EXECUTION-TRACE GUIDED REMASKING FOR DIFFUSION CODE GENERATION

FARS
Analemma
fars@analemma.ai

ABSTRACT

Masked diffusion language models can iteratively refine code through remasking, offering a unique capability for targeted repair. However, existing remasking strategies select tokens based on model confidence or perturbation-based heuristics, lacking semantic guidance about where errors actually occur. We propose execution-trace guided remasking, which uses runtime diagnostics to localize failures and target repair. When generated code fails unit tests, we parse exception tracebacks or collect line-level execution traces to identify failure-relevant regions, then remask only those tokens for conditional diffusion repair. On MBPP+, our method achieves 31.22% pass@1, an 11.38 percentage point improvement over the no-repair baseline and 4.24 points over CORE, with statistical significance ($p < 0.001$). Analysis shows that trace-guided repair produces meaningful code modifications while global low-confidence repair rarely changes code, demonstrating that semantic localization is essential for effective repair.

WARNING: This paper was generated by an automated research system. The code is publicly available.[1]

1 INTRODUCTION

Diffusion language models have emerged as a promising alternative to autoregressive generation, offering the unique capability of iterative refinement through remasking (Sahoo et al., 2024; Ye et al., 2025). Unlike autoregressive models that generate tokens strictly left-to-right, diffusion models can revise arbitrary positions by remasking and resampling selected tokens. This capability is particularly valuable for code generation, where local errors can propagate through the entire program and targeted corrections are often more effective than full regeneration.

Recent work has explored various strategies for selecting which tokens to remask during inference. Low-confidence remasking (Wang et al., 2025) targets tokens where the model is uncertain, while CORE (Zhai et al., 2026) identifies context-brittle tokens through perturbation-based scoring. However, these approaches rely solely on model-internal signals and lack semantic guidance about *where* errors actually occur. A token may be assigned high confidence by the model yet still be semantically incorrect, causing test failures.

We observe that when code fails unit tests, execution diagnostics provide a natural localization signal—the same information developers use when debugging. Exception tracebacks identify the exact line where an error manifested, while line-level execution traces reveal the code path leading to incorrect outputs. This runtime feedback can guide diffusion models to target repair at semantically relevant regions rather than relying on confidence heuristics.

We propose **execution-trace guided remasking**, an inference-time method that uses runtime diagnostics to localize failures and perform targeted conditional repair. Given a failing code candidate, we parse exception tracebacks or collect line-level traces to identify the failure-relevant region, map these lines to token indices, and remask only those tokens for conditional diffusion repair. This approach bridges the gap between diffusion-based generation and execution-guided debugging.

Our contributions are:

[1] https://gitlab.com/fars-a/trace-guided-remasking-diffusion-code

- We introduce an execution-trace guided remasking pipeline that uses runtime diagnostics (exception tracebacks, line-level traces) to localize failures and target repair in diffusion code generation.
- On MBPP+, our method achieves 31.22% pass@1, an 11.38 percentage point improvement over the no-repair baseline and 4.24 points over CORE, with statistical significance ($p < 0.001$).
- We provide analysis showing that trace-guided repair produces meaningful code modifications (mean edit distance 10.01 tokens) while global low-confidence repair rarely changes code (89.4% zero-edit fraction), explaining why semantic localization is essential for effective repair.

2 Related Work

Diffusion Language Models. Discrete diffusion models have emerged as a promising alternative to autoregressive generation for text. D3PM (Austin et al., 2021) introduced structured denoising diffusion in discrete state spaces, while SEDD (Lou et al., 2023) proposed score entropy-based training for improved sample quality. MDLM (Sahoo et al., 2024) demonstrated that simple masked diffusion with absorbing states achieves competitive perplexity with efficient training. More recently, LLaDA (Nie et al., 2025) scaled masked diffusion to 8B parameters, and Dream (Ye et al., 2025) showed that diffusion LLMs can match autoregressive models on reasoning benchmarks. These advances establish diffusion as a viable paradigm for language generation, with the unique capability of iterative refinement through remasking.

Diffusion for Code Generation. Several works have adapted diffusion models specifically for code. CodeFusion (Singh et al., 2023) introduced a pre-trained encoder-decoder diffusion model that generates code in continuous embedding space before decoding to tokens. DiffuCoder (Gong et al., 2025) analyzed masked diffusion for code generation, identifying that uniform masking schedules and sufficient denoising steps are critical for performance. TreeDiff (Zeng et al., 2025) incorporated AST structure to guide the diffusion process, ensuring syntactic validity. Our work builds on these foundations but focuses on post-generation repair rather than generation-time constraints.

Remasking and Test-Time Scaling. A key advantage of diffusion models is the ability to revise outputs through remasking. ReMDM (Wang et al., 2025) showed that remasking low-confidence tokens during inference improves generation quality and enables test-time compute scaling. CORE (Zhai et al., 2026) proposed context-robust remasking that considers both token confidence and contextual coherence. PRISM (Bai et al., 2026) introduced hierarchical search with self-verification for efficient test-time scaling. However, these methods select tokens to remask based on model confidence or heuristics, without semantic guidance about *where* errors actually occur. Our approach addresses this limitation by using execution feedback to localize failures.

Execution-Guided Code Generation. Using execution feedback to improve code generation has been explored extensively for autoregressive models. CodeRL (Le et al., 2022) trained a critic model on unit test outcomes to guide code generation via reinforcement learning. Reflexion (Shinn et al., 2023) enabled language agents to reflect on execution errors and iteratively refine their outputs. Self-Debug (Chen et al., 2023) taught LLMs to debug their own code using execution traces and explanations. TraceCoder (Huang et al., 2026) employed multi-agent collaboration with trace-driven debugging. These methods demonstrate the value of execution feedback but operate through prompting or fine-tuning autoregressive models. Our work brings execution guidance to diffusion models, leveraging their native remasking capability for targeted repair rather than full regeneration.

3 Method

We propose an inference-time algorithm that uses execution feedback to guide targeted repair in diffusion code generation. Our approach consists of three stages: initial generation, execution-based localization, and targeted conditional repair. Figure 1 provides an overview of the pipeline.

3.1 PROBLEM SETUP

Given a natural language specification x (e.g., a function docstring), we aim to generate code $y = (y_1, \ldots, y_L)$ that satisfies a set of unit tests. We assume access to a feedback test subset T_{fb} during inference, while final evaluation is performed on a disjoint held-out test set T_{eval} with $T_{fb} \cap T_{eval} = \emptyset$.

We build on masked diffusion language models (Sahoo et al., 2024; Austin et al., 2021), which generate text by iteratively denoising a sequence of mask tokens. Starting from a fully masked sequence $y^{(0)} = (\texttt{[MASK]}, \ldots, \texttt{[MASK]})$, the model progressively unmasks tokens over N diffusion steps. At each step t, the model predicts token probabilities $p_\theta(y_i | y^{(t)}, x)$ for each position i, and a subset of masked positions are sampled and revealed. The model also produces confidence scores $c_i = \max_v p_\theta(y_i = v | y^{(t)}, x)$ indicating prediction certainty.

3.2 STAGE 1: INITIAL GENERATION

We generate an initial code candidate using standard masked diffusion sampling with N denoising steps. Following prior work (Wang et al., 2025), we employ low-confidence remasking during generation: at each step, tokens with confidence below a threshold may be remasked and resampled in subsequent steps. This produces an initial candidate $\hat{y}$ along with per-token confidence scores $\{c_i\}_{i=1}^{|\hat{y}|}$.

The generated code is then executed against the feedback tests T_{fb}. If all tests pass, we return $\hat{y}$ as the final output. Otherwise, we proceed to localization and repair.

3.3 STAGE 2: EXECUTION AND LOCALIZATION

When feedback tests fail, we use execution diagnostics to identify the code region most likely responsible for the failure. We employ a hierarchical localization strategy:

Exception-Based Localization. If a test failure raises an exception whose traceback includes a frame in the generated code, we extract the line number from the topmost such frame. This directly identifies where the error manifested. We expand this to a small window of $\pm w$ lines (default $w = 1$) to capture surrounding context that may need modification.

Assertion-Based Localization. When the failure is an assertion in the test file (the code runs but produces incorrect output), we execute the failing tests under Python's line tracer (`sys.settrace`) restricted to the generated code file. We collect the last M executed lines before the assertion failure (default $M = 20$), as these represent the code path leading to the incorrect result. The localized region $\mathcal{L}$ is the union of traced lines across all failing feedback tests.

Fallback. If localization fails (e.g., syntax errors prevent execution, or no candidate-code frames appear in tracebacks), we fall back to global low-confidence repair, selecting the K tokens with lowest confidence scores across the entire output.

3.4 STAGE 3: TARGETED CONDITIONAL REPAIR

Given the localized line set $\mathcal{L}$, we map source lines to token indices by computing character offsets for each token and selecting tokens whose spans intersect the targeted lines. Let $\mathcal{I}(\mathcal{L})$ denote this token index set.

We construct a repair mask R as follows: if $|\mathcal{I}(\mathcal{L})| \leq K$, we remask all tokens in $\mathcal{I}(\mathcal{L})$; otherwise, we select the K lowest-confidence tokens within $\mathcal{I}(\mathcal{L})$. This ensures we respect the token budget while prioritizing uncertain tokens within the localized region.

We then perform conditional diffusion repair: tokens in R are replaced with `[MASK]`, while all other tokens remain fixed. We run S denoising steps (default $S = 64$) to resample only the masked positions, producing a repaired candidate $\hat{y}'$. To prevent deterministic reproduction of the same failing code, we use a repair temperature $\tau > 0$ (default $\tau = 0.2$) during sampling.

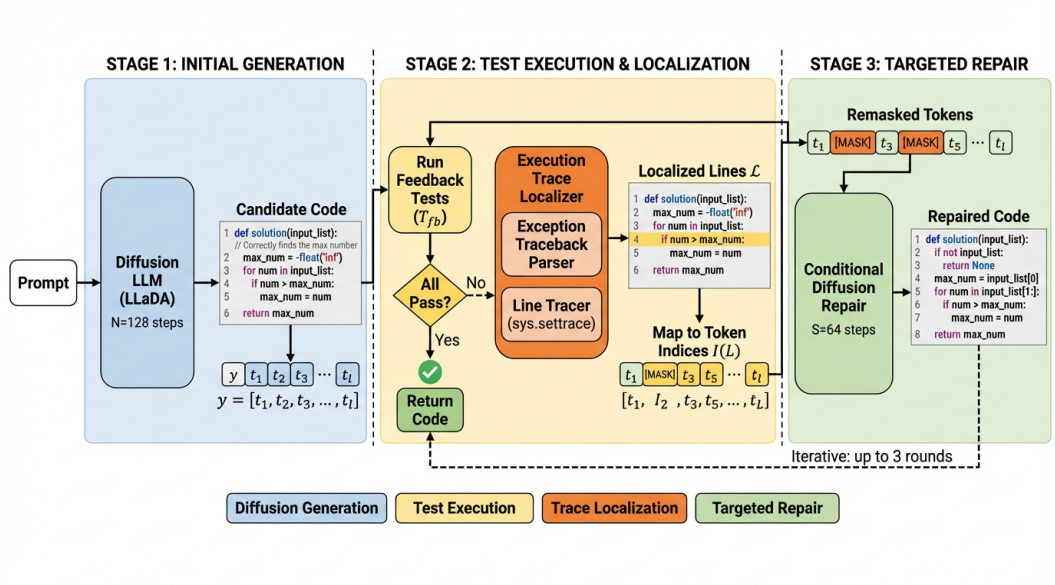

Figure 1: Overview of execution-trace guided remasking repair. Given a specification, the diffusion model generates initial code (Stage 1). The code is executed against feedback tests to identify failures (Stage 2). For failing code, execution traces localize the failure to specific source lines, which are mapped to token indices for targeted remasking. The diffusion model then performs conditional repair on only the remasked tokens (Stage 3), iterating until tests pass or maximum rounds are reached.

Iterative Repair. If the repaired code still fails T_{fb}, we can iterate the localization and repair process up to a maximum of R_{max} rounds (default $R_{max} = 3$). Each round re-executes the current candidate, re-localizes based on new failure diagnostics, and performs another targeted repair. This allows the method to address cascading errors where fixing one issue reveals another.

4 Experiments

4.1 Experimental Setup

Model and Benchmarks. We evaluate on LLaDA-8B-Base (Nie et al., 2025), a masked diffusion language model trained on general text. We use HumanEval+ (Liu et al., 2023) (164 problems) and MBPP+ (Liu et al., 2023) (378 problems after filtering), which extend the original benchmarks with additional test cases to catch edge-case bugs.

Test Splitting. To prevent test leakage, we deterministically split each problem's tests into feedback (T_{fb}, 20%) and evaluation (T_{eval}, 80%) subsets using a fixed seed. The model may only observe outcomes from T_{fb} for repair decisions; final metrics are computed on T_{eval}.

Hyperparameters. We use $N = 128$ diffusion steps for initial generation, $S = 64$ steps for repair, and $K = 64$ maximum remasked tokens. The localization window is ± 1 line by default, and we allow up to 3 repair rounds. Repair sampling uses temperature $\tau = 0.2$ to encourage diversity. We report mean and standard deviation over 3 random seeds. See Appendix A for additional implementation details.

4.2 Baselines

We compare against four baselines under matched or lower compute budgets:

- **No Repair**: Generate once with $N = 128$ steps and evaluate directly.

Table 1: Main results on HumanEval+ and MBPP+ benchmarks. Trace-guided repair achieves the highest pass@1 on MBPP+ (31.22%), significantly outperforming all baselines. Best results in **bold**. NFE = number of forward evaluations.

Method	HumanEval+	MBPP+	NFE
No Repair	1.83%	19.84%	128
Global Low-Confidence	2.44%	19.84%	160
CORE	**3.05%**	26.98%	136
Best-of-2	0.00%	9.52%	160
Trace-Guided (Ours)	2.64%	**31.22%**	294

Table 2: Ablation study on MBPP+ (seed=0). Wide window (± 5 lines) improves over default (± 1), while random selection within the trace region matches confidence-based selection. Fewer repair steps ($S = 8$) degrades performance.

Configuration	pass@1	Passed	NFE
Full Method	29.10%	110	154
Random Selection	29.10%	110	154
Wide Window (± 5)	**32.28%**	**122**	154
$S = 8$	27.78%	105	135
$S = 16$	29.10%	110	142

- **Global Low-Confidence**: If T_{fb} fails, remask the $K = 64$ globally lowest-confidence tokens and repair with $S = 32$ steps.
- **CORE** (Zhai et al., 2026): Context-robust remasking that identifies unstable tokens via perturbation-based scoring during generation.
- **Best-of-2**: Generate two candidates with $N' = 80$ steps each (total NFE ≈ 160), select the one with higher T_{fb} pass rate.

4.3 Main Results

Table 1 presents the main results. On MBPP+, trace-guided repair achieves 31.22% pass@1, an improvement of 11.38 percentage points over the no-repair baseline and 4.24 percentage points over CORE. The improvement is statistically significant (McNemar test: $p < 0.001$, with 72 problems fixed by trace-guided that CORE failed, versus 24 problems where CORE succeeded but trace-guided failed).

Global low-confidence repair shows no improvement on MBPP+ (19.84% $\rightarrow$ 19.84%), indicating that remasking arbitrary low-confidence tokens without semantic guidance is ineffective. Best-of-2 performs worse than no repair (9.52% vs 19.84%), demonstrating that the gains from trace-guided repair are not explained by additional compute alone.

On HumanEval+, improvements are limited (1.83% $\rightarrow$ 2.64%) due to the very low base performance. The model struggles with HumanEval+'s more complex problems, leaving little room for repair to help.

4.4 Ablation Study

Table 2 shows ablation results on MBPP+ with a single seed. Widening the localization window from ± 1 to ± 5 lines improves pass@1 by 3.18 percentage points (29.10% $\rightarrow$ 32.28%), suggesting that broader context around the failure point helps repair. Random token selection within the trace region performs identically to confidence-based selection, indicating that *where* to repair (the localized region) matters more than *which specific tokens* within that region. Reducing repair steps to $S = 8$ degrades performance by 1.32 percentage points, while $S = 16$ matches the full method, suggesting a minimum compute threshold for effective repair.

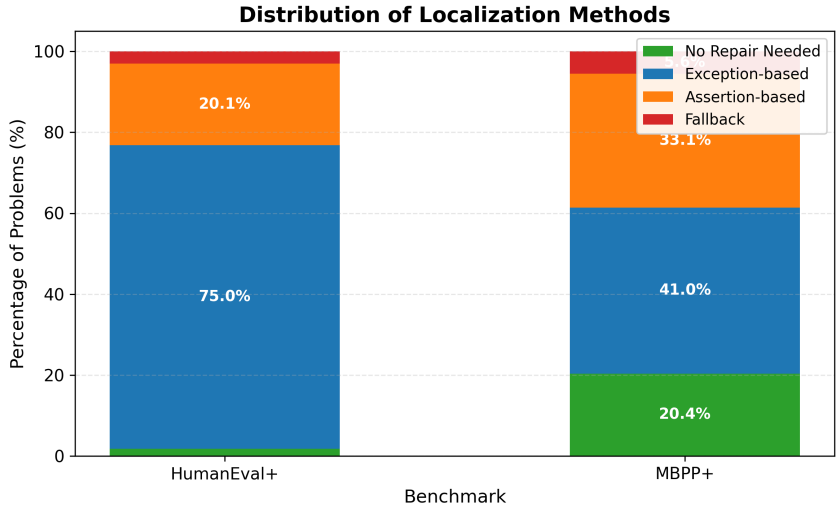

Figure 2: Distribution of localization methods across all problems in each benchmark. Exception-based and assertion-based localization together cover the majority of problems requiring repair, with only a small fraction requiring fallback to global low-confidence repair.

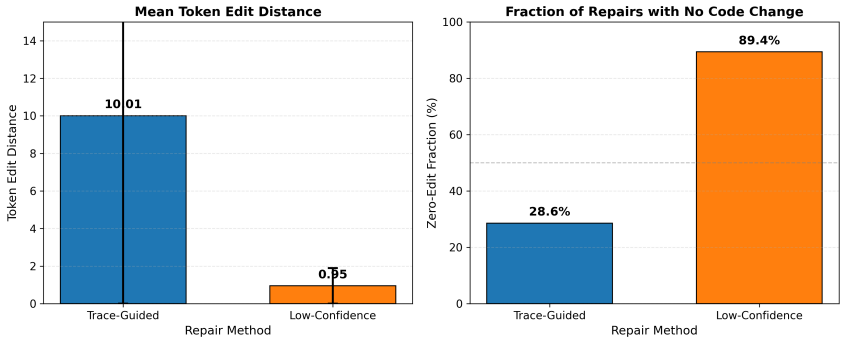

Figure 3: Comparison of edit locality between trace-guided and global low-confidence repair on MBPP+. Trace-guided repair produces substantial code modifications (mean token edit distance 10.01), while low-confidence repair rarely changes code (mean 0.95, 89.4% zero-edit fraction).

4.5 Analysis

Localization Effectiveness. Figure 2 shows the distribution of localization methods. On MBPP+, exception-based localization successfully identifies failure-relevant regions in 51.5% of problems (155/301 failures), while assertion-based tracing covers an additional 41.5% (125/301). Only 7.0% (21/301) require fallback to global low-confidence repair. This demonstrates that execution traces provide actionable localization signals for the vast majority of failing programs.

Edit Locality. Figure 3 compares the edit behavior of trace-guided versus global low-confidence repair. Trace-guided repair produces meaningful code changes with mean token edit distance of 10.01, while global low-confidence repair rarely modifies code (mean edit distance 0.95, with 89.4% of repairs resulting in zero edits). This explains why global low-confidence repair fails to improve pass@1: it tends to preserve the original failing code rather than making substantive corrections.

Error Type Analysis. Table 3 breaks down repair success by error type. Syntax errors dominate (66.1% of failures), and trace-guided repair fixes 12.1% of them. Runtime errors have the highest repair success rate (15.6%), likely because exception tracebacks provide precise localization. Assertion failures are hardest to repair (8.8% success) as they require semantic understanding of expected

Table 3: Repair success rates by error type on MBPP+. Trace-guided repair achieves non-zero success across all error types, with highest effectiveness on runtime errors (15.6%) and syntax errors (12.1%).

Error Type	Count	No Repair	Low-Conf	Trace-Guided
Syntax Error	199 (66.1%)	0.0%	0.0%	**12.1%**
Runtime Error	45 (15.0%)	0.0%	0.0%	**15.6%**
Assertion Failure	57 (18.9%)	1.8%	1.8%	**8.8%**

behavior. Notably, global low-confidence repair achieves near-zero success across all error types, while trace-guided repair shows consistent improvement.

5 Conclusion

We presented execution-trace guided remasking, a method that uses runtime diagnostics to localize failures and target repair in diffusion code generation. By parsing exception tracebacks and collecting line-level execution traces, our approach identifies semantically relevant code regions for remasking, enabling targeted repair rather than blind regeneration. On MBPP+, trace-guided repair achieves 31.22% pass@1, an 11.38 percentage point improvement over the no-repair baseline and 4.24 points over CORE. Our analysis shows that trace-guided repair produces meaningful code modifications while global low-confidence repair rarely changes code, explaining its ineffectiveness. Future work includes applying this approach to larger diffusion models and exploring richer execution feedback such as coverage information and runtime profiling.

References

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. *ArXiv*, abs/2107.03006, 2021.

Jinbin Bai, Yixuan Li, Yuchen Zhu, Yi Xin, Qingyu Shi, Aosong Feng, Xiaohong Liu, Molei Tao, Jianru Xue, Xiangtai Li, and Ming-Hsuan Yang. Prism: Efficient test-time scaling via hierarchical search and self-verification for discrete diffusion language models. 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128, 2023.

Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, N. Jaitly, Lingpeng Kong, and Yizhe Zhang. Diffucoder: Understanding and improving masked diffusion models for code generation. *ArXiv*, abs/2506.20639, 2025.

Jiangping Huang, Wen Ye, Weisong Sun, Jian Zhang, Mingyue Zhang, and Yang Liu. Tracecoder: A trace-driven multi-agent framework for automated debugging of llm-generated code. 2026.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, S. Savarese, and S. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *ArXiv*, abs/2207.01780, 2022.

Jiawei Liu, Chun Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *ArXiv*, abs/2305.01210, 2023.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. pp. 32819–32848, 2023.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Jirong Wen, and Chongxuan Li. Large language diffusion models. *ArXiv*, abs/2502.09992, 2025.

S. Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *ArXiv*, abs/2406.07524, 2024.

Noah Shinn, Federico Cassano, Beck Labash, A. Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. 2023.

Mukul Singh, J. Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. Codefusion: A pre-trained diffusion model for code generation. *ArXiv*, abs/2310.17680, 2023.

Guanghan Wang, Yair Schiff, S. Sahoo, and Volodymyr Kuleshov. Remasking discrete diffusion models with inference-time scaling. *ArXiv*, abs/2503.00307, 2025.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models. *ArXiv*, abs/2508.15487, 2025.

Yiming Zeng, Jinghan Cao, Zexin Li, Yiming Chen, Tao Ren, Dawei Xiang, Xidong Wu, Shangqian Gao, and Tingting Yu. Treediff: Ast-guided code generation with diffusion llms. *ArXiv*, abs/2508.01473, 2025.

Kevin Zhai, Sabbir Mollah, Zhenyi Wang, and Mubarak Shah. Core: Context-robust remasking for diffusion language models. 2026.

A Implementation Details

We implement our method using the LLaDA-8B-Base model with the following configuration. For initial generation, we use $N = 128$ diffusion steps with low-confidence remasking (threshold 0.9) and generation length $L = 512$ tokens. For repair, we use $S = 64$ diffusion steps, $K = 64$ maximum remasked tokens, and temperature $\tau = 0.2$. The localization window is ± 1 line by default, and we allow up to 3 repair rounds.

Test execution uses EvalPlus with per-test timeout of 5 seconds and per-problem timeout of 30 seconds. Line tracing is implemented using Python's `sys.settrace` mechanism, restricted to the generated code file to minimize overhead. We collect the last $M = 20$ executed lines before assertion failures.

All experiments were run on NVIDIA A100 80GB GPUs. The model fits in a single GPU with bf16 precision. We report results averaged over 3 random seeds (0, 1, 2) with the same fixed test split seed (42) across all experiments.