34	-36.21		28.22±12.82	-36.94	
		GPT3.5-0125	55.85±12.36	-22.97		49.02±11.19	-23.80		43.77±12.48	-35.46		29.93±8.29	-36.60	
		GPT4o-0513	88.66±1.23	-4.21	**72.43±7.54**	78.90±1.23	-3.21	**64.02±6.84**	71.12±1.14	-5.60	**53.48±12.12**	50.87±1.00	-5.64	**37.73±8.64**
		GPT4o-0806	87.07±2.24	-3.84	↓ 14.40%	77.01±2.07	-3.95	↓ 14.07%	70.56±1.6	-5.20	↓ 25.27%	50.59±1.51	-5.21	↓ 26.28%
		Claude3.5-0620	76.83±5.91	-14.05		67.07±5.01	-14.60		50.82±22.53	-33.11		35.13±16.31	-36.06	
		Claude3.5-1022	73.54±4.77	-14.71		64.51±4.26	-14.68		44.12±16.65	-39.04		31.66±11.91	-40.19	
Readability (*NFR-Aware*)	NFR Integrated	**GPT3.5-1106**	73.29±3.56	-4.15		64.82±2.88	-3.01		66.93±2.38	5.45		47.26±1.60	5.61	
		GPT3.5-0125	73.17±2.80	0.92		64.33±1.91	0.00		68.76±1.51	1.39		48.41±1.19	2.54	
		GPT4o-0513	92.74±1.30	0.19	**83.51±2.42**	81.89±1.52	0.45	**73.61±2.16**	73.72±1.32	-2.15	**69.89±2.75**	52.67±0.74	-2.30	**49.66±1.89**
		GPT4o-0806	91.40±1.64	0.94	↓ 1.30%	80.98±1.60	1.00	↓ 1.20%	75.04±0.87	0.82	↓ 2.34%	53.63±0.89	0.49	↓ 2.99%
		Claude3.5-0620	86.46±1.80	-3.28		75.85±1.81	-3.43		73.35±2.64	-3.45		51.66±1.52	-5.97	
		Claude3.5-1022	84.02±3.41	-2.55		73.78±3.26	-2.42		61.55±7.79	-14.95		44.31±5.41	-16.29	
	NFR Enhanced	**GPT3.5-1106**	62.56±14.36	-18.18		55.30±12.76	-17.25		52.44±8.67	-17.38		36.63±5.42	-18.15	
		GPT3.5-0125	62.20±6.10	-14.21		55.18±5.83	-14.22		57.35±3.98	-15.44		39.44±2.39	-16.46	
		GPT4o-0513	91.34±0.94	-1.32	**76.05±5.96**	80.49±0.76	-1.26	**66.87±5.29**	72.76±1.19	-3.42	**57.41±8.08**	51.76±1.14	-3.99	**40.67±5.51**
		GPT4o-0806	88.96±1.30	-1.76	↓ 10.12%	78.66±1.15	-1.90	↓ 10.25%	72.67±1.15	-2.36	↓ 19.78%	52.15±0.85	-2.29	↓ 20.55%
		Claude3.5-0620	80.85±5.57	-9.55		70.85±4.73	-9.79		55.18±25.14	-27.37		38.55±17.58	-29.83	
		Claude3.5-1022	70.37±7.50	-18.38		60.73±6.50	-19.68		34.05±8.32	-52.95		25.48±5.7	-51.86	
Reliability (*NFR-Aware*)	NFR Integrated	**GPT3.5-1106**	65.73±4.29	-14.03		57.62±4.40	-13.78		45.11±11.71	-28.93		30.80±8.21	-31.17	
		GPT3.5-0125	68.29±3.50	-5.81		59.09±3.62	-8.15		42.93±13.93	-36.70		29.46±9.60	-37.60	
		GPT4o-0513	89.09±1.92	-3.75	**77.71±2.64**	76.46±2.23	-6.21	**66.95±2.79**	71.59±0.83	-4.98	**58.00±5.56**	50.35±1.01	-6.60	**39.91±4.03**
		GPT4o-0806	88.29±1.25	-2.50	↓ 8.15%	76.22±1.52	-4.94	↓ 10.13%	71.59±0.88	-3.82	↓ 18.96%	50.02±0.7	-6.28	↓ 22.03%
		Claude3.5-0620	81.83±2.64	-8.46		70.12±2.96	-10.72		69.32±1.81	-8.75		46.79±1.96	-14.83	
		Claude3.5-1022	73.05±2.26	-15.27		62.20±2.02	-17.74		47.45±4.21	-34.43		32.04±2.68	-39.47	
	NFR Enhanced	**GPT3.5-1106**	62.07±9.94	-18.82		53.48±9.17	-19.98		54.71±10.60	-13.80		38.41±7.58	-14.17	
		GPT3.5-0125	66.52±3.15	-8.25		58.96±3.40	-8.35		64.45±1.99	-4.97		43.65±1.68	-7.54	
		GPT4o-0513	88.78±1.53	-4.08	**72.75±4.54**	75.85±1.71	-6.96	**62.09±4.36**	72.97±1.13	-3.15	**61.04±4.08**	50.82±0.77	-5.73	**42.38±2.93**
		GPT4o-0806	86.10±2.11	-4.91	↓ 14.02%	74.51±1.46	-7.07	↓ 16.66%	71.17±0.88	-4.38	↓ 14.71%	49.63±0.69	-7.01	↓ 17.21%
		Claude3.5-0620	76.71±4.59	-14.19		63.54±6.12	-19.10		66.84±2.62	-12.02		46.09±1.79	-16.11	
		Claude3.5-1022	56.34±5.89	-34.66		46.22±4.30	-38.87		36.11±7.24	-50.10		25.67±5.05	-51.50	
Performance (*NFR-Aware*)	NFR Integrated	**GPT3.5-1106**	72.26±1.58	-5.49		63.54±2.13	-4.92		65.95±2.16	3.91		47.14±1.41	5.34	
		GPT3.5-0125	70.79±3.45	-2.36		61.83±2.93	-3.89		66.63±1.92	-1.75		47.82±1.47	1.29	
		GPT4o-0513	90.18±1.56	-2.57	**81.32±1.98**	80.73±1.63	-0.97	**72.04±1.78**	73.54±0.47	-2.39	**70.49±1.57**	52.95±0.67	-1.78	**50.39±1.47**
		GPT4o-0806	89.33±2.12	-1.35	↓ 3.89%	80.18±1.61	0.00	↓ 3.30%	74.07±0.72	-0.48	↓ 1.50%	53.56±0.8	0.36	↓ 1.56%
		Claude3.5-0620	83.29±1.53	-6.82		74.02±1.40	-5.76		72.04±1.76	-5.17		51.43±2.08	-6.39	
		Claude3.5-1022	81.32±1.98	-4.81		71.95±0.96	-4.84		70.73±2.36	-2.27		49.41±2.39	-6.65	
	NFR Enhanced	**GPT3.5-1106**	67.26±2.54	-22.33		54.09±5.95	-19.06		65.71±1.38	3.53		47.28±1.03	5.65	
		GPT3.5-0125	87.56±1.16	-7.23		59.39±2.61	-7.68		66.35±1.70	-2.17		47.00±1.18	-0.44	
		GPT4o-0513	86.10±1.11	-5.40	**77.02±2.70**	78.72±1.30	-3.43	**68.23±2.49**	73.14±0.66	-2.92	**68.53±2.01**	52.72±0.64	-2.21	**48.98±1.67**
		GPT4o-0806	81.34±3.07	-4.91	↓ 8.97%	77.44±0.86	-3.42	↓ 8.41%	74.15±1.05	-0.38	↓ 4.24%	53.82±0.92	0.84	↓ 4.30%
		Claude3.5-0620	80.49±1.22	-9.01		71.71±3.67	-8.70		70.82±1.11	-6.78		49.79±1.27	-9.37	
		Claude3.5-1022	77.02±2.70	-6.65		68.05±0.55	-10.00		61.03±6.14	-15.67		43.28±4.98	-18.23	

Table 1: The *Pass@1* column represents the Pass@1 scores along with their STDEV across 10 semantically equivalent prompts. Δ indicates the percentage difference in Pass@1 of the same model version between the NFR-aware results and the *Function-Only* result. The *Average* column provides the average Pass@1 scores and STDEV across all models, as well as the percentage difference relative to the *Function-Only* results.

need for regression testing across versions.

> LLMs achieve better functional correctness when NFRs are specified in the same prompt. However, incorporating NFRs generally reduces Pass@1, which shows challenges for LLMs in balancing NFRs and functional correctness.

RQ2: How Do NFR-Aware Workflows Affect Non-Functional Code Quality?

Motivation. Apart from functional correctness (i.e., Pass@1), how NFRs are addressed is crucial in NFR-aware coding workflows. This RQ evaluates the generated code by studying NFR metrics.

Approach. We follow the approaches and metrics described in Section 3.4 to study the non-functional code quality. We study whether incorporating NFRs can enhance NFR metrics by comparing the baseline (*Function-Only*) with *NFR-Integrated* and *NFR-Enhanced*. We report only the results for HumanEval and MBPP because they share the same generated code with the ET version (the ET version contains more test cases). Since the problems have different difficulties and length, we measure the execution time only for problems that successfully

Task	Approach	Model	HumanEval				MBPP			
			code smell (Δ%)	unreadability (Δ%)	exception-handling (Δ%)	execution time (Δ%)	code smell (Δ%)	unreadability (Δ%)	exception-handling (Δ%)	execution time (Δ%)
Func.-Only (*Functional*)	Raw	**GPT3.5-1106**	0.38±0.01	2.77±0.04	0.011±0.003	110.78±46.55	0.32±0.01	3.64±0.02	0.006±0.000	48.84±1.33
		GPT3.5-0125	0.31±0.01	3.42±0.04	0.036±0.003	112.63±48.09	0.27±0.01	3.44±0.03	0.011±0.000	43.18±7.96
		GPT4o-0513	0.13±0.01	2.50±0.04	0.040±0.002	77.40±13.78	0.10±0.00	2.72±0.03	0.129±0.007	34.69±0.21
		GPT4o-0806	0.12±0.01	2.62±0.03	0.026±0.005	75.92±4.23	0.12±0.00	3.18±0.02	0.130±0.005	37.23±1.37
		Claude3.5-0620	0.10±0.01	2.03±0.01	0.037±0.003	57.06±2.93	0.08±0.00	2.67±0.01	0.069±0.002	40.92±3.68
		Claude3.5-1022	0.06±0.00	2.60±0.02	0.022±0.000	70.31±13.75	0.03±0.00	2.69±0.00	0.041±0.000	34.40±0.30
		Average	0.18±0.01	2.66±0.03	0.029±0.003	84.02±21.55	0.15±0.01	3.06±0.02	0.064±0.002	39.88±2.47
Code Design (*NFR-Aware*)	NFR Integrated	**GPT3.5-1106**	0.25±0.01 (-34.2%)	1.79±0.10 (-35.4%)	0.055±0.058 (400.0%)	97.55±49.53(-11.94%)	0.35±0.04 (9.4%)	3.71±0.18 (1.9%)	0.126±0.118 (2000.0%)	51.66±11.29(5.77%)
		GPT3.5-0125	0.22±0.03 (-29.0%)	2.68±0.20 (-21.6%)	0.051±0.038 (41.7%)	92.93±32.03(-17.49%)	0.28±0.03 (3.7%)	4.68±0.48 (36.0%)	0.117±0.106 (963.6%)	58.78±11.12(36.13%)
		GPT4o-0513	0.06±0.01 (-53.8%)	1.35±0.17 (-46.0%)	0.095±0.027 (137.5%)	81.57±6.25(5.39%)	0.06±0.01 (-40.0%)	2.23±0.12 (-18.0%)	0.363±0.049 (181.4%)	38.64±4.47(11.39%)
		GPT4o-0806	0.06±0.00 (-50.0%)	1.27±0.15 (-51.5%)	0.091±0.018 (250.0%)	81.22±6.32(6.98%)	0.05±0.00 (-58.3%)	1.79±0.08 (-43.7%)	0.363±0.029 (179.2%)	38.30±3.03(2.87%)
		Claude3.5-0620	0.02±0.01 (-80.0%)	0.79±0.10 (-61.1%)	0.148±0.064 (300.0%)	47.27±15.87(-17.16%)	0.03±0.01 (-62.5%)	1.66±0.18 (-37.8%)	0.454±0.142 (558.0%)	36.42±0.18(-11.00%)
		Claude3.5-1022	0.02±0.01 (-66.7%)	1.54±0.50 (-40.8%)	0.176±0.118 (700.0%)	55.95±1.23(-20.42%)	0.01±0.01 (-66.7%)	1.95±0.64 (-27.5%)	0.445±0.171 (985.4%)	35.18±1.83(2.27%)
		Average	0.10±0.01 (↓ **44.4%**)	1.57±0.20 (↓ **41.0%**)	0.103±0.054 (↑ **255.2%**)	76.08±18.54 (↓ **9.45%**)	0.13±0.02 (↓ **13.3%**)	2.67±0.28 (↓ **12.7%**)	0.311±0.103 (↑ **385.9%**)	43.16±5.32 (↑ **8.22%**)
	NFR Enhanced	**GPT3.5-1106**	0.14±0.05 (-63.2%)	1.01±0.33 (-63.5%)	0.015±0.017 (36.4%)	111.79±89.17(0.91%)	0.10±0.05 (-68.8%)	2.83±1.22 (-22.3%)	0.024±0.018 (300.0%)	49.19±4.57(0.72%)
		GPT3.5-0125	0.07±0.03 (-77.4%)	1.30±0.21 (-62.0%)	0.032±0.012 (-11.1%)	64.45±24.73(-42.78%)	0.08±0.02 (-70.4%)	3.27±0.86 (-4.9%)	0.049±0.042 (345.5%)	43.46±3.36(0.65%)
		GPT4o-0513	0.05±0.01 (-61.5%)	1.48±0.10 (-40.8%)	0.064±0.017 (60.0%)	72.02±10.94(-6.95%)	0.03±0.00 (-70.0%)	2.29±0.15 (-15.8%)	0.220±0.061 (70.5%)	41.55±0.69(19.78%)
		GPT4o-0806	0.05±0.01 (-58.3%)	1.60±0.08 (-38.9%)	0.075±0.022 (188.5%)	71.32±10.69(-6.06%)	0.03±0.01 (-75.0%)	2.30±0.11 (-27.7%)	0.227±0.062 (74.6%)	41.69±0.28(11.98%)
		Claude3.5-0620	0.02±0.01 (-80.0%)	0.97±0.25 (-52.2%)	0.155±0.088 (318.9%)	55.73±21.47(-2.33%)	0.01±0.00 (-87.5%)	0.78±0.24 (-70.8%)	0.384±0.076 (456.5%)	38.38±4.90(-6.21%)
		Claude3.5-1022	0.02±0.00 (-66.7%)	1.00±0.24 (-61.5%)	0.196±0.076 (790.9%)	62.89±4.48(-10.55%)	0.01±0.00 (-66.7%)	1.61±0.20 (-40.1%)	0.437±0.097 (965.9%)	43.84±19.54(27.44%)
		Average	0.06±0.02 (↓ **66.7%**)	1.23±0.20 (↓ **53.8%**)	0.090±0.039 (↑ **210.3%**)	73.03±26.91 (↓ **13.08%**)	0.05±0.02 (↓ **66.7%**)	2.18±0.46 (↓ **28.8%**)	0.224±0.059 (↑ **250.0%**)	43.02±5.56 (↑ **7.87%**)
Readability (*NFR-Aware*)	NFR Integrated	**GPT3.5-1106**	0.21±0.04 (-44.7%)	1.58±0.09 (-43.0%)	0.015±0.007 (36.4%)	97.58±46.15(-11.92%)	0.30±0.03 (-6.3%)	3.42±0.28 (-6.0%)	0.012±0.005 (100.0%)	52.02±5.54(6.51%)
		GPT3.5-0125	0.18±0.03 (-41.9%)	2.47±0.24 (-27.8%)	0.016±0.005 (-55.6%)	120.41±49.29(6.91%)	0.24±0.02 (-11.1%)	4.14±0.49 (20.3%)	0.013±0.006 (18.2%)	49.45±10.22(14.52%)
		GPT4o-0513	0.07±0.01 (-46.2%)	1.38±0.15 (-44.8%)	0.038±0.012 (-5.0%)	81.52±9.24(5.32%)	0.08±0.01 (-20.0%)	2.26±0.10 (-16.9%)	0.122±0.031 (-5.4%)	37.39±2.32(7.78%)
		GPT4o-0806	0.06±0.01 (-50.0%)	1.25±0.07 (-52.3%)	0.029±0.007 (11.5%)	84.49±5.31(11.29%)	0.07±0.01 (-41.7%)	1.86±0.19 (-41.5%)	0.107±0.033 (-17.7%)	36.39±2.33(-2.26%)
		Claude3.5-0620	0.04±0.02 (-60.0%)	0.97±0.20 (-52.2%)	0.084±0.042 (127.0%)	44.04±13.54(-22.82%)	0.05±0.02 (-37.5%)	1.54±0.24 (-42.3%)	0.261±0.097 (278.3%)	39.04±8.94(-4.59%)
		Claude3.5-1022	0.02±0.01 (-66.7%)	1.38±0.20 (-46.9%)	0.088±0.040 (300.0%)	61.56±10.61(-12.44%)	0.02±0.01 (-33.3%)	1.61±0.10 (-40.1%)	0.226±0.086 (451.2%)	35.33±1.56(2.70%)
		Average	0.10±0.02 (↓ **44.4%**)	1.51±0.16 (↓ **43.2%**)	0.045±0.019 (↑ **55.2%**)	81.60±22.35 (↓ **2.88%**)	0.12±0.02 (↓ **20.0%**)	2.47±0.23 (↓ **19.3%**)	0.124±0.043 (↑ **93.8%**)	41.60±5.15 (↑ **4.31%**)
	NFR Enhanced	**GPT3.5-1106**	0.18±0.05 (-52.6%)	1.16±0.27 (-58.1%)	0.007±0.004 (-36.4%)	96.66±51.73(-12.75%)	0.20±0.05 (-37.5%)	3.46±0.55 (-4.9%)	0.009±0.004 (50.0%)	52.68±3.40(7.86%)
		GPT3.5-0125	0.09±0.02 (-71.0%)	1.46±0.14 (-57.3%)	0.019±0.002 (-47.2%)	88.61±55.58(-21.33%)	0.14±0.02 (-48.1%)	4.25±0.23 (23.5%)	0.013±0.006 (18.2%)	45.40±10.62(5.14%)
		GPT4o-0513	0.05±0.01 (-61.5%)	1.48±0.08 (-40.8%)	0.032±0.006 (-20.0%)	78.37±11.33(1.25%)	0.04±0.01 (-60.0%)	2.23±0.05 (-18.0%)	0.147±0.013 (14.0%)	41.32±0.29(19.11%)
		GPT4o-0806	0.06±0.01 (-50.0%)	1.61±0.11 (-38.5%)	0.020±0.006 (-23.1%)	76.72±4.10(1.05%)	0.04±0.01 (-66.7%)	2.25±0.11 (-29.2%)	0.105±0.010 (-19.2%)	41.66±0.23(11.90%)
		Claude3.5-0620	0.02±0.01 (-80.0%)	0.90±0.18 (-55.7%)	0.083±0.024 (124.3%)	50.04±20.34(-12.30%)	0.02±0.01 (-75.0%)	0.88±0.40 (-67.0%)	0.272±0.100 (294.2%)	36.56±1.15(-10.65%)
		Claude3.5-1022	0.02±0.00 (-66.7%)	0.86±0.15 (-66.9%)	0.173±0.037 (686.4%)	60.46±4.04(-14.01%)	0.01±0.00 (-66.7%)	1.10±0.06 (-59.1%)	0.364±0.050 (787.8%)	47.31±24.16(37.53%)
		Average	0.07±0.02 (↓ **61.1%**)	1.24±0.15 (↓ **53.4%**)	0.056±0.013 (↑ **93.1%**)	75.14±24.52 (↓ **10.57%**)	0.07±0.02 (↓ **53.3%**)	2.36±0.23 (↓ **22.9%**)	0.152±0.031 (↑ **137.5%**)	44.15±6.64 (↑ **10.71%**)
Reliability (*NFR-Aware*)	NFR Integrated	**GPT3.5-1106**	0.40±0.10 (5.3%)	1.92±0.23 (-30.7%)	1.362±0.311 (12281.8%)	117.04±54.46(5.65%)	0.45±0.12 (40.6%)	2.72±0.54 (-25.3%)	1.785±0.212 (29650.0%)	42.01±3.51(-13.98%)
		GPT3.5-0125	0.34±0.10 (9.7%)	2.81±0.40 (-17.8%)	1.342±0.247 (3627.8%)	117.86±41.17(4.64%)	0.36±0.07 (33.3%)	3.25±0.62 (-5.5%)	1.601±0.222 (14454.5%)	40.96±0.67(-5.14%)
		GPT4o-0513	0.10±0.03 (-23.1%)	1.66±0.19 (-33.6%)	0.910±0.136 (2175.0%)	87.69±1.44(13.29%)	0.18±0.08 (80.0%)	2.75±0.15 (1.1%)	1.588±0.192 (1131.0%)	35.44±1.81(2.16%)
		GPT4o-0806	0.10±0.04 (-16.7%)	1.45±0.16 (-44.7%)	0.942±0.157 (3523.1%)	90.48±4.98(19.18%)	0.17±0.08 (41.7%)	2.61±0.19 (-17.9%)	1.584±0.204 (1118.5%)	35.09±0.27(-5.75%)
		Claude3.5-0620	0.05±0.01 (-50.0%)	1.07±0.05 (-47.3%)	1.177±0.152 (3081.1%)	53.22±7.62(-6.73%)	0.05±0.01 (-37.5%)	1.98±0.45 (-25.8%)	1.354±0.107 (1862.3%)	43.66±15.62(6.70%)
		Claude3.5-1022	0.03±0.01 (-50.0%)	1.76±0.16 (-32.3%)	1.006±0.079 (4472.7%)	81.29±41.04(15.62%)	0.01±0.00 (-66.7%)	1.48±0.20 (-45.0%)	1.115±0.075 (2619.5%)	34.69±0.43(0.84%)
		Average	0.17±0.05 (↓ **5.6%**)	1.78±0.20 (↓ **33.1%**)	1.123±0.180 (↑ **3772.4%**)	91.27±25.12 (↑ **8.63%**)	0.20±0.06 (↑ **33.3%**)	2.47±0.36 (↓ **19.3%**)	1.504±0.169 (↑ **2250.0%**)	38.64±3.72 (↓ **3.11%**)
	NFR Enhanced	**GPT3.5-1106**	0.19±0.05 (-50.0%)	0.98±0.18 (-64.6%)	0.726±0.250 (6500.0%)	93.26±64.29(-15.82%)	0.30±0.10 (-6.3%)	2.52±0.56 (-30.8%)	1.653±0.345 (27450.0%)	50.54±3.06(3.48%)
		GPT3.5-0125	0.15±0.04 (-51.6%)	1.73±0.22 (-49.4%)	0.675±0.163 (1775.0%)	106.82±31.82(-5.16%)	0.27±0.08 (0.0%)	4.03±0.37 (17.2%)	1.448±0.264 (13063.6%)	46.30±11.45(7.23%)
		GPT4o-0513	0.07±0.02 (-46.2%)	1.76±0.12 (-29.6%)	0.797±0.128 (1892.5%)	78.57±5.62(1.51%)	0.07±0.05 (-30.0%)	2.30±0.11 (-15.4%)	1.190±0.168 (822.5%)	41.09±0.25(18.45%)
		GPT4o-0806	0.08±0.02 (-33.3%)	1.93±0.14 (-26.3%)	0.997±0.134 (3734.6%)	77.81±3.87(2.49%)	0.08±0.05 (-33.3%)	2.44±0.15 (-23.3%)	1.322±0.156 (916.9%)	37.01±2.69(-0.59%)
		Claude3.5-0620	0.04±0.01 (-60.0%)	0.94±0.25 (-53.7%)	1.036±0.142 (2700.0%)	48.61±11.23(-14.81%)	0.04±0.01 (-50.0%)	0.91±0.08 (-65.9%)	1.323±0.098 (1817.4%)	35.64±1.74(-12.90%)
		Claude3.5-1022	0.01±0.00 (-83.3%)	1.09±0.07 (-58.1%)	0.900±0.086 (3990.9%)	83.95±46.81(19.40%)	0.01±0.00 (-66.7%)	1.17±0.19 (-56.5%)	1.027±0.062 (2404.9%)	36.11±3.57(4.97%)
		Average	0.09±0.02 (↓ **50.0%**)	1.41±0.16 (↓ **47.0%**)	0.855±0.150 (↑ **2848.3%**)	81.50±27.27 (↓ **3.00%**)	0.13±0.05 (↓ **13.3%**)	2.23±0.24 (↓ **27.1%**)	1.327±0.182 (↑ **1973.4%**)	41.12±3.79 (↑ **3.11%**)
Performance (*NFR-Aware*)	NFR Integrated	**GPT3.5-1106**	0.32±0.06 (-15.8%)	2.41±0.18 (-13.0%)	0.014±0.003 (27.3%)	62.87±3.01(-43.25%)	0.32±0.05 (0.0%)	5.18±0.22 (42.3%)	0.011±0.003 (83.3%)	47.05±6.29(-3.67%)
		GPT3.5-0125	0.27±0.04 (-12.9%)	3.21±0.22 (-6.1%)	0.016±0.005 (-55.6%)	63.48±33.85(-43.64%)	0.28±0.06 (3.7%)	5.99±0.30 (74.1%)	0.011±0.002 (0.0%)	51.61±10.93(19.52%)
		GPT4o-0513	0.07±0.01 (-46.2%)	1.38±0.11 (-44.8%)	0.023±0.008 (-42.5%)	79.44±8.68(2.64%)	0.13±0.02 (30.0%)	3.31±0.19 (21.7%)	0.107±0.039 (-17.1%)	36.25±2.48(4.50%)
		GPT4o-0806	0.08±0.00 (-33.3%)	1.64±0.13 (-37.4%)	0.027±0.010 (3.8%)	74.34±1.00(-2.08%)	0.12±0.02 (0.0%)	3.26±0.18 (2.5%)	0.103±0.026 (-20.8%)	34.50±0.43(-7.33%)
		Claude3.5-0620	0.05±0.02 (-50.0%)	1.67±0.11 (-17.7%)	0.028±0.005 (-24.3%)	35.20±1.72(-38.31%)	0.05±0.00 (-37.5%)	2.51±0.12 (-6.0%)	0.096±0.047 (39.1%)	34.62±0.64(-15.40%)
		Claude3.5-1022	0.02±0.01 (-66.7%)	2.33±0.08 (-10.4%)	0.033±0.008 (50.0%)	98.39±24.43(39.94%)	0.02±0.00 (-33.3%)	2.42±0.11 (-10.0%)	0.070±0.027 (70.7%)	37.19±2.53(8.11%)
		Average	0.14±0.02 (↓ **22.2%**)	2.11±0.14 (↓ **20.7%**)	0.024±0.006 (↓ **17.2%**)	68.95±12.11 (↓ **17.94%**)	0.15±0.02 (↓ **0.0%**)	3.78±0.19 (↑ **23.5%**)	0.066±0.024 (↑ **3.1%**)	40.21±3.88 (↑ **0.83%**)
	NFR Enhanced	**GPT3.5-1106**	0.17±0.04 (-55.3%)	1.25±0.15 (-54.9%)	0.007±0.003 (-36.4%)	72.26±34.20(-34.77%)	0.29±0.06 (-9.4%)	4.42±0.26 (21.4%)	0.013±0.003 (116.7%)	47.27±7.24(-3.21%)
		GPT3.5-0125	0.21±0.04 (-32.3%)	2.53±0.11 (-26.0%)	0.019±0.005 (-47.2%)	53.59±29.05(-52.42%)	0.25±0.04 (-7.4%)	5.04±0.29 (46.5%)	0.018±0.008 (63.6%)	44.76±2.69(3.66%)
		GPT4o-0513	0.07±0.01 (-46.2%)	1.26±0.06 (-49.6%)	0.030±0.011 (-25.0%)	88.52±8.30(14.37%)	0.08±0.01 (-20.0%)	2.80±0.16 (2.9%)	0.112±0.041 (-13.2%)	40.76±0.19(17.50%)
		GPT4o-0806	0.08±0.01 (-33.3%)	1.46±0.11 (-44.3%)	0.036±0.008 (38.5%)	76.42±2.33(0.66%)	0.08±0.01 (-33.3%)	2.59±0.15 (-18.6%)	0.114±0.046 (-12.3%)	41.35±0.27(11.07%)
		Claude3.5-0620	0.03±0.00 (-70.0%)	1.30±0.09 (-36.0%)	0.057±0.018 (54.1%)	67.80±29.01(18.82%)	0.04±0.01 (-50.0%)	2.21±0.13 (-17.2%)	0.142±0.055 (105.8%)	38.56±1.56(-5.77%)
		Claude3.5-1022	0.03±0.01 (-50.0%)	1.90±0.08 (-26.9%)	0.062±0.019 (181.8%)	111.85±10.65(59.08%)	0.01±0.00 (-66.7%)	1.75±0.15 (-34.9%)	0.135±0.062 (229.3%)	34.82±1.96(1.22%)
		Average	0.10±0.02 (↓ **44.4%**)	1.62±0.10 (↓ **39.1%**)	0.035±0.011 (↑ **20.7%**)	78.41±18.92 (↓ **6.68%**)	0.13±0.02 (↓ **13.3%**)	3.14±0.19 (↑ **2.6%**)	0.089±0.036 (↑ **39.1%**)	41.25±2.32 (↑ **3.44%**)

Table 2: Columns *code smell density*, *unreadability density*, *exception-handling density*, and *execution time (millisecond)* represent the NFR metrics (Section 3). Each metric includes standard deviations and Δ%, which indicates the percentage difference between NFR-aware results and *Function-Only* results. *Average* summarizes mean scores, standard deviations, and percentage differences relative to the *Function-Only* results across all models.

pass the tests across all workflows.

<u>*Result.*</u> ***Incorporating NFRs consistently enhances NFR metrics.*** Table 2 presents the NFR results for *Function-Only*, *NFR-Integrated*, and *NFR-Enhanced* across all four NFRs and benchmarks. Notably, incorporating NFRs consistently improves all NFR metrics, irrespective of the specific NFRs. For example, considering code design NFRs enhances exception-handling density by 210.3%–255.2%, ***suggesting that incorporating even just one NFR may improve other dimensions of non-functional code quality***.

Unlike Pass@1, NFR-Enhanced leads to a larger improvement in certain non-functional code quality than NFR-Integrated. While *NFR-Integrated* outperforms *NFR-Enhanced* at Pass@1 (RQ1), *NFR-Enhanced* excels in improving NFR metrics. For code smell density, *NFR-Integrated* achieves a reduction of 13.3% and 44.4% on HumanEval and MBPP, respectively, whereas *NFR-Enhanced* reduces by 66.7% for both datasets. Similarly, for readability, *NFR-Integrated* improves by 19.3%–43.2%, while *NFR-Enhanced* achieves 22.9%–53.4% enhancements. Interestingly, an inverse pattern emerges for reliability, where *NFR-Integrated* outperforms *NFR-Enhanced* with improvements of 2250.0%–3772.4% for HumanEval and MBPP, compared to *NFR-Enhanced*'s

1973.4%–2848.3%. A similar trend is observed for performance metrics, with *NFR-Integrated* reducing execution time by 17.94% compared to *NFR-Enhanced*'s 6.68% in HumanEval, but no statistically significant difference in MBPP (t-test's p-value > 0.05). Our findings suggest that the two NFR-aware workflows have varying benefits depending on the NFRs. While *NFR-Enhanced* is more effective for improving readability and reducing code designs, *NFR-Integrated* may be better suited for addressing runtime-related requirements like exception handling and performance.

On average, NFR-Integrated and NFR-Enhanced share similar levels of stability in the NFR metrics, yet some versions of the models show much higher variability. *Function-Only* has the lowest STDEV across all NFR metrics, partly because of its lack of consideration of NFRs. In comparison, *NFR-Integrated* and *NFR-Enhanced* have larger STDEVs, but the values are often stable. For example, code smell density has an STDEV of 0.01–0.02, and unreadability density has an STDEV of 0.15–0.23 for both NFR-aware workflows. Similar to RQ1, some versions of the LLMs have much larger variability across all workflows. For instance, in HumanEval, the STDEV for code smell density (the *NFR-Integrated* row in *Readability*) is 0.24 for GPT3.5-0125 and 0.07 for GPT4o-0806, whereas the newer model shows much lower variability. However, a slightly older model, GPT3.5-1106, shows a lower STDEV of 0.09. Yet, *NFR-Enhanced* in *Readability* has an opposite finding, where GPT3.5-1106 has a larger STDEV than GPT3.5-0125 (0.27 vs. 0.14). ***This finding suggests that NFR's stability can be affected by specific model refinements, and the effect can be different for different NFR-aware workflows***, which may not always correlate with the model's general improvements. Future research should consider regression testing and data selection strategies during fine-tuning and model training to consider NFR and improve stability.

Incorporating NFRs improves the metrics of NFRs, with *NFR-Enhanced* excelling in readability and code structure-related design, and *NFR-Integrated* in exception handling and runtime-related performance. Variability across models highlights the need for careful regression testing and data selection to ensure consistent performance.

5 Discussion & Conclusion

5.1 Discussion of Implications

Our findings highlight implications for two key groups of stakeholders: (i) practitioners and (ii) LLM developers.

For Practitioners. Our findings suggest that practitioners should prioritize the *NFR-Integrated* when aiming to optimize both functional and non-functional requirements within a single iteration. This approach demonstrates lower variability and improved balance between competing objectives (i.e., Pass@1 vs. non-functional code quality).

For LLM Developers. The observed trade-offs between functional correctness and non-functional quality highlight the future direction to improve training process and fine-tuning. Future studies on LLMs may focus on enabling models to effectively address both functional and non-functional requirements, thereby reducing the observed trade-offs and variability. Moreover, future research could investigate advanced prompt engineering techniques or optimization mechanisms to mitigate performance variability and achieve superior alignment with complex software requirements.

5.2 Conclusion

This study investigates the challenges and opportunities associated with integrating NFRs into code generation workflows using LLMs. We propose *NFRGen*, a generalizable framework for evaluating LLM-generated code, which incorporates diverse workflows and non-functional quality metrics. The findings from our results underscore significant trade-offs between functional correctness and non-functional code quality attributes, such as design, readability, reliability, and performance.

Our study demonstrates that while incorporating NFRs reduces the functional correctness metric (i.e., Pass@1), notable improvements are observed in non-functional code quality metrics, including reductions in code smells and enhanced exception-handling density. The analysis of workflows reveals complementary strengths: the *NFR-Integrated* performs better in runtime-oriented aspects, such as performance and exception handling, whereas the *NFR-Enhanced* demonstrates higher efficiency in addressing structural aspects, such as readability and design improvements. By providing real-time feedback, *NFRGen* can be used to improve code quality, reduce manual testing, and enhance development efficiency.

Limitations

We use a certain set of widely used LLMs to conduct the experiments. The results may not apply to all models, as results may vary across different architectures and training methods. Future studies could benefit from incorporating a broader range of models to validate the results.

In this study, we have primarily examined Python datasets. While Python is a widely used language, the generalizability of our framework to other programming languages remains to be fully explored. However, our framework is not inherently language-specific. It is expected to be applicable to other languages and can be further verified by future studies.

The main objective of our framework is to evaluate the impact of different NFR-aware coding workflows on Pass@1 and non-functional code quality. Although *NFRGen* is not explicitly pre-trained for code refinement, it aligns with how developers use LLMs (e.g., zero-shot) for both code generation and refinement tasks. The insights derived from our evaluation can guide improvements in future model architectures, help prioritize areas for code optimization, and inform strategies for more effective handling of NFRs in generated code. Future work could investigate adjusting model training processes or providing more targeted NFR optimization during code generation and refinement.

Ethics Statement

We declare that all authors of this paper adhere to the ACM Code of Ethics and uphold its code of conduct. The aim of our work is to assess the robustness of LLMs in incorporating non-functional requirements (NFRs) to improve both functional correctness (Pass@1) and non-functional code quality. Our findings demonstrate that LLMs are capable of enhancing both dimensions, providing valuable insights for future research, and potential implications for industrial adoption, as commercial projects must adhere to various quality assurance practices, including non-functional requirements. Nevertheless, our results indicate that LLMs still require further refinement to achieve a better balance between functional and non-functional quality.

References

Authors Anonymous. 2024. Link to our replication package. https://anonymous.4open.science/r/NFRGen-8175.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Junkai Chen, Zhenhao Li, Xing Hu, and Xin Xia. 2024. Nlperturbator: Studying the robustness of code llms to natural language variations. *arXiv preprint arXiv:2406.19783*.

Lingjiao Chen, Matei Zaharia, and James Zou. 2023. How is chatgpt's behavior changing over time? *arXiv preprint arXiv:2307.09009*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Lawrence Chung and Julio Cesar Sampaio do Prado Leite. 2009. On non-functional requirements in software engineering. *Conceptual modeling: Foundations and applications: Essays in honor of john mylopoulos*, pages 363–379.

Copilot. 2024a. An overview about using copilot in visual studio. https://code.visualstudio.com/docs/copilot/overview. Accessed: 2024-11-05.

Copilot. 2024b. Using github copilot to ask some coding-related questions. https://docs.github.com/en/copilot/using-github-copilot/asking-github-copilot-questions-in-your-ide. Accessed: 2024-11-05.

Cursor. 2024. Cursor features. https://www.cursor.com/features. Accessed: 2024-11-05.

Guilherme B De Padua and Weiyi Shang. 2017. Revisiting exception handling practices with exception flow analysis. In *2017 IEEE 17th International working conference on source code analysis and manipulation (SCAM)*, pages 11–20.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2023. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.

Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 254–266.

Martin Glinz. 2007. On non-functional requirements. In *15th IEEE international requirements engineering conference (RE 2007)*, pages 21–26. IEEE.

Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. 2024. Archcode: Incorporating software requirements in code generation with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 13520–13552.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Seonghyeon Lee, Sanghwan Jang, Seongbo Jang, Dongha Lee, and Hwanjo Yu. 2024. Exploring language model's code generation ability with auxiliary functions. *arXiv preprint arXiv:2403.10575*.

Zhenhao Li, An Ran Chen, Xing Hu, Xin Xia, Tse-Hsun Chen, and Weiyi Shang. 2023. Are they all good? studying practitioners' expectations on the readability of log messages. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 129–140.

Feng Lin, Dong Jae Kim, Tse-Husn, and Chen. 2024. Soen-101: Code generation by emulating software process models using large language model agents. *arXiv preprint arXiv:2403.15852*.

Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. 2013. Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th international conference on software engineering (ICSE)*, pages 1012–1021.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. 2024. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*.

OpenAI. 2023. Chatgpt. https://chatgpt.com/.

José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. 2022. Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering*, pages 47–94.

Hoang Pham. 2000. *Software reliability*. Springer Science & Business Media.

Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. 2020. How does code readability change during software evolution? *Empirical Software Engineering*, pages 5374–5412.

PyCQA. 2024a. Pylint user guide: Messages overview - refactor. https://pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html#refactor. Accessed: 2024-12-03.

PyCQA. 2024b. Pylint user guide: Messages overview-convention. https://pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html#convention. Accessed:2024/12/03.

Pylint. 2024. What is pylint?

Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024. Ai-powered code review with llms: Early results.

Martin Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36.

Atsushi Shirafuji, Yutaka Watanobe, Takumi Ito, Makoto Morishita, Yuki Nakamura, Yusuke Oda, and Jun Suzuki. 2023. Exploring the robustness of large language models for solving programming problems. *arXiv preprint arXiv:2306.14583*.

Antonio Vitale, Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. 2023. Using deep learning to automatically improve code readability. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 573–584.

Bartosz Walter and Tarek Alkhaeir. 2016. The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, pages 127–142.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022. Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*.

Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2024. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for Effective Software Development*, pages 71–108.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. 2020. Learning to handle exceptions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 29–41.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2024. A survey on large language models for software engineering.

A Prompt Templates for NFR-Aware Code Generation

Table 3 shows the semantically equivalent prompt templates to consider the four NFRs in code-generation.

Error-handle	Code Smell	Readability	Performance
Incorporate various *error handling* techniques	Investigate various strategies to handle *code smell*	Evaluate different coding practices for *readability*	Optimize for *performance*
Implement multiple *exception handling* strategies	Minimize *code smell*	Investigate various techniques to enhance *readability*	Focus on enhancing *performance*
Apply different *error handling* mechanisms	Eliminate *code smell*	Improve the code *readability*	Ensure the code *runs efficiently*
Investigate different methods of *managing exceptions*	Identify and address different *code smells*	Ensure the code is *readable*	Prioritize *runtime optimization*
Integrate diverse *error handling* approaches	Apply best practices to reduce *code smell*	Apply coding practices that enhance *readability*	Keep *performance* in mind while solving
Utilize multiple *error management* techniques	Mitigate *code smell*	Focus on *readability*	Aim for *high-performance* execution
Experiment with various ways to *handle exceptions*	Tackle different *code smell* issues	Enhance the *readability* of the code	Reduce *computational overhead*
Combine different *error handling* practices	Implement techniques to prevent *code smell*	Implement strategies to make the code more *readable*	Emphasize *speed and efficiency*
Evaluate multiple *exception management* strategies	Resolve *code smell* problems	Optimize the code for better *readability*	Ensure minimal *resource consumption*
Develop a range of *error handling* solutions	Optimize code to avoid *code smell*	Adopt coding practices for improved *readability*	Maximize *performance* in your solution

Table 3: LLM generated prompt templates to consider non-functional requirements in code generation.

B Discussion on Metric Selection and Pylint

We measure the ability of LLMs to generate code based on non-functional requirements (NFRs) by focusing on specific dimensions such as maintainability, reliability, and performance. These dimensions are commonly used in software quality assurance and directly influence the quality of the generated code (Glinz, 2007). Morover, given the nature of datasets, generating function-level code, these NFRs are reasonable to evaluate compared to other NFRs such as portability and scalability.

In our study, we chose code design as a proxy for maintainability. For code design, some prior studies use metrics like cyclomatic complexity (Shepperd, 1988), where high complexity makes it hard to maintain code. However, in our research, we utilize code smell and readability as separate proxies for code design. We differentiate code design and readability since maintainability is too broad and may encompass both dimensions. Moreover, for LLM-generated code, factors such as readability, adherence to coding standards, and code smell (recurring bad design patterns) provide more interpretable and valuable meaning for maintainability for developers as opposed to control flow complexity (Shepperd, 1988).

We rely on Pylint, the most popular Python-based linter, to measure code smells and readability (Pylint, 2024). It identifies readability issues as "convention", which detects common coding errors like unused imports, and inconsistent naming conventions (PyCQA, 2024b). It also identifies code smell issues as "refactoring" (PyCQA, 2024a). Refactoring is a small structural characteristic in code that indicates a potential problem (code smells), suggesting that the code should be structurally changed, without changing its behavior, to improve design (Fowler, 2018).

Reliability is another NFR we study in LLM-generated code. Reliability may involve responding to unexpected events when a computer program runs (Pham, 2000). In particular, we measure whether the generated code includes exception-handling mechanisms, such as try-catch blocks, to gain insight into how well the code anticipates and manages potential errors.

C Failure Examples When The LLM Attempts To Address Both Functional And Non-Functional Requirements

In this section, we present a few code examples exposed by *NFRGen*, demonstrating that LLMs make some mistakes when addressing non-functional requirements such as reliability, readability, performance, and code design.

C.1 Reliability

To enhance reliability, LLMs often include additional exception-handling statements in the code. However, they sometimes make errors, such as using incorrect try-except formats or raising generic exceptions instead of returning specific results as described in the problem requirements.

Here is an example where the LLM made an error in the try-except format. As shown in Code 1, although the LLM attempted to incorporate error-handling logic, it failed to include the required except statements. This oversight caused the code to malfunction and resulted in an IndentationError when executed during evaluation.

```
def find_char_long(text):
    '''Write a function to find all words which are
     at least 4 characters long in a string.
    '''

    try:
    words = text.split()
    long_words = [word for word in words if len(word
    ) >= 4]
    return long_words
    print("Input must be a string")
    print("An error occurred:", e)
```

Code 1: MBPP-7 Test Failure: LLM Added Error Handling but Made a Format Error.

Even when the LLM ensures the correct format, the additional exception-handling statements may sometimes contradict the functional requirements. For example, as shown in Code 2, the error-handling code introduces extra checks for the triangle inequality theorem on lines 23-24, leading to failure. Specifically, the function's docstring states that such inputs should return False. However, the error-handling logic raises a ValueError instead, resulting in failure. This discrepancy highlights that while NFR-aware code generation can produce functionally correct logic, it may conflict with the specified evaluation criteria.

```python
1 def right_angle_triangle(a, b, c):
2     '''
3     Given the lengths of the three sides of a triangle. Return True if the three
4     sides form a right-angled triangle, False otherwise.
5     A right-angled triangle is a triangle in which one angle is right angle or
6     90 degree.
7 
8     Example:
9     right_angle_triangle(3, 4, 5) == True
10    right_angle_triangle(1, 2, 3) == False
11    '''
12
13    # Type checking
14    for side in (a, b, c):
15        if not isinstance(side, (int, float)):
16            raise TypeError(f"Invalid type: {side}. All sides must be numbers (int or float).")
17
18    # Value checking
19    if a <= 0 or b <= 0 or c <= 0:
20        raise ValueError("All sides must be positive numbers.")
21
22    # Triangle inequality theorem
23    if a + b <= c or a + c <= b or b + c <= a:
24        raise ValueError("The given sides do not form a triangle.")
25
26    # Check for right angle using Pythagorean theorem
27    sides = sorted([a, b, c])  # Sort the sides to identify the longest side
28    return sides[0]**2 + sides[1]**2 == sides[2]**2
```

Code 2: HumanEval-157 Test Failure Due to *raise ValueError* Which Violates the Functional Requirement.

C.2 Code Design

When provided with only functional requirements, the LLM can generate correct code; however, the generated code may still exhibit code smell issues. For instance, as shown in Code 3, the function-only solution passes the evaluation but relies heavily on multiple if-else statements for validation checks, introducing potential maintainability concerns.

```python
def valid_date(date):
    """You have to write a function which validates a given date string and
    returns True if the date is valid otherwise False.
    The date is valid if all of the following rules are satisfied:
    1. The date string is not empty.
    2. The number of days is not less than 1 or higher than 31 days for months 1,3,5,7,8,10,12. And the number of days is not less than 1 or higher than 30 days for months 4,6,9,11. And, the number of days is not less than 1 or higher than 29 for the month 2.
    3. The months should not be less than 1 or higher than 12.
    4. The date should be in the format: mm-dd-yyyy
    for example:
    valid_date('03-11-2000') => True
    valid_date('15-01-2012') => False
    valid_date('04-0-2040') => False
    valid_date('06-04-2020') => True
    valid_date('06/04/2020') => False
    """
    # Check if the date string is empty
    if not date:
        return False
    # Check if the date string is in the correct format
    if len(date) != 10 or date[2] != '-' or date[5] != '-':
        return False
    # Split the date string into month, day, and year
    try:
        month, day, year = map(int, date.split('-'))
    except ValueError:
        return False
    # Check if the month is valid
    if month < 1 or month > 12:
        return False
    # Check if the day is valid based on the month
    if month in {1, 3, 5, 7, 8, 10, 12}:
        if day < 1 or day > 31:
            return False
    elif month in {4, 6, 9, 11}:
        if day < 1 or day > 30:
            return False
    elif month == 2:
        if day < 1 or day > 29:
            return False
    # If all checks pass, the date is valid
    return True
```

Code 3: HumanEval-124 correct but has code smell.

LLMs may have the capability to avoid code smells, but they can sometimes make mistakes during the process. For instance, as shown in Code 4, when prompted to avoid a specific code smell, the LLM attempted to mitigate it by using a regular expression pattern. However, it failed to produce a correct and functional pattern. These inaccuracies resulted in AssertionError exceptions during evaluation when the function was executed.

```python
import re
def valid_date(date):
    # Check if the date string is empty
    if not date:
        return False

    # Define the regex pattern for the date format
    date_pattern = re.compile(r'^(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])-(\d{4})$')

    # Check if the date matches the pattern
    match = date_pattern.match(date)
    if not match:
        return False

    # Extract month, day, and year from the date
    month, day, year = map(int, match.groups())

    # Define the number of days in each month
    days_in_month = {
        1: 31, 2: 29, 3: 31, 4: 30, 5: 31, 6: 30,
        7: 31, 8: 31, 9: 30, 10: 31, 11: 30, 12: 31
    }

    # Check if the day is valid for the given month
    if day < 1 or day > days_in_month[month]:
        return False
    return True
```

```
assert candidate("03-15-957") == True # match fail
```

Code 4: HumanEval-124 Test Failure while LLM using re to avoid code smell but introduce other error.

C.3 Readability

Similar to its ability to address code design, LLMs can enhance the readability of generated code. However, we observed that LLMs sometimes prioritize suggesting strategies over producing more readable code. For example, as shown in Code 5, while the LLM attempted to improve code readability, it occasionally failed to generate valid code. Instead, it returned strategy descriptions as responses, which led to evaluation failures. When executed, this resulted in a SyntaxError during evaluation.

```
One way to enhance readability and improve the code is to add comments to explain the logic and steps of the algorithm. This will make it easier for others (and yourself) to understand the code.
Another way to improve the code is to use more descriptive variable names. For example, instead of using "l" as the variable name for the list, you could use a more descriptive name such as "input_list".
Additionally, you could extract the logic inside the function into smaller, more descriptive functions. For example, you could create a function to check if a triple of elements sum to zero, and then call that function inside the main function.
Lastly, you could add type hints to the function signature to make it clear what type of input the function expects and what type of output it returns. This can improve readability and help catch potential errors early on.
```

Code 5: HumanEval-40 Test Failure while LLM improve readability.

C.4 Performance

In the absence of specific requirements for code generation, LLMs may prioritize generating correct code while overlooking other critical factors, such as execution time. For instance, as shown in Code 6, the function-only solution implements an $O(n^2)$ algorithm. While the code is functionally correct, it is computationally inefficient.

```
def car_race_collision(n: int):
    """
    Simulates the car collisions explicitly and calculates the number of collisions.
    Arguments:
    n -- Number of cars in each group (left-to-right and right-to-left)
    Returns:
    int -- Number of collisions
    """

    # Initialize collision count
    collisions = 0

    # Iterate through each car in the left-to-right group
    for _ in range(n):
        # Each car collides with all cars in the right-to-left group
        for _ in range(n):
            collisions += 1
    return collisions
```

Code 6: HumanEval-41, Function-only, correct but slow.

When tasked with optimizing an algorithm's runtime, the LLM occasionally makes mistakes, such as using incorrect formulas to solve the problem. For instance, in Code 7, the correct formula for calculating the number of collisions is $n \times n$. However, the LLM sometimes employed alternative mathematical formulas, resulting in incorrect outputs. During evaluation, these inaccuracies led to AssertionError exceptions when the function was executed.

```
# Performance-Code-1: Correct and Efficient
def car_race_collision(n: int):
    return n * n

# Performance-Code-2: Efficient but Incorrect
def car_race_collision(n: int):
    return n * (n - 1) // 2
```

Code 7: HumanEval-41 Test Failure while LLM improve performance but use wrong formula.