

EVALOOOP: A Self-Consistency-Centered Framework for Assessing Large Language Model Robustness in Programming

Sen Fang Weiyuan Ding Bowen Xu NC State University 

{sfang9, wding8, bxu22}@ncsu.edu

Abstract

Evaluating the programming robustness of large language models (LLMs) is paramount for ensuring their reliability in AI-based software development. However, adversarial attacks exhibit fundamental limitations that compromise fair robustness assessment: they demonstrate contradictory evaluation outcomes where different attack strategies tend to favor different models, and more critically, they operate solely through external perturbations, failing to capture the intrinsic stability essential for autonomous coding agents where subsequent inputs are endogenously generated by the model itself. We introduce EVALOOOP, a novel assessment framework that evaluates robustness from a self-consistency perspective, leveraging the natural duality inherent in software engineering tasks (*e.g.*, code generation and code summarization). EVALOOOP establishes a self-contained feedback loop where an LLM iteratively transforms between code and natural language until functional failure occurs, with robustness quantified by a novel Average Sustainable Loops (ASL) metric—the mean number of iterations maintaining functional correctness across benchmark tasks. This cyclical strategy intrinsically evaluates robustness without relying on external attack configurations, providing a unified metric that reveals how effectively LLMs preserve semantic integrity through sustained self-referential transformations. We evaluate 96 popular LLMs, ranging from 0.5B to 685B parameters, on EVALOOOP equipped with the MBPP Plus benchmark, and found that EVALOOOP typically induces a 2.65%–47.62% absolute drop in pass@1 accuracy within ten loops. Intriguingly, robustness does not always align with initial performance (*i.e.*, one-time query); for instance, Qwen3-235B-A22B-Instruct-2507, despite inferior initial code generation compared to OpenAI’s o-series models and DeepSeek-V3, demonstrated the superior robustness (ASL score). EVALOOOP reveals distinct degradation patterns across different models, providing developers with a practical framework for evaluating LLM functional coherence through iterative transformations and offering complementary insights that support more informed model selection decisions.

1 Introduction

Robustness is a critical aspect for measuring the performance of Large Language Models (LLMs) in code generation [11, 38, 33, 28, 1, 26]. The reasons are threefold. First, **real-world consequences of non-robust code generation can be severe**: Research demonstrates that 11.56% of websites using LLM-generated PHP code could be compromised, with 26% having at least one exploitable

vulnerability [15], while Meta’s CyberSecEval studies reveal that approximately one in three AI-generated code pieces contain vulnerabilities [10, 9, 42]. Production incidents have included SQL injection vulnerabilities from string concatenation patterns, exposure of hard-coded API keys, and OS command injection flaws introduced through AI-suggested code [31, 48]. Second, **modern software development increasingly relies on LLM-powered tools**: Studies show that LLM-generated code has become integral to contemporary development workflows, with developers increasingly accepting AI-assisted code suggestions despite inherent security risks [29, 37, 21]. Third, **the shift toward agent-based software engineering** means that LLMs must maintain consistency across multiple interconnected tasks, where cascading inconsistencies can propagate through entire systems, turning minor semantic drift into complete workflow failures.

Evaluating the robustness of LLMs in programming often involves adversarial attacks [4, 43, 20, 32, 12, 52, 19, 45, 51, 8]. However, we find that existing works on evaluating LLM robustness are often biased due to their reliance on different adversarial attack strategies. For example, when we apply three adversarial attacks [39], *i.e.*, Case Transformation, Structural Reformatting, and Redundant Elaboration, to evaluate the robustness of deepseek-coder-7b-instruct-v1.5 and Qwen2.5-Coder- 32B-Instruct, we observe completely contradictory robustness patterns: deepseek-coder-7b-instruct-v1.5 demonstrates severe vulnerability to structural attacks (performance drops from 64.6% to 1.1%) while maintaining robustness under verbose prompt attacks, whereas Qwen2.5-Coder-32B-Instruct shows the opposite behavior—resilience to structural changes but significant degradation under verbose prompts (76.2% to 54.5%). As a result, both model developers and consumers could be easily misled depending on which attack methodology is employed for evaluation.

From the software engineering workflow perspective, agent-based frameworks represent the trending solution for complex programming tasks [53, 13, 46]. In these frameworks, one task’s output frequently serves as another task’s input, with each task handled by an LLM agent operating in sequence or parallel configurations. Notably, these agent systems typically employ post-trained LLMs that have undergone reinforcement learning [36, 40] or instruction-based fine-tuning [35, 16] to enhance their reliability and instruction-following capabilities. However, our empirical analysis reveals that such post-trained models demonstrate significant resilience to adversarial attacks, rendering traditional adversarial evaluation approaches largely ineffective for assessing their robustness. This creates a critical evaluation limitation: while adversarial attacks fail to meaningfully stress-test these resilient models, agent deployments require assessing a fundamentally different dimension of robustness—the model’s ability to maintain functional coherence through sustained self-referential transformations where subsequent inputs are endogenously generated by the model itself rather than provided by external adversaries. **These gaps underscores the urgent need for a unified approach for LLMs robustness evaluation in programming that does not depend on external attack configurations.**

Our solution. Inspired by cognitive resilience principles in human learning, we introduce EVALOOOP, a novel framework for evaluating LLM robustness in programming that addresses these fundamental limitations. Drawing from the Levels of Processing theory [14], deep cognitive engagement—such as achieving bilingual fluency or mastering complex dual relationships—builds robust knowledge structures that are inherently resistant to minor perturbations. Modern LLMs, having been trained on vast code corpus, demonstrably acquiring sophisticated coding capabilities and constructed resilient knowledge representations that should theoretically withstand iterative transformations.

Specifically, EVALOOOP practices this principle by leveraging the natural duality [44] between complementary software engineering tasks (e.g., code generation [24] and code summarization [2]) to create a self-contained feedback loop. Take the popular dual tasks, code generation and code summarization, as an example, starting from an initial natural language specification, the framework instructs an LLM to generate corresponding code, then tasks the same model with summarizing this generated code back into a new natural language specification. This cyclical process continues iteratively until the generated code fails to pass functional correctness tests, with robustness quantified by our proposed Average Sustainable Loops (ASL) metric—the mean number of successful iterations before functional failure occurs. EVALOOOP essentially constitutes a minimal agent-based system where each loop represents an autonomous agent interaction. Higher ASL scores indicate superior self-consistency, suggesting that the LLM is better suited as a foundation model for complex agent-based software systems. More crucially, EVALOOOP eliminates all external perturbations—beyond the initial input, all subsequent inputs are endogenously generated by the model itself—thereby

providing a fair, unbiased comparison framework that captures the intrinsic stability essential for real-world deployment scenarios.

Listing 1.1: The initial solution generated by o3-mini. 🍌

```
Initial prompt: Write a python function to determine whether a given list of numbers forms a full
sequence of consecutive integers without any duplicates.

assert check_Consecutive([1,2,3,4,5]) == True

Solution:

def check_Consecutive(lst):
    if not lst:
        return False
    s = set(lst)
    return max(lst) - min(lst) + 1 == len(lst) and s == set(range(min(lst), max(lst) + 1))
```

Listing 1.2: The solution generated by o3-mini in 5-th loop. 🗨️

```
Derived prompt: Write a python function to determine if a list of numbers contains an unbroken
sequence of consecutive integers from its minimum to its maximum value.

assert check_Consecutive([1,2,3,4,5]) == True

Solution:

def check_Consecutive(lst):
    if not lst:
        return False
    return set(lst) == set(range(min(lst), max(lst) + 1))

assert check_Consecutive([1,2,3,4,5]) == True
```

Figure 1: The generated solutions for MBPP⁺ No. 472 task by o3-mini.

Figure 1 demonstrates EVALOOP’s effectiveness through OpenAI o3-mini’s performance on MBPP Plus task No.472. Initially, the model generates a correct solution that properly checks both consecutive sequence formation and duplicate absence (Listing 1.1). However, by the 5th loop, despite receiving a semantically equivalent prompt derived from its own previous output, o3-mini produces a fundamentally flawed implementation (Listing 1.2) that fails to verify duplicate absence—erroneously returning True for inputs containing duplicates. This degradation exemplifies the critical self-consistency failures that adversarial attacks cannot capture. While external perturbations might leave this model seemingly robust, EVALOOP reveals how the same LLM loses functional coherence through endogenous transformations—precisely the type of cascading failures that compromise agent-based systems in practice. Such intrinsic instability, measurable only through sustained self-referential evaluation, directly impacts the reliability of autonomous coding agents where one model’s output becomes another’s input across multiple reasoning steps.

Contributions. Our main contributions are:

- **Problem Identification and Empirical Evidence:** We systematically identify and empirically validate critical limitations of adversarial attack-based robustness evaluation methods through comprehensive experiments across 13 state-of-the-art LLMs. Our findings reveal that these approaches exhibit negligible effectiveness against modern post-trained models (e.g., OpenAI’s o-series) and demonstrate inherent evaluation bias where different attack strategies tend to favor different models, leading to contradictory robustness conclusions that compromise fair comparison and practical applicability.
- **Novel Evaluation Framework (EVALOOP):** We propose EVALOOP, a self-consistency framework that leverages the natural duality between code generation and code summarization within a self-contained feedback loop to provide unified and unbiased evaluation. Our proposed Average Sustainable Loops (ASL) metric quantifies robustness through endogenous transformations, eliminating dependence on external attack configurations while capturing the intrinsic stability crucial for agent-based systems.
- **Large-scale Empirical Validation and Insights:** Through evaluation of 96 LLMs (0.5B-685B parameters) on MBPP Plus benchmark [27], we reveal that EVALOOP induces 2.65%-47.62% absolute performance drops within ten loops, with robustness patterns that diverge from initial performance rankings. Notably, models like Qwen3-235B-A22B-Instruct-2507 demonstrate superior

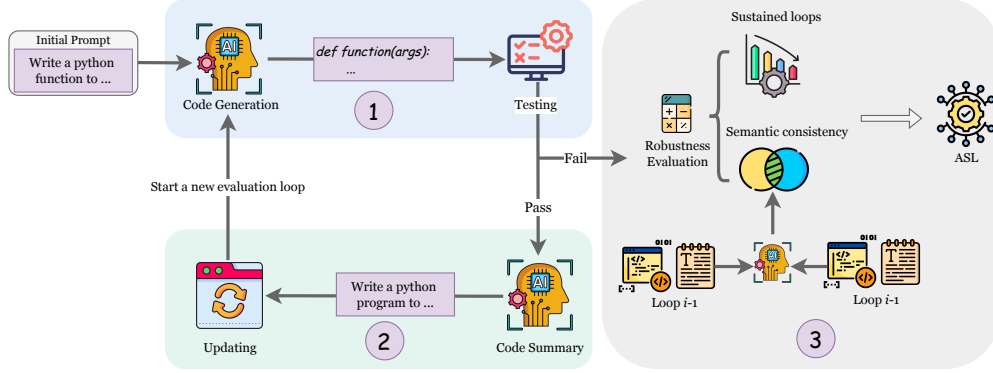


Figure 2: An overview of EVALOOP implemented by integrating code generation and summarization loop.

robustness despite inferior initial capabilities compared to OpenAI’s o-series, suggesting that sustained self-consistency may indicate deeper comprehension rather than memorization—offering developers practical guidance for selecting optimal foundation models in iterative, agent-based software systems. To our knowledge, this represents the first comprehensive unified assessment of LLM programming robustness at this scale, encompassing nearly 10 times more models than prior relevant studies and providing the research community with a reliable robustness ranking across the majority of modern LLMs.

- **Framework Extensibility through Code Translation:** We demonstrate EVALOOP’s generalization by extending it to code translation tasks, establishing dual transformation loops between different programming languages. This extension support our framework’s general applicability beyond generation-summarization pairs, showcasing its potential for evaluating robustness across diverse software engineering scenarios and proving its easily extensible nature for various dual-task configurations.
- **Open-Source Leaderboard and Resources:** We release a comprehensive online Leaderboard (Available at: <https://evaloop.github.io/>) that enables practitioners to interactively explore model robustness rankings across different ASL metrics, facilitating informed model selection for automated software engineering workflow.

2 Proposed Framework: EVALOOP

Figure 2 presents an overview of our novel framework EVALOOP for evaluating LLM robustness through self-consistency assessment. The core insight driving our approach is that robust LLMs should demonstrate consistent semantic understanding when processing complementary software engineering tasks—such as generating code from natural language specifications and subsequently summarizing that code back into natural language. Unlike adversarial approaches that rely on external perturbations, our framework creates endogenous transformations where models iteratively process their own outputs, revealing intrinsic stability patterns crucial in informed model selection for automated software engineering workflow. Moreover, this iterative evaluation provides a unified, bias-free metric for comparing LLM stability across diverse architectures and training paradigms.

2.1 Duality Loop

As illustrated in Figure 2, the evaluation process establishes a cyclical transformation where task outputs are iteratively converted between complementary representations. The code generation-summarization pair serves as our exemplar implementation due to its universal applicability and clear semantic relationship: code generation demands translating abstract requirements into concrete implementations, while code summarization requires extracting and articulating essential logic from existing code. However, the core principle of our framework readily extends to other dual task configurations in software engineering, such as iterative code translation across multiple programming languages, as we demonstrate in Section 5 through a code translation-based extension of EVALOOP.

Specifically, the evaluation loop operates through the following three key steps: ① Starting with an initial natural language prompt specifying a programming task, the target LLM generates executable code that undergoes rigorous testing against predefined test suites. ② Upon successful validation, the same LLM summarizes its generated code back into natural language, creating a new specification for the subsequent iteration. This process continues until the generated code fails functional correctness tests. ③ Upon failed validation, we compute the robustness of the given LLM on the programming task using our proposed Average Sustainable Loops (ASL) metric, which quantifies both loop sustainability and semantic consistency.

The bidirectional nature of this transformation is crucial—it requires the model to maintain semantic fidelity across two distinct cognitive tasks. Code generation demands translating abstract requirements into concrete implementations, while code summarization requires extracting and articulating the essential logic from existing code. A robust LLM should demonstrate consistency across both directions, maintaining the core functionality through successive transformations of its own creation while preserving semantic coherence throughout the iterative evaluation process.

2.2 Proposed Metric: Average Sustainable Loops (ASL)

To quantify LLM robustness through our iterative evaluation framework, we introduce the Average Sustainable Loops (ASL) metric. Effective robustness assessment requires capturing two critical dimensions simultaneously: the model’s ability to maintain functional performance across increasing loop iterations and the semantic consistency preserved between consecutive transformations. As illustrated in step 3 of Figure 2, our metric must account for both performance degradation patterns and the quality of semantic preservation throughout the evaluation cycle. Traditional robustness metrics typically focus on binary success/failure outcomes or simple performance drops, failing to distinguish between catastrophic failures due to sudden semantic collapse versus gradual degradation through accumulated drift. Our ASL metric addresses this limitation by incorporating semantic similarity analysis that reveals whether functional failures result from significant prompt evolution or sudden robustness breakdown despite minimal semantic changes.

Specifically, we define ASL as a comprehensive measure that incorporates both sustainability duration and semantic consistency:

$$\text{ASL} = \frac{\sum_{i=1}^M n_i \times i^2 \times s_i}{MT}, \quad (1)$$

where M represents the maximum number of evaluation loops, n_i denotes the number of tasks that sustain functionality through exactly i loops, T represents the total number of tasks in the benchmark dataset, and s_i represents the average semantic similarity score for tasks failing at loop i . The quadratic weighting i^2 assigns progressively higher importance to sustained performance across multiple iterations.

For semantic similarity computation, consider a task τ that sustains l loops before failure. We denote p_τ^j as the natural language prompt used at loop j for code generation, where p_τ^1 represents the original task specification and p_τ^j for $j > 1$ represents the code summary generated in the previous loops. The key insight driving our similarity assignment stems from the observation that, from the perspective of LLMs: 1) two programs that could pass identical test cases should be regarded as semantically equivalent; 2) following this principle, the prompts used to generate these functionally correct programs should also be considered semantically equivalent, as they specify the same requirements. Therefore, for all successful loops, we assign:

$$s_\tau^j = 1 \quad \text{for } j < l \text{ or } l = M. \quad (2)$$

For the critical failure boundary where the task terminates at loop $l < M$, we require nuanced evaluation to distinguish between failure due to significant semantic drift versus sudden robustness collapse despite minimal prompt changes:

$$s_\tau^l = \text{Sim}(p_\tau^l, p_\tau^{l+1}) \in [0, 1]. \quad (3)$$

Evaluating semantic similarity at failure boundaries presents significant challenges that standard embedding-based approaches cannot adequately address. Traditional similarity metrics rely solely on natural language embeddings, failing to capture the subtle semantic nuances that distinguish functionally correct specifications from those leading to implementation failures. The core difficulty

lies in requiring comprehensive assessment that considers both the natural language prompt and its corresponding generated code simultaneously— only through this joint evaluation can we determine whether failure resulted from prompt ambiguity, semantic drift, or sudden model robustness collapse.

Listing 2.1: Prompt for LLM-based Semantic Similarity Evaluation.

```
Compare the semantic similarity of these two code generation prompts. Consider both the prompts and
their generated code outputs.

Prompt 1: {prompt1}
Generated Code 1: {code1}

Prompt 2: {prompt2}
Generated Code 2: {code2}

Return only a similarity score between 0 and 1, where:
- 1.0 = semantically equivalent (same intent, requirements, and expected output)
- 0.0 = completely different semantic meaning
- Values between 0 and 1 represent partial semantic overlap

Score:
```

To address this challenge, we employ an LLM-based evaluator with task-specific prompts designed to perform holistic similarity assessment. This approach enables simultaneous consideration of both prompt semantics and code functionality, providing the nuanced evaluation necessary for distinguishing different failure modes at loop boundaries. The evaluation prompt (shown in Listing 2.1) instructs the evaluator to generate similarity scores between 0 and 1, where higher scores indicate failure occurred despite minimal semantic drift between consecutive prompts, suggesting sudden robustness collapse, while lower scores indicate significant semantic divergence at the failure boundary, which may result from either accumulated drift across multiple loops or sudden misinterpretation.

We compute the overall prompt similarity for each task as the per-loop average:

$$\bar{s}_\tau = \frac{1}{l} \sum_{j=1}^l s_\tau^j. \quad (4)$$

Finally, denoting tasks that sustain exactly i loops as $m_i(1), \dots, m_i(n_i)$, we obtain s_i by averaging over all tasks in this category:

$$s_i = \frac{1}{n_i} \sum_{k=1}^{n_i} \bar{s}_{m_i(k)} = \frac{\sum_{k=1}^{n_i} \sum_{j=1}^i s_{m_i(k)}^j}{n_i \cdot i}. \quad (5)$$

ASL metric comprehensively addresses the two critical considerations identified earlier. First, the quadratic weighting scheme with loop count i^2 recognizes that maintaining functionality through multiple iterations demonstrates deeper semantic understanding than initial success alone, implementing a non-linear reward structure where sustained performance across cascading transformations indicates superior robustness. Second, the semantic similarity component s_i provides more precise robustness assessment by incorporating the quality of semantic preservation throughout the evaluation process, enabling our metric to account for both the duration of sustained performance and the consistency of model behavior across transformations.

Compared with adversarial attack-based methods that depend on external perturbation strategies, ASL operates through endogenous evaluation where robustness differences reflect genuine model capabilities rather than vulnerability to specific attack methodologies. The semantic similarity integration provides fine-grained analysis of failure modes, enabling developers to distinguish between models that gradually degrade through semantic drift versus those experiencing sudden robustness collapse. Furthermore, the continuous scoring scale from sustained loop counts naturally accommodates models with varying capability levels, providing informative rankings across the entire performance spectrum rather than binary classifications.

2.3 Novel Advantages of EVALOOP

We summarize the following key advantages of EVALOOP:

- **Compared to adversarial attack-based robustness assessment, EVALOOP is unbiased.** Building upon the duality loop mechanism, EVALOOP quantifies robustness by measuring how many

consecutive transformation cycles a model can sustain before functional crash. This iterative assessment addresses the fundamental limitations of adversarial attack-based evaluation: beyond their diminished effectiveness against post-trained LLMs and inherent bias where different attack strategies favor different models, adversarial approaches rely on external perturbations and fail to capture the internal robustness (i.e., self-consistency) of LLMs—a critical capability for deploying models in complex AI systems where sustained coherence across interactions determines overall reliability.

- **EVALLOOP iteratively exposes robustness weakness from the internal of the model.** Unlike many adversarial attacks that rely on externally crafted perturbations, the failing examples in EVALLOOP are generated by the model itself, providing authentic insights into intrinsic model limitations. The iterative framework reveals distinct degradation trajectories that vary significantly across models. Some LLMs demonstrate gradual semantic drift over many loops, while others experience sudden collapse after a few iterations. These patterns reflect fundamental differences in how models represent and maintain semantic information across self-referential transformations. This approach captures degradation emerging from the model’s own processing limitations and internal inconsistencies, offering a more genuine assessment of model reliability.

The sustained loop approach operates independently of external attack configurations. After the initial prompt, all subsequent inputs are generated by the model itself, creating a self-contained evaluation environment. This endogenous design eliminates biases introduced by specific adversarial strategies and enables fair comparison across different model architectures, as robustness differences reflect genuine capabilities rather than vulnerability to particular perturbation methods.

- **EVALLOOP is easy to implement, enabling widespread adoption with significant practical impact.**

Our comprehensive evaluation demonstrates this scalability through assessment of 96 popular LLMs spanning from 0.5B to 685B parameters—substantially exceeding the scope of existing robustness evaluation studies. For comparison, recent adversarial attack studies typically evaluate 5-15 models [45, 20], while our framework enables evaluation of 5-10 times more models due to its straightforward implementation requirements. This broad applicability facilitates systematic robustness analysis across the entire spectrum of contemporary LLMs. To further support practical adoption, we provide an interactive online Leaderboard (Available at: <https://evaloop.github.io/>) that enables practitioners to explore model robustness rankings, facilitating informed model selection for various deployment scenarios.

- **EVALLOOP can be easily extended with critical software engineering tasks.**

Beyond the core generation-summarization cycle, we demonstrate how our framework readily extends to other critical software engineering tasks by integrating different software engineering tasks. Specifically, we implement and evaluate a code translation-based extension where models iteratively translate code across multiple programming languages (e.g., Python → PHP → Ruby → JavaScript → Perl → Python), with each successful translation representing one evaluation loop, as detailed in Section 5. Our framework’s extensibility also enables integration of other dual task configurations such as code and pseudocode conversion [34] and natural language specification and pseudocode transformation [47].

- **EVALLOOP offers a customized metric to quantify model robustness accurately.**

Since our iterative evaluation framework represents a novel approach to robustness assessment, we developed the Average Sustainable Loops (ASL) metric specifically tailored to capture the nuanced robustness patterns revealed through sustained loop evaluation. Unlike traditional binary pass/fail metrics or simple performance drops, ASL incorporates both loop sustainability duration and semantic consistency quality, providing model developers with interpretable scores that directly inform model selection decisions. The metric’s design enables clear differentiation between models exhibiting gradual degradation versus sudden collapse, guiding developers toward models best suited for their specific deployment requirements where sustained reliability is paramount.

3 Experimental Setup

3.1 Research Questions

- **RQ1 (Reliability of Adversarial Attacks for Robustness Assessment):** How reliable are adversarial attacks in assessing the robustness of LLMs in programming?

- **RQ2 (Effectiveness of the Proposed Framework):** How effectively does EVALLOOP measure LLM robustness in programming?
- **RQ3 (Reliability of the Proposed Framework):** How reliable is EVALLOOP to initial prompt variations and temperature settings?

3.2 Methodology for Answering RQ1

To investigate the effectiveness of adversarial attacks on post-trained LLMs, we employ three representative attack strategies from a recent work by Sclar et al. [39]. Each attack perturbs the original prompt from different angles. As illustrated in Listing 3.1, these attacks represent distinct perturbation categories:

- **Case Transformation:** Converting the entire prompt to uppercase to test sensitivity to typographic variations.
- **Structural Reformatting:** Reorganizing prompt components with explicit labels and formatting to assess structural dependency.
- **Redundant Elaboration:** Injecting verbose instructions and explanatory text to evaluate resistance to information dilution.

We systematically apply these three attack methods to measure performance degradation of LLMs by comparing pre- and post-attack pass@1 accuracy. This approach allows us to quantify each model’s vulnerability to different perturbation types while maintaining functional equivalence of the underlying tasks.

Models and Benchmark We evaluate 13 state-of-the-art LLMs that have undergone substantial post-training, such as reinforcement learning and instruction-based fine-tuning. Our selection encompasses both proprietary closed-source models (e.g., OpenAI’s o-series) and open-source alternatives (e.g., Qwen, LLaMA) to ensure comprehensive coverage of contemporary model architectures. For each model, we report performance change on MBPP Plus benchmark [27] under each attack type, providing a quantitative assessment of adversarial robustness across different perturbation strategies.

Listing 3.1: Examples of the three adversarial attacks.

```
Initial prompt:
Write a function to find the shared elements from the given two lists.
assert set(similar_elements((3, 4, 5, 6),(5, 7, 4, 10))) == set((4, 5))
```

```
1. Case Transformation:
WRITE A FUNCTION TO FIND THE SHARED ELEMENTS FROM THE GIVEN TWO LISTS.
assert set(similar_elements((3, 4, 5, 6),(5, 7, 4, 10))) == set((4, 5))
```

```
2. Structural Reformatting:
Task: Write a function to find the shared elements from the given two lists.
Test Case:
assert set(similar_elements((3, 4, 5, 6),(5, 7, 4, 10))) == set((4, 5))
```

```
3. Redundant Elaboration:
Please implement the following Python function according to the detailed specification provided below.
Function Requirement: Write a function to find the shared elements from the given two lists.
Your implementation must satisfy the following test case:
assert set(similar_elements((3, 4, 5, 6),(5, 7, 4, 10))) == set((4, 5))
Please ensure your function handles all edge cases and follows Python best practices.
```

3.3 Methodology for Answering RQ2

This investigation aims to validate EVALLOOP’s ability to differentiate robustness levels and reveal stability patterns that conventional metrics cannot capture. In detail, we evaluate 96 LLMs spanning parameter scales from 0.5B to 685B, covering the majority of mainstream models currently available, including both open-source and close-source. Our evaluation employs MBPP Plus [27], an enhanced version of the original MBPP benchmark [6] that applies rigorous quality filtering to programming

tasks and expands the test case corpus by 35-fold. This enhanced benchmark provides comprehensive evaluation coverage specifically suited for our robustness assessment framework. To maintain consistency with the benchmark’s code generation prompts, we design complementary prompts for the code summarization task, as shown the initial prompt in Listing 3.2. It instruct LLMs to generate code summaries that follow the same format as the initial generation prompts, ensuring semantic alignment across the dual-task evaluation loop. We analyze robustness rankings using our proposed ASL metric to identify LLMs that demonstrate superior self-consistency independent of initial performance. This includes examining cases where models with relatively lower initial performance exhibit higher robustness scores, potentially indicating deeper semantic understanding versus memorization patterns. The comprehensive scale of our evaluation enables robust statistical analysis of robustness patterns across model families, parameter scales, and training approaches.

Listing 3.2: An example of five variants prompts.

```
Initial prompt: Use one sentence to summarize the following code and start with write a python
function to:
“““\n{code}\n“““
“““\nwrite a python function to\n“““

Prompt 1 (Generate detailed summarization): Summarize what the following code does in natural lan
guage:
“““\n{code}\n“““
Provide a brief explanation in 2-3 sentences. Describe what the code does and its basic inputs/outputs,
but avoid technical implementation details and repetitive explanations. Keep it straightforward and con
cise.

Prompt 2 (Case transformation): USE ONE SENTENCE TO SUMMARIZE THE FOLLOWING CODE AND START WITH WRITE
A PYTHON FUNCTION TO:
“““\n{code}\n“““
“““\nWRITE A PYTHON FUNCTION TO\n“““

Prompt 3 (Structural reformatting): Task: Use one sentence to summarize the following code and start
with write a python function to:
1. Input Code:
“““\n{code}\n“““
2. Output Format:
“““\nwrite a python function to\n“““

Prompt 4 (Simplify): Summarize this code in one sentence starting with ‘write a python function to’:
“““\n{code}\n“““
Summary: write a python function to

Prompt 5 (Redundant elaboration): Please analyze the provided Python code below and generate a concise
one-sentence summary. Your response must begin with the exact phrase ‘write a python function to’.
Code to analyze:
“““python\n{code}\n“““
Please provide your summary in the following format:
“““\nwrite a python function to\n“““
```

3.4 Methodology for Answering RQ3

To assess the reliability of EVALOOP under varying experimental conditions, we investigate two critical factors that could influence ASL measurements: prompt variations and temperature settings. This analysis ensures our framework provides stable robustness assessments across different deployment configurations.

Prompt Reliability Analysis We assess the reliability of EVALOOP to prompt variations by focusing on the code summarization component, which drives the iterative transformation loops in our evaluation. Different prompt formulations are applied to every loop iteration to determine whether observed robustness differences reflect genuine model capabilities rather than prompt-specific artifacts. As shown in Listing 3.2, we employ five distinct prompt variants: the three adversarial strategies from Methodology for RQ1 (Case Transformation, Structural Reformatting, and Redundant Elaboration) plus two additional approaches—generating detailed summarization and prompt simplification. Unlike RQ1, which examines model vulnerability to these perturbations, RQ3 investigates whether EVALOOP produces consistent robustness rankings despite input variations across the summarization phase. A robust evaluation framework should demonstrate minimal ranking

perturbations across semantically equivalent prompts, indicating that observed robustness differences reflect genuine model capabilities rather than prompt-specific artifacts.

Temperature Reliability Analysis We systematically evaluate ASL score variations across different sampling temperatures to understand the framework’s behavior under varying generation stochasticity. We test four additional temperature settings: 0.2, 0.4, 0.6, and 0.8, comparing them against the default temperature configuration.

In this RQ, we experiment on the same 13 LLMs from RQ1, ensuring consistency with our adversarial attack comparison while maintaining experimental feasibility. To quantify the consistency of model rankings across different prompts, we calculate the *Spearman rank correlation coefficient* between the baseline rankings and the average prompt and temperature rankings. A high correlation coefficient generally indicates a strong positive correlation, demonstrating that the relative performance of LLMs remains highly stable across different prompt variations and temperature settings.

3.5 Greedy Decoding vs. Temperature Sampling

Figure 3 presents the performance distribution of average sustainable loops per task in the MBPP Plus benchmark across 13 LLMs (same as RQ1 and RQ3) using either greedy decoding or temperature-controlled stochastic sampling. Our comprehensive evaluation, conducted through five times experimental runs and averaged results, reveals an intriguing finding: both decoding strategies achieve nearly identical performance, with greedy decoding averaging 6.38 sustainable loops and temperature sampling averaging 6.37 sustainable loops. This convergence in performance is particularly noteworthy given that temperature sampling is widely adopted as the default decoding mechanism in many production LLMs (e.g., ChatGPT employs a default temperature of 0.7), primarily valued for its capacity to introduce variability and enhance human-like expression diversity. However, our results suggest that in the context of code generation tasks, the stochastic exploration benefits of temperature sampling do not translate to measurable performance improvements. This might be because code generation inherently demands strict adherence to precise semantic and syntactic rules, creating a more constrained solution space where the highest-probability tokens selected by greedy decoding are often optimal. Besides, the deterministic nature of greedy decoding may actually be advantageous in programming tasks, where consistency and logical coherence are paramount over linguistic creativity. **Given the equivalent performance and the superior reproducibility afforded by greedy decoding, we adopt greedy decoding as our primary evaluation strategy for all subsequent experiments.**

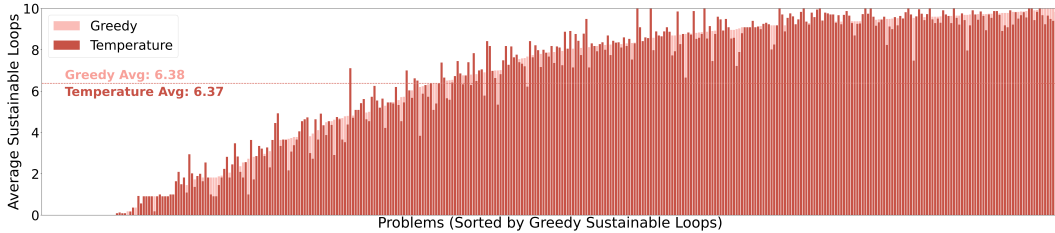


Figure 3: Distribution of average sustainable loops per task in MBPP Plus.

3.6 Implementation

We evaluate proprietary LLMs through their official APIs (OpenAI, Anthropic, etc.) and deploy open-source LLMs locally using VLLM [23] across eight NVIDIA H200 GPUs. All experiments use consistent hyperparameters: temperature=0.0 and top-p=1.0 for deterministic generation (greedy decoding), maximum 1024 tokens for code generation and summarization. Besides, we set the maximum number of evaluation loop (*i.e.*, M) as 10 and use gpt-4-turbo-2024-04-09 for similarity evaluator.

Table 1: Model performance (in terms of pass@1) comparison under adversarial attacks.

| Model | Original | ATT_{upper} | $ATT_{structure}$ | ATT_{detail} |
|---------------------------------|----------|---------------|-------------------|----------------|
| Codestral-22B-v0.1 | 0.607 | 0.648 | 0.706 | 0.669 |
| deepseek-coder-7b-instruct-v1.5 | 0.646 | 0.291 | 0.011 | 0.638 |
| DeepSeek-Coder-V2-Lite-Instruct | 0.713 | 0.722 | 0.722 | 0.704 |
| DeepSeek-Coder-V2-Instruct | 0.769 | 0.767 | 0.765 | 0.773 |
| Llama-3.1-8B-Instruct | 0.586 | 0.468 | 0.458 | 0.360 |
| OpenCoder-8B-Instruct | 0.703 | 0.683 | 0.698 | 0.693 |
| CodeQwen1.5-7B-Chat | 0.698 | 0.704 | 0.698 | 0.706 |
| Qwen2.5-Coder-32B-Instruct | 0.762 | 0.759 | 0.751 | 0.545 |
| Qwen3-235B-A22B-Instruct-2507 | 0.789 | 0.796 | 0.791 | 0.788 |
| GPT-4.1 | 0.767 | 0.775 | 0.775 | 0.743 |
| GPT-4o | 0.767 | 0.754 | 0.767 | 0.759 |
| o3-mini | 0.815 | 0.804 | 0.807 | 0.818 |
| o4-mini | 0.818 | 0.818 | 0.810 | 0.794 |

4 Results

4.1 Answer to RQ1: Adversarial Attack Limitations

Table 1 presents the comparative analysis of model performance under three adversarial attack methods against baseline performance on MBPP Plus. We evaluate robustness by measuring pass@1 accuracy change across Case Transformation (ATT_{upper}), Structural Reformatting ($ATT_{structure}$), and Redundant Elaboration (ATT_{detail}) attacks.

Overall Attack Ineffectiveness The experimental results reveal that adversarial attacks demonstrate limited effectiveness against modern post-trained LLMs. The majority of evaluated models show minimal performance degradation across attack variants. For instance, Qwen3-235B-A22B-Instruct-2507 maintain near-identical performance (0.789 \rightarrow 0.796, 0.791, 0.788), while OpenAI’s o-series models (o3-mini, o4-mini) exhibit negligible drops, with some variants even showing slight improvements—a phenomenon consistent with observations by Fang et al. [17]. Similarly, DeepSeek-Coder-V2-Instruct maintains consistent performance (0.769 \rightarrow 0.767, 0.765, 0.773) across all attack types, indicating robust resilience to external perturbations.

Contradictory Vulnerability Patterns Among LLMs that do exhibit performance drops, we observe contradictory sensitivity patterns that highlight fundamental limitations in adversarial evaluation. deepseek-coder-7b-instruct-v1.5 demonstrates severe vulnerability to Case Transformation (0.646 \rightarrow 0.291) and catastrophic failure under Structural Reformatting (0.646 \rightarrow 0.011), yet maintains baseline performance under Redundant Elaboration (0.646 \rightarrow 0.638). Conversely, Qwen2.5-Coder-32B-Instruct shows minimal sensitivity to the first two attacks (0.762 \rightarrow 0.759, 0.751) but significant degradation under Redundant Elaboration (0.762 \rightarrow 0.545). This contradictory behavior creates an evaluation paradox: which model is more robust than the other? deepseek-coder-7b-instruct-v1.5 performs excellently under verbose prompts but fails catastrophically with structural changes, while Qwen2.5-Coder-32B-Instruct demonstrates the opposite pattern. Such inconsistency makes comparative robustness assessment unreliable for model selection, as conclusions heavily rely on the specific attack method employed.

Answer Summary to RQ1

Our analysis highlights two key limits of adversarial attacks: (1) *Diminished effectiveness* — post-trained LLMs resist external perturbations, making attacks largely ineffective; (2) *Inherent evaluation bias* — different attack methods favor different models, leading to conflicting robustness results and undermining fair comparison.

Table 2: Robustness ranking benchmarked by EVALOOP. Numbers in parentheses denote the change in ASL ranking relative to the PassRate ranking.

| Model | ASL | Model | ASL |
|---------------------------------------|--------------|--|--------------|
| 1 Claude Opus 4 | 7.786 (↑ 2) | 49 Mistral-Small-3.2-24B-Instruct-2506 | 6.029 (↑ 1) |
| 2 Qwen3-235B-A22B-Instruct-2507 | 7.689 (↑ 6) | 50 Llama-3.1-70B-Instruct | 6.023 (↓ 7) |
| 3 Qwen3-Coder-480B-A35B-Instruct | 7.599 (↑ 6) | 51 Meta-Llama-3-70B-Instruct | 6.017 (↑ 9) |
| 4 Claude Sonnet 4 | 7.518 (↑ 6) | 52 Qwen3-4B-Instruct-2507 | 5.986 (↑ 7) |
| 5 DeepSeek-V3 | 7.481 (↑ 1) | 53 Hermes-3-Llama-3.1-70B | 5.983 (↑ 1) |
| 6 o3-mini | 7.456 (↓ 2) | 54 Qwen2.5-Coder-3B-Instruct | 5.946 (↑ 7) |
| 7 o3 | 7.383 (=) | 55 Mistral-Small-3.1-24B-Instruct-2503 | 5.937 (↑ 2) |
| 8 GPT-4.1 | 7.355 (↑ 4) | 56 DeepSeek-V2-Chat | 5.854 (↓ 4) |
| 9 o4-mini | 7.317 (↓ 8) | 57 Baichuan-M2-32B | 5.827 (↓ 25) |
| 10 GPT-4.1-mini | 7.290 (↑ 1) | 58 deepseek-coder-33b-instruct | 5.767 (↓ 7) |
| 11 o1 | 7.285 (↓ 9) | 59 gemma-3-4b-it | 5.766 (↑ 4) |
| 12 o1-mini | 7.254 (↓ 7) | 60 Mistral-Small-Instruct-2409 | 5.700 (↑ 7) |
| 13 Qwen2.5-Coder-32B-Instruct | 7.253 (↑ 6) | 61 Mixtral-8x22B-Instruct-v0.1 | 5.575 (↑ 5) |
| 14 Gemini 2.5 Flash-Lite | 7.178 (↑ 1) | 62 Codestral-22B-v0.1 | 5.443 (↑ 10) |
| 15 Qwen3-30B-A3B-Instruct-2507 | 7.124 (↑ 11) | 63 deepseek-coder-7b-instruct-v1.5 | 5.423 (↑ 1) |
| 16 Claude Haiku 3.5 | 7.097 (↑ 1) | 64 Qwen2.5-Coder-1.5B-Instruct | 5.319 (↑ 11) |
| 17 DeepSeek-Coder-V2-Instruct | 7.070 (↑ 3) | 65 Llama-4-Scout-17B-16E-Instruct | 5.224 (↓ 29) |
| 18 Gemini 2.0 Flash | 7.041 (↓ 2) | 66 gemma-3n-E2B-it | 5.051 (↓ 4) |
| 19 GPT-4o | 7.039 (↑ 4) | 67 Phi-4-reasoning | 4.897 (↓ 12) |
| 20 DeepSeek-V2.5 | 6.945 (↑ 1) | 68 Ministral-8B-Instruct-2410 | 4.844 (↑ 11) |
| 21 Llama-4-Maverick-17B-128E-Instruct | 6.922 (↓ 8) | 69 LFM-7B | 4.743 (↑ 11) |
| 22 Gemini 2.5 Flash | 6.922 (↓ 4) | 70 Phi-3.5-MoE-instruct | 4.713 (↑ 1) |
| 23 gemma-3-27b-it | 6.905 (↑ 4) | 71 Phi-3.5-mini-instruct | 4.706 (↑ 12) |
| 24 Gemini 2.5 Pro | 6.842 (↓ 10) | 72 Mixtral-8x7B-Instruct-v0.1 | 4.687 (↑ 6) |
| 25 gemma-3-12b-it | 6.839 (↑ 12) | 73 Ling-plus | 4.521 (↑ 8) |
| 26 GPT-4-Turbo | 6.820 (↑ 2) | 74 Hermes-3-Llama-3.1-8B | 4.446 (↑ 8) |
| 27 GLM-4.5-Air | 6.809 (↑ 6) | 75 Meta-Llama-3-8B-Instruct | 4.310 (↑ 2) |
| 28 Llama-3.1-405B-Instruct | 6.787 (↑ 10) | 76 Phi-4-mini-instruct | 4.008 (↑ 10) |
| 29 Gemini 2.0 Flash-Lite | 6.775 (=) | 77 Qwen3-235B-A22B | 3.933 (↓ 9) |
| 30 Qwen2.5-Coder-14B-Instruct | 6.676 (=) | 78 Qwen2.5-Coder-0.5B-Instruct | 3.424 (↑ 11) |
| 31 Qwen3-Coder-30B-A3B-Instruct | 6.669 (↓ 9) | 79 DeepSeek-V2-Lite-Chat | 3.280 (↑ 5) |
| 32 NextCoder-32B | 6.610 (↓ 8) | 80 Moonlight-16B-A3B-Instruct | 3.181 (↓ 11) |
| 33 NextCoder-14B | 6.520 (↑ 1) | 81 Mistral-7B-Instruct-v0.3 | 3.129 (↑ 7) |
| 34 Llama-3.3-70B-Instruct | 6.517 (↑ 10) | 82 NextCoder-7B | 3.123 (↓ 24) |
| 35 OpenCoder-8B-Instruct | 6.490 (↑ 12) | 83 Seed-Coder-8B-Reasoning | 3.106 (↑ 4) |
| 36 Seed-Coder-8B-Instruct | 6.452 (↓ 5) | 84 Phi-3-mini-4k-instruct | 3.012 (↓ 11) |
| 37 Mistral-Large-Instruct-2407 | 6.422 (↑ 8) | 85 Mistral-7B-Instruct-v0.2 | 2.895 (↑ 6) |
| 38 CodeQwen1.5-7B-Chat | 6.410 (↑ 11) | 86 Llama-3.1-8B-Instruct | 2.640 (↓ 10) |
| 39 DeepSeek-Coder-V2-Lite-Instruct | 6.377 (↑ 7) | 87 DeepSeek-R1-Distill-Llama-70B | 2.373 (↓ 17) |
| 40 Qwen2.5-Coder-7B-Instruct | 6.340 (=) | 88 Phi-3-medium-4k-instruct | 2.368 (↑ 4) |
| 41 Hermes-3-Llama-3.1-405B | 6.295 (=) | 89 OpenCoder-1.5B-Instruct | 2.352 (↓ 15) |
| 42 gemma-3n-E4B-it | 6.279 (↓ 3) | 90 Qwen3-4B-Thinking-2507 | 2.092 (=) |
| 43 GPT-3.5-Turbo | 6.257 (↑ 5) | 91 GLM-4-32B-0414 | 1.959 (↓ 26) |
| 44 GLM-4.5 | 6.226 (↓ 19) | 92 Jan-v1-4B | 1.847 (↓ 7) |
| 45 GLM-4-9B-0414 | 6.198 (↑ 8) | 93 Qwen3-1.7B | 1.636 (=) |
| 46 Ling-lite-1.5 | 6.177 (↓ 4) | 94 Claude Haiku 3 | 1.571 (↑ 1) |
| 47 Pixtral-Large-Instruct-2411 | 6.150 (↑ 9) | 95 deepseek-coder-1.3b-instruct | 1.340 (↓ 1) |
| 48 MiniCPM4-8B | 6.137 (↓ 13) | 96 deepseek-coder-6.7b-instruct | 1.107 (=) |

4.2 Answer to RQ2: Framework Effectiveness

Table 2 presents the comprehensive robustness evaluation of 96 LLMs using EVALOOP, ranked by their ASL scores from highest to lowest. Models with high ASL scores demonstrate both the ability to maintain functional correctness across extended iterations and semantic coherence throughout self-referential transformations. The numbers in parentheses indicate the change in ASL ranking relative to each model’s pass@1 accuracy ranking, revealing how robustness assessment differs from conventional performance metrics.

Robustness vs. Initial Performance Divergence The results demonstrate substantial divergence between robustness rankings and initial performance capabilities. The majority of models exhibit significant ranking changes when evaluated through the robustness lens: 88 out of 96 models (91.7%) show ranking shifts, with a mean absolute deviation of 6.9 positions. Over one-quarter of models (27.1%) experience large shifts of 10 or more positions, and nearly one in ten model pairs have their relative ordering reversed between PassRate and ASL evaluations. Notable examples include Qwen3-235B-A22B-Instruct-2507, which rises 6 positions in robustness ranking despite already strong initial performance, and conversely, o1 and o4-mini, which drop 9 and 8 positions respectively, indicating that high initial performance does not guarantee sustained stability through iterative transformations.

Besides, there are some models experiencing significant drops (e.g., Llama-4-Scout-17B-16E-Instruct dropping 29 positions, GLM-4-32B-0414 dropping 26 positions), which reveals brittleness that becomes apparent only through sustained iterative evaluation. Conversely, models with positive ranking changes demonstrate robustness capabilities that exceed their initial performance suggestions, offering valuable insights for model selection in deployment scenarios requiring sustained reliability.

The ASL scores reveal pronounced differences in intrinsic stability across models, with scores ranging from 7.786 (Claude Opus 4) to 1.107 (deepseek-coder-6.7b-instruct). This wide distribution confirms that robustness represents a distinct capability dimension independent of initial coding competence, validating the necessity of specialized robustness evaluation frameworks. Remarkably, several models demonstrate superior robustness despite lower initial performance rankings—for instance, gemma-3-12b-it rises 12 positions, and OpenCoder-8B-Instruct gains 12 positions, suggesting that sustained self-consistency may indicate deeper comprehension mechanisms rather than mere memorization patterns.

Correlation with Agent System Performance Our robustness rankings demonstrate meaningful correlation with empirical observations from real-world agent deployments. Cross-referencing with SWE-bench Bash Only Leaderboard [22]¹—which evaluates models in minimal agent environments using only bash commands—reveals a **Spearman correlation coefficient** of 0.586 between the two ranking systems. Specifically, among the 15 overlapping models:

- Perfect alignment: Claude Opus 4 maintain the top-1 ranking in both evaluations.
- Close correspondence: Claude Sonnet 4, o3, Qwen3-Coder-480B-A35B-Instruct, GPT-4o, o4-mini, Llama-4-Scout-17B-16E-Instruct, and Llama-4-Maverick-17B-128E-Instruct maintain close rankings in both evaluations (± 2 position change).
- Significant divergences: GLM-4.5, Gemini 2.5 Pro, GPT-4.1, GPT-4.1-mini, Gemini 2.5 Flash, Gemini 2.0 Flash, and Qwen2.5-Coder-32B-Instruct exhibit notable ranking discrepancies between two evaluation (≥ 3 position change).

These divergences highlight the complementary nature of different robustness assessment approaches. While SWE-bench evaluates robustness through complex, multi-step software engineering challenges, EVALOOP assesses robustness through sustained self-consistency across iterative transformations—a distinct but equally important dimension of model reliability. The observed ranking differences reflect each framework’s unique focus: SWE-bench emphasizes comprehensive problem-solving capabilities under complex scenarios, while EVALOOP specifically targets the fundamental ability to maintain semantic coherence when processing self-generated outputs. This focused evaluation design enables EVALOOP to isolate and measure intrinsic stability patterns that may be masked in more complex evaluation environments.

The positive Spearman correlation coefficient (0.586) demonstrates that EVALOOP captures stability characteristics genuinely relevant to agent deployments, while the moderate correlation strength indicates that robustness assessment through self-consistency represents a complementary evaluation dimension rather than merely replicating existing performance metrics. This distinction proves essential for comprehensive model evaluation, where sustained coherence across self-referential interactions represents a fundamental capability for reliable autonomous operation, distinct from but complementary to complex problem-solving proficiency.

Answer Summary to RQ2

Our large-scale assessment on 96 LLMs reveals: (1) *Robustness-performance divergence* - 88 out of 96 models show significant ranking changes (-29 to +12) when evaluated for sustained stability versus initial performance; (2) *Complementary insights* - a moderate correlation (0.586) with agent system performance validates practical relevance while maintaining independence from conventional metrics, enabling identification of models with superior long-term reliability despite lower initial performance.

Table 3: Prompt sensitivity results. Numbers in parentheses are the rank within the current column; the final averaged prompt rank is the mean of the five prompt-specific ranks (Prompt1–Prompt5).

| Model | Baseline | Prompt1 | Prompt2 | Prompt3 | Prompt4 | Prompt5 | Avg. Prompt Rank |
|---------------------------------|------------|------------|------------|------------|------------|------------|------------------|
| Qwen3-235B-A22B-Instruct-2507 | 7.689 (1) | 7.654 (1) | 7.603 (1) | 7.655 (1) | 7.710 (1) | 7.553 (1) | 1.00 |
| o3-mini | 7.456 (2) | 7.272 (4) | 7.489 (2) | 7.481 (2) | 7.523 (2) | 7.355 (2) | 2.40 |
| GPT-4.1 | 7.355 (3) | 7.475 (3) | 7.372 (3) | 7.431 (3) | 7.445 (4) | 7.348 (3) | 3.20 |
| o4-mini | 7.317 (4) | 7.482 (2) | 7.299 (4) | 7.367 (4) | 7.447 (3) | 7.335 (4) | 3.40 |
| Qwen2.5-Coder-32B-Instruct | 7.253 (5) | 7.029 (7) | 7.022 (7) | 7.087 (6) | 7.003 (7) | 7.020 (7) | 6.80 |
| DeepSeek-Coder-V2-Instruct | 7.070 (6) | 7.250 (5) | 7.147 (6) | 6.908 (7) | 7.054 (6) | 7.081 (6) | 6.00 |
| GPT-4o | 7.039 (7) | 7.231 (6) | 7.207 (5) | 7.163 (5) | 7.148 (5) | 7.131 (5) | 5.20 |
| OpenCoder-8B-Instruct | 6.490 (8) | 6.236 (10) | 6.749 (8) | 6.776 (8) | 2.913 (12) | 6.085 (10) | 9.60 |
| DeepSeek-Coder-V2-Lite-Instruct | 6.377 (9) | 6.597 (9) | 6.234 (10) | 6.293 (9) | 5.949 (9) | 6.094 (9) | 9.20 |
| CodeQwen1.5-7B-Chat | 6.340 (10) | 6.753 (8) | 6.277 (9) | 6.198 (10) | 6.173 (8) | 6.146 (8) | 8.60 |
| Codestral-22B-v0.1 | 5.443 (11) | 5.530 (12) | 5.209 (12) | 5.221 (11) | 5.404 (10) | 5.272 (12) | 11.40 |
| deepseek-coder-7b-instruct-v1.5 | 5.423 (12) | 5.804 (11) | 5.572 (11) | 2.428 (12) | 4.969 (11) | 5.580 (11) | 11.20 |
| Llama-3.1-8B-Instruct | 2.640 (13) | 2.683 (13) | 3.469 (13) | 2.065 (13) | 1.583 (13) | 5.189 (13) | 13.00 |

Table 4: Temperature sensitivity results. Numbers in parentheses are the rank within the current column; the final averaged temperature rank is the mean of the four temperature-specific ranks (0.2, 0.4, 0.6, and 0.8).

| Model | Baseline | 0.2 | 0.4 | 0.6 | 0.8 | Avg. Temperature Rank |
|---------------------------------|------------|------------|------------|------------|------------|-----------------------|
| Qwen3-235B-A22B-Instruct-2507 | 7.689 (1) | 7.643 (1) | 7.673 (1) | 7.662 (1) | 7.686 (1) | 1.00 |
| o3-mini | 7.456 (2) | 7.375 (3) | 7.458 (2) | 7.410 (2) | 7.389 (3) | 2.50 |
| GPT-4.1 | 7.355 (3) | 7.209 (4) | 7.315 (4) | 7.389 (4) | 7.214 (4) | 4.00 |
| o4-mini | 7.317 (4) | 7.622 (2) | 7.389 (3) | 7.395 (3) | 7.431 (2) | 2.50 |
| Qwen2.5-Coder-32B-Instruct | 7.253 (5) | 7.140 (6) | 7.082 (6) | 7.158 (5) | 7.067 (6) | 5.75 |
| DeepSeek-Coder-V2-Instruct | 7.070 (6) | 7.186 (5) | 7.124 (5) | 7.141 (6) | 7.151 (5) | 5.25 |
| GPT-4o | 7.039 (7) | 6.961 (7) | 7.036 (7) | 6.783 (7) | 6.611 (7) | 7.00 |
| OpenCoder-8B-Instruct | 6.490 (8) | 5.838 (10) | 5.827 (9) | 5.924 (9) | 5.625 (9) | 9.25 |
| DeepSeek-Coder-V2-Lite-Instruct | 6.377 (9) | 6.419 (8) | 6.385 (8) | 6.183 (8) | 6.273 (8) | 8.00 |
| CodeQwen1.5-7B-Chat | 6.340 (10) | 6.334 (9) | 5.381 (11) | 4.916 (12) | 4.784 (10) | 10.50 |
| Codestral-22B-v0.1 | 5.443 (11) | 5.207 (12) | 5.025 (12) | 4.959 (11) | 4.734 (11) | 11.50 |
| deepseek-coder-7b-instruct-v1.5 | 5.423 (12) | 5.438 (11) | 5.467 (10) | 5.392 (10) | 4.701 (12) | 10.75 |
| Llama-3.1-8B-Instruct | 2.640 (13) | 2.804 (13) | 2.525 (13) | 1.985 (13) | 1.246 (13) | 13.00 |

4.3 Answer to RQ3: Framework Reliability

Tables 3 and 4 present the reliability analysis of EVALOOOP across prompt variations and different temperature settings. We evaluate our framework stability by examining the variation of standard ASL score and ranking consistency across different experimental conditions, with rankings shown in parentheses for each model.

Prompt Reliability Analysis To assess prompt reliability, we compute the ranking for each LLM across all prompt variants (Prompts 1-5) and compare it with the original rankings. Our analysis reveals remarkable stability in model rankings despite prompt variations. For instance, Qwen3-235B-A22B-Instruct-2507 maintains consistent top ranking (rank 1) across all prompt conditions, while models like Llama-3.1-8B-Instruct consistently occupy the bottom position (rank 13). Computing the average rankings across prompt variants yields: Qwen3 (1.0), o3-mini (2.4), gpt-4.1 (3.2), o4-mini (3.4), demonstrating minimal deviation from baseline rankings. Moreover, **the Spearman rank correlation coefficient** between original and average prompt rankings is **0.951**, indicating exceptionally high framework stability against prompt modifications.

Temperature Reliability Analysis Our temperature reliability analysis also reveals similar stability patterns across sampling configurations. Computing average rankings across temperatures 0.2-0.8: Qwen3 (1.0), o3-mini (2.5), gpt-4.1 (4.0), DeepSeek-Coder-V2-Instruct (5.25), Qwen2.5-Coder-32B (5.75), show minimal ranking perturbations from baseline. **The Spearman rank correlation coefficient** between original and average temperature rankings is **0.974**, demonstrating even higher stability than prompt variations. Notably, higher temperatures generally correlate with slight ASL score reductions (e.g., Codestral-22B-v0.1: 5.443 \rightarrow 4.734, CodeQwen1.5-7B-Chat: 6.340 \rightarrow 4.784 at temperature 0.8), reflecting increased generation stochasticity. However, relative model rankings remain largely preserved despite these score variations.

¹<https://www.swebench.com/bash-only.html>

Answer Summary to RQ3

Our reliability analysis reveals: (1) *Prompt robustness* - Spearman correlation of 0.951 between baseline and prompt-varied rankings, with minimal ranking perturbations (± 1 -2 positions) across semantic-preserving modifications; (2) *Temperature stability* - Even higher correlation of 0.974 across temperature settings (0.2-0.8), indicating that observed robustness differences reflect genuine model capabilities rather than experimental artifacts, validating EVALOOP as a reliable evaluation framework for fair model comparison.

5 Discussion: The Extension of EVALOOP by Code Translation Loop

To demonstrate the extensibility of our framework, we implement a code translation variant that evaluates robustness through iterative translation across multiple programming languages. Instead of alternating between code generation and summarization, this extension maintains the code domain throughout while testing semantic preservation across different programming paradigms. The code translation loop operates by establishing a continuous translation chain where each successful conversion represents one evaluation iteration. Starting with functionally correct source code in an initial programming language, the LLM receives translation prompts (e.g., "Convert the following Python code to PHP"), performs the translation, and undergoes rigorous testing through language-specific test suites. Upon successful validation, the process continues to the next language in the predetermined sequence. Each successful translation step contributes one loop count to the ASL calculation, with failure occurring when the translated code fails to pass the test.

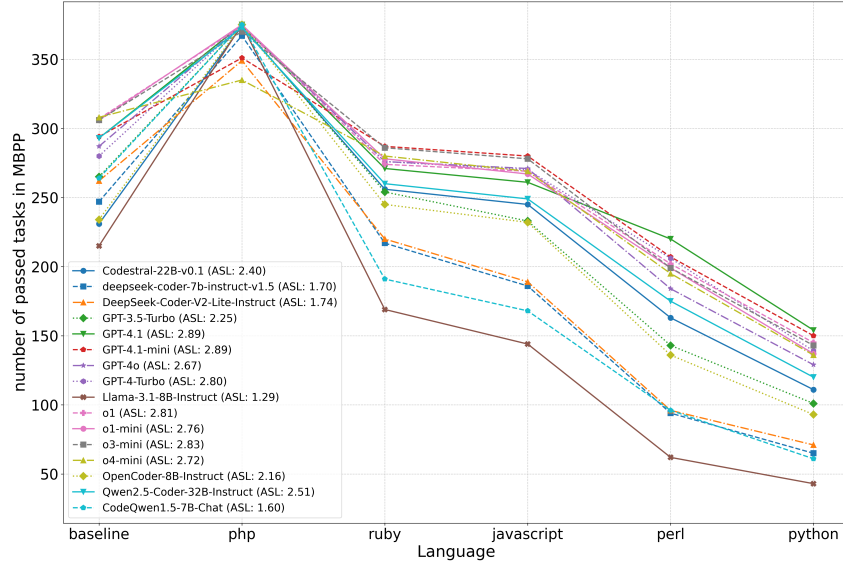


Figure 4: Robustness assessment of LLMs on Framework 2.

We construct a five-step translation chain: Python \rightarrow PHP \rightarrow Ruby \rightarrow JavaScript \rightarrow Perl \rightarrow Python, creating a complete loop that returns to the original language. This design tests the model’s ability to preserve program semantics across diverse paradigms including scripting languages, web-oriented languages, and general-purpose programming environments. We evaluate this extension using the MXEVAL benchmark [5], which provides comprehensive test suites for cross-language code evaluation. As shown in Fig. 4, the experimental results on 16 mainstream LLMs reveal substantial performance degradation across successive translation loops, with pass@1 accuracy declining by 32.00%–54.13% compared to the initial performance. Notably, the robustness rankings mirror those observed in our original framework, where initial translation performance does not necessarily correlate with sustained robustness across the complete chain. GPT-4.1 and GPT-4.1-mini achieve the highest ASL scores of 2.89, followed by o3-mini (2.83), demonstrating superior semantic preservation

capabilities. Conversely, open-source models exhibit greater challenges, with Llama-3.1-8B-Instruct recording the lowest ASL score of 1.29.

6 Related Work

LLMs’ Robustness in Programming. Code generation has emerged as the predominant practical application of AI in programming environments [25], catalyzing extensive research into the robustness of LLMs in this domain. Empirical assessments of commercial systems reveal concerning vulnerabilities: GitHub Copilot produces inconsistent implementations for semantically equivalent prompts in 46% of cases, with correctness variations reaching 28% between prompt variants [29]. Similarly, Codex exhibits substantial sensitivity to minor linguistic perturbations in task specifications [41, 54]. The research community has responded with LLMs’ robustness through diverse adversarial techniques [12, 52, 20, 7] alongside corresponding defensive strategies [19, 49, 18]. However, different adversarial attacks might "favor" different LLMs, potentially leading to contradictory conclusions about a model’s robustness depending on the specific attack employed, struggling in a unified evaluation. EVALOOP addresses this critical gap by leveraging the duality between generation and understanding, enabling unified robustness assessment across LLMs with the average number of sustainable loops, *i.e.*, ASL. This novel approach provides a natural measure of LLMs’ robustness in programming.

Duality in Software Engineering Tasks. The concept of duality in software engineering was first proposed by Wei et al [44], which demonstrated that these complementary tasks require consistent semantic understanding from opposing perspectives. This foundational concept has been leveraged for different evaluation purposes. Min et al. [30] propose IdentityChain, a framework that evaluates whether Code LLMs can maintain semantic consistency when translating between natural language and code in both directions. They introduce the Test Output Match (TOM) to compare the exact outputs of two programs across all test cases rather than just pass/fail results, capturing fine-grained semantic differences by examining specific return values and complete error messages for syntax or runtime errors. Allamanis et al. [3] introduce Round-Trip Correctness (RTC) as an unsupervised evaluation method for Code LLMs that leverages the natural duality between tasks like code generation and code summarization, where a model generates a prediction, feeds it back through the inverse task, and checks if the round-trip preserves semantic equivalence to the original input. RTC demonstrates strong correlation with existing metrics on narrow-domain benchmarks like HumanEval [11] and ARCADE [50] while enabling evaluation across a much broader spectrum of real-world software domains without requiring costly human annotations.

Our framework introduces three key innovations that distinguish it from prior duality-based approaches: (1) *iterative stress-testing* through multiple consecutive transformation loops that progressively challenge model stability rather than single round-trip evaluations, (2) the *Average Sustainable Loops (ASL)* metric that quantifies long-term degradation patterns and robustness thresholds rather than binary correctness or semantic equivalence at individual time points, and (3) *framework extensibility* across diverse dual task configurations, demonstrated through both generation-summarization cycles and cross-language code translation chains. Crucially, our empirical findings reveal a key insight that neither IdentityChain nor RTC captures: *initial performance does not reliably predict sustained robustness*. LLMs with superior single-task accuracy cannot always exhibit superior stability across multiple iterations, while some LLMs with competitive initial capabilities demonstrate superior resilience under iterative stress—a phenomenon critical for understanding LLM reliability in production environments but invisible to single-round evaluations.

7 Conclusion

In this paper, we addressed critical limitations in current LLM robustness evaluation for programming, which rely on biased adversarial attacks and overlook sustained consistency capabilities. We introduced EVALOOP, a novel framework that assesses robustness through self-contained iterative loops between dual software engineering tasks such as code generation and summarization. Our comprehensive evaluation of 96 LLMs reveals that initial programming proficiency does not correlate with sustained robustness, with 85 models showing significant ranking changes when evaluated for long-term stability. The proposed Average Sustainable Loops (ASL) metric provides a unified and unbiased robustness assessment that correlates meaningfully with agent system performance

while maintaining exceptional framework stability across experimental variations. EVALOOP offers authentic insights into model reliability for deployment scenarios requiring sustained autonomous operation, contributing to understanding of self-consistency as a fundamental dimension of LLM robustness beyond adversarial resilience.

Looking ahead, we plan to conduct qualitative analysis of experimental results to systematically investigate why certain models exhibit superior robustness while others fail quickly, we aim to develop targeted enhancement strategies that improve model stability within our evaluation paradigm, ultimately providing more reliable LLM foundations for agent system construction.

References

- [1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- [2] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, 2020.
- [3] M. Allamanis, S. Panthaplackel, and P. Yin. Unsupervised evaluation of code llms with round-trip correctness. *arXiv preprint arXiv:2402.08699*, 2024.
- [4] M. Anand, P. Kayal, and M. Singh. Adversarial robustness of program synthesis models. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*, 2021.
- [5] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, et al. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*, 2022.
- [6] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- [7] B. Bačić, C. Feng, and W. Li. Jy61 imu sensor external validity: A framework for advanced pedometer algorithm personalisation. *ISBS Proceedings Archive*, 42(1):60, 2024.
- [8] B. Bačić, C. Vasile, C. Feng, and M. G. Ciucă. Towards nation-wide analytical healthcare infrastructures: A privacy-preserving augmented knee rehabilitation case study. *arXiv preprint arXiv:2412.20733*, 2024.
- [9] M. Bhatt, S. Chennabasappa, Y. Li, C. Nikolaidis, D. Song, S. Wan, F. Ahmad, C. Aschermann, Y. Chen, D. Kapil, et al. Cyberseceval 2: A wide-ranging cybersecurity evaluation suite for large language models. *arXiv preprint arXiv:2404.13161*, 2024.
- [10] M. Bhatt, S. Chennabasappa, C. Nikolaidis, S. Wan, I. Evtimov, D. Gabi, D. Song, F. Ahmad, C. Aschermann, L. Fontana, et al. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724*, 2023.
- [11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [12] N. Chen, Q. Sun, J. Wang, M. Gao, X. Li, and X. Li. Evaluating and enhancing the robustness of code pre-trained models through structure-aware adversarial samples generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14857–14873, 2023.
- [13] Z. Chen, Y. Pan, S. Lu, J. Xu, C. L. Goues, M. Monperrus, and H. Ye. Prometheus: Unified knowledge graphs for issue resolution in multilingual codebases. *arXiv preprint arXiv:2507.19942*, 2025.
- [14] F. I. Craik. Levels of processing: Past, present... and future? *Memory*, 10(5-6):305–318, 2002.

- [15] S. Dora, D. Lunkad, N. Aslam, S. Venkatesan, and S. K. Shukla. The hidden risks of llm-generated web application code: A security-centric evaluation of code generation capabilities in large language models. *arXiv preprint arXiv:2504.20612*, 2025.
- [16] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [17] S. Fang, W. Ding, A. Mastropaolo, and B. Xu. Smaller= weaker? benchmarking robustness of quantized llms in code generation. *arXiv preprint arXiv:2506.22776*, 2025.
- [18] Y. Ge, W. Sun, Y. Lou, C. Fang, Y. Zhang, Y. Li, X. Zhang, Y. Liu, Z. Zhao, and Z. Chen. Demonstration attack against in-context learning for code intelligence. *arXiv preprint arXiv:2410.02841*, 2024.
- [19] C. Improta, P. Liguori, R. Natella, B. Cukic, and D. Cotroneo. Enhancing robustness of ai offensive code generators via data augmentation. *Empirical Software Engineering*, 30(1):7, 2025.
- [20] A. Jha and C. K. Reddy. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 14892–14900, 2023.
- [21] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *12th International Conference on Learning Representations, ICLR 2024*, 2024.
- [22] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.
- [23] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.
- [24] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
- [25] J. T. Liang, C. Yang, and B. A. Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13, 2024.
- [26] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
- [27] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.
- [28] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.
- [29] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota. On the robustness of code generation techniques: An empirical study on github copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2149–2160. IEEE, 2023.
- [30] M. J. Min, Y. Ding, L. Buratti, S. Pujar, G. Kaiser, S. Jana, and B. Ray. Beyond accuracy: Evaluating self-consistency of code large language models with identitychain. *arXiv preprint arXiv:2310.14053*, 2023.
- [31] A. Mohsin, H. Janicke, A. Wood, I. H. Sarker, L. Maglaras, and N. Janjua. Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms. *arXiv preprint arXiv:2406.12513*, 2024.

- [32] T.-D. Nguyen, Y. Zhou, X. B. D. Le, P. Thongtanunam, and D. Lo. Adversarial attacks on code models with discriminative graph patterns. *arXiv preprint arXiv:2308.11161*, 2023.
- [33] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.
- [34] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE, 2015.
- [35] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.
- [36] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.
- [37] S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer. An empirical study on usage and perceptions of llms in a software engineering project. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 111–118, 2024.
- [38] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- [39] M. Sclar, Y. Choi, Y. Tsvetkov, and A. Suhr. Quantifying language models’ sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. In *The Twelfth International Conference on Learning Representations*, 2024.
- [40] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- [41] A. Shirafuji, Y. Watanobe, T. Ito, M. Morishita, Y. Nakamura, Y. Oda, and J. Suzuki. Exploring the robustness of large language models for solving programming problems. *arXiv preprint arXiv:2306.14583*, 2023.
- [42] S. Wan, C. Nikolaidis, D. Song, D. Molnar, J. Crnkovich, J. Grace, M. Bhatt, S. Chennabasappa, S. Whitman, S. Ding, et al. Cyberseceval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models. *arXiv preprint arXiv:2408.01605*, 2024.
- [43] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, et al. Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*, 2022.
- [44] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32, 2019.
- [45] X. Wei, S. K. Gonugondla, S. Wang, W. Ahmad, B. Ray, H. Qian, X. Li, V. Kumar, Z. Wang, Y. Tian, et al. Towards greener yet powerful code generation via quantization: An empirical study. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 224–236, 2023.
- [46] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.
- [47] S. Xu, Z. Li, K. Mei, and Y. Zhang. Core: Llm as interpreter for natural language programming, pseudo-code programming, and flow programming of ai agents. *arXiv preprint arXiv:2405.06907*, 2, 2024.

- [48] H. Yan, S. S. Vaidya, X. Zhang, and Z. Yao. Guiding ai to fix its own flaws: An empirical study on llm-driven secure code generation. *arXiv preprint arXiv:2506.23034*, 2025.
- [49] Z. Yang, Z. Sun, T. Z. Yue, P. Devanbu, and D. Lo. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *arXiv preprint arXiv:2403.07506*, 2024.
- [50] P. Yin, W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi, J. Howland, P. Bailey, M. Catasta, H. Michalewski, et al. Natural language to code generation in interactive data science notebooks. *arXiv preprint arXiv:2212.09248*, 2022.
- [51] W. Zaremba, E. Nitishinskaya, B. Barak, S. Lin, S. Toyer, Y. Yu, R. Dias, E. Wallace, K. Xiao, J. Heidecke, et al. Trading inference-time compute for adversarial robustness. *arXiv preprint arXiv:2501.18841*, 2025.
- [52] C. Zhang, Z. Wang, R. Zhao, R. Mangal, M. Fredrikson, L. Jia, and C. Pasareanu. Attacks and defenses for large language models on coding tasks. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2268–2272, 2024.
- [53] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.
- [54] T. Y. Zhuo, Z. Li, Y. Huang, F. Shiri, W. Wang, G. Haffari, and Y.-F. Li. On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex. *arXiv preprint arXiv:2301.12868*, 2023.