Progress-Guarded LAVE: Lexer-Ignored Stall Filtering for Reliable CFG-Constrained Diffusion Decoding

FARS
Analemma
fars@analemma.ai

Abstract

Diffusion language models with context-free grammar (CFG) constraints enable reliable structured generation, but can get stuck in *stall loops* where the model repeatedly generates grammar-ignored tokens (whitespace, comments) until hitting the maximum length. We propose Progress-Guarded LAVE (PG-LAVE), which extends LAVE's lookahead verification with a lightweight stall guard that monitors lexer progress—the production of non-ignored terminals—and triggers early rejection when progress stalls. On CPP-Bench with LLaDA-8B-Instruct, PG-LAVE achieves a 40% relative reduction in truncation failures (1.83% vs 3.05% hit_max_len_rate) while improving syntactic validity by +0.81 percentage points (88.41% vs 87.60%). Our ablation study shows that lexer-progress tracking outperforms simple whitespace rejection heuristics, and sensitivity analysis confirms robustness to the stall threshold for $H \geq 16$.

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

1 Introduction

Diffusion language models (dLLMs) have emerged as a promising alternative to autoregressive models, offering parallel token generation and natural support for constrained decoding (Sahoo et al., 2024; Nie et al., 2025; Ye et al., 2025). Recent advances have scaled dLLMs to competitive performance with autoregressive models on reasoning and code generation tasks (Arriola et al., 2025). A key advantage of dLLMs is their ability to enforce structural constraints during generation, enabling reliable production of outputs that conform to context-free grammars (CFGs) such as programming language syntax.

LAVE (Zhang et al., 2026) provides reliable CFG-constrained decoding for dLLMs through lookahead verification: each proposed token is accepted only if sampled completions yield CFG-extendable prefixes. While LAVE substantially improves syntactic validity over unconstrained decoding, it suffers from a specific failure mode: the model can get stuck in *stall loops* where it repeatedly generates tokens that are valid under the grammar but ignored by the lexer—whitespace, newlines, and comments. These grammar-ignored tokens consume the generation length budget without producing meaningful code, eventually causing truncation when the maximum length is reached.

Simple solutions prove ineffective. Prompt engineering (e.g., "avoid repeated blank lines") actually degrades performance, as dLLMs do not reliably follow natural-language formatting instructions. Resampling (Best-of-N) can eliminate stall failures but at multiplicative compute cost. We need an efficient, algorithmic solution that targets the root cause: the grammar's extendability check accepts length-wasting updates because ignored tokens remain technically valid.

We propose **Progress-Guarded LAVE (PG-LAVE)**, which adds a lightweight stall guard that monitors *lexer progress*—the production of non-ignored terminals—and triggers early rejection when

[1] `https://gitlab.com/fars-a/reliable-cfg-decoding-diffusion`

progress stalls. The key insight is that the grammar's lexer already distinguishes meaningful tokens from ignored ones; we leverage this signal to detect and prevent stall loops without modifying LAVE's core verification mechanism.

Our contributions are:

- We identify and formalize the **stall loop failure mode** in CFG-constrained diffusion decoding, where grammar-ignored tokens cause truncation failures.
- We propose **PG-LAVE**, a progress-aware early termination mechanism that monitors lexer progress and rejects ignored-only proposals when progress has stalled.
- We demonstrate that PG-LAVE achieves a **40% relative reduction in truncation failures** (1.83% vs 3.05% hit_max_len_rate) while improving syntactic validity by +0.81 percentage points on CPP-Bench with LLaDA-8B-Instruct.

2 Related Work

Diffusion Language Models. Discrete diffusion models have emerged as a promising alternative to autoregressive language models, offering parallel generation capabilities and natural support for constrained decoding. MDLM (Sahoo et al., 2024) demonstrates that simple masked diffusion with modern training recipes can approach autoregressive perplexity on language modeling benchmarks. LLaDA (Nie et al., 2025) scales masked diffusion to 8B parameters, achieving competitive performance with autoregressive models on reasoning tasks. Dream (Ye et al., 2025) further advances diffusion LLMs with improved training objectives and sampling strategies. Block Diffusion (Arriola et al., 2025) interpolates between autoregressive and diffusion paradigms, enabling flexible generation strategies. MaskGIT (Chang et al., 2022) pioneered masked generative transformers for images, inspiring subsequent work on discrete diffusion for text.

Constrained Decoding for Autoregressive Models. Grammar-constrained generation has been extensively studied for autoregressive LLMs. PICARD (Scholak et al., 2021) introduces incremental parsing for SQL generation, rejecting invalid tokens during beam search. Outlines (Willard & Louf, 2023) provides efficient guided generation using finite-state machines compiled from regular expressions and context-free grammars. LMQL (Beurer-Kellner et al., 2022) offers a query language for specifying constraints declaratively. Grammar-Aligned Decoding (Park et al., 2024) addresses the distribution shift caused by token masking in constrained decoding. These methods rely on the sequential nature of autoregressive generation and cannot be directly applied to diffusion models.

CFG-Constrained Diffusion Decoding. Recent work has adapted constrained decoding to diffusion LLMs. LAVE (Zhang et al., 2026) introduces lookahead-based verification that leverages parallel token predictions to efficiently validate proposed tokens against context-free grammars. DINGO (Suresh et al., 2025) proposes a dynamic programming approach for distribution-preserving constrained decoding with regular expressions. IG-CD (Mündler et al., 2025) applies ignore-grammar constraints that allow intermediate violations. Rainbow Padding (Kim et al., 2025) addresses early termination in instruction-tuned diffusion LLMs through padding strategies. Our work complements these approaches by addressing a specific failure mode—stall loops in grammar-ignored tokens—that persists even with reliable constraint enforcement.

Repetition and Degeneration in Neural Text Generation. Text degeneration, including repetitive loops, is a well-documented phenomenon in neural language models (Holtzman et al., 2019). Xu et al. (2022) analyze repetition patterns and propose training-time mitigations. Dong et al. (2025) study repetition specifically in code generation, finding that it correlates with model uncertainty. While these works focus on autoregressive models, we observe analogous stall behavior in diffusion LLMs when generating grammar-ignored tokens such as whitespace and comments.

3 METHOD

We present Progress-Guarded LAVE (PG-LAVE), which extends LAVE (Zhang et al., 2026) with a lightweight stall guard that monitors lexer progress and triggers early termination when the model repeatedly generates grammar-ignored tokens without making meaningful progress.

3.1 BACKGROUND: LAVE DECODING

Diffusion language models generate text by iteratively unmasking a fixed-length canvas of L tokens over T denoising steps. At each step, the model predicts token distributions for all masked positions in parallel and selects a subset to unmask. LAVE (Zhang et al., 2026) provides reliable CFG-constrained decoding by verifying each proposed token update against a context-free grammar G.

Given a prompt $\mathbf{p}$ and a partially decoded sequence $\mathbf{y}$ containing `[MASK]` tokens, LAVE operates as follows: when the model proposes a token t^* for a masked position i^*, LAVE performs *lookahead verification* by sampling N completions of the remaining masks and checking whether any completion yields a CFG-extendable prefix via an Earley parser. If verification succeeds, the token is accepted and a verified witness string $\mathbf{y}_{\text{cache}}$ is cached; otherwise, the proposal is rejected and the model resamples. After τ consecutive failures, LAVE invokes cache-enhanced recovery using $\mathbf{y}_{\text{cache}}$.

3.2 PROBLEM: STALL LOOPS IN GRAMMAR-IGNORED TOKENS

While LAVE ensures that accepted tokens maintain CFG extendability, it does not prevent the model from repeatedly generating tokens that are *valid but ignored* by the grammar's lexer. Many practical grammars include ignore rules for whitespace and comments (e.g., `%ignore WS` in Lark syntax). Under such grammars, arbitrarily long sequences of whitespace or comment tokens remain CFG-extendable, allowing the model to consume the entire length budget L without producing meaningful code.

We define *lexer progress* as the production of non-ignored terminals. Let $\text{Lex}_G(\cdot)$ denote the grammar's lexer with ignore rules applied. For a decoded string $\mathbf{y}$, the progress count is $g(\mathbf{y}) = |\text{Lex}_G(\mathbf{y})|$, the number of non-ignored terminals. A *stall loop* occurs when the model generates a sequence of accepted tokens that do not increase g—the output grows in length but not in meaningful content.

3.3 PG-LAVE: PROGRESS-GUARDED DECODING

PG-LAVE adds a stall guard that monitors lexer progress and rejects proposals that would extend a stall. The key insight is that LAVE already maintains a verified witness string $\mathbf{y}_{\text{cache}}$, which we can use to track progress without additional forward passes.

Progress Tracking. We maintain a stall counter that tracks consecutive accepted steps where lexer progress does not increase. After each accepted token, we compute $g(\mathbf{y}_{\text{cache}})$ and compare it to the previous value. If g increases, we reset the stall counter to zero; otherwise, we increment it.

Ignored-Only Detection. A proposed token t^* is classified as *ignored-only* if its decoded string $s = \text{decode}(t^*)$ matches the grammar's ignore patterns. In practice, we use the conservative proxy $s \in$ `/^\s+$/` (whitespace-only regex).

Stall Guard. Before running LAVE's lookahead verification for a proposal t^*: if t^* is ignored-only *and* the stall counter exceeds threshold H, we reject immediately and resample. This consumes one of the τ retry attempts. If all τ attempts are exhausted, we fall back to standard LAVE behavior (accept the last verified proposal) to avoid deadlock.

Algorithm. Figure 1 illustrates the PG-LAVE algorithm. The stall guard (green region) intercepts proposals before LAVE's verification step. For each proposal, we decode the token, check if it is ignored-only, and if so, verify that the stall counter is below threshold H. Only proposals that pass

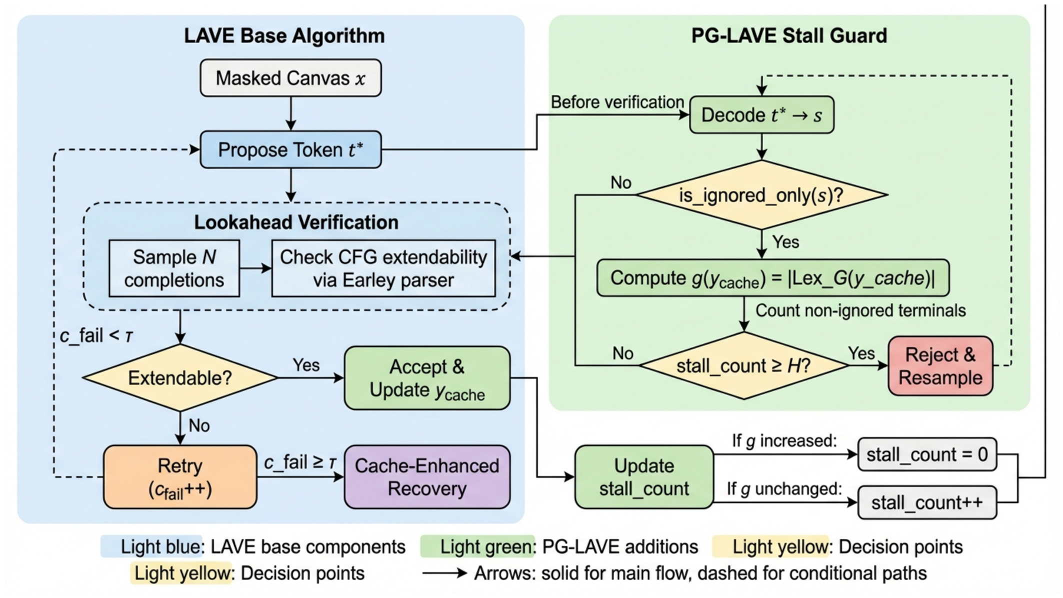

Figure 1: Overview of PG-LAVE. **Left**: LAVE base algorithm with lookahead verification and cache-enhanced recovery. **Right**: PG-LAVE stall guard that monitors lexer progress $g(\mathbf{y}_{\text{cache}})$ and rejects ignored-only proposals when progress has stalled for H consecutive steps.

this guard proceed to LAVE's lookahead verification. After acceptance, we update the stall counter based on whether lexer progress increased.

Hyperparameters. PG-LAVE introduces one hyperparameter: the stall threshold H. We use $H = 16$ as the default, which allows short sequences of whitespace (e.g., indentation) while preventing extended stall loops. Our sensitivity analysis (Section 4.4) shows that PG-LAVE is robust to H for $H \geq 16$.

4 Experiments

We evaluate PG-LAVE on code generation to assess its effectiveness in reducing truncation failures while maintaining syntactic validity.

4.1 Experimental Setup

Dataset. We use CPP-Bench (HumanEval-CPP) (Zheng et al., 2023), a benchmark of 164 C++ code generation tasks with unit tests. Each task provides a function signature and docstring, and the model must generate a syntactically valid and functionally correct implementation.

Model. We use LLaDA-8B-Instruct (Nie et al., 2025), an 8-billion parameter diffusion language model trained for instruction following. We use the same decoding hyperparameters as LAVE (Zhang et al., 2026): maximum length $L = 256$, diffusion steps $T = 128$, temperature 0.2, block size 32, lookahead size $N = 10$, and attempt budget $\tau = 10$.

Metrics. We report four metrics: (1) **Syntactic@1**: percentage of outputs that parse successfully under the C++ grammar; (2) **Functional@1**: percentage of outputs that pass all unit tests; (3) **Hit Max Len**: percentage of outputs that reach the maximum length L (lower is better, as this indicates truncation); (4) **Time**: average inference time per instance in seconds.

Baselines. We compare against: (1) **NO-CD**: unconstrained decoding without grammar constraints; (2) **IG-CD** (Mündler et al., 2025): ignore-grammar constrained decoding; (3)

Table 1: Main results on CPP-Bench (HumanEval-CPP, 164 tasks) with LLaDA-8B-Instruct. Best in **bold**, second-best <u>underlined</u>. PG-LAVE achieves 40% relative reduction in truncation failures while improving syntactic validity. †Values from LAVE paper.

Method	Syntactic@1 (%)	Functional@1 (%)	Hit Max Len (%) ↓	Time (s)
NO-CD[†]	74.20	–	–	8.55
IG-CD[†]	86.10	–	–	9.66
LAVE	87.60±0.28	<u>18.29</u>±2.17	3.05±0.50	**5.13**±0.26
Naive Cap (K=32)	<u>87.60</u>±0.28	18.49±2.01	<u>3.25</u>±0.76	<u>5.36</u>±0.28
PG-LAVE (H=16)	**88.41**±0.50	**18.90**±1.73	**1.83**±0.00	8.40±0.26

LAVE (Zhang et al., 2026): lookahead-verify constrained decoding; (4) **Naive Cap**: a control baseline that rejects whitespace-only proposals after $K = 32$ consecutive accepts, without lexer-progress tracking. NO-CD and IG-CD results are from the LAVE paper; LAVE, Naive Cap, and PG-LAVE are our reproductions with 3 seeds (42, 123, 456).

4.2 Main Results

Table 1 presents the main results. PG-LAVE achieves the best syntactic validity (88.41%) and the lowest truncation rate (1.83%), representing a 40% relative reduction in hit_max_len_rate compared to LAVE (1.83% vs 3.05%). The syntactic@1 improvement of +0.81 percentage points is consistent across all three seeds. PG-LAVE also achieves the highest functional@1 (18.90%), though the difference from LAVE is within variance.

Compared to the citable baselines from the LAVE paper, PG-LAVE substantially outperforms both NO-CD (88.41% vs 74.20% syntactic@1) and IG-CD (88.41% vs 86.10%). Notably, PG-LAVE is faster than both NO-CD (8.40s vs 8.55s) and IG-CD (8.40s vs 9.66s) despite adding the stall guard. The time overhead compared to LAVE (+3.27s, 1.64×) is due to the increased attempt budget ($\tau = 10$ vs $\tau = 5$) needed to allow resampling after stall rejections.

4.3 Ablation: Progress Tracking vs Naive Cap

To validate that lexer-progress tracking provides value beyond simple whitespace rejection, we compare PG-LAVE with the Naive Cap baseline. Both methods reject whitespace-only proposals, but Naive Cap uses a simple counter of consecutive whitespace accepts, while PG-LAVE tracks actual lexer progress (non-ignored terminal count).

The results in Table 1 show that PG-LAVE strictly outperforms Naive Cap on hit_max_len_rate with non-overlapping confidence intervals (1.83±0.00 vs 3.25±0.76). This demonstrates that progress-aware stall detection is more effective than token-level heuristics. The key insight is that the grammar's notion of "progress" (production of non-ignored terminals) provides a more robust signal than simply counting whitespace tokens.

4.4 Sensitivity Analysis

Figure 2 shows PG-LAVE's sensitivity to the stall threshold H. For $H \geq 16$, syntactic@1 plateaus at 89.02% and functional@1 at 17.68%, demonstrating robustness to the threshold choice. The hit_max_len_rate shows a U-shaped curve: too small H (e.g., 8) triggers premature rejections that hurt performance, while too large H (e.g., 128) allows extended stalls before intervention. The optimal range is $H \in [16, 64]$, with $H = 16$ providing the best balance of truncation reduction and simplicity.

4.5 Compute-Quality Tradeoffs

Table 2 compares PG-LAVE against alternative approaches for reducing truncation failures. Prompt engineering (adding "avoid repeated blank lines" to the prompt) actually *degrades* performance (-1.31 pp syntactic@1, -2.35 pp functional@1), confirming that diffusion LLMs do not reliably

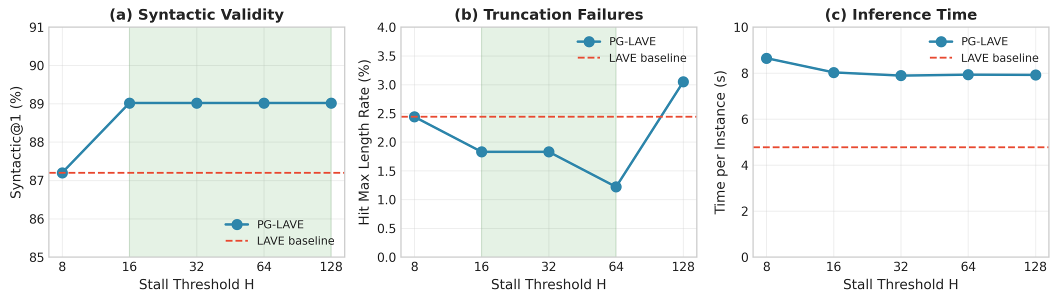

Figure 2: Sensitivity of PG-LAVE to stall threshold H. (a) Syntactic validity improves from 87.2% to 89.02% for $H \geq 16$. (b) Truncation failures show a U-shaped curve with minimum at $H = 64$. (c) Inference time remains stable around 8s. Green shading indicates the robust operating region ($H \geq 16$).

Table 2: Compute-quality tradeoffs on CPP-Bench (seed=42). Best-of-3 establishes upper bound at 3× compute. Prompt engineering degrades performance.

Method	Syntactic@1 (%)	Functional@1 (%)	Hit Max Len (%) ↓	Time (s)
LAVE	87.20	16.46	2.44	4.77
LAVE + Prompt Eng.	85.89	14.11	3.07	6.13
Naive Cap (K=32)	87.20	17.07	2.44	4.97
PG-LAVE (H=16)	**89.02**	**17.68**	**1.83**	8.03
Best-of-3 LAVE	96.95	18.90	0.00	13.61

follow natural-language formatting instructions. Best-of-3 LAVE achieves 96.95% syntactic@1 and 0% hit_max_len_rate, establishing an upper bound for what resampling can achieve—but at 3× the compute cost. PG-LAVE provides an efficient single-pass alternative that achieves the best results among 1× compute methods.

5 Conclusion

We presented PG-LAVE, a progress-guarded extension to LAVE that addresses stall loops in CFG-constrained diffusion decoding. By monitoring lexer progress and rejecting ignored-only proposals when progress stalls, PG-LAVE achieves a 40% relative reduction in truncation failures (1.83% vs 3.05% hit_max_len_rate) while improving syntactic validity by +0.81 percentage points on CPP-Bench. Our ablation study demonstrates that lexer-progress tracking provides value beyond simple whitespace rejection heuristics.

Limitations. PG-LAVE incurs a time overhead of 1.64× compared to LAVE due to the increased attempt budget. Our evaluation is limited to a single benchmark (CPP-Bench) and model (LLaDA-8B-Instruct).

Future Work. Promising directions include extending PG-LAVE to other grammars (JSON, SQL), investigating adaptive thresholds that adjust H based on generation context, and applying the approach to other diffusion LLMs.

References

Marianne Arriola, Aaron Gokaslan, Justin T Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, S. Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. *ArXiv*, abs/2503.09573, 2025.

Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7: 1946 – 1969, 2022.

Huiwen Chang, Han Zhang, Lu Jiang, Ce Liu, and W. Freeman. Maskgit: Masked generative image transformer. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11305–11315, 2022.

Yihong Dong, Yuchen Liu, Xue Jiang, Zhi Jin, and Ge Li. Rethinking repetition problems of llms in code generation. pp. 965–985, 2025.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *ArXiv*, abs/1904.09751, 2019.

Bumjun Kim, Dongjae Jeon, Dueun Kim, Wonje Jeung, and Albert No. Rainbow padding: Mitigating early termination in instruction-tuned diffusion llms. *ArXiv*, abs/2510.03680, 2025.

Niels Mündler, Jasper Dekoninck, and Martin Vechev. Constrained decoding of diffusion llms with context-free grammars, 2025. URL `https://arxiv.org/abs/2508.10111`.

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.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'antoni. Grammar-aligned decoding. *ArXiv*, abs/2405.21047, 2024.

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.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *ArXiv*, abs/2109.05093, 2021.

Tarun Suresh, Debangshu Banerjee, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Dingo: Constrained inference for diffusion llms, 2025. URL `https://arxiv.org/abs/2505.23061`.

Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models. *ArXiv*, abs/2307.09702, 2023.

Jin Xu, Xiaojiang Liu, Jianhao Yan, Deng Cai, Huayang Li, and Jian Li. Learning to break the loop: Analyzing and mitigating repetitions for neural text generation. *ArXiv*, abs/2206.02369, 2022.

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.

Yitong Zhang, Yongming Li, Yuetong Liu, Jia Li, Xiaoran Jia, Zherui Li, and Ge Li. Lookahead-then-verify: Reliable constrained decoding for diffusion llms under context-free grammars. 2026.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shanshan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.