

ONE STEP FORWARD, TWO STEPS BACK: REGRESSION ERRORS AND COST INEFFICIENCIES IN LLM ITERATIVE REFINEMENT FOR CODE GENERATION

Lucas T. Borges & Ricardo Rios

Institute of Computing
Federal University of Bahia
Salvador, Bahia, Brazil
{lucastb, ricardoar}@ufba.br

ABSTRACT

Self-Refine, a self-reflection framework, has proven effective for improving Large Language Model performance in stylistic tasks, such as code readability. However, its utility for ensuring functional correctness in code generation is often undermined by significant reliability and cost trade-offs. In this work, we characterize the specific failure modes of Self-Refine in this domain and analyze the impact of "Asymmetric Self-Refine" (utilizing smaller, cost-efficient models as critics) to mitigate high inference costs of iterative refinement. Evaluating on the LiveCodeBench dataset, we uncover a critical negative result: the framework fails to reliably improve Pass@1 scores and often degrades them due to hallucination-induced regression errors. Critics hallucinated flaws in functionally correct code, prompting the actor to introduce bugs into valid solutions. This instability, present even in symmetric configurations, is exacerbated by smaller critics, which drive total inference costs higher due to excessive refinement loops, contradicting the expectation of cost savings.

1 INTRODUCTION

While Large Language Models (LLMs) consistently produce coherent and contextually relevant outputs, they often fall short on the first attempt when faced with intricate tasks. Iterative refinement has emerged as a compelling paradigm for enhancing responses without supervised fine-tuning. Frameworks such as Self-Refine (Madaan et al., 2023) demonstrate that LLMs can effectively critique and improve their own outputs, achieving significant gains in tasks like sentiment reversal and code readability improvement.

However, this iterative process introduces a massive computational overhead, increasing token usage with each refinement iteration. To address this, we proposed an Asymmetric Self-Refine strategy, hypothesizing that the critic's role (identifying syntax or logic errors) is computationally simpler than generation and could be offloaded to smaller, cheaper models without compromising quality.

Furthermore, while other self-reflection frameworks such as Reflexion (Shinn et al., 2023) have displayed substantial gains in code generation tasks, they often require an external oracle to verify each proposed solution before the self-reflection step. Such a tool is not always available in real-world scenarios, which raises the question of whether Self-Refine can serve as a fallback in these situations, since its potential for improving functional correctness in code generation has not been firmly established.

In this work, we evaluate the Self-Refine framework on the LiveCodeBench dataset (Jain et al., 2024) and report a negative result that challenges the practical applications of this algorithm for the generation of functionally correct code. While prior studies have indicated that self-repair gains in code generation are often modest when accounting for the increased inference cost (Olausson et al., 2024), our work demonstrates that iterative refinement can actively degrade functional correctness through hallucination-induced regression errors. Furthermore, we show that attempting to

Table 1: Trajectory analysis. Scores are Pass@1 (%); parentheses show absolute change from Raw (Single-Pass). Regr. is regression errors. Avg. It. is the average refinement steps. Cost is normalized to single-pass (n/a for local models). Iterations indicate the maximum, but early stopping is allowed.

Family	Actor / Critic	Raw	1 Iteration	2 Iterations	3 Iterations	Regr.	Avg. It.	Cost
GPT-4.1	Mini / Mini	38.3	37.9 (-0.4)	38.3 (0.0)	38.7 (+0.4)	3	1.25	2.15x
	Mini / Nano	38.3	39.9 (+1.6)	41.5 (+3.2)	38.7 (+0.4)	8	1.70	2.56x
Qwen2.5	32B / 32B	22.6	21.8 (-0.8)	23.8 (+1.2)	23.0 (+0.4)	13	2.02	–
	32B / 7B	23.0	21.4 (-1.6)	21.8 (-1.2)	19.4 (-3.6)	18	2.37	–
	32B / 3B	23.8	25.0 (+1.2)	23.4 (-0.4)	23.4 (-0.4)	13	2.63	–

mitigate these overheads through asymmetric model pairings (utilizing smaller models as critics) can counterintuitively increase total costs due to a surge in unnecessary refinement iterations.

2 ANALYSIS OF NEGATIVE RESULTS

We evaluated the efficacy of Self-Refine on 248 problems from the LiveCodeBench dataset. Problems were selected from the post-June 2024 pool to avoid data contamination (Cheng et al., 2025). Moreover, the problems were sampled so as to keep the difficulty distribution as close as possible to the distribution from the complete dataset. We compared symmetric baselines (e.g., GPT-4.1-Mini as both actor and critic) against asymmetric pairs (e.g., Mini Actor, Nano Critic). We used the "Instruct" variations for the Qwen2.5-Coder models (Hui et al., 2024) and compared the results using the Pass@1 metric (Chen et al., 2021). Following Madaan et al. (2023), refinement was capped at 3 iterations, with early stopping triggered upon critic acceptance. Execution details and a link to the repository with the source code and full experimental results can be found in the Reproducibility Statement.

2.1 HALLUCINATION-INDUCED REGRESSIONS

Our analysis characterizes the specific mechanisms that make iterative refinement unreliable for code generation, specifically focusing on hallucination-induced regressions. We define a Regression Error in this context as a valid, passing solution rejected by the critic and subsequently refined into a failing one. As shown in Table 1, additional refinement steps actually decreased Pass@1 scores in multiple instances due to introduced errors outnumbering new correct solutions for that iteration.

2.2 THE COST-INEFFICIENCY PARADOX

We hypothesized that a cheaper critic would reduce costs. Results show the opposite: the Asymmetric Mini/Nano pair was approximately 19% more expensive than the symmetric one ($2.56\times$ vs $2.15\times$ the single-pass baseline), despite GPT-4.1-Nano being 4 times cheaper than Mini (OpenAI, 2026a;b). Root cause analysis via confusion matrices (see Appendix A) revealed that less robust critics were stricter and significantly more prone to triggering refinement. This behavior inflates the average iteration count (1.70 vs 1.25 for the GPT-4.1 family), negating per-token savings and increasing the likelihood of regression errors. This trend extended to the Qwen family, where average iteration counts proved inversely proportional to model size, confirming that the inefficiency pattern persists even where direct API cost comparisons do not apply.

2.3 QUALITATIVE ANALYSIS OF REGRESSION ERRORS

To investigate the mechanism driving performance drops, we conducted a qualitative analysis of the refinement logs. We identified a recurring pattern in the cases where regression was observed: critics often fail to comprehend the actor’s algorithmic logic. Instead of verifying functional correctness against the provided constraints, the critic hallucinates missing edge cases or misinterprets the problem statement itself, demanding unnecessary (and sometimes breaking) changes. In attempting to satisfy the flawed feedback, the actor is sometimes misled to alter correct code, directly introducing genuine bugs.

This behavior explains why smaller critics exhibit a higher tendency to reject valid code and request refinement that results in the actor breaking an otherwise correct solution. As a direct result of its lessened logic capabilities, a smaller critic may lack the robustness necessary to understand the solution generated by its larger counterpart. Thus, the agent rejects it and requests unnecessary refinement, as illustrated by the case study in Appendix C. In the analyzed instance, a 7B critic induces a 32B actor to corrupt a perfect zero-shot solution through faulty feedback, only to accept the ruined solution in the subsequent iteration.

3 CONCLUSION

Our investigation corroborates and extends the finding that Self-Refine is currently inefficient for zero-shot code generation. The lack of grounded verification allows critics to hallucinate non-existent errors or fail to identify real mistakes, leading to regression errors and failing to significantly improve pass rates while substantially increasing costs. This result is consistent with the known Self-Refine limitation regarding math reasoning tasks (Madaan et al., 2023) and existing observations in the coding domain (Olausson et al., 2024), both of which suggest that intrinsic feedback is a bottleneck for complex logic. Furthermore, we demonstrate that optimizing for cost via smaller critics backfires: the drop in reasoning capability leads to strict, noisy feedback that inflates iteration counts and total costs.

Future work should investigate more robust stopping criteria and explore hybrid architectures, such as pairing general-purpose actors with specialized code-verification critics. A deeper causal investigation into regression errors is also needed to identify which specific critique types are most prone to inducing bugs. For instance, our qualitative analysis suggested that a significant portion of flawed critiques originated from the critic identifying edge cases that do not impact the functional requirements of the solution. We suspect that explicit instructions to analyze "edge case failures" may have inadvertently biased smaller critics toward hallucinating nonexistent flaws. Investigating different prompting structures and few-shot examples could determine if this pedantic behavior can be mitigated through prompt engineering or if it represents a fundamental capability ceiling for smaller models.

ETHICS STATEMENT

This work investigates the limitations of automated code generation using Large Language Models. We acknowledge that relying on LLM-generated code in safety-critical workflows carries inherent risks, as these models may introduce subtle logic bugs or security vulnerabilities regardless of the refinement framework employed. Our findings specifically highlight how iterative feedback loops can exacerbate these reliability issues by breaking functionally correct code. Furthermore, the significant increase in inference costs without corresponding performance gains observed in this study contributes to the ongoing discourse regarding responsible and sustainable AI compute usage. This research utilizes exclusively publicly available datasets and does not involve human subjects.

REPRODUCIBILITY STATEMENT

We have taken steps to ensure the reproducibility of our results through open code and transparent data selection. A general overview of the experimental design is provided in Section 2. We utilize the LiveCodeBench dataset (v6), which is publicly available; the specific problem IDs used in the experiment are provided in our repository to allow exact replication. To maximize consistency and determinism, all models were queried with a temperature of 0. The full implementation, including generation, testing, and analysis scripts, along with the experimental results such as all generated solutions, tables and plots, is available at <https://github.com/LucasTBorges/Asymmetric-Self-Refine>. Prompt templates are also listed in Appendix B.

LLM USAGE STATEMENT

We utilized LLMs to assist with grammar, spelling, ensuring correct LaTeX formatting, and drafting code documentation. All experimental design, data collection, analysis, and scientific conclusions are the original work of the authors.

REFERENCES

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL <https://arxiv.org/abs/2107.03374>.
- Yuxing Cheng, Yi Chang, and Yuan Wu. A survey on data contamination for large language models, 2025. URL <https://arxiv.org/abs/2502.14425>.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL <https://arxiv.org/abs/2403.07974>.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL <https://arxiv.org/abs/2303.17651>.
- Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 36545–36593, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/9ddc141bdf9d1db510cefff56c586ad-Paper-Conference.pdf.
- OpenAI. GPT-4.1 mini Model | OpenAI API, 2026a. URL <https://platform.openai.com/docs/models/gpt-4.1-mini>. Accessed: 2026-01-14.
- OpenAI. GPT-4.1 nano Model | OpenAI API, 2026b. URL <https://platform.openai.com/docs/models/gpt-4.1-nano>. Accessed: 2026-01-14.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*, Red Hook, NY, USA, 2023. Curran Associates Inc.

A CRITIC CONFUSION MATRICES

We analyze the decision-making patterns of the critic agents by comparing their verdicts (Pass/Refine) against the ground truth provided by unit tests (Pass/Fail). We interpret the matrices as follows: the top-left and bottom-right quadrants represent correct decisions (True Positives and True Negatives). The top-right quadrant indicates *False Rejections*, where the critic rejects a functionally correct solution. Conversely, the bottom-left quadrant represents *False Acceptance*, indicating "lenient" behavior where bugs are overlooked.

This analysis reveals the mechanism behind the *Cost-Inefficiency Paradox* discussed in Section 2. We observe an inverse relationship between model size and strictness: less robust critics from the same family consistently exhibit higher rejection rates, even for correct solutions. This "pedantic" behavior drastically inflates the number of refinement requests and total critiques (visible in the increasing sample size N in each chart), resulting in a counter-intuitive cost increase when utilizing cheaper models.

Conversely, more robust critics (like GPT-4.1-Mini) demonstrate a conservative bias, favoring acceptance even when solutions are incorrect. While this behavior caps costs by limiting iterations, it also limits the potential for improvement, as the critic effectively gives up too early.

A.1 GPT-4.1 FAMILY

The symmetric Mini/Mini experiment demonstrates a clear bias towards acceptance. The confusion matrix reveals a high False Acceptance rate (the highest among all experiments) indicating that the critic struggles to identify subtle logic errors in incorrect code. However, its high recall (low False Rejection) means it rarely hallucinates errors in correct code, preserving valid solutions.

In contrast, the asymmetric Mini/Nano pair demonstrated superior rejection of invalid solutions but in detriment to its ability to correctly accept valid answers. This strictness trend is even more pronounced in the Qwen2.5-Coder family, exacerbating as the critic's parameter count decreases.

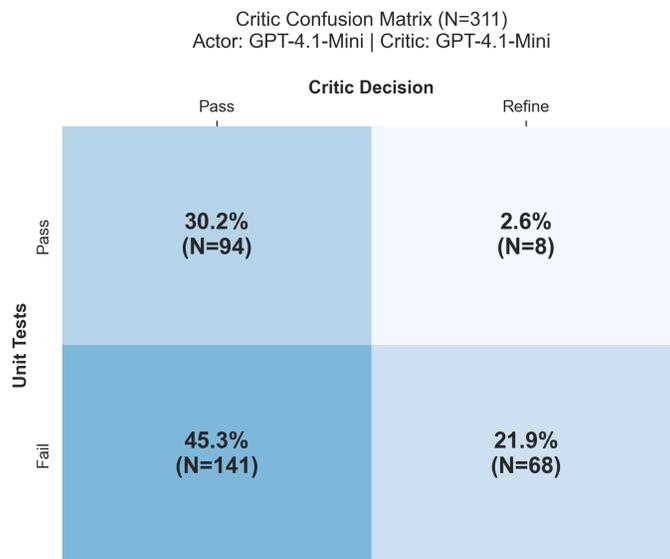


Figure 1: The confusion matrix for the Mini/Mini pair. The critic maintained an acceptance rate of over 75%, explaining the low average iteration count (1.25) and the minimal number of regression errors (3).

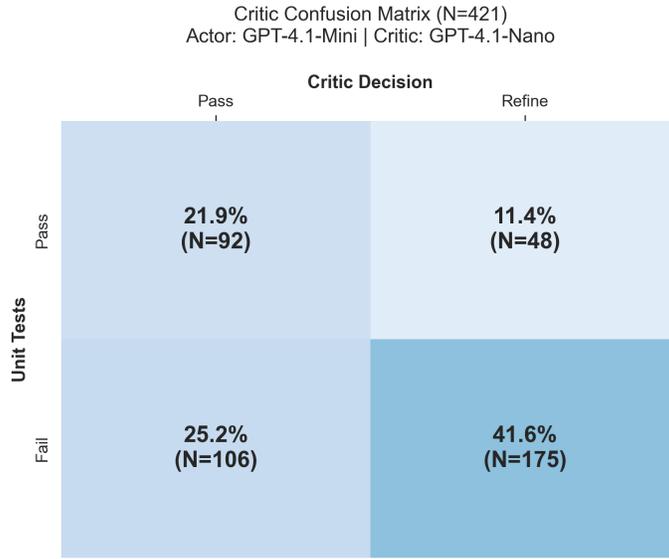


Figure 2: In comparison to its symmetric counterpart, the asymmetric Mini/Nano pair exhibits a high False Rejection rate. This pedantic behavior forces the actor to refine valid code, inflating costs and increasing the risk of introducing regression errors.

A.2 QWEN2.5 FAMILY

The Qwen2.5 experiments further highlight the risks of utilizing weaker critics in the code generation domain. The symmetric 32B/32B pair was the only configuration to show a net improvement (albeit modest) in Pass@1 scores at the 3-iteration budget. The experiments using this family yielded significantly more regression errors than the GPT-4.1 models. The bias towards rejection intensifies as the critic size shrinks, culminating in the 32B/3B pair, which granted a "Pass" verdict in only $\approx 10.2\%$ of cases.

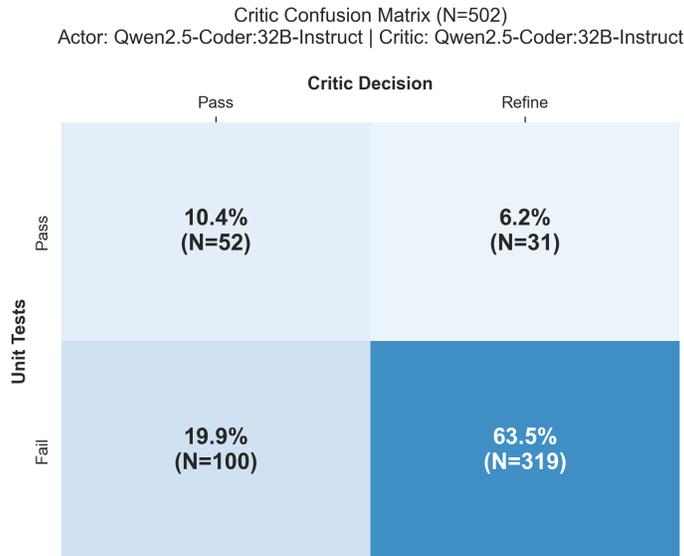


Figure 3: The symmetric 32B/32B baseline provides a reference for the stability of the Qwen2.5-Coder family on this task, showing a balanced distribution of errors compared to the asymmetric pairs.

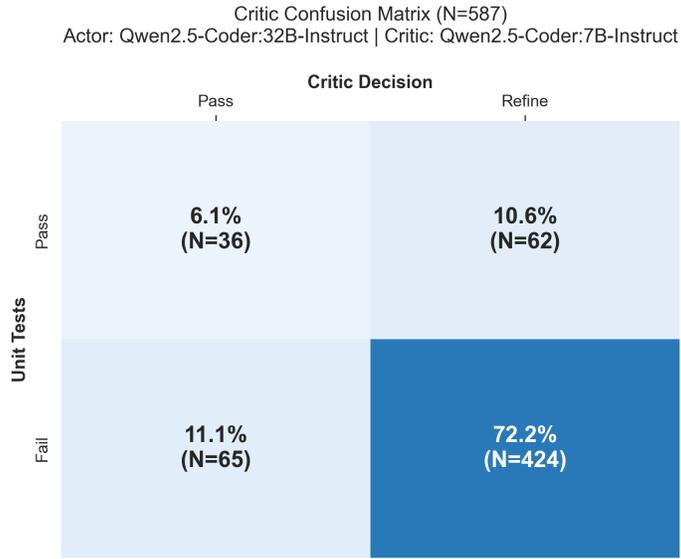


Figure 4: The 32B/7B asymmetric pair exhibits the highest instability of all experiments. The 7B critic frequently hallucinates errors in correct code (high False Rejection), directly correlating with the 18 Regression Errors and the net performance loss reported in the main text.

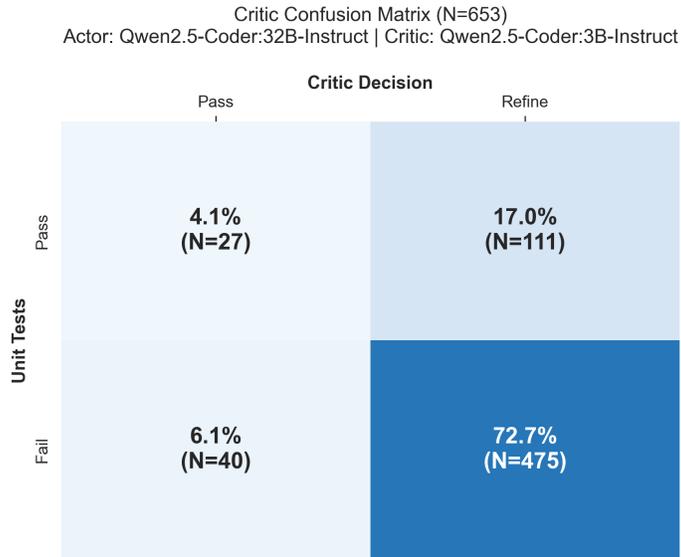


Figure 5: The 32B/3B pair displays strictness trends similar to the 7B critic, with an even sharper bias towards rejection, despite producing fewer total regression errors and yielding overall better results than the 7B configuration.

B PROMPT TEMPLATES

To ensure deterministic parsing of the critic’s feedback, we utilized Structured Outputs (specifically enforcing a JSON schema via Pydantic models) for the critic agent. This guaranteed that the decision to stop or continue refining was based on a boolean flag (`passed`) rather than heuristic text parsing.

The specific prompts for each stage of the refinement loop are detailed below. Placeholders (e.g., {problem}) are replaced with instance-specific data at runtime.

B.1 STRUCTURED OUTPUT SCHEMA

The following Pydantic model was used to enforce the critic's response format using LangChain's Structured Output feature:

```
class CriticResponse(BaseModel):
    """
    Model for the critic's structured response.
    """
    critique: str = Field(description="Detailed critique of the code.")
    passed: bool = Field(description="Indicates if the code is fully correct.")
```

B.2 ACTOR: INITIAL GENERATION

System Prompt:

```
You are an expert Python programmer. You will be given a question
(problem specification) and will generate a correct Python program
that matches the specification and passes all tests.
```

User Prompt:

```
### Question:
{problem}

### Format:
Read the inputs from stdin, solve the problem, and write the answer to
stdout.
Enclose your code within delimiters as follows. Ensure that when the
python
program runs, it reads the inputs, runs the algorithm, and writes output
to
STDOUT.

```python
YOUR CODE HERE
```

Answer: (use the provided format with backticks)
```

B.3 CRITIC: FEEDBACK GENERATION

System Prompt:

```
You are a senior code reviewer. Analyze the provided Python code for
logical errors, syntax issues, or edge case failures. The code MUST
read from Standard Input (stdin) and print to Standard Output
(stdout). Do NOT simply rewrite the code. Focus on identifying
specific bugs.

Your task is to:

1. Provide a detailed critique of the code, briefly describing any
mistakes.

2. Determine if the code is fully correct (PASS) or has bugs (NOT PASS).

Respond in JSON format as follows:
{
```

```
"critique": "<detailed critique here>",
"passed": <true or false>
}
```

User Prompt:

```
Problem Statement: {problem} Code to review: <code_block> {code}
</code_block>
```

B.4 ACTOR: REFINEMENT

System Prompt:

```
You are an expert Python programmer. You previously wrote code that a
reviewer flagged as incorrect. Fix the code based on the feedback.
Output ONLY the full fixed code, inside a markdown code block.
```

User Prompt:

```
Original Problem: {problem} Your Previous Code:

```python
{code}
```

Reviewer Feedback: {critique} Please rewrite the code to fix the issues.
Ensure it reads from STDIN and prints to STDOUT.
```

C QUALITATIVE CASE STUDY: HALLUCINATED CRITIC FEEDBACK

This section provides a concrete example of a hallucination-induced regression error, directly addressing the causal mechanism behind the performance degradation observed in the experiment. The following logs are taken from the Qwen 32B (actor) and 7B (critic) experiment on the LiveCodeBench problem abc357_a ("Sanitize Hands").

The original problem statement contains three input and output samples and a breakdown of the logic behind the first output, which have been omitted from the subsection below for the sake of simplicity. However, the samples were included in the prompt during the experiment and are available on the LiveCodeBench dataset.

C.1 PROBLEM STATEMENT:

There is a bottle of disinfectant that can disinfect exactly M hands.

N aliens come one by one to disinfect their hands.

The i -th alien ($1 \leq i \leq N$) has H_i hands and wants to disinfect all of their hands once.

Determine how many aliens can disinfect all of their hands.

Here, even if there is not enough disinfectant left for an alien to disinfect all of their hands when they start, they will use up the remaining disinfectant.

Input

The input is given from Standard Input in the following format:

```
 $N$   $M$ 
 $H_1$   $H_2$  ...  $H_N$ 
```

Output

Print the number of aliens who can disinfect all of their hands.

Constraints

- $1 \leq N, M \leq 100$
- $1 \leq H_i \leq 100$
- All input values are integers.

C.2 ZERO-SHOT SOLUTION (QWEN 32B ACTOR):

The actor generated a functionally correct solution on the first attempt. The logic correctly iterates through the queue of aliens, consumes the necessary disinfectant and increments the count if the remaining liquid is sufficient, and explicitly halts the process the moment an alien requires more disinfectant than what remains (matching the problem's constraint that the remaining liquid is entirely consumed).

Zero-shot Solution:

```
import sys
def main():
    import sys
    input = sys.stdin.read().split()
    N = int(input[0])
    M = int(input[1])
    H = list(map(int, input[2:]))
    count = 0
    for hands in H:
        if M >= hands:
            count += 1
            M -= hands
        else:
            break
    print(count)

if __name__ == "__main__":
    main()
```

C.3 THE REJECTION (QWEN 7B CRITIC):

Although the initial solution was functionally correct and passed all unit tests, the 7B critic rejected it. The feedback it generated was fundamentally flawed, as it assumes we should consider all aliens from the line, even when the disinfectant is over, which contradicts the problem statement:

"The code is mostly correct, but it has a logical error. The issue lies in the loop that iterates through the aliens' hands. Once an alien's hands exceed the remaining disinfectant (M), the loop breaks prematurely, which means not all aliens are considered. The correct approach should continue iterating through all aliens even if some of them cannot be fully disinfected."

C.4 THE REGRESSION ERROR (ACTOR'S REFINEMENT):

Following the critic's explicit instructions, the actor removed the `break` statement. By doing so, it introduced a fatal bug: the code now simply skips an alien if they have too many hands, and continues to disinfect subsequent aliens with fewer hands. This violates the clearly stated rule that the current alien would "use up the remaining disinfectant," leaving nothing for the rest of the queue.

The actor did follow the instructions though, resulting in the following refined code (which did not pass the unit tests):

```
import sys
def main():
    input = sys.stdin.read().split()
    N = int(input[0])
    M = int(input[1])
    H = list(map(int, input[2:]))
    count = 0
    for hands in H:
        if M >= hands:
            count += 1
            M -= hands
    print(count)

if __name__ == "__main__":
    main()
```

C.5 THE EARLY STOP (QWEN 7B CRITIC):

The critic then evaluated this newly bugged code and confidently accepted it, demonstrating its inability to grasp the sequential constraints of the problem.

"The code is logically correct and handles the problem as described. It reads input from standard input, processes it to determine how many aliens can disinfect all their hands with the given amount of disinfectant, and prints the result. There are no syntax issues or edge case failures identified."

This example highlights the danger of using Self-Refine for code verification: the critic confidently enforced incorrect constraints, overriding the original logic of the actor.