SemGuard: Real-Time Semantic Evaluator for Correcting LLM-Generated Code

Qinglin Wang
Shandong Normal University
Jinan, China
2023317094@stu.sdnu.edu.cn

Zhihong Sun
Shandong Normal University
Jinan, China
2022021002@stu.sdnu.edu.cn

Ruyun Wang
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
wangruyun@iie.ac.cn

Tao Huang
Shandong Normal University
Jinan, China
2022317095@stu.sdnu.edu.cn

Zhi Jin
Key Lab of HCST (PKU), MOE; SCS
Beijing, China
zhijin@pku.edu.cn

Ge Li
Key Lab of HCST (PKU), MOE; SCS
Beijing, China
lige@pku.edu.cn

Chen Lyu*
Shandong Normal University
Jinan, China
lvchen@sdnu.edu.cn

***Abstract*—*Large Language Models* (LLMs) can translate natural language requirements into code, yet empirical analyses of representative models reveal that *semantic errors*—programs that compile but behave incorrectly—constitute the majority of observed faults (e.g., >60% on DeepSeek-Coder-6.7B and QwenCoder-7B). Post-hoc repair pipelines detect such faults only *after* execution, incurring latency, relying on incomplete test suites, and often mis-localizing the defect. Since semantic drift originates in the autoregressive decoding process, *intervening while the code is being generated* is a direct way to stop error propagation. Constrained-decoding approaches such as ROCODE attempt this, but still wait until the entire program runs to obtain feedback and use entropy heuristics that do not truly capture semantics. A more effective solution must inject *semantic* signals—early and precisely—into the decoding process.We present SemGuard, a semantic-evaluator-driven framework that performs real-time, line-level semantic supervision. To train the evaluator, we build *SemDiff*, the first dataset with fine-grained annotations that mark the exact line where a correct and an incorrect implementation diverge. The evaluator, once embedded in the LLM's decoder, flags deviations on partial code, rolls back to the faulty line, and guides regeneration—without executing the program or requiring test cases. Across four benchmarks, SemGuard consistently outperforms state-of-the-art baselines. It lowers the semantic error rate by 19.86% on *SemDiff* relative to ROCODE, and lifts Pass@1 by 48.92% on the real-world *LiveCodeBench* with CodeLlama-7B. Similar gains hold for StarCoder2-7B on *MBPP* and for DeepSeekCoder-6.7B on the Java benchmark *SemDiff-Java*, demonstrating model- and language-agnostic effectiveness.**

***Index Terms*—Semantic Supervision, Large Language Models, Code Generation.**

I. Introduction

Large Language Models (LLMs), exemplified by GPT-o3 [1] and DeepSeek-R1 [2], have significantly advanced automated software engineering tasks by translating natural language into executable code [3]. However, the generated programs frequently contain subtle yet critical *semantic errors*, where code compiles successfully but deviates from intended functionality. As shown in Figure 1, semantic errors account for over 60% of faults in LLM-generated code, surpassing syntax and runtime errors combined. This prevalence arises primarily because existing debugging tools (compilers, static analyzers, and unit tests) lack sufficient semantic awareness, and the autoregressive decoding process of LLMs propagates initial semantic deviations throughout subsequent code generation, exacerbating early mistakes.

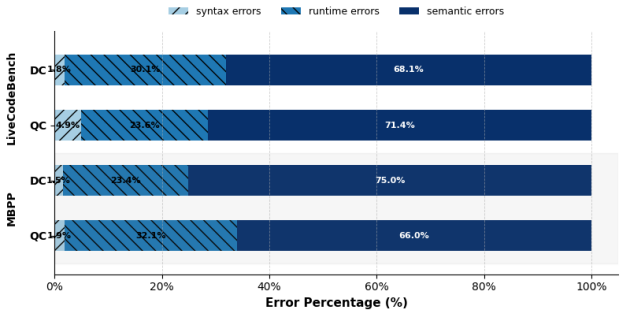


Fig. 1: Distribution of error types in LLM-generated code. Results are collected using greedy decoding with DeepSeek-Coder-6.7B (**DC**) and Qwen-Coder-7B (**QC**) on the *MBPP* and *LiveCodeBench* benchmarks.

Previous efforts, notably ROCODE [4], have attempted to mitigate these semantic inaccuracies by integrating traditional program analysis tools within the decoding phase of LLMs. Although ROCODE effectively curbs syntactic and runtime errors, it remains limited in addressing semantic errors due to two critical drawbacks: ❶ **Post-generation semantic detection.** ROCODE verifies semantic correctness only after the complete program has been generated and corresponding test cases executed, resulting in delayed identification and rectification of semantic errors. This post-generation validation

*Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00160

not only reduces development efficiency but also introduces potential security risks by running potentially erroneous and unverified code [5]. ❷ **Imprecise backtracking-point localization.** ROCODE employs entropy-based heuristics to pinpoint the origins of semantic deviations. However, entropy inherently reflects prediction uncertainty rather than causal relationships, leading to inaccurate identification of the true sources of errors. Consequently, substantial segments of correctly generated code may be incorrectly discarded during backtracking, causing unnecessary computational overhead and diminished generation quality.

To address these limitations, we integrate a lightweight **semantic evaluator** into the decoding process, allowing real-time assessment of semantic correctness in *partial code*. By identifying semantic deviations before code execution, our approach immediately backtracks to the erroneous line and regenerates code, effectively curbing error propagation. However, the primary difficulty lies in **training a dependable semantic evaluator**: unlike syntax errors, semantic correctness is implicit and highly context-dependent, complicating large-scale formalization, detection, and annotation.

Specifically, we encounter two core challenges: ❶ **Scarcity of line-level semantic annotations.** Existing benchmarks such as *APPS* [6], *CodeContests* [7], and *CodeNet* [8] rarely indicate exactly *where* semantic deviations occur or precisely *how* the erroneous code diverges from correct solutions. ❷ **Undecidable semantic validity of partial code.** Without full program context or reference specifications, even experts find it challenging to reliably determine semantic correctness for isolated code fragments. Manual annotation is thus slow and error-prone, while automated error-localization methods—such as spectrum-based fault localization (e.g., Tarantula [9], Ochiai [10], Dstar [11])—are ineffective, as generated partial fragments often lack complete execution traces for meaningful analysis.

To overcome these obstacles, we devise a dedicated data pipeline to generate fine-grained annotations of semantic deviations at the code-fragment level. Specifically, we extract pairs of highly similar correct and incorrect implementations from the *CodeNet* dataset, identifying precise lines where semantic deviation begins. For code pairs with minimal differences, segmentation occurs directly at the differing line; for more complex cases, we utilize LLMs to semi-automatically pinpoint semantic deviation points, enabling scalable and high-quality annotation.

Leveraging this pipeline, we construct *SemDiff*, a dataset providing fine-grained annotations of semantic deviations. Using *SemDiff*, we train a semantic evaluator that judges partial code correctness in real time. This evaluator is embedded into the decoding process of an LLM, monitoring semantic validity of intermediate outputs continuously. Upon detecting a deviation, the LLM promptly backtracks and regenerates from the faulty line, preventing error propagation. We term this framework **SemGuard**, a *semantic-evaluator-driven* code generation system combining real-time semantic detection and targeted backtracking to proactively safeguard logical correctness in LLM-generated code.

We extensively evaluate SemGuard across four benchmarks—*SemDiff*, *MBPP*, *LiveCodeBench*, and *SemDiff-Java*—to verify its effectiveness, generalizability, and cross-language adaptability. Experimental results consistently show that SemGuard substantially outperforms state-of-the-art baselines in reducing semantic errors. Specifically, on *SemDiff*, SemGuard achieves a **19.86%** lower semantic error rate compared to ROCODE, without requiring code execution or test cases. On Python tasks, SemGuard boosts the Pass@1 scores significantly: by **11.14%** with StarCoder2-7B on *MBPP* and by **48.92%** with CodeLlama-7B on the challenging *LiveCodeBench*. SemGuard also demonstrates strong language-independent performance, improving DeepSeekCoder-6.7B's Pass@1 by **25.09%** on *SemDiff-Java*. These outcomes confirm SemGuard's robust applicability across diverse models and programming languages.

In summary, our work makes three key contributions:

- **SemGuard: Real-time Semantic Detection and Intervention.** We propose SemGuard, a lightweight semantic evaluator embedded within LLM decoding, enabling immediate line-level semantic checking and targeted backtracking. This effectively prevents semantic errors from propagating, preserving previously generated correct code.
- **SemDiff: Automated Semantic Annotation Pipeline.** We introduce *SemDiff*, the first dataset providing precise line-level semantic annotations. Built via a diff-guided and LLM-assisted pipeline, *SemDiff* pairs semantically correct and incorrect code fragments, pinpointing exact semantic deviation points to facilitate training robust semantic evaluators.
- **Cross-language and Cross-model Validation.** We conduct comprehensive evaluations using four representative LLMs—StarCoder2, CodeLlama, DeepSeek-Coder, and QwenCoder—on diverse benchmarks covering both Python and Java tasks. The consistent performance improvements across languages and models demonstrate SemGuard's broad applicability and cross-language generalizability. For reproducibility and future research, we publicly release the *SemDiff* dataset, trained evaluators, and the complete, open-source implementation of SemGuard.[1]

II. Motivation

A. A Concrete Example: Limitations of ROCODE

ROCDOE reduces the occurrence of syntax and runtime errors by introducing a backtracking mechanism based on program analysis and test cases to locate decoding deviations. However, its design reveals certain limitations when dealing with semantic errors caused by semantic deviation during the decoding process.

As shown in Figure 2, we examine a representative task from CodeNet that asks the model to transform a string of parentheses into a balanced form using the fewest insertions. Among multiple valid answers, the lexicographically smallest one should be returned. ROCODE first generates an initial

[1] https://github.com/wwwql/SemGuard

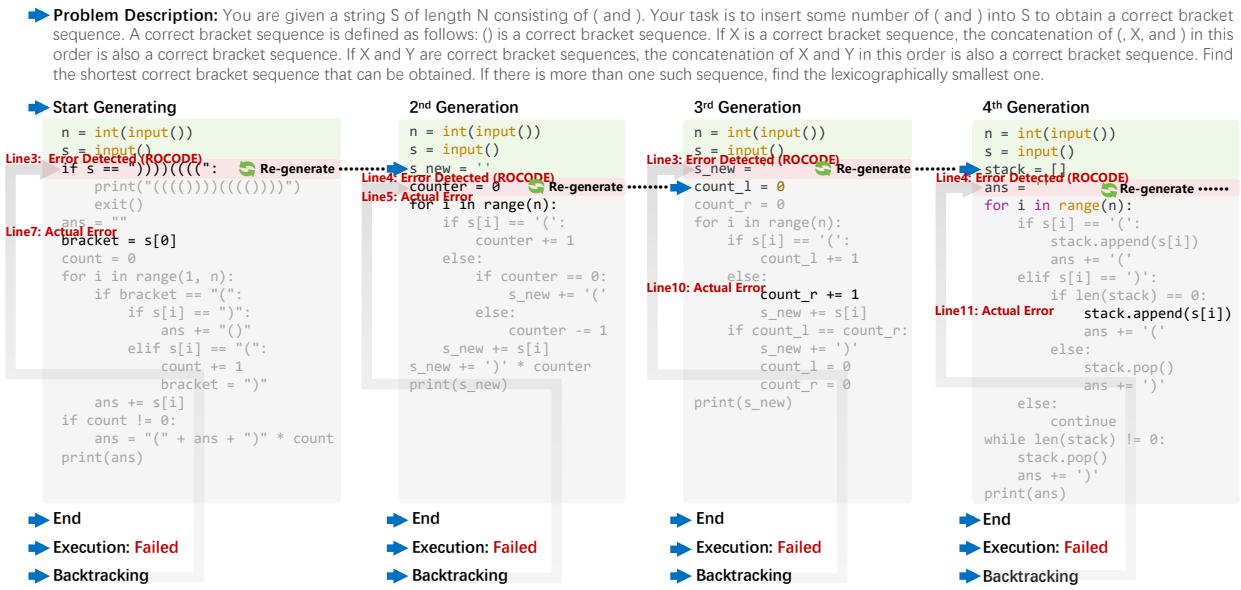


Fig. 2: ROCODE backtracking sequence (DeepSeekCoder-6.7B) on CodeNet task #03696.

program, then applies three rounds of backtracking, but still fails to produce the correct result.

The root cause lies in an early semantic error: during the first generation, the variable assignment `bracket = s[0]` captures only the first character's bracket type. Subsequent logic relies on this local reference rather than tracking the actual number of unmatched left parentheses, which is crucial for correct insertion logic. This subtle semantic deviation gradually shifts the model's reasoning trajectory, leading to accumulation of errors.

Despite backtracking, ROCODE cannot effectively recover. This failure reflects two deeper limitations:

- **Delayed semantic detection.** ROCODE performs semantic validation only after the full program is generated and executed on test cases. This example underwent four complete rounds of code generation, with semantic errors detected after execution in each round. This delay not only wastes computation but also poses security risks by executing unverified code.
- **Inaccurate rollback localization.** ROCODE uses entropy-based heuristics to choose rollback points, but entropy reflects uncertainty—not semantic deviation. For instance, during the three backtracking processes, ROCODE repeatedly locates the backtracking point incorrectly at the beginning of the code, rather than at the actual location where the semantic deviation occurs. Especially during the process from the third generation to the fourth generation, the true semantic error appears at line 10 (`count_r +=`), but the model rolls back to line 3 (`s_new = ''`). This misidentification truncates correct code and causes large-scale, unnecessary regeneration.

B. Key Idea: Real-Time, Granular Semantic Supervision

This example underscores that post-hoc test-based correction alone is insufficient; semantic correctness must instead be proactively ensured *during* code generation. To achieve this, we propose integrating a lightweight **semantic evaluator** directly within the LLM decoding process, enabling real-time, fine-grained semantic checking. The evaluator continuously analyzes partial programs as they are generated, immediately detects semantic deviations, and initiates precise, line-level rollback and regeneration. This approach effectively halts error propagation with minimal disruption to previously correct code fragments.

Implementing such an evaluator is non-trivial because fine-grained semantic labels are scarce and partial-code semantics are hard to judge—challenges already detailed in Section I. In the next section we explain how the proposed SemGuard framework overcomes these obstacles.

III. Approach

We next detail SemGuard, our framework for ensuring semantic correctness via real-time evaluation of code fragments (Figure 3). The workflow comprises three stages: ❶ *dataset construction*, which produces line-level semantic-deviation pairs; ❷ *training*, where the annotated corpus is used to train a lightweight evaluator; and ❸ *inference*, in which the trained evaluator is embedded into the LLM's decoding process to detect deviations on-the-fly and trigger targeted backtracking.

A. Dataset Constructing

Training a fragment-level semantic evaluator requires data that pinpoints where a program first diverges from the intended logic—information missing from existing public corpora. We

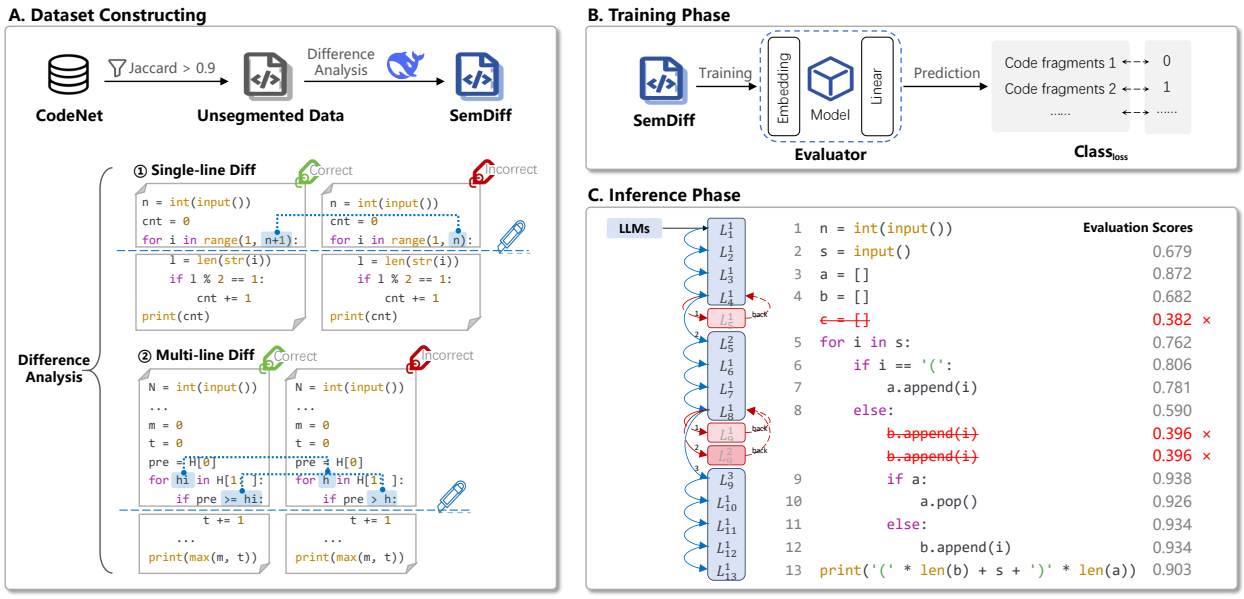


Fig. 3: SemGuard Architecture, consisting of three components: (1) Dataset Construction, (2) Training Phase, (3) Inference Phase. In Phase C, L_i^j denotes the i-th line of code after the j-th attempt (j=1 initial, j>1 after backtracking).

therefore build *SemDiff*, a new dataset with train, validation, and test splits in which every erroneous solution is paired with a correct counterpart and tagged at the exact line of semantic deviation. These line-level annotations allow us to slice each program into *prefix* + *faulty line* + *suffix* segments, providing the fine-grained supervision needed to teach the evaluator to judge the correctness of arbitrary code fragments.

As shown in Figure 3, we draw our source corpus from *CodeNet*, which comprises more than 14 million submissions in over 50 programming languages and covers roughly 4 000 competitive- programming tasks. We treat each submission as an ordered line sequence $C = \{ l_1, l_2, \ldots, l_n \}$, where l_i is the i-th line of code.

Verification. Every solution labelled *correct* in CodeNet is re-executed in a local sandbox; samples whose output deviates from the ground-truth answer are discarded. For solutions labelled *incorrect*, we keep only those that fail for semantic reasons, filtering out purely syntactic or runtime faults.

Pairing. For each user we pair a retained erroneous submission C_{err} with a correct one C_{corr}. Let T_{corr} and T_{err} be their respective n-gram sets; we compute the Jaccard similarity

$$J(C_{\text{corr}}, C_{\text{err}}) = \frac{|T_{\text{corr}} \cap T_{\text{err}}|}{|T_{\text{corr}} \cup T_{\text{err}}|} \tag{1}$$

Pairs with $J > 0.9$ are retained, ensuring that the two versions differ lexically as little as possible—typically by a single, localized semantic deviation.

For every retained pair $(C_{\text{corr}}, C_{\text{err}})$ we perform a line–by–line diff and record the index set

$$D = \{ i \mid l_i^{\text{corr}} \neq l_i^{\text{err}} \} \tag{2}$$

whose cardinality $|D|$ is the number of differing lines. If $|D| = 1$ the pair differs in a single line (see *Single-line Diff* case); otherwise the changes span multiple lines (see *Multi-line Diff* case). Let $i^* = \min D$ be the first point of deviation. We slice each program at i^* to obtain a *semantic prefix* that will serve as a training instance:

$$\begin{aligned} S_{\text{corr}} &= \langle l_1^{\text{corr}}, l_2^{\text{corr}}, \ldots, l_{i^*}^{\text{corr}} \rangle \\ S_{\text{err}} &= \langle l_1^{\text{err}}, l_2^{\text{err}}, \ldots, l_{i^*}^{\text{err}} \rangle \end{aligned} \tag{3}$$

The two prefixes differ only in the final line l_{i^*}, giving the evaluator a minimal contrast between a correct and an erroneous semantic unit.

The *Multi-line Diff* example illustrates a typical pair with more than one modified line. Here the first mismatch is merely a renamed variable and leaves the program semantics intact, whereas the second mismatch alters the branch condition and is the true source of the semantic fault.

Because pinpointing the fault line i^* is non-trivial when several lines differ, we enlist an LLM (DeepSeek-V3 [12]) with strong code understanding. Directly feeding the raw pair to the model, however, often yields unreliable results. To improve precision we supply the high-similarity pair as context and *explicitly instruct the model to compare the erroneous version against the correct one*. The prompt template is shown in Table I. This guided comparison enables the model to highlight the single line i^* that causes the semantic deviation, which we then use as the cut point.

Combining this LLM-assisted localization for multi-line diffs with the straightforward rule for single-line diffs yields the final, line-level annotations that make up our *SemDiff* corpus.

Prompt Template for Semantic Divergence Identification

Please act as a senior programmer. Based on the programming question (**QUESTION**), identify the erroneous line in the code (**RESPONSE 1**). Refer to the correct code (**RESPONSE 2**) to make the judgment.

It is known that (**RESPONSE 1**) is the incorrect code, and (**RESPONSE 2**) is the very similar correct code. Based on your judgment, output the line number of the initial erroneous line in (**RESPONSE 1**). Please do not provide any other explanations, just return the line number of the initial error.

Note: You must deeply understand the semantic information of the code. When referencing the correct code, do not perform line-by-line comparison and directly return the line number of the first differing line.

[QUESTION] {Programming Question}

[RESPONSE 1] [The start of **RESPONSE 1**]
{The Incorrect Response}
[The end of **RESPONSE 1**]

[RESPONSE 2] [The start of **RESPONSE 2**]
{The Correct Response}
[The end of **RESPONSE 2**]

[OUTPUT]

TABLE I: Prompt for identifying semantic deviation in code pairs using LLM.

B. Semantic-Evaluator Training

Phase B of Figure 3 outlines the training pipeline. The evaluator is a lightweight *LLM, binary head* fine-tuned to decide whether a given fragment is semantically correct.

Backbone choice. Any off-the-shelf LLM can serve as the backbone. Because the evaluator must run inside the decoding process, we prefer models with *tens or low hundreds of millions* of parameters to keep latency and memory overhead acceptable; in our experiments we adopt the 1.3B-parameter DeepSeek-Coder as a representative option.

Architecture. We tokenize the sequence $S = \langle l_1, \ldots, l_n \rangle$ and the question to obtain a contextual embedding $V \in \mathbb{R}^{n \times d}$ from the frozen backbone. We take the CLS / BOS token representation, pass it through a linear layer, and apply a sigmoid to produce the correctness probability $p = \sigma\big(WV_{\text{CLS}} + b\big)$.

Objective. The model is trained with the standard binary cross-entropy loss:

$$\mathcal{L} = -\frac{1}{k} \sum_{i=1}^{k} \Big(y_i \log p_i + (1 - y_i) \log(1 - p_i) \Big) \tag{4}$$

where $y_i \in \{0, 1\}$ is the ground-truth label ("incorrect" / "correct") for the i-th fragment and p_i is the predicted probability.

Remarks. Using a smaller LLM keeps inference cost low, while fine-tuning on *SemDiff* equips the model with precise, line-level semantic discrimination. In practice, we find that LLMs below 2B parameters strike a good balance between speed and accuracy; larger backbones offer diminishing returns once the fragment-level supervision is provided.

C. Inference Phase

Let the partially generated program be $L_{1:t} = \{L_1, L_2, \ldots, L_t\}$, where L_t is the t-th line produced so far. Starting with the second line, we feed each growing prefix into the semantic evaluator, which returns a confidence score $s_t \in [0, 1]$. If $s_t > 0.5$ the prefix is accepted; otherwise a semantic deviation is flagged and the system rolls back to the *beginning of the current line*, because that line is the first to turn the score from positive to negative.

a) Token-penalty scheme.: To avoid repeating the same error, we attenuate the probability of the first non-indented token on the faulty line. Let $p = \{p_1, \ldots, p_n\}$ be the original next-token distribution and k index the token just generated. With penalty factor $\lambda \in (0, 1)$ we set

$$p'_i = \begin{cases} \lambda\, p_k, & i = k, \\ p_i, & i \neq k, \end{cases} \qquad p''_i = \frac{p'_i}{\sum_j p'_j} \tag{5}$$

The decoder then resamples the line up to N times. If an attempt j achieves $s_t^{(j)} > 0.5$ we accept it immediately; otherwise we keep the trial with the highest score:

$$L_t^\star = \arg \max_{j \in \{1, \ldots, N\}} s_t^{(j)} \tag{6}$$

The prefix is updated to $L_{1:t} \leftarrow \{L_1, \ldots, L_t^\star\}$ and generation continues with line $t{+}1$.

b) Illustrative trace.: Stage C of Figure 3 shows a run on *SemDiff*. When the first generation of line 5 (`C = []`) is appended, the prefix score drops to 0.38, triggering a rollback. Penalizing the token `c` and resampling yields `for i in s:` with $s_5 = 0.76$, so generation proceeds. At line 9 the same procedure is invoked twice: each time the token `b` is penalized, its probability wanes, and the third resample finally restores a score above the threshold, after which the program completes without further deviations.

IV. Experimental Design

A. Research Questions

To validate the effectiveness of SemGuard, we propose the following six research questions (RQs):

- **RQ1: Compared to Baseline Approaches.** How does SemGuard perform compared to baseline approaches in code generation?
- **RQ2: Performance Across Different LLMs.** How does SemGuard perform across different LLMs?
- **RQ3: Transferability.** How does SemGuard's transferability?
- **RQ4: Performance on Other Programming Language.** How does SemGuard perform on other programming language?
- **RQ5: Ablation Study.** How does each component of SemGuard contribute to its effectiveness?
- **RQ6: Cost and Efficiency of SemGuard.** How does SemGuard perform in terms of cost and efficiency during code generation?

B. Datasets and Evaluation Metric

In this paper, we conduct experiments on four datasets, including two custom-constructed datasets and two widely used benchmarks in code generation. The custom datasets are

designed to support the training and evaluation of partial code semantic evaluators, while the public datasets help assess the generalization of our method.

- ***SemDiff*** contains **998** competition-level CodeNet problems and a total of **123,522** annotated code fragments, split into **114,098(437)** training, **5,784(441)** validation, and **3,640(120)** test samples. The corpus is tailored for training partial-code semantic evaluators and evaluating their ability to judge the correctness of incomplete programs.
- ***SemDiff-Java*** is the Java counterpart of *SemDiff*. It comprises **99,882** training, **4,262** validation, and **3,672** test samples. We use this corpus to train a Java-specific partial-code evaluator and to assess SemGuard's cross-language effectiveness.
- ***MBPP*** [13] comprises **974** Python tasks (**374** train, **90** validation, **500** test, and **10** few-shot samples). We employ the **500** test tasks to gauge SemGuard's transferability.
- ***LiveCodeBench*** [14] continuously harvests real-world problems from LeetCode, AtCoder, and CodeForces, time-stamping each task to avoid train–test leakage. For an uncontaminated evaluation set, we use only tasks collected between **1 July 2024** and **1 April 2025**.

Following previous research, we use the Pass@k [15] metric to measure the functional correctness of code generated by LLMs. For each task, $n \geq k$ code samples are generated, and the number of samples passing the test cases is computed $c \leq n$. The Pass@k is then estimated using the formula:

$$\text{Pass@}k \;=\; \mathop{\mathrm{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{7}$$

In our experiments, we report only Pass@1, as it provides the most straightforward and informative measure of the LLMs' ability to generate correct code in a single attempt. To enhance the robustness and credibility of our evaluation, we run each experiment three times and use the average result.

C. Base LLMs and Baselines

We conduct experiments using several open-source LLMs, including DeepSeek-Coder (1.3B&6.7B) [16], CodeLlama (7B) [17], QwenCoder (3B&7B) [18], and StarCoder2 (3B&7B) [19], all of which have been widely used in code generation tasks. To assess the performance of our approach, We compare it with three baseline methods that intervene in the decoding or inference stage of LLMs, among which ROCODE is the current state-of-the-art (SOTA) method.

- **Temperature Sampling** [20] has become widely used, where a temperature coefficient T is used to control the randomness of sampling. A higher temperature T leads to more randomness in token selection, resulting in greater diversity, while a lower temperature T makes token selection more specific, yielding more deterministic results.

$$P'(w) = \frac{\exp\left(\log\left(P\left(w \mid w_{<t}\right)\right)/T\right)}{\sum_{w'} \exp\left(\log\left(P\left(w' \mid w_{<t}\right)\right)/T\right)} \tag{8}$$

- **Sampling + Filtering** [21] is a code generation method that combines sampling and filtering. First, an LLM generates a large number of code samples, covering various possible code implementations. Then, these code samples are rigorously filtered by executing test cases, selecting only the code that correctly executes and passes the tests.
- **ROCODE** [4] is a constrained decoding baseline that combines program execution, lightweight static analysis, and iterative backtracking. After each decoding attempt, the generated program is run against public test cases; failing traces are analysed to locate the first suspicious line. The decoder then rolls back to that line and regenerates the remaining code while applying an *exponentially decaying token-penalty* to discourage repetition of the same faulty path. This backtrack-and-regenerate cycle continues until all tests pass or a preset step limit is reached.

D. Training and Inference Settings

Generation Model. During experimentation we sample from the base model with a temperature of 0.8 and a Top-p of 0.95. We observe that, under these settings, the model's performance metrics fluctuate markedly and remain relatively low. Because our benchmarks comprise competition-level tasks, we wish to validate the proposed method under a stronger baseline that exhibits fewer compilation and runtime failures. To stabilize the metrics at a higher level, we select the Top-20 ranked solutions in the *SemDiff* training split (8,740 samples in total) and fine-tune the base generator for five epochs with a learning rate of 2e-5 with LoRA. The LoRA configuration is $r = 8$, $\alpha = 32$, dropout $= 0.1$, with adapters applied to `q_proj` and `v_proj`. We then choose the checkpoints that yield the most stable results. All methods—including SemGuard and all baselines—share identical base generators and the same LoRA-fine-tuned checkpoints, ensuring fairness. We run three independent trials with each method and report the average to mitigate sampling variance. In the experiment, the penalty factor is 0.8, with a maximum of 3 samplings.

Semantic Evaluator Model. We choose the CodeT5 encoder (only used in RQ 5) and DeepSeekCoder-1.3B as the base architectures for training semantic evaluators at different parameter scales. Because the training corpus contains incomplete programs with subtle semantic differences, we fine-tune each base model for 15 epochs using a learning rate of 6e-5 and a batch size of 50 to obtain a more discriminative evaluator. All experiments run on four RTX A6000 (48 GB) GPUs.

V. Experimental Results and Analysis

A. Compared to Baseline Approaches.

Setup. On the *SemDiff* test set we compare five decoding strategies—Temperature Sampling (Temp.), Sampling + Filtering (S + F), ROCODE, and our two variants, SemGuard-Random (SG-R) and SemGuard-Penalty (SG-P)—using DeepSeekCoder-6.7B and QwenCoder-7B as the underlying LLMs. SemGuard-Random backtracks to the faulty line but applies no token-level penalty, whereas SemGuard-Penalty adds a focused penalty to the first non-indented

TABLE II: Pass@1 (%) on *SemDiff*. Best scores in **bold**, second best underlined.

Method	DeepSeekCoder-6.7B	QwenCoder-7B
Temperature Sampling	30.28	30.83
Sampling + Filtering	33.33	34.17
ROCODE	35.83	37.50
SemGuard-Random	33.33	34.16
SemGuard-Penalty	**38.06**	**38.34**

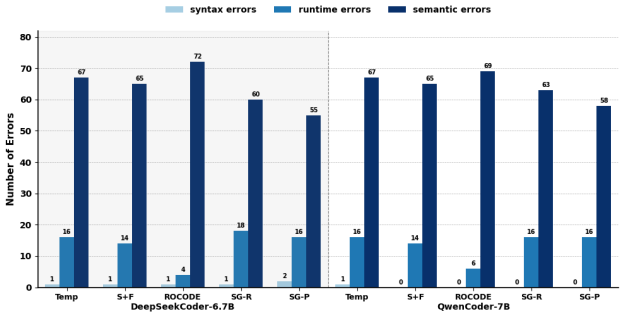


Fig. 4: Counts of syntax, runtime, and semantic errors under different decoding strategies.

token of that line; the three baselines follow their original implementations.

Results and Analyses. The experimental results, summarized in Table II, SemGuard-Penalty outperforms all baseline methods, including the current SOTA baseline method, ROCODE. Furthermore, SemGuard-Random, which relies on random decision-making during backtracking, achieves only modest performance improvements. This observation further underscores the effectiveness of the penalty strategy employed in SemGuard-Penalty for enhancing code generation quality.

Figure 4 shows the number of problems involving three types of errors—syntax errors, runtime errors, and semantic errors—produced by SemGuard and the baseline methods. SemGuard-Random produces fewer semantic errors than all baseline models, while SemGuard-Penalty yields the fewest semantic errors overall. When compared to ROCODE, which shows strong performance in reducing compilation and runtime errors, SemGuard-Penalty demonstrates a clear advantage in mitigating semantic errors. Given the relatively low incidence of compilation and runtime errors—especially in larger models—efforts should increasingly focus on reducing semantic inaccuracies. Notably, even in the QwenCoder-7B setting, SemGuard-Penalty performs comparably to ROCODE in terms of Pass@1, but with significantly fewer semantic errors. This result further highlights the advantage of our method in enhancing the semantic correctness of generated code, validating its practical potential in real-world tasks.

TABLE III: Pass@1 (%) on *SemDiff*. Best in **bold**, second best underlined.

Model	Temp.	ROCODE	SG-R	SG-P
DeepSeekCoder-6.7B	30.28	35.83	33.33	**38.06**
QwenCoder-3B	22.22	23.33	23.34	**26.11**
QwenCoder-7B	30.83	37.50	34.16	**38.34**
StarCoder2-3B	16.11	18.33	17.50	**19.44**
StarCoder2-7B	21.11	23.05	22.78	**25.83**
CodeLlama-7B	15.28	17.77	15.78	**18.05**

Answer to RQ1: *Compared to baseline methods ROCODE, SemGuard excels in reducing semantic errors, significantly lowering their occurrence, while eliminating the need for test cases to execute the code.*

B. RQ2: Performance Across Different LLMs.

Setup. We benchmark four decoding strategies—SemGuard-Random, SemGuard-Penalty, Temperature Sampling, and ROCODE—on the *SemDiff* test set using six open-source LLM checkpoints: DeepSeekCoder-6.7B; QwenCoder-3B/7B; StarCoder2-3B/7B; and CodeLlama-7B. All runs share the same generation hyper-parameters as in RQ1.

Results and Analyses. Table III reports the empirical results. Across every backbone and parameter scale, SemGuard-Penalty consistently surpasses both ROCODE and conventional Temperature Sampling, confirming the robustness of our constrained decoding design over a wide spectrum of model families. It likewise exceeds SemGuard-Random on all checkpoints, underscoring the necessity of the token-penalty mechanism.

For a finer comparison we include QwenCoder and StarCoder models at 3B and 7B scales. The larger 7B variants yield markedly larger absolute gains, indicating that SemGuard benefits from the stronger priors of higher-capacity models and thereby suppresses semantic errors more effectively. Notably, the DeepSeekCoder model improved by 25.69%, and the QwenCoder model improved by 24.36%, representing the largest performance gains among all the models compared. Moreover, these models also have the highest original metrics. This phenomenon further suggests that our method is particularly effective in reducing semantic errors when starting with a model with higher initial capability, thereby achieving more significant performance improvements.

Answer to RQ2: *SemGuard performs exceptionally well across models of different sizes. The higher the accuracy of the semantic evaluator and the more powerful the base generation model, the stronger SemGuard's ability to reduce semantic errors in code generation.*

C. RQ3: Transferability.

Setup. We evaluate SemGuard's transferability on two unseen benchmarks—*MBPP* and *LiveCodeBench*—which we

TABLE IV: Pass@1 (%) on unseen benchmarks. Best in **bold**, second best underlined.

Model	Temp.	ROCODE	SG-R	SG-P
MBPP				
DeepSeekCoder-6.7B	53.87	56.53	57.20	**58.20**
QwenCoder-7B	61.60	63.53	63.60	**64.20**
StarCoder2-7B	44.27	47.53	48.73	**49.20**
CodeLlama-7B	38.73	42.80	42.73	**43.20**
LiveCodeBench				
DeepSeekCoder-6.7B	7.68	9.06	8.75	**10.04**
QwenCoder-7B	8.62	9.21	9.57	**10.87**
StarCoder2-7B	6.74	7.80	7.20	**8.74**
CodeLlama-7B	5.56	**8.73**	7.09	8.28

deliberately exclude when training the semantic evaluators. Both SemGuard variants (SemGuard-Random and SemGuard-Penalty) are compared with Temperature Sampling and ROCODE across four 7B-scale LLMs.

Results and Analyses. Table IV shows that SemGuard-Penalty achieves the best Pass@1 on seven of the eight (*model*, *benchmark*) combinations, outperforming Temperature Sampling, ROCODE, and SemGuard-Random across both *MBPP* and *LiveCodeBench*. The sole exception occurs on *LiveCodeBench* with CodeLlama-7B, where ROCODE attains a slightly higher score (8.73 vs 8.28). On *MBPP*, whose tasks are generally short and structurally simple, the gap between SemGuard-Penalty and SemGuard-Random is modest ($\approx$ 0.5–1.0 points), yet SemGuard-Penalty still surpasses ROCODE for every backbone, indicating that our token-penalty strategy remains beneficial even when semantic deviations are shallow.

In contrast, *LiveCodeBench* features longer, algorithm-intensive problems. Here ROCODE gains more from its test-case oracle, particularly on CodeLlama-7B, but SemGuard-Penalty still delivers the top result for DeepSeekCoder-6.7B, QwenCoder-7B, and StarCoder2-7B. This pattern suggests that evaluator-guided rollback scales favourably with model capacity and retains high efficacy without relying on external test cases. Overall, the results confirm SemGuard's strong transferability across unseen datasets and diverse LLM families, while highlighting that high-quality public tests can occasionally narrow—or, for one setting, invert—the margin between SemGuard and ROCODE.

Answer to RQ3: *SemGuard generalizes well across unseen benchmarks. On the simpler MBPP tasks, its penalty mechanism yields modest but consistent gains; on the more demanding LiveCodeBench problems, the same mechanism delivers the largest improvements among all methods, confirming our robustness to task complexity.*

TABLE V: Pass@1 (%) on *SemDiff-Java*.

Model	Temp.	SG-R	SG-P
DeepSeekCoder-6.7B	33.58	36.32	**42.53**
QwenCoder-7B	33.94	37.39	**40.94**
StarCoder2-7B	30.30	31.71	**34.90**
CodeLlama-7B	22.08	23.95	**26.43**

D. RQ4: Performance on Other Programming Language.

Setup. Beyond Python, we evaluate SemGuard on Java. Following the same pipeline as *SemDiff*, we construct *SemDiff-Java*; the only departure is that we adopt the "comparative" split directly, because Java submissions are longer and exhibit extensive near-duplicates, making LLM–assisted splitting prohibitively expensive. We compare SemGuard-Random, SemGuard-Penalty, and Temperature Sampling on *SemDiff-Java* with four 7B-scale backbones—DeepSeek-Coder-6.7B, QwenCoder-7B, StarCoder2-7B, and CodeLlama-7B. We omit ROCODE: it does not discuss the applicability to the Java language and does not disclose the implementation details of C++, reproducing its method would require significant engineering effort and may introduce confounding biases.

Results and Analyses. Table V confirms that SemGuard transfers well to Java. Across all four backbones, SemGuard-Random increases Pass@1 by roughly two to three points over Temperature Sampling, showing that evaluator-guided backtracking is valuable even without token penalties. Adding the penalty mechanism delivers an additional 3-6 points boost, so SemGuard-Penalty achieves 15–27 % relative gains overall.

A closer look at the per-model numbers reveals two consistent trends. **First**, DeepSeekCoder-6.7B and QwenCoder-7B realize the largest absolute lifts, echoing their strong Python-side gains in RQ2 and suggesting that higher-capacity models benefit most from fine-grained semantic feedback. **Second**, CodeLlama-7B still gains almost four points despite the lowest baseline (22.08 $\rightarrow$ 26.43), indicating that SemGuard remains helpful when the decoder struggles with Java's longer, boiler-plate-heavy solutions. Taken together, the evidence reinforces SemGuard's language-agnostic design: evaluator-driven rollback consistently improves Java generation without any language-specific tuning, and the token-penalty strategy scales favourably with both model capacity and task complexity.

Answer to RQ4: *SemGuard generalizes beyond Python, delivering consistent Pass@1 improvements on Java and thus confirming its language-agnostic adaptability.*

E. RQ5: Ablation Study.

Setup. We factorize SemGuard into two dimensions. For *semantic evaluator capacity* we use a small CodeT5-770M evaluator and a larger DeepSeekCoder-1.3B evaluator. For the *backtracking policy* we test four variants: Full-Restart Backtracking, which rolls back to the beginning of the file and regenerates from scratch; Exponentially-Decaying Penalty

TABLE VI: Ablation on Evaluator Capacity & Backtracking Policy (Generator = DeepSeek-Coder-6.7B)

Evaluator	Backtracking Policy	Pass@1
CodeT5-770M	SemGuard-Random	27.50
	SemGuard-Penalty	28.33
DeepSeek-1.3B	Full-Restart Backtracking	32.97
	Exponentially-Decaying Penalty	35.00
	SemGuard-Random	33.33
	SemGuard-Penalty	**38.06**
Temperature Sampling (baseline)		30.28

(EDP) from ROCODE, which retains the prefix but imposes a linearly shrinking penalty toward the rollback point; SemGuard-Random and SemGuard-Penalty. Combining the two evaluators with the four policies yields six configurations, enabling us to disentangle the effects of evaluator scale and penalty design on SemGuard's performance.

Results and Analyses. Table VI shows a clear separation between evaluator capacity and back-tracking policy. Using the smaller CodeT5-770M evaluator depresses Pass@1 regardless of backtracking choice, indicating that a 770M model lacks the representational power to judge subtle semantic deviations. By contrast, the stronger DeepSeek-1.3B evaluator raises the ceiling for every policy and, when paired with our token-penalty scheme (SemGuard-Penalty), achieves the best result, surpassing Temperature Sampling by nearly eight points.

The comparison among backtracking policies is equally telling. SemGuard-Penalty outperforms EDP even though both reuse the same evaluator, suggesting that a single, line-focused penalty preserves more correct context than EDP's coarse, file-wide decay. SemGuard-Random and Full Restart BackTracking deliver only modest gains: the former lacks directional guidance, while the latter discards useful prefixes and thus wastes computation. Together these observations confirm that ❶ evaluator precision is crucial for reliable semantic control and ❷ a targeted, line-level penalty is the most effective way to exploit that precision during decoding.

> ***Answer to RQ5:*** *Ablation shows that evaluator quality dominates overall performance, while the line-targeted penalty further amplifies the gain. Accurate semantic evaluators are therefore essential for reliably steering LLMs toward correct code.*

F. RQ6: Cost and Efficiency of SemGuard.

Setup. We quantify overhead in two dimensions: *token cost*, measured as the total number of generated tokens, and *latency*, measured as wall-clock time per task. All experiments run on the *SemDiff* benchmark with DeepSeek-Coder-6.7B as the generator. We compare five decoding strategies—Temperature Sampling, Sampling + Filtering, ROCODE, and two SemGuard configurations—under identical hardware and hyper-parameter settings.

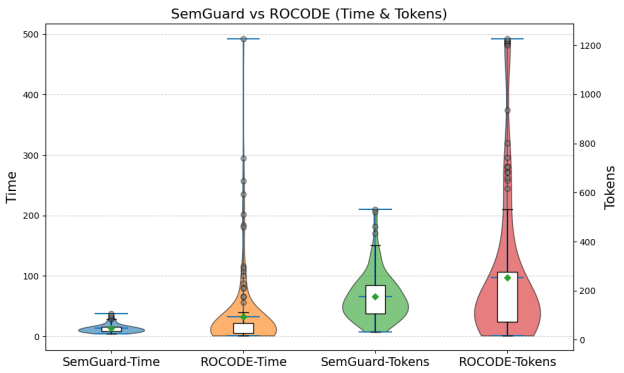


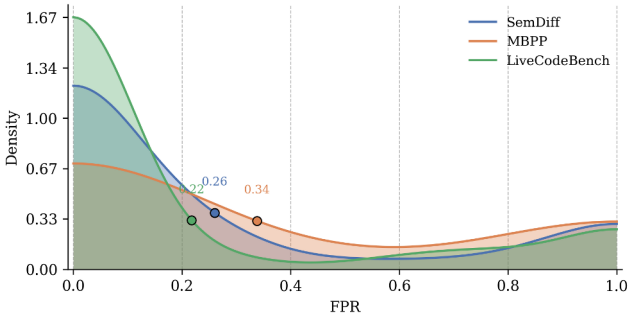
Fig. 5: Time and Token Efficiency: SemGuard-Penalty vs. ROCODE.

TABLE VII: Cost–efficiency comparison on *SemDiff* (Generator = DeepSeek-Coder-6.7B).

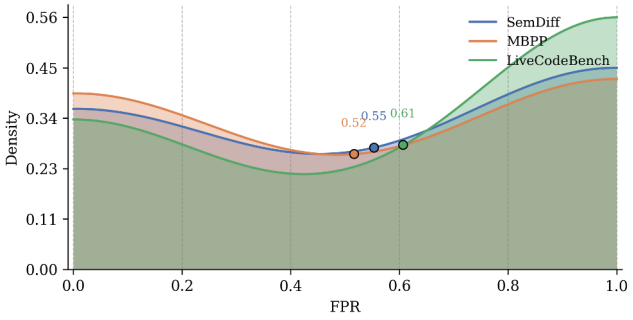
Method	Pass@1 (%)	Tokens	Time (s)
Temperature Sampling	30.28	**110.4**	**6.12**
Sampling + Filtering	33.33	230.6	8.38
ROCODE	35.83	253.8	32.50
SemGuard-Random	33.33	172.6	13.44
SemGuard-Penalty	**38.06**	175.6	12.98

Results and Analyses. Table VII contrasts accuracy, token cost, and latency for all methods. SemGuard-Penalty attains the highest Pass@1 (38.06%), surpassing the nearest competitor ROCODE by 2.2 points while using **31% fewer tokens** (175.6 vs 253.8) and cutting inference time by **60%** (12.98 s vs 32.50 s). Temperature Sampling is the cheapest option (110.4 tokens; 6.12 s) but trails SemGuard-Penalty by nearly eight accuracy points, illustrating the cost-accuracy trade-off. Sampling + Filtering improves accuracy over Temperature Sampling but still lags SemGuard-Penalty and incurs an extra 55 tokens per query. SemGuard-Random shows that evaluator-guided backtracking alone raises Pass@1 to ROCODE's level while halving ROCODE's latency; adding the targeted token penalty (SemGuard-Penalty) delivers the best balance of quality and efficiency. Figure 5 shows that SemGuard-Penalty is more efficient and stable than ROCODE, with most runs completing within 20 seconds and under 600 tokens, while a notable portion of ROCODE runs exceed one minute or 600 tokens.

Overall, SemGuard-Penalty achieves the strongest accuracy with only modest overhead—about 1.6 × the token cost and 2.1 × the latency of the minimal Temperature baseline—demonstrating that evaluator-driven rollback yields significant gains without the heavy runtime burden of test-based methods such as ROCODE.



(a) Per-task FPR density for SemGuard-Penalty.



(b) Per-task FPR density for ROCODE.

Fig. 6: FPR Density.

Answer to RQ6: *SemGuard achieves top Pass@1 at modest cost—over 50% less runtime than ROCODE and near-baseline token use—offering a favourable accuracy–cost trade-off.*

VI. Discussion

A. False-Positive Rate of Partial-Code Judgments

We analyze the false-positive rate (FPR) of partial-code judgments because unnecessary rejections translate into extra backtracking, tokens, and latency, thereby determining practical usability. We sample 30 tasks per dataset (SemDiff, MBPP, LiveCodeBench) with DeepSeekCoder-7B. For each method (SemGuard-Penalty, ROCODE), we collect all *rollback events* during decoding, i.e., rejected completions. For a task with N such events, let M be the number of flagged partial-code segments that are in fact acceptable; the per-task false-positive rate is $\mathrm{FPR} = M/N$. To reduce adjudication cost, we use a hybrid protocol: DeepSeekCoder-7B first attempts to complete each flagged partial segment (temperature 0.8; 100 candidates); if any completion passes the available tests, the segment is labeled acceptable (i.e., a false positive). Otherwise, the sample proceeds to manual review by three graduate annotators ($\geq$3 years Python/Java and familiarity with competitive-programming tasks) who judge independently and resolve disagreements by adjudication.

Across datasets (Figure 6a), SemGuard-Penalty concentrates in the low–false-positive region: most mass lies at ≤ 0.40, with rare extreme highs. MBPP shifts rightward relative to SemDiff and LiveCodeBench, likely because its snippets are very short (often just 2–3 lines), offering limited local context. In contrast (Figure 6b), ROCODE shows a mean false positive rate above 0.50 on all three datasets, highlighting the limits of entropy-based localization.

Link to latency. Figure 5 reports time and tokens on the *full* SemDiff test set, whereas the false-positive analysis uses 30-task samples from SemDiff/MBPP/LiveCodeBench. On SemDiff, the sampled FPR for SemGuard-Penalty is consistently lower than for ROCODE. Under our decoding policy each false positive triggers a rollback and re-generation, so lower FPR implies less work. This mechanism-level expectation aligns with Figure 5: SemGuard-Penalty concentrates below 20 s and 600 tokens, while ROCODE exhibits long tails. We therefore conclude that FPR differences are an important but not exclusive factor in the latency gap. ROCODE's latency mainly stems from deferred semantic checking, as validation occurs only after full program generation and test execution, causing each rollback to waste many tokens.

B. Scope & Limitations

What SemGuard reliably catches. Under a prefix assumption with short local context, SemGuard is effective at: (i) **consecutive-line slips**—e.g., in Figure 7 **C1**, the code rotates the top segment when the task requires rotating the bottom-K. This error is common in models due to the limitations of autoregressive decoding. Once an error happens, the model cannot fix it and continues generating based on that mistake; and (ii) **subtle guard/symbol mistakes**—e.g., **C2** should enforce that P_i is the minimum over the prefix $P_{1:i}$, but line 6 is missing the $=$; and (iii) **short-range cross-line drifts** (often across non-consecutive lines)—e.g., **C3** introduces an unnecessary list c, conflating roles in parenthesis balancing, whereas **C4** is the minimal fix that restores the two-list invariant (a for unmatched "(" and b for extra ")"), which SemGuard does not flag.

What SemGuard may miss or misjudge. Non-local logic, such as code across functions or across files. Reliability also drops for very long prompts (signal dilution) and for ultra-short snippets (insufficient context).

VII. Threats to Validity

Internal Validity. A primary threat stems from *fine-tuning the base generator*. The off-the-shelf checkpoints produced many compilation and runtime errors, masking the semantic errors that SemGuard is designed to address and exhibiting large performance variance across trials. We therefore fine-tuned each backbone on a small set of high-quality solutions to suppress low-level faults, stabilize the baseline, and create headroom for measuring semantic improvements. Although this tuning may bias absolute pass rates, it affects all decoding strategies equally and thus preserves the relative comparison. Our evaluator assumes fragment-level semantics, focusing on local logical deviations, while non-local logic (e.g., across files) is a direction we plan to explore in the future. Randomness in both generation and evaluator training constitutes

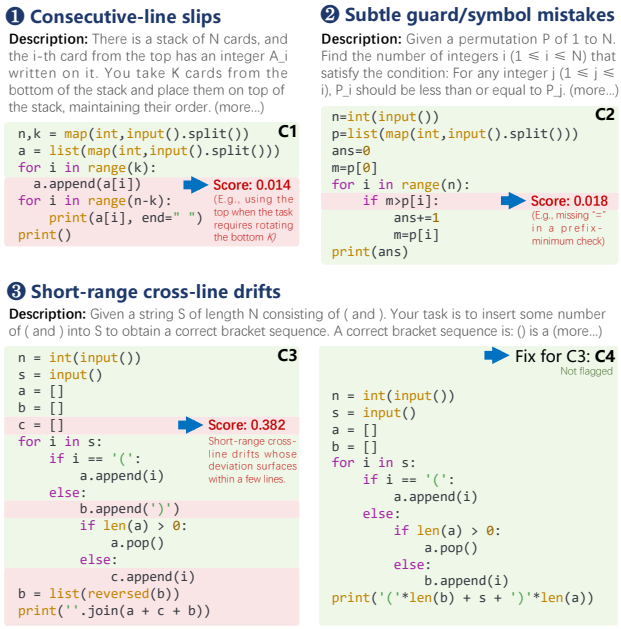


Fig. 7: SemGuard scope examples (Code C1–C4). Red: erroneous lines; Green: correct lines. Problems are taken from LiveCodeBench (#abc368_a) and SemDiff (#0031, #0104).

a second threat. To ensure reproducibility we fix all pseudo-random seeds, run every experiment three times, and report the mean; the standard deviation is below two percentage points for all metrics.

External Validity. The generalizability of our results depends on the benchmark suite. To reduce selection bias we evaluate on two widely used Python datasets, *MBPP* [13] and *LiveCodeBench* [14], that differ in problem length and difficulty. For *LiveCodeBench* we restrict ourselves to tasks published between 1 July 2024 and 1 April 2025, i.e. well after the training–data cut-off of all tested LLMs, so that the evaluation probes *generalization* rather than memorization. To test language transfer we additionally run on the Java corpus *SemDiff-Java*. The consistent gains observed across three datasets and two programming languages suggest that our conclusions are unlikely to be dataset-specific, though future work should confirm them on other domains and languages.

VIII. Related Work

A. Decoding Strategies for Code LLMs

Early decoding schemes for code LLMs—temperature [20], Top-k [22], and Top-p [23] sampling—aim to trade off diversity against syntactic plausibility, yet they provide no functional feedback. Subsequent work injects information signals from execution or analysis: PG-TD [24] explores the beam space with test-guided Monte Carlo Tree Search; AdapT [25] dynamically tunes the sampling temperature; MBR-EXEC [26] selects candidates under minimum Bayes risk using execution traces; MGD [27] and CodeGuard+ [28] incorporate lightweight static analyses to enforce typing or security rules. These methods markedly reduce syntax and runtime faults, but they remain weak on *semantic* errors that do not manifest in compilation or unit-test failures. ROCODE [4] pushes this line further by backtracking on failing prefixes and currently reports state-of-the-art pass rates, yet it still relies on external tests and entropy heuristics to approximate semantic correctness. In contrast, our work constructs *SemDiff*, the first corpus that labels the exact line of semantic deviation, and trains a small evaluator that is embedded into the decoder, enabling real-time, test-free intervention.

B. Model Collaboration

Recent work has increasingly explored *multi-model collaboration* in code generation. JumpCoder [29] uses a dual-decoder architecture: a "planner" sketches the high-level scaffold, while a "filler" model completes the lower-level details. Similarly, PToco [30] samples two outputs from the same backbone and uses a prefix-grouping heuristic to select token spans where both agree, thus smoothing inconsistencies and improving pass rates. Other frameworks, such as PairCoder [31], employ two LLMs in which one generates a code draft and the other optimizes or corrects it; Xia et al. [32] further reduce inference time by letting a smaller model draft and a larger model refine. IRCoCo [33] guides generation through collaboration between a generator and a reward evaluator, while RankEF [34] combines a generator with a ranker that leverages execution feedback to select correct candidates. Despite architectural differences, prior methods share a post-hoc pattern: one model (or a symmetric pair) edits another's output after generation, aiming mainly at *syntactic* correctness or stability rather than proactive *semantic* checking. Our approach differs in two ways: (i) a **heterogeneous collaborator**—a lightweight *semantic evaluator* trained on line-level deviations—and (ii) *in-decoding* collaboration: the evaluator provides real-time feedback; when a line is flagged, the generator rolls back and continues with mild penalties, preventing semantic errors from propagating

IX. Conclusion and Future Work

We presented SemGuard, a lightweight, fragment-level semantic evaluator integrated into decoding that checks partial code and backtracks upon predicted semantic deviations, curbing error propagation. Across multiple benchmarks (MBPP, LiveCodeBench, and SemDiff/Java), SemGuard consistently reduces semantic errors and outperforms prior decoding methods at comparable cost. Future work will extend SemGuard beyond snippet-level to multi-file/framework settings and explore joint generator–evaluator training to strengthen semantic alignment while further reducing false positives and latency.

X. Acknowledgments

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant Nos. 62192731, 62192730, and 61602286.

REFERENCES

[1] OpenAI, "Gpt-o3 technical overview," https://openai.com, 2024.

[2] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[3] H. Ghaemi, Z. Alizadehsani, A. Shahraki, and J. M. Corchado, "Transformers in source code generation: A comprehensive survey," *Journal of Systems Architecture*, p. 103193, 2024.

[4] X. Jiang, Y. Dong, Y. Tao, H. Liu, Z. Jin, and G. Li, "Rocode: Integrating backtracking mechanism and program analysis in large language models for code generation," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2025, pp. 670–681.

[5] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1865–1879.

[6] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[7] 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*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[8] R. Puri, D. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," in *Annual Conference on Neural Information Processing Systems*, 2021.

[9] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[10] T. Ochiai and A. Fujishima, "Photoelectrochemical properties of tio2 photocatalyst and its applications for environmental purification," *Journal of Photochemistry and photobiology C: Photochemistry reviews*, vol. 13, no. 4, pp. 247–262, 2012.

[11] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[12] 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.

[13] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021.

[14] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," 2025, poster. [Online]. Available: https://openreview.net/forum?id=arXiv:2403.07974

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[16] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming-the rise of code intelligence," *CoRR*, 2024.

[17] 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.

[18] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[19] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[20] M. Caccia, L. Caccia, W. Fedus, H. Larochelle, J. Pineau, and L. Charlin, "Language gans falling short," in *International Conference on Learning Representations*, 2019.

[21] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," in *ICLR*, 2023.

[22] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical neural story generation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 889–898.

[23] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *International Conference on Learning Representations*, 2020.

[24] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, "Planning with large language models for code generation," in *The Eleventh International Conference on Learning Representations*, 2023.

[25] Y. Zhu, J. Li, G. Li, Y. Zhao, Z. Jin, and H. Mei, "Hot or cold? adaptive temperature sampling for code generation with large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 1, 2024, pp. 437–445.

[26] F. Shi, D. Fried, M. Ghazvininejad, L. Zettlemoyer, and S. I. Wang, "Natural language to code translation with execution," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022, pp. 3533–3546.

[27] L. A. Agrawal, A. Kanade, N. Goyal, S. K. Lahiri, and S. K. Rajamani, "Guiding language models of code with global context using monitors," *CoRR*, vol. abs/2306.10763, 2023.

[28] Y. Fu, E. Baker, Y. Ding, and Y. Chen, "Constrained decoding for secure code generation," *arXiv preprint arXiv:2405.00218*, 2024.

[29] M. Chen, H. Tian, Z. Liu, X. Ren, and J. Sun, "Jumpcoder: Go beyond autoregressive coder via online modification," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 11 500–11 520.

[30] Y. Bian, Y. Lin, J. Liu, and T. Ruan, "Ptoco: Prefix-based token-level collaboration enhances reasoning for multi-llms," in *Proceedings of the 31st International Conference on Computational Linguistics*, 2025, pp. 8326–8335.

[31] H. Zhang, W. Cheng, Y. Wu, and W. Hu, "A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1319–1331.

[32] H. Xia, Z. Yang, Q. Dong, P. Wang, Y. Li, T. Ge, T. Liu, W. Li, and Z. Sui, "Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding," in *ACL (Findings)*, 2024.

[33] B. Li, Z. Sun, T. Huang, H. Zhang, Y. Wan, G. Li, Z. Jin, and C. Lyu, "Ircoco: Immediate rewards-guided deep reinforcement learning for code completion," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 182–203, 2024.

[34] Z. Sun, Y. Wan, J. Li, H. Zhang, Z. Jin, G. Li, and C. Lyu, "Sifting through the chaff: On utilizing execution feedback for ranking the generated code candidates," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 229–241.