LOCATE, MASK, AND REGENERATE: PARSER-GUIDED SELF-REPAIR FOR STRUCTURED GENERATION WITH DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion language models (DLMs) have attracted growing attention as promising backbones for large language model (LLM)-based agents due to their capabilities for parallel generation and partial revision. To interact successfully with external environments, these agents must reliably generate structured outputs that conform to strict syntactic rules. However, we consistently observe low parsing success rates across a range of structured generation tasks, with further declines as the number of denoising steps decreases. To address this issue, we introduce a new post-generation strategy, namely Parser-Guided Self-Repair (PGSR), which reinterprets the parser feedback as an explicit localization signal for selective refinement. Using the parser-reported error position as an anchor, PGSR remasks a small surrounding span and reconstructs it through DLM infilling. It is designed to perform localized repair without additional training and to generalize across diverse structured output domains in a format-agnostic manner. We conduct comprehensive experiments with LLaDA-8B and Dream-7B across four structured generation tasks: tool calling, code generation, text-to-structured query language (SQL), and molecular generation. Experimental results demonstrate that PGSR consistently improves both structural validity and downstream task performance across different denoising budgets. It is also confirmed that the proposed approach is compatible with existing decoding strategies, yielding further gains when combined with constrained decoding.

1 INTRODUCTION

Large language model (LLM)-based agents have recently emerged as a powerful paradigm for solving complex real-world problems through interactions with external environments (Du et al., 2026; He et al., 2026; Park & Kwon, 2026; Xu et al., 2026). Many of these interactions require models to produce task-specific structured outputs, such as JavaScript object notation (JSON) objects for tool calling, executable code for code generation, and structured query language (SQL) queries for text-to-SQL. Unlike free-form text, these outputs must follow predefined syntactic and structural requirements, as even minor violations can cause failures in downstream parsing or execution. When such errors occur, existing LLM-based agents typically use the invalid output together with the resulting parser feedback to generate a new response from scratch (Wang et al., 2026; Cheng et al., 2026; Yin et al., 2025), rather than selectively revising the relevant region. However, regenerating the entire response at each retry can introduce substantial inference latency and computational overhead, with these costs further accumulating over multiple rounds of repeated failure (Duanis et al., 2025; Zong et al., 2024).

In this context, diffusion language models (DLMs) have attracted growing attention as an alternative backbone for agentic systems (Lu et al., 2026; Zhao et al., 2026; Zhen et al., 2026). Their parallel token refinement mitigates the sequential bottleneck of autoregressive (AR) decoding, thereby improving inference efficiency. Beyond this computational advantage, iterative denoising also allows DLMs to revisit and selectively revise earlier predictions, naturally supporting localized error correction. Despite this potential, DLM-based agents exhibit substantially lower overall performance than their AR counterparts, with prior studies reporting higher rates of invalid actions arising from unreliable structured output generation, particularly in tool calling settings (Lu et al., 2026; Zhen et al.,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

2026). Here, we experimentally observe that this limitation extends beyond tool calling to a broad range of structured generation settings, including code generation, text-to-SQL, and molecular generation. The problem becomes particularly severe under low denoising-step budgets. At 4 denoising steps, parsing success rates range from 0.0% to only 54.7% across the evaluated configurations. Increasing the step budget improves structural validity but does not fully close the gap: even at 64 steps, only 43.3% of Python outputs from LLaDA-8B are successfully parsed, whereas the AR reference achieves 97.8–99.9% across all four domains. Prior work has sought to alleviate this problem through constrained decoding (Mündler et al., 2026; Zhang et al., 2026; Jin & Han, 2026), which enforces syntactic constraints throughout the denoising process. This requires grammar-aware checks at every decoding step, even when no correction is ultimately needed, adding extra inference-time computation. An alternative is to leverage the partial-revision capability of DLMs and trigger correction only after a structural error is detected. Yet, post-generation repair for structured outputs remains largely unexplored and how to effectively localize and correct such errors in DLMs is still an unresolved question.

To address this challenge, we introduce Parser-Guided Self-Repair (PGSR), a post-generation framework for improving the structural reliability of DLM outputs. This work departs from prior error mitigation approaches for DLMs (Zou et al., 2026; Ren et al., 2026; Mündler et al., 2026; Zhang et al., 2026; Jin & Han, 2026) in three key ways. First, PGSR reinterprets parser feedback as an explicit localization signal for refinement. Based on the identified error position, a local span is selectively remasked and reconstructed through DLM infilling. This design not only avoids repeating the full decoding process but also requires no additional model training, making PGSR entirely training-free. Second, PGSR uses format-specific parsers only to locate errors, leaving the subsequent correction procedure unchanged. This separation enables PGSR to perform format-agnostic repair across diverse structured output domains. Third, PGSR can be combined with different decoding strategies to achieve further accuracy gains. When the DLM infills the masked span, the paired decoding strategy guides the selection of replacement tokens.

We conduct extensive experiments with LLaDA-8B (Nie et al., 2026) and Dream-7B (Ye et al., 2025) across four structured generation domains: tool calling, code generation, text-to-SQL, and molecular generation. Empirically, it is verified that PGSR consistently improves both structural validity and downstream task accuracy across these domains. On the tool calling benchmark Berkeley Function Calling Leaderboard (BFCL) (Patil et al., 2025), PGSR raises simple Python accuracy from 71.00% to 80.75% with 4 diffusion steps, surpassing that of the unrepaired baseline at 32 steps. These improvements persist across the evaluated budgets of 4, 8, 16, 32, and 64 diffusion steps, with further gains in several categories when PGSR is combined with constrained decoding. This trend extends across diverse structured generation tasks and both DLM backbones, demonstrating the broad applicability of PGSR.

2 Related Work

Diffusion Language Models in Agentic Systems. Recent studies on DLM-based agents have examined how diffusion generation can be adapted to multi-step interaction, tool use, and iterative decision making. Lu et al. (2026) systematically evaluate general-purpose DLMs in embodied and tool calling workflows, revealing substantial limitations in long-horizon replanning and structured action generation. Subsequent studies have sought to overcome these limitations through agent-specific adaptation. DLLM-Searcher (Zhao et al., 2026) combines agentic post-training with a tool-first diffusion schedule to improve search performance and interaction efficiency, while Zhen et al. (2026) show that diffusion-backed agents can reduce end-to-end latency but remain more susceptible to invalid structured actions. More recently, LLaDA2.2 (Bie et al., 2026) introduces learned insertion and deletion operations to support multi-turn tool use and error correction. This growing body of work highlights the potential of DLM-based agents while also underscoring the need for reliable interaction with tools and environments. PGSR complements this direction through deterministic, training-free post-generation repair, improving the reliability of structured outputs generated by DLMs.

Reliable Structured Output Generation. Reliability in formal-output tasks has largely been pursued by constraining the decoding trajectory so that intermediate predictions remain compatible with a target grammar or schema. For DLMs, Mündler et al. (2026) formulate this problem through

grammar-constrained additive infilling, rejecting assignments that make the partial sequence impossible to complete. LAVE (Zhang et al., 2026) uses model-predicted lookahead information to verify candidate assignments before commitment, and EPIC (Jin & Han, 2026) reduces the cost of CFG validation to commit multiple compatible candidates in parallel. In contrast, a separate line of work revises an already generated sequence. DRR (Zou et al., 2026) trains a token-level discriminator with synthetic token-level supervision to remask semantically unsupported spans for faithfulness repair in evolving-context summarization, while ME-DLM (Ren et al., 2026) learns replacement, deletion, and insertion operations through three sequential stages of full-parameter finetuning on large-scale instruction and reasoning corpora. These approaches rely on learned components or additional training for refinement. We instead focus our comparisons on constrained decoding methods that, like PGSR, can be applied to DLMs without additional training.

3 Method: Parser-Guided Self-Repair

3.1 Preliminary: Generation and Infilling in DLMs

A masked DLM (Sahoo et al., 2024) defines a discrete forward corruption distribution q that progressively corrupts a token sequence $\mathbf{x}_0 = (x_0^1, \ldots, x_0^L)$ by independently replacing each token with a special `[MASK]` token m according to a masking schedule $\gamma_t \in [0, 1]$:

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \prod_{i=1}^{L} \left[\gamma_t\, \delta(x_t^i, m) + (1 - \gamma_t)\, \delta(x_t^i, x_0^i)\right], \tag{1}$$

where δ denotes the Kronecker delta and γ_t is the probability that a token is masked at timestep t, with $\gamma_0 = 0$ and $\gamma_T = 1$. Generation reverses this corruption process using the DLM p_θ. Starting from a fully masked sequence $\mathbf{x}_T = (m, \ldots, m)$, the model predicts a distribution $p_\theta(x_0^i \mid \mathbf{x}_t)$ over the clean token at every masked position. A subset of these positions is filled at each of the T denoising steps according to a confidence-based schedule. The step budget T therefore controls the computation required for generation, and reducing it can degrade output quality.

The same reverse procedure can be applied to a partially masked sequence $\tilde{\mathbf{x}}$. Let $\mathcal{M}$ denote the positions carrying the mask token m, while tokens outside $\mathcal{M}$ retain their current values. We write $\textsc{Infill}(p_\theta, \tilde{\mathbf{x}}, T')$ for the process of denoising only the positions in $\mathcal{M}$ over T' steps while leaving the remaining positions unchanged. Because the unmasked tokens provide rich surrounding context, partial infilling can reconstruct a selected region more reliably than generating it from scratch, thereby providing a natural mechanism for localized revision.

3.2 Problem Setting: Structural Validity under a Small Step Budget

As noted earlier, producing structurally valid outputs remains challenging for DLMs. We empirically examine the extent of this problem across structured generation tasks and denoising step budgets. Figure 1 reports how often the standard parser of each format accepts a tool call, program, query, or molecule from LLaDA-8B and Dream-7B, each adapted to its task with the setup of §4.1. All eight series sit lower at $T = 4$ than at $T = 64$. At the small budget, nothing is accepted more than 54.7% of the time, and molecules from LLaDA-8B fall from 74.6% to 0.0%. Structural reliability is thus weakest exactly at the budgets that make these models fast, and raising T is no escape, since it gives up that speed and still leaves programs from LLaDA-8B parsing only 43.3% of the time at $T = 64$. This limitation cannot be attributed solely to the output formats, as the Qwen2.5-7B AR baseline adapted using the same four corpora achieves parsing success rates of 97.8%–99.9% across the four domains. PGSR therefore aims to recover this lost reliability at small denoising budgets without sacrificing speed by increasing T.

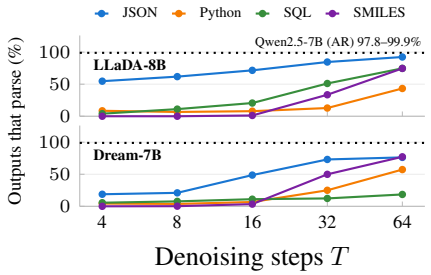

Figure 1: Parsing success rates across denoising budgets T for LLaDA-8B (top) and Dream-7B (bottom). The dotted line denotes the Qwen2.5-7B AR baseline. All models use greedy decoding.

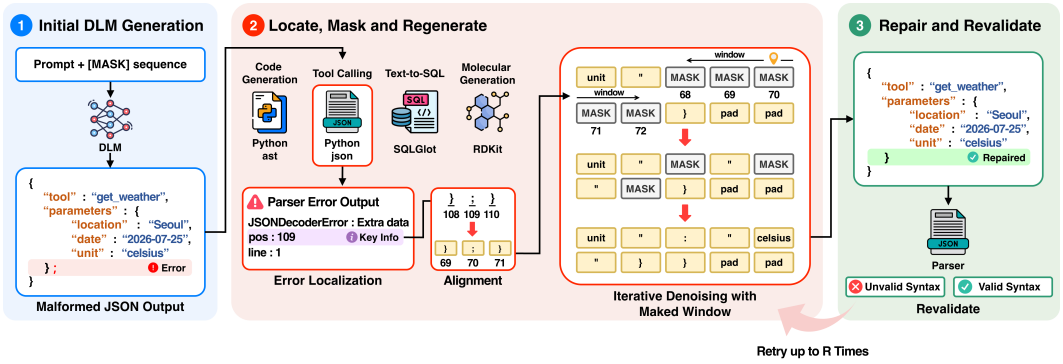

Figure 2: Overview of PGSR. A deterministic parser localizes the syntax error position in the DLM output, and the surrounding tokens are selectively remasked and regenerated. This locate–mask–regenerate cycle is repeated until the output passes validation or R repair rounds are exhausted.

3.3 Locate, Mask, and Regenerate

Given a structured output format, such as JSON for tool calling, Python for code generation, SQL for text-to-SQL, or SMILES for molecular generation, our goal is to improve the structural validity of the DLM outputs. Figure 2 illustrates PGSR, which corrects malformed outputs by selectively revising the suspected span rather than repeating the full decoding process. Let $\mathbf{y}^{(0)}$ be the initial output of p_θ, and let $\mathbf{y}^{(r-1)} = (y_1^{(r-1)}, \ldots, y_N^{(r-1)})$ denote the sequence at round r. Using a format-specific error diagnostic, PGSR identifies a token-level span, remasks the selected tokens, and applies DLM infilling. The process stops when the parser accepts the updated sequence or when R rounds have been completed.

Error Localization and Targeted Repair. Let $s = \mathrm{decode}(\mathbf{y}^{(r-1)})$ be the decoded output. A format-specific parser $\mathcal{P}$ examines s and, when possible, returns the character offset at which parsing fails:

$$c^{(r)} = \mathcal{P}(s) = \begin{cases} \varnothing & s \text{ accepted, or rejected with no offset,} \\ \text{the offset } \mathcal{P} \text{ reports} & \text{otherwise.} \end{cases} \tag{2}$$

If $c^{(r)} = \varnothing$, PGSR leaves $\mathbf{y}^{(r-1)}$ unchanged, preventing modification when no reliable repair location is available. Otherwise, the character offset is mapped to the corresponding token position. Writing $\mathrm{cs}(y_i)$ for the starting character offset of token y_i in s, with $\mathrm{cs}(y_1) = 0$ and $\mathrm{cs}(y_{i+1}) = \mathrm{cs}(y_i) + |\mathrm{decode}(y_i)|$, we define

$$e^{(r)} = \max\{\, i : \mathrm{cs}\big(y_i^{(r-1)}\big) \le c^{(r)} \,\}, \quad \text{i.e.} \quad \mathrm{cs}\big(y_{e^{(r)}}\big) \le c^{(r)} < \mathrm{cs}\big(y_{e^{(r)}+1}\big), \tag{3}$$

where $e^{(r)}$ serves as an anchor for repair. Because the reported offset indicates where parsing fails rather than necessarily where the error originates, PGSR masks a symmetric window around the anchor. This window captures nearby tokens that may have caused the failure while accounting for discrepancies between character-level parser offsets and token boundaries. We instantiate $\mathcal{P}$ using Python's `json` module for JSON, `ast.parse` for Python code, SQLGlot for SQL, and RDKit for SMILES.

Using $e^{(r)}$, PGSR masks a local region extending w tokens in both directions and applies infilling only to the selected positions:

$$\mathcal{M}^{(r)} = \{\, i \in \{1, \ldots, N\} : |i - e^{(r)}| \le w \,\}, \qquad \tilde{y}_i^{(r)} = \begin{cases} m & \text{if } i \in \mathcal{M}^{(r)}, \\ y_i^{(r-1)} & \text{otherwise,} \end{cases} \tag{4}$$

$$\mathbf{y}^{(r)} = \textsc{Infill}\big(p_\theta,\ [\mathbf{p};\ \tilde{\mathbf{y}}^{(r)};\ \texttt{EOT}],\ T_{\mathrm{rep}}\big), \qquad r = 1, \ldots, R. \tag{5}$$

The masked sequence is concatenated with the original prompt $\mathbf{p}$ and the end-of-turn delimiter, allowing the model to reconstruct the selected region while conditioning on both the task instruction and the intact remainder of the output. PGSR therefore treats an invalid generation not as a failed

result to be discarded, but as a partially correct solution that can be efficiently refined. Unlike full regeneration, which may alter already correct content, localized infilling preserves the global structure and semantics of the original output while revising only the region implicated in the failure. The complete PGSR procedure is summarized in Appendix A.

Adaptive and Compute-Efficient Refinement. After each repair round, PGSR revalidates the updated output and, if another usable error offset is returned, identifies a new region for the next round. This iterative process allows multiple errors to be corrected sequentially.

Let $R_i \leq R$ denote the number of repair rounds performed for sample i, where $R_i = 0$ if $\mathcal{P}$ accepts $\mathbf{y}^{(0)}$ or returns no usable offset, and let ρ be the fraction of initial outputs rejected by the parser. Counting forward passes,

$$N_i = T + R_i T_{\text{rep}}, \qquad \mathbb{E}[N_i] \leq T + \rho R T_{\text{rep}}. \tag{6}$$

Since $T_{\text{rep}} = 1$, the expected relative overhead of PGSR over the initial T-step decoding cost is at most $\rho R/T$. In contrast, each full-sequence retry adds another T forward passes, resulting in a total cost of up to $(1 + R)T$.

Training-Free Design. Existing post-generation refinement methods for DLMs typically require additional learning to determine which parts of an output should be revised or how the revision should be performed. For example, prior approaches train an auxiliary discriminator to identify unreliable tokens or further finetune the DLM to learn explicit editing operations (Zou et al., 2026; Ren et al., 2026). Such methods therefore introduce repair-specific supervision and parameter updates after the original task adaptation. PGSR removes this additional learning stage by assigning these roles to components that are already available at inference time. A deterministic parser provides the location of a structural failure, and the pretrained DLM itself reconstructs the selected span through its native infilling capability. As a result, PGSR introduces no repair-specific training, allowing the post-generation repair procedure to remain entirely training-free.

Format-Agnostic Repair Mechanism. Beyond this training-free design, its applicability across structured formats stems from separating format-specific error localization from the subsequent repair procedure. Format-specific parsers are used only to identify where parsing fails, while the same remasking and infilling procedure is applied regardless of the output format. Consequently, adapting PGSR to a new structured domain only requires a parser that can provide a usable failure location, without modifying the underlying model or repair mechanism. This design enables training-free and format-agnostic refinement across different DLM backbones and structured generation tasks.

Integration with Constrained Decoding. PGSR is complementary to existing constrained decoding methods, as their token-level constraints can also be incorporated into local regeneration. In our experiments, when a compatible decoding constraint $\mathcal{G}$ was available, we used it only during local infilling by replacing INFILL in equation 5 with constrained infilling, CD-INFILL$(p_\theta, \cdot, T_{\text{rep}}, \mathcal{G})$. This operation restricts $\mathcal{G}$ to the masked repair region $\mathcal{M}^{(r)}$. Since at most $2w+1$ tokens are subject to constraint checking within a single denoising step, the resulting overhead is substantially lower than enforcing constraints over the entire output throughout the decoding process.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Tasks and Benchmarks. Our evaluation covered four domains, each defined by distinct syntactic and structural requirements. We used execution- or semantic-level metrics rather than surface-form matching, ensuring that improvements could not be achieved merely by making outputs parseable. For tool calling, BFCL v4 (Patil et al., 2025) served as the standard benchmark, with results reported on its six non-live and four live categories using the official abstract syntax tree (AST)-based checker. This evaluator compared the predicted function calls against the reference calls at the structural level, including function names and arguments. Code generation was evaluated by pass@1 on HumanEval+ and MBPP+ using EvalPlus (Liu et al., 2023), along with the original HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) results. Text-to-SQL performance was measured

Table 1: BFCL accuracy (%) of LLaDA-8B across diffusion steps. Subscripts denote the gain or drop over Raw at the same step. Best per category per step in **bold**.

Steps	Method	Non-Live						Live			
		Py	Java	JS	Mult.	Par.	P.M.	L.Simp.	L.Mult.	L.Par.	L.P.M.
4	Raw	71.00	22.00	24.00	63.00	47.50	19.50	43.80	43.37	12.50	15.00
	C-Diff	71.25 +0.25	21.00 −1.00	26.00 +2.00	63.00 +0.00	56.00 +8.50	26.00 +6.50	40.70 −3.10	35.29 −8.08	**31.25** +18.75	15.00 +0.00
	LAVE (b=128)	76.25 +5.25	**26.00** +4.00	**28.00** +4.00	73.50 +10.50	36.00 −11.50	18.00 −1.50	50.78 +6.98	37.49 −5.88	12.50 +0.00	**25.00** +10.00
	EPIC	71.25 +0.25	21.00 −1.00	26.00 +2.00	63.00 +0.00	56.00 +8.50	26.00 +6.50	40.70 −3.10	35.64 −7.73	**31.25** +18.75	15.00 +0.00
	PGSR	**80.75** +9.75	**26.00** +4.00	26.00 +2.00	**76.50** +13.50	51.50 +4.00	23.00 +3.50	**52.33** +8.53	**54.44** +11.07	12.50 +0.00	20.00 +5.00
	PGSR + C-Diff	79.00 +8.00	22.00 +0.00	24.00 +0.00	73.00 +10.00	**58.00** +10.50	**39.00** +19.50	51.16 +7.36	51.33 +7.96	12.50 +0.00	15.00 +0.00
	PGSR + EPIC	79.00 +8.00	22.00 +0.00	24.00 +0.00	73.00 +10.00	**58.00** +10.50	**39.00** +19.50	51.16 +7.36	51.33 +7.96	12.50 +0.00	15.00 +0.00
8	Raw	71.75	23.00	26.00	65.50	71.00	51.00	46.12	43.71	**37.50**	**30.00**
	C-Diff	76.75 +5.00	**27.00** +4.00	**28.00** +2.00	70.50 +5.00	71.50 +0.50	49.50 −1.50	50.00 +3.88	40.60 −3.11	31.25 −6.25	**30.00** +0.00
	LAVE (b=64)	77.25 +5.50	25.00 +2.00	**28.00** +2.00	74.00 +8.50	53.00 −18.00	35.00 −16.00	51.55 +5.43	38.87 −4.84	18.75 −18.75	**30.00** +0.00
	EPIC	76.75 +5.00	**27.00** +4.00	**28.00** +2.00	70.50 +5.00	71.50 +0.50	50.50 −0.50	50.00 +3.88	41.64 −2.07	31.25 −6.25	**30.00** +0.00
	PGSR	**79.75** +8.00	25.00 +2.00	**28.00** +2.00	**77.00** +11.50	74.00 +3.00	55.00 +4.00	**51.94** +5.82	**53.98** +10.27	**37.50** +0.00	**30.00** +0.00
	PGSR + C-Diff	79.25 +7.50	23.00 +0.00	26.00 +0.00	75.50 +10.00	**75.00** +4.00	**62.50** +11.50	51.55 +5.43	50.75 +7.04	**37.50** +0.00	**30.00** +0.00
	PGSR + EPIC	79.25 +7.50	23.00 +0.00	26.00 +0.00	76.00 +10.50	**75.00** +4.00	**62.50** +11.50	51.55 +5.43	50.63 +6.92	**37.50** +0.00	**30.00** +0.00
16	Raw	73.50	26.00	28.00	67.50	76.00	63.50	52.33	47.64	**56.25**	35.00
	C-Diff	77.25 +3.75	**28.00** +2.00	28.00 +0.00	74.00 +6.50	**79.50** +3.50	61.50 −2.00	54.65 +2.32	41.98 −5.66	50.00 −6.25	30.00 −5.00
	LAVE (b=32)	81.00 +7.50	24.00 −2.00	28.00 +0.00	**82.00** +14.50	79.00 +3.00	63.00 −0.50	52.71 +0.38	47.75 +0.11	31.25 −25.00	**40.00** +5.00
	EPIC	77.25 +3.75	**28.00** +2.00	28.00 +0.00	74.00 +6.50	**79.50** +3.50	64.50 +1.00	54.65 +2.32	44.52 −3.12	50.00 −6.25	30.00 −5.00
	PGSR	81.25 +7.75	27.00 +1.00	**30.00** +2.00	77.00 +9.50	77.50 +1.50	67.00 +3.50	54.65 +2.32	**54.67** +7.03	**56.25** +0.00	**40.00** +5.00
	PGSR + C-Diff	**82.75** +9.25	26.00 +0.00	28.00 +0.00	79.00 +11.50	78.00 +2.00	**68.50** +5.00	**57.75** +5.42	52.83 +5.19	**56.25** +0.00	**40.00** +5.00
	PGSR + EPIC	**82.75** +9.25	26.00 +0.00	28.00 +0.00	79.00 +11.50	78.00 +2.00	**68.50** +5.00	**57.75** +5.42	52.83 +5.19	**56.25** +0.00	**40.00** +5.00
32	Raw	78.50	36.00	**36.00**	77.50	82.00	74.50	58.14	53.52	**56.25**	45.00
	C-Diff	81.00 +2.50	33.00 −3.00	26.00 −10.00	79.00 +1.50	80.50 −1.50	61.00 −13.50	58.53 +0.39	42.68 −10.84	50.00 −6.25	40.00 −5.00
	LAVE (b=32)	84.00 +5.50	32.00 −4.00	28.00 −8.00	82.50 +5.00	80.50 −1.50	74.00 −0.50	57.75 −0.39	50.52 −3.00	37.50 −18.75	**50.00** +5.00
	EPIC	81.00 +2.50	33.00 −3.00	26.00 −10.00	79.00 +1.50	81.00 −1.00	70.00 −4.50	58.53 +0.39	46.25 −7.27	50.00 −6.25	45.00 +0.00
	PGSR	85.50 +7.00	**37.00** +1.00	**36.00** +0.00	**83.00** +5.50	**82.50** +0.50	**76.50** +2.00	60.08 +1.94	**56.52** +3.00	**56.25** +0.00	45.00 +0.00
	PGSR + C-Diff	**85.75** +7.25	36.00 +0.00	**36.00** +0.00	82.50 +5.00	**82.50** +0.50	**76.50** +2.00	**62.40** +4.26	55.71 +2.19	**56.25** +0.00	45.00 +0.00
	PGSR + EPIC	**85.75** +7.25	36.00 +0.00	**36.00** +0.00	82.50 +5.00	**82.50** +0.50	**76.50** +2.00	**62.40** +4.26	55.71 +2.19	**56.25** +0.00	45.00 +0.00
64	Raw	88.00	**36.00**	**36.00**	83.00	82.50	**80.00**	59.69	53.86	43.75	45.00
	C-Diff	83.75 −4.25	30.00 −6.00	22.00 −14.00	81.50 −1.50	79.50 −3.00	66.00 −14.00	61.24 +1.55	36.68 −17.18	43.75 +0.00	45.00 +0.00
	LAVE (b=32)	87.00 −1.00	35.00 −1.00	28.00 −8.00	83.50 +0.50	**84.50** +2.00	79.50 −0.50	**62.79** +3.10	54.09 +0.23	**56.25** +12.50	**55.00** +10.00
	EPIC	83.75 −4.25	30.00 −6.00	22.00 −14.00	82.00 −1.00	80.00 −2.50	70.50 −9.50	61.24 +1.55	45.10 −8.76	43.75 +0.00	45.00 +0.00
	PGSR	89.25 +1.25	**36.00** +0.00	**36.00** +0.00	86.50 +3.50	83.00 +0.50	**80.00** +0.00	61.24 +1.55	**54.67** +0.81	43.75 +0.00	45.00 +0.00
	PGSR + C-Diff	**89.75** +1.75	**36.00** +0.00	**36.00** +0.00	**87.50** +4.50	82.50 +0.00	**80.00** +0.00	62.40 +2.71	54.09 +0.23	50.00 +6.25	45.00 +0.00
	PGSR + EPIC	**89.75** +1.75	**36.00** +0.00	**36.00** +0.00	**87.50** +4.50	82.50 +0.00	**80.00** +0.00	62.40 +2.71	54.09 +0.23	50.00 +6.25	45.00 +0.00

by execution accuracy (EX) on the cross-domain Spider dev set (Yu et al., 2018), which determined whether the predicted SQL query produced the same execution result as the reference query. For molecular generation, molecules were represented as SMILES strings, and performance on ZINC-250k (Gómez-Bombarelli et al., 2018) was assessed using validity and Fréchet ChemNet Distance (FCD) (Preuer et al., 2018), capturing the proportion of valid molecules and similarity to the reference distribution, respectively.

Implementation Details. Experiments were conducted using two pretrained DLMs, LLaDA-8B-Instruct (Nie et al., 2026) and Dream-v0-Instruct-7B (Ye et al., 2025). Each backbone was adapted separately to the four domains using QLoRA (Dettmers et al., 2023) with rank 32 for 3 epochs, and the resulting checkpoints were held fixed across all methods. Baselines included unrepaired generation, denoted as Raw, Constrained-Diffusion (Mündler et al., 2026), LAVE (Zhang et al., 2026), and EPIC-Decoding (Jin & Han, 2026). Constrained-Diffusion and EPIC-Decoding were referred to as C-Diff and EPIC throughout, respectively.

All methods used greedy decoding. Except for LAVE, the diffusion block size was set equal to the generation length, enabling fully bidirectional rather than semi-autoregressive decoding. LAVE instead followed its original semi-autoregressive decoding procedure with block length b. Unless otherwise stated, PGSR was configured with $w = 5$, $R = 2$, and $T_{\text{rep}} = 1$, while the initial generation budget was varied over $T \in \{4, 8, 16, 32, 64\}$. The combined variants PGSR + C-Diff and PGSR + EPIC applied the corresponding decoding constraint only during local regeneration. All training and evaluation were conducted on a single NVIDIA RTX PRO 6000 Blackwell GPU, and latency was measured as wall-clock time with a batch size of 1. Additional implementation details are provided in Appendix G.

Table 2: BFCL accuracy (%) of Dream-7B across diffusion steps $T \in \{8, 16, 32\}$. Subscripts denote the gain or drop over Raw at the same step. Best per category per step in **bold**.

Steps	Method	Non-Live						Live			
		Py	Java	JS	Mult.	Par.	P.M.	L.Simp.	L.Mult.	L.Par.	L.P.M.
8	Raw	60.00	16.00	24.00	55.00	16.00	3.00	40.31	41.38	**6.25**	4.35
	C-Diff	20.75 $_{-39.25}$	2.00 $_{-14.00}$	6.00 $_{-18.00}$	19.00 $_{-36.00}$	1.50 $_{-14.50}$	1.00 $_{-2.00}$	20.16 $_{-20.15}$	10.96 $_{-30.42}$	0.00 $_{-6.25}$	0.00 $_{-4.35}$
	LAVE (b=64)	46.00 $_{-14.00}$	10.00 $_{-6.00}$	18.00 $_{-6.00}$	46.00 $_{-9.00}$	16.00 $_{+0.00}$	**8.50** $_{+5.50}$	39.53 $_{-0.78}$	31.91 $_{-9.47}$	0.00 $_{-6.25}$	**13.04** $_{+8.69}$
	EPIC	20.75 $_{-39.25}$	2.00 $_{-14.00}$	6.00 $_{-18.00}$	19.00 $_{-36.00}$	1.50 $_{-14.50}$	1.00 $_{-2.00}$	20.16 $_{-20.15}$	10.96 $_{-30.42}$	0.00 $_{-6.25}$	0.00 $_{-4.35}$
	PGSR	67.00 $_{+7.00}$	20.00 $_{+4.00}$	**26.00** $_{+2.00}$	63.00 $_{+8.00}$	**19.50** $_{+3.50}$	7.00 $_{+4.00}$	44.57 $_{+4.26}$	46.91 $_{+5.53}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
	PGSR + C-Diff	**71.50** $_{+11.50}$	**23.00** $_{+7.00}$	**26.00** $_{+2.00}$	**69.00** $_{+14.00}$	**19.50** $_{+3.50}$	5.00 $_{+2.00}$	**50.00** $_{+9.69}$	**48.40** $_{+7.02}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
	PGSR + EPIC	**71.50** $_{+11.50}$	**23.00** $_{+7.00}$	**26.00** $_{+2.00}$	**69.00** $_{+14.00}$	**19.50** $_{+3.50}$	5.00 $_{+2.00}$	**50.00** $_{+9.69}$	**48.40** $_{+7.02}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
16	Raw	70.00	14.00	26.00	59.50	24.00	14.50	45.74	45.96	25.00	13.04
	C-Diff	73.25 $_{+3.25}$	18.00 $_{+4.00}$	24.00 $_{-2.00}$	**73.00** $_{+13.50}$	27.00 $_{+3.00}$	**28.50** $_{+14.00}$	**54.65** $_{+8.91}$	45.85 $_{-0.11}$	25.00 $_{+0.00}$	34.78 $_{+21.74}$
	LAVE (b=32)	55.25 $_{-14.75}$	6.00 $_{-8.00}$	14.00 $_{-12.00}$	50.50 $_{-9.00}$	27.00 $_{+3.00}$	20.00 $_{+5.50}$	41.86 $_{-3.88}$	35.11 $_{-10.85}$	**43.75** $_{+18.75}$	**39.13** $_{+26.09}$
	EPIC	73.25 $_{+3.25}$	18.00 $_{+4.00}$	24.00 $_{-2.00}$	**73.00** $_{+13.50}$	27.00 $_{+3.00}$	**28.50** $_{+14.00}$	**54.65** $_{+8.91}$	45.85 $_{-0.11}$	25.00 $_{+0.00}$	34.78 $_{+21.74}$
	PGSR	76.25 $_{+6.25}$	15.00 $_{+1.00}$	28.00 $_{+2.00}$	68.50 $_{+9.00}$	**28.00** $_{+4.00}$	17.00 $_{+2.50}$	49.22 $_{+3.48}$	52.02 $_{+6.06}$	25.00 $_{+0.00}$	13.04 $_{+0.00}$
	PGSR + C-Diff	**78.75** $_{+8.75}$	**25.00** $_{+11.00}$	**30.00** $_{+4.00}$	72.50 $_{+13.00}$	27.50 $_{+3.50}$	18.00 $_{+3.50}$	52.71 $_{+6.97}$	**53.40** $_{+7.44}$	25.00 $_{+0.00}$	21.74 $_{+8.70}$
	PGSR + EPIC	**78.75** $_{+8.75}$	**25.00** $_{+11.00}$	**30.00** $_{+4.00}$	72.50 $_{+13.00}$	27.50 $_{+3.50}$	18.00 $_{+3.50}$	52.71 $_{+6.97}$	**53.40** $_{+7.44}$	25.00 $_{+0.00}$	21.74 $_{+8.70}$
32	Raw	80.25	25.00	30.00	77.50	62.00	52.50	52.71	57.45	31.25	47.83
	C-Diff	77.50 $_{-2.75}$	27.00 $_{+2.00}$	30.00 $_{+0.00}$	79.50 $_{+2.00}$	63.50 $_{+1.50}$	60.00 $_{+7.50}$	58.91 $_{+6.20}$	51.06 $_{-6.39}$	43.75 $_{+12.50}$	47.83 $_{+0.00}$
	LAVE (b=32)	75.75 $_{-4.50}$	24.00 $_{-1.00}$	28.00 $_{-2.00}$	76.50 $_{-1.00}$	**70.00** $_{+8.00}$	**63.00** $_{+10.50}$	**60.47** $_{+7.76}$	57.34 $_{-0.11}$	**56.25** $_{+25.00}$	47.83 $_{+0.00}$
	EPIC	77.50 $_{-2.75}$	27.00 $_{+2.00}$	30.00 $_{+0.00}$	79.50 $_{+2.00}$	63.50 $_{+1.50}$	60.00 $_{+7.50}$	58.91 $_{+6.20}$	51.28 $_{-6.17}$	43.75 $_{+12.50}$	47.83 $_{+0.00}$
	PGSR	82.50 $_{+2.25}$	26.00 $_{+1.00}$	30.00 $_{+0.00}$	84.00 $_{+6.50}$	63.00 $_{+1.00}$	54.00 $_{+1.50}$	55.43 $_{+2.72}$	59.47 $_{+2.02}$	31.25 $_{+0.00}$	47.83 $_{+0.00}$
	PGSR + C-Diff	**87.50** $_{+7.25}$	**30.00** $_{+5.00}$	**32.00** $_{+2.00}$	**85.00** $_{+7.50}$	63.00 $_{+1.00}$	56.50 $_{+4.00}$	**60.47** $_{+7.76}$	**60.74** $_{+3.29}$	31.25 $_{+0.00}$	**60.87** $_{+13.04}$
	PGSR + EPIC	**87.50** $_{+7.25}$	**30.00** $_{+5.00}$	**32.00** $_{+2.00}$	**85.00** $_{+7.50}$	63.00 $_{+1.00}$	56.50 $_{+4.00}$	**60.47** $_{+7.76}$	**60.74** $_{+3.29}$	31.25 $_{+0.00}$	**60.87** $_{+13.04}$

4.2 Experimental Results

Tool Calling. Tables 1 and 2 report BFCL accuracy for LLaDA-8B and Dream-7B. On LLaDA, PGSR showed its largest gains at small denoising budgets. At $T = 4$, it raised simple Python accuracy from 71.00 to 80.75 and multiple-function accuracy from 63.00 to 76.50, while requiring at most two additional forward passes. PGSR at $T = 4$ also surpassed Raw at larger T values, indicating that targeted post-generation correction could recover task performance more efficiently than simply increasing T. As T increased, more initial outputs became syntactically valid, leaving fewer parser-detectable errors to correct. The remaining failures were increasingly dominated by semantic errors that did not trigger parser feedback and therefore provided no repair location for PGSR, explaining its reduced benefit at larger T.

On Dream, PGSR raised simple Python accuracy from 60.00 to 67.00 at $T = 8$. In contrast, C-Diff and EPIC reduced the same score to 20.75, showing that enforcing grammatical constraints throughout generation did not necessarily preserve task correctness. Restricting these constraints to the repair stage avoided this degradation and yielded additional gains. PGSR + C-Diff reached 71.50 at $T = 8$ and 87.50 at $T = 32$, outperforming Raw by 11.50 and 7.25 points, respectively. Overall, results across both backbones showed that PGSR was more consistently effective than constrained decoding alone and benefited further when grammatical constraints were confined to targeted repair.

Code Generation, Text-to-SQL, and Molecular Generation. Table 3 reports PGSR results across Python, SQL, and SMILES outputs. The method consistently enhanced downstream performance across these structured formats and denoising budgets. On MBPP, pass@1 increased by 4.7 to 5.8 points for LLaDA and by up to 9.3 points for Dream. Similar trends on MBPP+ showed that the repaired programs satisfied stricter functional tests rather than merely becoming syntactically valid. For molecular generation, PGSR raised LLaDA validity from 1.24% to 4.05% at $T = 16$ while reducing FCD from 34.02 to 22.72, indicating better validity while preserving similarity to the reference molecular distribution.

On Spider, PGSR also improved EX across all evaluated budgets, although the gains were relatively limited, particularly for Dream. For LLaDA, EX increased from 11.0% to 13.1% at $T = 32$, whereas Dream remained at low absolute EX despite consistent improvements after repair. This suggests that the benefit of structural repair is naturally limited when the underlying model itself struggles with the target task. Nevertheless, the consistent EX gains show that localized structural repair can still translate into downstream improvements even in this more challenging setting. Overall, the results showed that PGSR benefited not only parser acceptance but also functional correctness in code, EX in SQL, and distributional quality in molecular generation.

Table 3: Results before (Bef.) and after (Aft.) PGSR across code generation, text-to-SQL, and molecular generation. Subscripts show changes, and the best Aft. results are bolded. FCD is reported as N/A when too few valid molecules are available.

		Code Generation (pass@1 ↑)								Text-to-SQL		Molecular Generation			
		HumanEval		HumanEval+		MBPP		MBPP+		Spider EX ↑		Validity ↑		FCD ↓	
Model	**Steps**	**Bef.**	**Aft.**	**Bef.**	**Aft.**	**Bef.**	**Aft.**	**Bef.**	**Aft.**	**Bef.**	**Aft.**	**Bef.**	**Aft.**	**Bef.**	**Aft.**
LLaDA-8B	4	0.0	1.2$_{+1.2}$	0.0	1.2$_{+1.2}$	1.6	7.4$_{+5.8}$	1.3	6.3$_{+5.0}$	0.3	2.2$_{+1.9}$	0.02	0.06$_{+0.04}$	N/A	
	8	0.0	2.4$_{+2.4}$	0.0	2.4$_{+2.4}$	2.6	8.2$_{+5.6}$	2.6	7.4$_{+4.8}$	1.2	2.2$_{+1.0}$	0.00	0.26$_{+0.26}$	N/A	53.81
	16	0.0	0.0$_{+0.0}$	0.0	0.0$_{+0.0}$	7.7	12.4$_{+4.7}$	6.9	11.6$_{+4.7}$	2.7	3.9$_{+1.2}$	1.24	4.05$_{+2.81}$	34.02	**22.72**$_{-11.30}$
	32	1.8	4.3$_{+2.5}$	1.8	4.3$_{+2.5}$	14.8	20.4$_{+5.6}$	12.4	17.5$_{+5.1}$	11.0	13.1$_{+2.1}$	33.51	39.52$_{+6.01}$	25.92	24.01$_{-1.91}$
	64	13.4	**17.1**$_{+3.7}$	12.8	**16.5**$_{+3.7}$	29.1	**34.1**$_{+5.0}$	25.4	**29.1**$_{+3.7}$	19.4	**20.2**$_{+0.8}$	74.59	**77.05**$_{+2.46}$	26.79	26.24$_{-0.55}$
Dream-7B	4	0.6	1.8$_{+1.2}$	0.6	1.8$_{+1.2}$	3.2	6.9$_{+3.7}$	2.9	5.8$_{+2.9}$	0.0	0.8$_{+0.8}$	0.00	0.02$_{+0.02}$	N/A	
	8	1.2	4.3$_{+3.1}$	1.2	4.3$_{+3.1}$	1.1	8.5$_{+7.4}$	0.8	7.4$_{+6.6}$	0.0	0.8$_{+0.8}$	0.20	0.48$_{+0.28}$	54.89	40.30$_{-14.59}$
	16	0.0	3.7$_{+3.7}$	0.0	3.7$_{+3.7}$	1.3	8.7$_{+7.4}$	1.3	7.7$_{+6.4}$	0.3	0.6$_{+0.3}$	3.43	5.79$_{+2.36}$	30.40	24.35$_{-6.05}$
	32	6.7	11.0$_{+4.3}$	6.1	9.8$_{+3.7}$	12.7	22.0$_{+9.3}$	11.1	19.6$_{+8.5}$	0.2	0.8$_{+0.6}$	49.90	55.07$_{+5.17}$	23.15	22.03$_{-1.12}$
	64	14.6	**16.5**$_{+1.9}$	12.8	**13.4**$_{+0.6}$	24.9	**27.2**$_{+2.3}$	22.5	**24.3**$_{+1.8}$	2.0	**2.2**$_{+0.2}$	77.17	**78.50**$_{+1.33}$	20.51	**20.32**$_{-0.19}$

5 Analysis

5.1 Malformed Calls Slip Late and Fail to Close

Each row of Table 4 represents the first error reported by the JSON parser. Missing commas were the most frequent failure mode, accounting for 19.9% of all outputs at $T = 4$, nearly half of the 45.3% rejected outputs. Because parsing stops at the first failure, these categories indicate where each output first became invalid rather than all errors it may contain. Failures also tended to occur late, with median break positions ranging from 0.72 to 0.94 of the output length. This suggests that parsing failures often emerged only after most of the tool call had already been produced in a structurally valid form, highlighting the inefficiency of full regeneration commonly used in LLM-based agents. By selectively revising such localized failures, PGSR reduced every failure mode across all denoising budgets and lowered the overall rejection rate by roughly half at $T = 4$.

Table 4: Malformed JSON tool calls from LLaDA-8B before and after PGSR, grouped by parser message. Percentages are over all outputs from ten BFCL AST categories. Break is the normalized parser offset pooled over T, with bars showing the interquartile range and notches the median.

		Rejected (%) at T					Break
Failure mode		4	8	16	32	64	0 1
Missing comma	Raw	19.9	17.7	12.6	6.9	4.3	
	PGSR	10.7	9.0	6.1	2.9	2.0	
Missing colon	Raw	9.4	6.9	3.7	1.7	0.8	
	PGSR	5.5	3.9	1.6	0.6	0.3	
Unterminated	Raw	7.6	6.3	5.9	3.2	1.3	
	PGSR	2.1	1.7	1.6	0.7	0.1	
Malformed key	Raw	4.5	3.9	4.0	2.4	0.6	
	PGSR	1.9	1.4	1.5	0.8	0.3	
Missing value	Raw	3.9	3.4	2.2	1.1	0.5	
	PGSR	2.3	2.1	1.3	0.6	0.3	
All rejected	*Raw*	*45.3*	*38.2*	*28.4*	*15.4*	*7.6*	
	PGSR	*22.6*	*18.1*	*12.1*	*5.6*	*3.0*	

5.2 Repair Pays per Failure, Constrained Decoding Pays per Step

Figure 3 reports wall-clock latency for Raw, the two constrained decoding baselines, and PGSR. LAVE was excluded because its semi-autoregressive decoding made direct latency comparison difficult. C-Diff and EPIC ran 1.1 to 5.4 times slower than Raw, although this relative gap decreased as the denoising budget increased. However, larger denoising budgets inherently diminish the efficiency benefit of DLMs, making low-step decoding the more practically relevant setting. In this regime, PGSR maintained latency close to that of Raw because additional computation was incurred only when repair was required. As detailed in Appendix B, PGSR introduced an overhead of only 8.0% at $T = 4$ and 0.08% at $T = 64$. Overall, PGSR largely preserves the inference efficiency of the underlying DLM while providing selective structural repair.

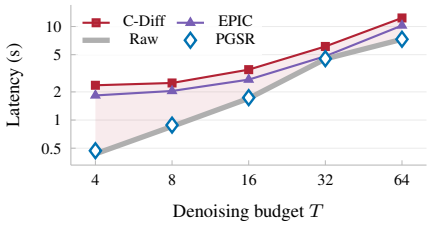

Figure 3: Mean latency on 400 BFCL non-live simple Python requests. The band indicates grammar-enforcement overhead.

Table 5: Two BFCL requests and what each method emits on them, LLaDA-8B at $T = 16$. Red marks what is wrong in an output and green what PGSR regenerated in its place. A tinted character is the parser-reported offset that the repair window is centred on, and . . . elides fields the rows of a block share. EPIC is omitted because it emits the same strings as C-Diff.

Method	Verdict	Emitted tool call
"I'm thinking about finding a salon in Berkeley" LIVE_MULTIPLE		
Raw	× parser rejects	`[{... "arguments": {"city": "Berkeley, CA", "is_un`
C-Diff	× argument invented	`[{... "arguments": {"city": "Berkeley, CA", "is_unisex": true}}]`
LAVE	× value corrupted	`[{... "arguments": {"city": "Ber,, CA", "is_unisex": true}}]`
PGSR	✓ accepted	`[{... "arguments": {"city": "Berkeley, CA"}}]`
"Find lawyers specializing in criminal law near me in New York." MULTIPLE		
Raw	× parser rejects	`[{... "specialization": ["Criminalriminal"]"]]`
C-Diff	× value not a word	`[{... "specialization": ["Criminalriminal"]}}]`
LAVE	× 16 calls, 1 expected	`[{... "specialization": ["criminal", " "]}}, ... 15 more identical calls]`
PGSR	✓ accepted	`[{... "specialization": ["Criminal law"]}}]`

5.3 Qualitative Analysis of Localized Repair

Table 5 presents two BFCL examples illustrating the different behaviors of constrained decoding and PGSR. Although C-Diff and LAVE produced syntactically valid outputs, they introduced incorrect arguments, corrupted values, or redundant tool calls, showing that grammatical validity did not ensure functional correctness. In contrast, PGSR regenerated a local window around the parser-reported offset, removing the truncated argument in the first example and recovering `"Criminal law"` in the second. While PGSR did not explicitly target semantic errors, contextual infilling could also restore corrupted content when it fell within a syntax-triggered repair window.

5.4 Sensitivity to the Repair Hyperparameters

Table 6 varies one hyperparameter at a time from the default setting of $w = 5$, $R = 2$, and $T_{\text{rep}} = 1$. All twelve settings improved the parse rate over unrepaired decoding by at least 6.9 percentage points, demonstrating that PGSR was robust to its hyperparameter choices. For the repair window, $w = 1$ was too narrow to revise malformed keys or values effectively, whereas moderate windows performed better, with $w = 10$ achieving the highest overall AST accuracy. Increasing R or T_{rep} further improved parsing success but yielded only marginal changes in AST accuracy, indicating that additional repair computation mainly resolved syntactic errors. Notably, two rounds with $T_{\text{rep}} = 1$ achieved a parse rate of 92.44%, exceeding the 91.12% obtained with one round using $T_{\text{rep}} = 4$, highlighting the benefit of repeated parser feedback.

Table 6: PGSR sensitivity at $T = 16$ on four BFCL AST categories. Shading marks the default and Parse denotes parsing success.

	AST accuracy (%)				
Setting	**Py**	**Mult.**	**Par.**	**P.M.**	**Parse**
Raw (no repair)	73.50	67.50	76.00	63.50	79.81
Default	81.25	77.00	77.50	67.00	92.44
Window w					
$w = 1$	79.25	74.00	76.50	64.50	86.69
$w = 3$	81.25	76.00	77.50	67.00	91.62
$w = 10$	**82.75**	**79.00**	78.00	**68.50**	93.19
$w = 20$	81.00	77.00	**78.50**	**68.50**	92.81
Rounds R					
$R = 1$	80.00	77.00	77.50	66.50	89.94
$R = 3$	82.00	77.50	77.50	66.50	93.56
$R = 4$	82.00	77.50	77.50	66.50	93.81
Repair steps T_{rep}					
$T_{\text{rep}} = 2$	82.25	77.00	77.50	66.50	92.81
$T_{\text{rep}} = 4$	82.50	77.00	77.50	67.00	**93.94**
One round, more steps					
R=1, T_{rep}=2	81.25	77.00	77.50	66.50	90.44
R=1, T_{rep}=4	81.50	76.50	77.50	67.00	91.12

6 Conclusion

We introduced PGSR, a training-free framework that uses parser feedback to localize and regenerate malformed regions in DLM outputs. Experiments across four structured generation domains show that PGSR consistently improves structural validity and downstream performance, particularly under limited denoising budgets. Its low repair overhead and compatibility with constrained decoding make PGSR a practical approach for improving the reliability of DLM-based structured generation.

AI USE STATEMENT

No generative AI tools were used in the preparation of this manuscript.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Tiwei Bie, Jiansheng Cai, Maosong Cao, Xiang Cao, Binbin Chen, Fuyuan Chen, Kun Chen, Yuqi Ding, Lun Du, Zhuocheng Gong, et al. Llada2. 2: Enabling agentic diffusion language models via levenshtein editing. Technical report, CSPaper OpenPrint, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Junyan Cheng, Ankit Srivastava, Jessie Zeng, Milenko Drinic, and Jack W Stokes. Apeiron: A scalable llm-agentic framework for autonomous full-lifecycle demand-optimized application synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 3868–3899, 2026.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

Weihua Du, Zhan Ling, Kang Liu, Lingfeng Shen, Xuesong Yao, Yufei Xu, Dingyuan Shi, Yiming Yang, and Jiecao Chen. Generalizable end-to-end tool-use rl with synthetic codegym. In *International Conference on Learning Representations*, volume 2026, pp. 17733–17756, 2026.

Sarel Duanis, Asnat Greenstein-Messica, and Eliya Habba. Json whisperer: Efficient json editing with llms. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1265–1274, 2025.

Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268, 2018.

Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi Gu, Hui Su, and Xunliang Cai. Vitabench: Benchmarking llm agents with versatile interactive tasks in real-world applications. In *International Conference on Learning Representations*, volume 2026, pp. 91654–91681, 2026.

Hyundong Jin and Yo-Sub Han. Epic: Efficient and parallel inference under cfg constraints for diffusion language models. *arXiv preprint arXiv:2606.00722*, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Qingyu Lu, Liang Ding, Kanjian Zhang, Jinxia Zhang, and Dacheng Tao. The bitter lesson of diffusion language models for agentic workflows: A comprehensive reality check. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 43997–44020, 2026.

Niels Mündler, Jasper Dekoninck, and Martin Vechev. Constrained decoding of diffusion llms with context-free grammars. In *International Conference on Learning Representations*, volume 2026, pp. 99362–99408, 2026.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *Advances in Neural Information Processing Systems*, 38:50608–50646, 2026.

Sangeun Park and Minhae Kwon. Multi˙2: Hierarchical multi-agent decision-making with llm-based agents in interactive environments. *arXiv preprint arXiv:2606.03698*, 2026.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Kristina Preuer, Philipp Renz, Thomas Unterthiner, Sepp Hochreiter, and Gunter Klambauer. Fréchet chemnet distance: a metric for generative models for molecules in drug discovery. *Journal of chemical information and modeling*, 58(9):1736–1741, 2018.

Houxing Ren, Mingjie Zhan, Zimu Lu, Ke Wang, Yunqiao Yang, Haotian Hou, Junting Pan, and Hongsheng Li. Edit-based refinement for parallel masked diffusion language models. *arXiv preprint arXiv:2605.09603*, 2026.

Subham 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. *Advances in Neural Information Processing Systems*, 37:130136–130184, 2024.

Jiaxuan Wang, Yulan Hu, Wenjin Yang, Zheng Pan, Xin Li, and Lan-Zhe Guo. Aligning agents via planning: A benchmark for trajectory-level reward modeling. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23174–23200, 2026.

Frank Fangzheng Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. Theagentcompany: benchmarking llm agents on consequential real world tasks. *Advances in Neural Information Processing Systems*, 38, 2026.

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

Guoli Yin, Haoping Bai, Shuang Ma, Feng Nan, Yanchao Sun, Zhaoyang Xu, Shen Ma, Jiarui Lu, Xiang Kong, Aonan Zhang, et al. Mmau: A holistic benchmark of agent capabilities across diverse domains. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 4752–4780, 2025.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 3911–3921, 2018.

Yitong Zhang, Yongmin 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. *arXiv preprint arXiv:2602.00612*, 2026.

Jiahao Zhao, Shaoxuan Xu, Zhongxiang Sun, Fengqi Zhu, Jingyang Ou, Yuling Shi, Chongxuan Li, Xiao Zhang, and Jun Xu. Dllm-searcher: Adapting diffusion large language model for search agents. *arXiv preprint arXiv:2602.07035*, 2026.

Huiling Zhen, Weizhe Lin, Renxi Liu, Kai Han, Yiming Li, Yuchuan Tian, Hanting Chen, Xiaoguang Li, Xiaosong Li, Chen Chen, et al. Dllm agent: See farther, run faster. *arXiv preprint arXiv:2602.07451*, 2026.

Chang Zong, Yuchen Yan, Weiming Lu, Jian Shao, Yongfeng Huang, Heng Chang, and Yueting Zhuang. Triad: A framework leveraging a multi-role llm-based agent to solve knowledge base question answering. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 1698–1710, 2024.

Hao Zou, Zachary Horvitz, Chandhru Karthick, Zhou Yu, and Kathleen McKeown. Detect, remask, repair: Diffusion editing for faithful summarization of evolving contexts. *arXiv preprint arXiv:2606.12807*, 2026.

A PGSR ALGORITHM

Algorithm 1 summarizes the complete PGSR procedure described in §3.3. Following the initial T-step generation, the parser validates the output and, when possible, returns the character offset at which parsing fails. Using the parser-reported offset as a token-level anchor, PGSR defines a local repair window around the identified position. The selected span is then remasked and reconstructed through DLM infilling. The updated output is then revalidated, and the locate–mask–regenerate cycle is repeated until the parser accepts the sequence or the maximum number of repair rounds R is reached. When a compatible decoding constraint $\mathcal{G}$ is available, the same procedure uses constrained infilling instead of standard infilling, with $\mathcal{G}$ enforced only within the masked repair region.

Algorithm 1: PGSR

Input : DLM p_θ; prompt $\mathbf{p}$; parser $\mathcal{P}$; window w; rounds R; steps T, T_{rep}; optional decoding constraint $\mathcal{G}$

Output: Repaired output token sequence

$\mathbf{y}^{(0)} \leftarrow \textsc{Sample}(p_\theta, \mathbf{p}, T)$ *// Initial generation*

for $r \leftarrow 1$ **to** R **do**

  $c^{(r)} \leftarrow \mathcal{P}(\text{decode}(\mathbf{y}^{(r-1)}))$ *// Locate the first violation*

  if $c^{(r)} = \varnothing$ **then return** $\mathbf{y}^{(r-1)}$ *// Accepted, or no usable offset*

  $e^{(r)} \leftarrow \max\{i : \text{cs}(y_i^{(r-1)}) \le c^{(r)}\}$ *// Map character offset to token index*

  $\mathcal{M}^{(r)} \leftarrow \{i : |i - e^{(r)}| \le w\} \cap \{1, \dots, N\}$

  $\tilde{\mathbf{y}}^{(r)} \leftarrow \textsc{Mask}(\mathbf{y}^{(r-1)}, \mathcal{M}^{(r)})$ *// Remask the local window*

  if $\mathcal{G}$ *is provided* **then**

    $\mathbf{y}^{(r)} \leftarrow \text{CD-}\textsc{Infill}(p_\theta, [\mathbf{p}; \tilde{\mathbf{y}}^{(r)}; \texttt{EOT}], T_{\text{rep}}, \mathcal{G})$

  else

    $\mathbf{y}^{(r)} \leftarrow \textsc{Infill}(p_\theta, [\mathbf{p}; \tilde{\mathbf{y}}^{(r)}; \texttt{EOT}], T_{\text{rep}})$

  end

end

return $\mathbf{y}^{(R)}$

B REPAIR OVERHEAD

Figure 4 further decomposes the latency difference between Raw and PGSR shown in Figure 3. The average latency introduced by repair remained small across denoising budgets, ranging from approximately 26 to 35 ms for $T \in \{4, 8, 16, 32\}$ and decreasing to only 6 ms at $T = 64$. This limited overhead follows from the adaptive nature of PGSR: additional computation is performed only when the initial output is rejected by the parser and a usable failure location is available, while successfully parsed outputs incur no repair cost. Moreover, each repair round uses only $T_{\text{rep}} = 1$ denoising step over a localized span rather than repeating full-sequence generation. As a result, the relative overhead over Raw decreases from 8.0% at $T = 4$ to 0.08% at $T = 64$. This behavior is consistent with the adaptive computation characterized in equation 6, where additional forward passes are allocated selectively to outputs requiring post-generation refinement.

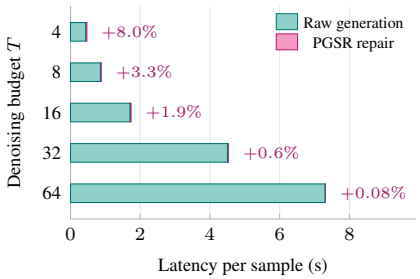

Figure 4: Latency decomposition of PGSR under the same setting as Figure 3. Bars separate Raw and repair latency, with labels showing the relative overhead over Raw.

Table 7: Comparison of PGSR with additional denoising steps on code generation, text-to-SQL, and molecular generation. $T + 2$ applies two additional denoising steps to every sample, whereas PGSR uses up to two repair steps only when repair is triggered. Bold indicates the best value for each model and metric. FCD is reported as N/A when too few valid molecules are available for estimation.

		Code Generation (pass@1 ↑)												**Text-to-SQL**			**Molecular Generation**					
		HumanEval			HumanEval+			MBPP			MBPP+			Spider EX ↑			Validity ↑			FCD ↓		
Model	T	**Raw**	$T+2$	**PGSR**	**Raw**	$T+2$	**PGSR**	**Raw**	$T+2$	**PGSR**	**Raw**	$T+2$	**PGSR**	**Raw**	$T+2$	**PGSR**	**Raw**	$T+2$	**PGSR**	**Raw**	$T+2$	**PGSR**
LLaDA-8B	4	0.0	0.0	1.2	0.0	0.0	1.2	1.6	1.9	7.4	1.3	1.9	6.3	0.3	0.1	2.2	0.02	0.00	0.06		N/A	
	8	0.0	0.0	2.4	0.0	0.0	2.4	2.6	3.7	8.2	2.6	3.7	7.4	1.2	1.1	2.2	0.00	0.02	0.26	N/A	N/A	53.81
	16	0.0	0.6	0.0	0.0	0.6	0.0	7.7	8.2	12.4	6.9	7.1	11.6	2.7	4.1	3.9	1.24	2.77	4.05	34.02	42.82	**22.72**
	32	1.8	3.0	4.3	1.8	3.0	4.3	14.8	18.5	20.4	12.4	15.1	17.5	11.0	12.2	13.1	33.51	36.31	39.52	25.92	24.97	24.01
	64	13.4	11.6	**17.1**	12.8	11.0	**16.5**	29.1	29.1	**34.1**	25.4	25.1	**29.1**	19.4	**22.0**	20.2	74.59	**79.04**	77.05	26.79	26.59	26.24
Dream-7B	4	0.6	0.6	1.8	0.6	0.6	1.8	3.2	2.1	6.9	2.9	1.9	5.8	0.0	0.0	0.8	0.00	0.04	0.02		N/A	
	8	1.2	1.2	4.3	1.2	1.2	4.3	1.1	1.1	8.5	0.8	1.1	7.4	0.0	0.1	0.8	0.20	0.74	0.48	54.89	45.31	40.30
	16	0.0	2.4	3.7	0.0	2.4	3.7	1.3	1.1	8.7	1.3	1.1	7.7	0.3	0.0	0.6	3.43	7.56	5.79	30.40	24.49	24.35
	32	6.7	7.3	11.0	6.1	6.1	9.8	12.7	9.8	22.0	11.1	8.7	19.6	0.2	0.2	0.8	49.90	45.95	55.07	23.15	20.61	22.03
	64	14.6	**18.9**	16.5	12.8	**16.5**	13.4	24.9	21.4	**27.2**	22.5	19.0	**24.3**	2.0	**2.2**	**2.2**	77.17	**79.46**	78.50	20.51	**19.71**	20.32

C Targeted Repair vs. Additional Denoising Steps

To determine whether PGSR's gains arise from targeted repair rather than simply increasing the denoising budget, we compared it with additional generation steps. In Table 7, Raw denotes unrepaired generation with T steps, $T + 2$ adds two denoising steps to every sample, and PGSR uses up to two repair steps only when the initial output is rejected with a usable failure location. Despite this selective allocation of additional computation, PGSR outperformed $T + 2$ in 53 of the 67 comparable cells, whereas $T + 2$ performed better in 13 and the two methods achieved identical performance in one cell.

The difference was most pronounced at small denoising budgets, with PGSR outperforming $T + 2$ in 23 of 25 comparable cases for $T \leq 8$. The gap narrowed as T increased because larger denoising budgets already reduced structural errors in the initial outputs, leaving fewer failures to repair. At $T = 64$, seven cells favored PGSR and six favored $T + 2$, while one showed identical performance. A notable exception was SMILES validity for Dream-7B, where $T + 2$ performed better at four of the five budgets. As discussed in Appendix G, RDKit does not provide a usable failure offset for some invalid molecules, such as those involving valence errors, in which case PGSR leaves the output unchanged.

D PGSR after Constrained Decoding

The combined variants in the main text apply grammar constraints only during local regeneration. Here, we examined whether PGSR could also be applied after constrained decoding had already been performed. Specifically, C-Diff or EPIC was first used to produce the output, after which PGSR was applied only when the result remained unparsable. Table 8 reports this setting for C-Diff and EPIC. LAVE is omitted because a failed lookahead can return an empty output, leaving no span for PGSR to repair. For the remaining malformed outputs produced by LAVE, repair did not recover any additional parsable outputs.

Applying PGSR after constrained decoding improved 117 of the 200 reported cells and left the remaining cells unchanged, with an average gain of 1.84 percentage points. These results demonstrate that PGSR can provide additional improvements even after grammatical constraints have already been enforced during decoding. The average gain decreased from 2.57 points at $T = 8$ to 0.72 points at $T = 64$, as larger denoising budgets left fewer unparsable outputs for subsequent repair. This behavior is consistent with PGSR's selective repair mechanism, which leaves outputs unchanged once they are accepted by the parser.

Table 8: Repairing the output of a constrained decoder, BFCL accuracy (%). Each shaded row applies PGSR to the generations of the row above it, and the subscript is the gain or drop against that decoder without repair. Compare with the opposite composition, PGSR + C-Diff and PGSR + EPIC in Tables 1 and 2, which constrains only the repair window.

Steps	Method	Non-Live						Live			
		Py	Java	JS	Mult.	Par.	P.M.	L.Simp.	L.Mult.	L.Par.	L.P.M.
LLaDA-8B											
4	C-Diff	71.25	21.00	26.00	63.00	56.00	26.00	40.70	35.29	31.25	15.00
	C-Diff + PGSR	73.00 +1.75	21.00 +0.00	26.00 +0.00	69.00 +6.00	56.00 +0.00	27.50 +1.50	43.02 +2.33	40.48 +5.19	31.25 +0.00	15.00 +0.00
	EPIC	71.25	21.00	26.00	63.00	56.00	26.00	40.70	35.64	31.25	15.00
	EPIC + PGSR	73.00 +1.75	21.00 +0.00	26.00 +0.00	69.00 +6.00	56.00 +0.00	27.50 +1.50	43.02 +2.33	41.41 +5.77	31.25 +0.00	15.00 +0.00
8	C-Diff	76.75	27.00	28.00	70.50	71.50	49.50	50.00	40.60	31.25	30.00
	C-Diff + PGSR	79.25 +2.50	27.00 +0.00	28.00 +0.00	74.00 +3.50	71.50 +0.00	50.50 +1.00	51.55 +1.55	44.29 +3.69	37.50 +6.25	30.00 +0.00
	EPIC	76.75	27.00	28.00	70.50	71.50	50.50	50.00	41.64	31.25	30.00
	EPIC + PGSR	79.25 +2.50	27.00 +0.00	28.00 +0.00	74.00 +3.50	71.50 +0.00	51.50 +1.00	51.55 +1.55	45.67 +4.04	37.50 +6.25	30.00 +0.00
16	C-Diff	77.25	28.00	28.00	74.00	79.50	61.50	54.65	41.98	50.00	30.00
	C-Diff + PGSR	80.00 +2.75	28.00 +0.00	28.00 +0.00	78.00 +4.00	79.50 +0.00	62.50 +1.00	55.81 +1.16	46.02 +4.04	50.00 +0.00	30.00 +0.00
	EPIC	77.25	28.00	28.00	74.00	79.50	64.50	54.65	44.52	50.00	30.00
	EPIC + PGSR	80.00 +2.75	28.00 +0.00	28.00 +0.00	78.00 +4.00	79.50 +0.00	65.50 +1.00	55.81 +1.16	48.67 +4.15	50.00 +0.00	30.00 +0.00
32	C-Diff	81.00	33.00	26.00	79.00	80.50	61.00	58.53	42.68	50.00	40.00
	C-Diff + PGSR	82.75 +1.75	33.00 +0.00	28.00 +2.00	81.50 +2.50	80.50 +0.00	61.00 +0.00	60.85 +2.33	46.25 +3.58	56.25 +6.25	40.00 +0.00
	EPIC	81.00	33.00	26.00	79.00	81.00	70.00	58.53	46.25	50.00	45.00
	EPIC + PGSR	82.75 +1.75	33.00 +0.00	28.00 +2.00	81.50 +2.50	81.00 +0.00	70.00 +0.00	60.85 +2.33	50.29 +4.04	56.25 +6.25	45.00 +0.00
64	C-Diff	83.75	30.00	22.00	81.50	79.50	66.00	61.24	36.68	43.75	45.00
	C-Diff + PGSR	85.50 +1.75	30.00 +0.00	22.00 +0.00	83.00 +1.50	79.50 +0.00	66.00 +0.00	62.40 +1.16	37.95 +1.27	43.75 +0.00	45.00 +0.00
	EPIC	83.75	30.00	22.00	82.00	80.00	70.50	61.24	45.10	43.75	45.00
	EPIC + PGSR	85.50 +1.75	30.00 +0.00	22.00 +0.00	83.50 +1.50	80.00 +0.00	70.50 +0.00	62.40 +1.16	47.06 +1.96	43.75 +0.00	45.00 +0.00
Dream-7B											
4	C-Diff	20.50	2.00	6.00	18.00	0.00	0.00	20.16	10.85	0.00	0.00
	C-Diff + PGSR	28.50 +8.00	4.00 +2.00	6.00 +0.00	27.00 +9.00	0.00 +0.00	0.00 +0.00	24.81 +4.65	13.30 +2.45	0.00 +0.00	0.00 +0.00
	EPIC	20.50	2.00	6.00	18.00	0.00	0.00	20.16	10.85	0.00	0.00
	EPIC + PGSR	28.50 +8.00	4.00 +2.00	6.00 +0.00	26.50 +8.50	0.00 +0.00	0.00 +0.00	24.42 +4.26	13.40 +2.55	0.00 +0.00	0.00 +0.00
8	C-Diff	20.75	2.00	6.00	19.00	1.50	1.00	20.16	10.96	0.00	0.00
	C-Diff + PGSR	29.00 +8.25	3.00 +1.00	6.00 +0.00	27.00 +8.00	3.50 +2.00	2.00 +1.00	25.19 +5.04	13.19 +2.23	0.00 +0.00	4.35 +4.35
	EPIC	20.75	2.00	6.00	19.00	1.50	1.00	20.16	10.96	0.00	0.00
	EPIC + PGSR	29.00 +8.25	4.00 +2.00	6.00 +0.00	27.00 +8.00	3.50 +2.00	2.50 +1.50	25.19 +5.04	13.30 +2.34	0.00 +0.00	4.35 +4.35
16	C-Diff	73.25	18.00	24.00	73.00	27.00	28.50	54.65	45.85	25.00	34.78
	C-Diff + PGSR	75.00 +1.75	18.00 +0.00	24.00 +0.00	74.00 +1.00	33.00 +6.00	34.00 +5.50	56.20 +1.55	48.40 +2.55	31.25 +6.25	34.78 +0.00
	EPIC	73.25	18.00	24.00	73.00	27.00	28.50	54.65	45.85	25.00	34.78
	EPIC + PGSR	75.00 +1.75	18.00 +0.00	24.00 +0.00	74.00 +1.00	33.00 +6.00	34.00 +5.50	56.20 +1.55	48.72 +2.87	31.25 +6.25	39.13 +4.35
32	C-Diff	77.50	27.00	30.00	79.50	63.50	60.00	58.91	51.06	43.75	47.83
	C-Diff + PGSR	78.50 +1.00	27.00 +0.00	30.00 +0.00	81.00 +1.50	69.00 +5.50	66.50 +6.50	59.30 +0.39	52.34 +1.28	43.75 +0.00	47.83 +0.00
	EPIC	77.50	27.00	30.00	79.50	63.50	60.00	58.91	51.28	43.75	47.83
	EPIC + PGSR	78.50 +1.00	27.00 +0.00	30.00 +0.00	81.00 +1.50	69.50 +6.00	66.00 +6.00	59.30 +0.39	52.98 +1.70	43.75 +0.00	47.83 +0.00
64	C-Diff	84.75	28.00	30.00	87.00	81.00	73.50	63.18	55.00	43.75	60.87
	C-Diff + PGSR	85.00 +0.25	28.00 +0.00	30.00 +0.00	88.00 +1.00	82.00 +1.00	78.50 +5.00	63.57 +0.39	55.96 +0.96	43.75 +0.00	60.87 +0.00
	EPIC	84.75	28.00	30.00	87.00	81.00	73.50	63.18	55.00	43.75	60.87
	EPIC + PGSR	85.00 +0.25	28.00 +0.00	30.00 +0.00	88.00 +1.00	82.00 +1.00	78.00 +4.50	63.57 +0.39	56.17 +1.17	43.75 +0.00	60.87 +0.00

E FULL BFCL RESULTS FOR DREAM-7B

Table 2 in the main text reports Dream-7B at $T \in \{8, 16, 32\}$. Table 9 extends the evaluation to $T \in \{4, 8, 16, 32, 64\}$ across all ten BFCL AST categories.

The additional budgets followed the trends observed in the main text. At $T = 4$, C-Diff and EPIC performed below Raw across all ten categories, including a drop in simple Python accuracy from 41.25 to 20.50. In contrast, PGSR improved the same category to 52.75, while PGSR + C-Diff reached 56.50. The gains became smaller at $T = 64$, where the initial outputs were more structurally reliable. Averaged across the ten categories, PGSR improved over Raw by 3.99 points at $T = 4$ and 1.28 points at $T = 64$, while PGSR + C-Diff improved by 5.69 and 2.51 points, respectively. Across all 50 step–category cells, PGSR never reduced BFCL accuracy relative to Raw.

Table 9: BFCL accuracy (%) of Dream-7B across the full sweep $T \in \{4, 8, 16, 32, 64\}$. Subscripts denote the gain or drop over Raw at the same step. Best per category per step in **bold**. The LAVE block length b follows the main text and is chosen per budget.

Steps	Method	Non-Live						Live			
		Py	Java	JS	Mult.	Par.	P.M.	L.Simp.	L.Mult.	L.Par.	L.P.M.
4	Raw	41.25	12.00	24.00	44.50	7.00	2.50	33.33	35.32	**6.25**	4.35
	C-Diff	20.50 $_{-20.75}$	2.00 $_{-10.00}$	6.00 $_{-18.00}$	18.00 $_{-26.50}$	0.00 $_{-7.00}$	0.00 $_{-2.50}$	20.16 $_{-13.17}$	10.85 $_{-24.47}$	0.00 $_{-6.25}$	0.00 $_{-4.35}$
	LAVE (b=128)	47.00 $_{+5.75}$	10.00 $_{-2.00}$	18.00 $_{-6.00}$	46.00 $_{+1.50}$	7.00 $_{+0.00}$	4.50 $_{+2.00}$	40.31 $_{+6.98}$	30.85 $_{-4.47}$	0.00 $_{-6.25}$	**8.70** $_{+4.35}$
	EPIC	20.50 $_{-20.75}$	2.00 $_{-10.00}$	6.00 $_{-18.00}$	18.00 $_{-26.50}$	0.00 $_{-7.00}$	0.00 $_{-2.50}$	20.16 $_{-13.17}$	10.85 $_{-24.47}$	0.00 $_{-6.25}$	0.00 $_{-4.35}$
	PGSR	52.75 $_{+11.5}$	**16.00** $_{+4.00}$	24.00 $_{+0.00}$	54.00 $_{+9.50}$	**9.00** $_{+2.00}$	**5.00** $_{+2.50}$	38.76 $_{+5.43}$	40.32 $_{+5.00}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
	PGSR + C-Diff	**56.50** $_{+15.2}$	**16.00** $_{+4.00}$	**26.00** $_{+2.00}$	**61.50** $_{+17.0}$	8.50 $_{+1.50}$	3.50 $_{+1.00}$	**43.41** $_{+10.08}$	**41.38** $_{+6.06}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
	PGSR + EPIC	**56.50** $_{+15.2}$	**16.00** $_{+4.00}$	**26.00** $_{+2.00}$	**61.50** $_{+17.0}$	8.50 $_{+1.50}$	3.50 $_{+1.00}$	**43.41** $_{+10.08}$	**41.38** $_{+6.06}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
8	Raw	60.00	16.00	24.00	55.00	16.00	3.00	40.31	41.38	**6.25**	4.35
	C-Diff	20.75 $_{-39.25}$	2.00 $_{-14.00}$	6.00 $_{-18.00}$	19.00 $_{-36.00}$	1.50 $_{-14.50}$	1.00 $_{-2.00}$	20.16 $_{-20.15}$	10.96 $_{-30.42}$	0.00 $_{-6.25}$	0.00 $_{-4.35}$
	LAVE (b=64)	46.00 $_{-14.00}$	10.00 $_{-6.00}$	18.00 $_{-6.00}$	46.00 $_{-9.00}$	16.00 $_{+0.00}$	**8.50** $_{+5.50}$	39.53 $_{-0.78}$	31.91 $_{-9.47}$	0.00 $_{-6.25}$	**13.04** $_{+8.69}$
	EPIC	20.75 $_{-39.25}$	2.00 $_{-14.00}$	6.00 $_{-18.00}$	19.00 $_{-36.00}$	1.50 $_{-14.50}$	1.00 $_{-2.00}$	20.16 $_{-20.15}$	10.96 $_{-30.42}$	0.00 $_{-6.25}$	0.00 $_{-4.35}$
	PGSR	67.00 $_{+7.00}$	20.00 $_{+4.00}$	**26.00** $_{+2.00}$	63.00 $_{+8.00}$	**19.50** $_{+3.50}$	7.00 $_{+4.00}$	44.57 $_{+4.26}$	46.91 $_{+5.53}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
	PGSR + C-Diff	**71.50** $_{+11.5}$	**23.00** $_{+7.00}$	**26.00** $_{+2.00}$	**69.00** $_{+14.0}$	**19.50** $_{+3.50}$	5.00 $_{+2.00}$	**50.00** $_{+9.69}$	**48.40** $_{+7.02}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
	PGSR + EPIC	**71.50** $_{+11.5}$	**23.00** $_{+7.00}$	**26.00** $_{+2.00}$	**69.00** $_{+14.0}$	**19.50** $_{+3.50}$	5.00 $_{+2.00}$	**50.00** $_{+9.69}$	**48.40** $_{+7.02}$	**6.25** $_{+0.00}$	4.35 $_{+0.00}$
16	Raw	70.00	14.00	26.00	59.50	24.00	14.50	45.74	45.96	25.00	13.04
	C-Diff	73.25 $_{+3.25}$	18.00 $_{+4.00}$	24.00 $_{-2.00}$	**73.00** $_{+13.50}$	27.00 $_{+3.00}$	**28.50** $_{+14.00}$	**54.65** $_{+8.91}$	45.85 $_{-0.11}$	25.00 $_{+0.00}$	34.78 $_{+21.74}$
	LAVE (b=32)	55.25 $_{-14.75}$	6.00 $_{-8.00}$	14.00 $_{-12.00}$	50.50 $_{-9.00}$	27.00 $_{+3.00}$	20.00 $_{+5.50}$	41.86 $_{-3.88}$	35.11 $_{-10.85}$	**43.75** $_{+18.75}$	**39.13** $_{+26.09}$
	EPIC	73.25 $_{+3.25}$	18.00 $_{+4.00}$	24.00 $_{-2.00}$	**73.00** $_{+13.50}$	27.00 $_{+3.00}$	**28.50** $_{+14.00}$	**54.65** $_{+8.91}$	45.85 $_{-0.11}$	25.00 $_{+0.00}$	34.78 $_{+21.74}$
	PGSR	76.25 $_{+6.25}$	15.00 $_{+1.00}$	28.00 $_{+2.00}$	68.50 $_{+9.00}$	**28.00** $_{+4.00}$	17.00 $_{+2.50}$	49.22 $_{+3.48}$	52.02 $_{+6.06}$	25.00 $_{+0.00}$	13.04 $_{+0.00}$
	PGSR + C-Diff	**78.75** $_{+8.75}$	**25.00** $_{+11.0}$	**30.00** $_{+4.00}$	72.50 $_{+13.0}$	27.50 $_{+3.50}$	18.00 $_{+3.50}$	52.71 $_{+6.97}$	**53.40** $_{+7.44}$	25.00 $_{+0.00}$	21.74 $_{+8.70}$
	PGSR + EPIC	**78.75** $_{+8.75}$	**25.00** $_{+11.0}$	**30.00** $_{+4.00}$	72.50 $_{+13.0}$	27.50 $_{+3.50}$	18.00 $_{+3.50}$	52.71 $_{+6.97}$	**53.40** $_{+7.44}$	25.00 $_{+0.00}$	21.74 $_{+8.70}$
32	Raw	80.25	25.00	30.00	77.50	62.00	52.50	52.71	57.45	31.25	47.83
	C-Diff	77.50 $_{-2.75}$	27.00 $_{+2.00}$	30.00 $_{+0.00}$	79.50 $_{+2.00}$	63.50 $_{+1.50}$	60.00 $_{+7.50}$	58.91 $_{+6.20}$	51.06 $_{-6.39}$	43.75 $_{+12.50}$	47.83 $_{+0.00}$
	LAVE (b=32)	75.75 $_{-4.50}$	24.00 $_{-1.00}$	28.00 $_{-2.00}$	76.50 $_{-1.00}$	**70.00** $_{+8.00}$	**63.00** $_{+10.50}$	**60.47** $_{+7.76}$	57.34 $_{-0.11}$	**56.25** $_{+25.00}$	47.83 $_{+0.00}$
	EPIC	77.50 $_{-2.75}$	27.00 $_{+2.00}$	30.00 $_{+0.00}$	79.50 $_{+2.00}$	63.50 $_{+1.50}$	60.00 $_{+7.50}$	58.91 $_{+6.20}$	51.28 $_{-6.17}$	43.75 $_{+12.50}$	47.83 $_{+0.00}$
	PGSR	82.50 $_{+2.25}$	26.00 $_{+1.00}$	30.00 $_{+0.00}$	84.00 $_{+6.50}$	63.00 $_{+1.00}$	54.00 $_{+1.50}$	55.43 $_{+2.72}$	59.47 $_{+2.02}$	31.25 $_{+0.00}$	47.83 $_{+0.00}$
	PGSR + C-Diff	**87.50** $_{+7.25}$	**30.00** $_{+5.00}$	**32.00** $_{+2.00}$	**85.00** $_{+7.50}$	63.00 $_{+1.00}$	56.50 $_{+4.00}$	**60.47** $_{+7.76}$	**60.74** $_{+3.29}$	31.25 $_{+0.00}$	**60.87** $_{+13.04}$
	PGSR + EPIC	**87.50** $_{+7.25}$	**30.00** $_{+5.00}$	**32.00** $_{+2.00}$	**85.00** $_{+7.50}$	63.00 $_{+1.00}$	56.50 $_{+4.00}$	**60.47** $_{+7.76}$	**60.74** $_{+3.29}$	31.25 $_{+0.00}$	**60.87** $_{+13.04}$
64	Raw	85.50	27.00	**30.00**	83.00	77.50	68.50	57.75	57.77	**50.00**	60.87
	C-Diff	84.75 $_{-0.75}$	28.00 $_{+1.00}$	**30.00** $_{+0.00}$	**87.00** $_{+4.00}$	**81.00** $_{+3.50}$	73.50 $_{+5.00}$	63.18 $_{+5.43}$	55.00 $_{-2.77}$	43.75 $_{-6.25}$	60.87 $_{+0.00}$
	LAVE (b=32)	**88.75** $_{+3.25}$	**34.00** $_{+7.00}$	**30.00** $_{+0.00}$	85.00 $_{+2.00}$	79.50 $_{+2.00}$	**75.50** $_{+7.00}$	**67.44** $_{+9.69}$	**61.17** $_{+3.40}$	**50.00** $_{+0.00}$	52.17 $_{-8.70}$
	EPIC	84.75 $_{-0.75}$	28.00 $_{+1.00}$	**30.00** $_{+0.00}$	**87.00** $_{+4.00}$	**81.00** $_{+3.50}$	73.50 $_{+5.00}$	63.18 $_{+5.43}$	55.00 $_{-2.77}$	43.75 $_{-6.25}$	60.87 $_{+0.00}$
	PGSR	86.25 $_{+0.75}$	27.00 $_{+0.00}$	**30.00** $_{+0.00}$	85.50 $_{+2.50}$	78.50 $_{+1.00}$	69.50 $_{+1.00}$	59.69 $_{+1.94}$	59.04 $_{+1.27}$	**50.00** $_{+0.00}$	**65.22** $_{+4.35}$
	PGSR + C-Diff	87.00 $_{+1.50}$	29.00 $_{+2.00}$	**30.00** $_{+0.00}$	**87.00** $_{+4.00}$	79.00 $_{+1.50}$	73.00 $_{+4.50}$	62.79 $_{+5.04}$	60.00 $_{+2.23}$	**50.00** $_{+0.00}$	**65.22** $_{+4.35}$
	PGSR + EPIC	87.00 $_{+1.50}$	29.00 $_{+2.00}$	**30.00** $_{+0.00}$	**87.00** $_{+4.00}$	79.00 $_{+1.50}$	73.00 $_{+4.50}$	62.79 $_{+5.04}$	60.00 $_{+2.23}$	**50.00** $_{+0.00}$	**65.22** $_{+4.35}$

F HYPERPARAMETER SENSITIVITY AT EVERY BUDGET

Table 6 in the main text reports the sensitivity analysis at $T = 16$. Table 10 extends the same twelve configurations to the four remaining denoising budgets. For each budget, we report AST accuracy and parsing success for the four non-live BFCL categories used in the main analysis.

The trends observed at $T = 16$ remained consistent across the full denoising-budget sweep. The benefit of repair was most pronounced at small denoising budgets, with mean improvements in parsing success ranging from $+10.69$ to $+27.19$ points at $T = 4$, compared with $+2.06$ to $+3.31$ points at $T = 64$. The difference between the highest and lowest mean AST accuracy among the tested repair settings also decreased from 5.82 points at $T = 4$ to 0.87 points at $T = 64$.

Moderate-to-wide repair windows consistently performed better than $w = 1$, with $w = 10$ or $w = 20$ achieving the highest mean AST accuracy at every budget. Increasing the number of repair rounds also improved parsing success, particularly at small T. Notably, the default configuration with two one-step rounds yielded higher mean parsing success than a single four-step repair at every budget, despite using fewer additional forward passes. Improved parsing, however, did not always translate into a comparable gain in task accuracy. At $T = 4$, for example, the default setting increased parsing success for multiple-function calling by 22.5 points and AST accuracy by 13.5 points, whereas the corresponding gains for parallel function calling were 18.0 and 4.0 points. This difference reflects the scope of PGSR: it addresses structural failures, while repaired outputs may still contain semantically incorrect function calls or arguments.

Table 10: PGSR sensitivity at $T \in 4, 8, 32, 64$ on LLaDA-8B. The $T = 16$ results are reported in Table 6 of the main text. Each setting varies one quantity from the shaded default ($w = 5$, $R = 2$, $T_{\text{rep}} = 1$). Parsing success denotes the proportion of outputs accepted by the JSON parser. Bold indicates the best repair setting within each budget.

	AST accuracy (%)					Parse rate (%)				
Setting	**Py**	**Mult.**	**Par.**	**P.M.**	**Mean**	**Py**	**Mult.**	**Par.**	**P.M.**	**Mean**
$T = 4$										
Raw (no repair)	71.00	63.00	47.50	19.50	50.25	75.25	69.00	52.00	21.50	54.44
Default	80.75	76.50	51.50	23.00	57.94	91.75	91.50	70.00	42.00	73.81
$w = 1$	77.75	72.00	48.50	22.00	55.06	85.50	83.00	59.50	32.50	65.12
$w = 3$	79.00	77.00	51.50	22.50	57.50	89.00	89.00	66.00	39.00	70.75
$w = 10$	80.75	78.50	56.00	27.50	60.69	93.25	93.00	77.00	52.00	78.81
$w = 20$	79.50	79.00	55.00	30.00	**60.88**	91.50	93.00	76.50	53.00	78.50
$R = 1$	80.00	75.50	50.50	23.00	57.25	89.75	88.50	63.00	33.50	68.69
$R = 3$	80.50	77.00	52.00	25.00	58.62	93.00	92.50	75.00	51.50	78.00
$R = 4$	80.50	77.00	52.00	25.00	58.62	93.00	93.00	80.50	60.00	**81.62**
$T_{\text{rep}} = 2$	80.75	76.50	52.00	24.00	58.31	94.25	91.00	73.00	42.50	75.19
$T_{\text{rep}} = 4$	80.50	78.50	51.50	24.50	58.75	95.00	93.50	73.50	46.50	77.12
R=1, T_{rep}=2	80.00	75.50	51.00	23.00	57.38	91.50	88.50	64.50	33.50	69.50
R=1, T_{rep}=4	80.25	77.50	50.50	23.50	57.94	92.75	91.00	65.50	36.50	71.44
$T = 8$										
Raw (no repair)	71.75	65.50	71.00	51.00	64.81	78.00	72.50	77.50	54.50	70.62
Default	79.75	77.00	74.00	55.00	71.44	93.50	91.50	85.00	70.00	85.00
$w = 1$	77.25	73.00	72.00	53.50	68.94	86.50	87.00	82.50	63.50	79.88
$w = 3$	79.25	77.00	73.50	55.00	71.19	91.75	91.00	84.50	71.00	84.56
$w = 10$	81.00	77.00	75.00	60.50	73.38	94.50	93.00	87.00	77.50	88.00
$w = 20$	81.00	76.50	77.50	61.00	**74.00**	93.75	93.00	91.50	75.50	**88.44**
$R = 1$	79.00	77.00	74.00	55.00	71.25	89.75	90.00	81.50	65.50	81.69
$R = 3$	80.25	78.00	74.50	56.50	72.31	94.25	92.50	87.50	73.50	86.94
$R = 4$	80.25	78.00	75.00	56.50	72.44	94.50	93.50	88.00	76.00	88.00
$T_{\text{rep}} = 2$	80.50	78.00	74.00	55.50	72.00	95.00	91.50	87.00	69.50	85.75
$T_{\text{rep}} = 4$	80.25	78.00	74.50	55.00	71.94	95.50	95.50	87.50	73.50	88.00
R=1, T_{rep}=2	79.75	77.00	74.00	55.00	71.44	91.25	90.00	82.00	65.50	82.19
R=1, T_{rep}=4	79.75	77.00	74.00	55.00	71.44	92.50	92.50	83.50	67.00	83.88
$T = 32$										
Raw (no repair)	78.50	77.50	82.00	74.50	78.12	86.00	84.00	95.50	89.00	88.62
Default	85.50	83.00	82.50	76.50	81.88	96.75	94.50	98.00	95.00	96.06
$w = 1$	81.00	81.00	82.00	76.00	80.00	91.50	90.50	96.50	93.00	92.88
$w = 3$	83.00	82.00	82.50	76.50	81.00	94.75	94.00	98.50	95.50	95.69
$w = 10$	85.00	84.50	82.50	77.00	**82.25**	96.75	97.50	99.00	96.50	**97.44**
$w = 20$	84.50	83.50	82.50	77.50	82.00	95.50	95.50	98.00	96.00	96.25
$R = 1$	84.50	82.50	82.50	76.50	81.50	94.25	94.00	97.50	94.50	95.06
$R = 3$	85.50	83.00	82.50	76.50	81.88	96.75	95.50	99.00	96.00	96.81
$R = 4$	85.50	83.00	82.50	76.50	81.88	96.75	95.50	99.00	96.50	96.94
$T_{\text{rep}} = 2$	84.75	83.00	82.50	76.50	81.69	96.75	94.50	98.00	95.00	96.06
$T_{\text{rep}} = 4$	85.00	83.50	82.50	77.00	82.00	98.25	94.50	99.00	95.00	96.69
R=1, T_{rep}=2	84.00	82.50	82.50	76.50	81.38	95.25	94.00	97.50	94.50	95.31
R=1, T_{rep}=4	84.50	83.00	82.50	77.00	81.75	96.75	93.50	98.00	95.00	95.81
$T = 64$										
Raw (no repair)	88.00	83.00	82.50	80.00	83.38	95.25	92.50	98.00	95.50	95.31
Default	89.25	86.50	83.00	80.00	84.69	98.50	99.00	98.50	97.00	98.25
$w = 1$	88.75	85.00	83.00	80.00	84.19	98.00	95.50	98.50	97.50	97.38
$w = 3$	89.00	86.00	83.00	80.00	84.50	98.50	98.50	98.50	97.50	98.25
$w = 10$	89.75	86.50	83.00	80.50	84.94	99.00	99.00	98.50	98.00	**98.62**
$w = 20$	90.25	86.50	83.00	80.50	**85.06**	98.50	98.00	98.50	98.50	98.38
$R = 1$	89.00	86.50	83.00	80.00	84.62	98.00	98.00	98.50	97.00	97.88
$R = 3$	89.25	86.50	83.00	80.50	84.81	98.50	99.00	98.50	97.50	98.38
$R = 4$	89.25	86.50	83.00	80.50	84.81	98.50	99.00	98.50	97.50	98.38
$T_{\text{rep}} = 2$	89.25	86.50	83.00	80.00	84.69	98.75	99.00	98.50	97.00	98.31
$T_{\text{rep}} = 4$	89.25	86.50	83.00	80.00	84.69	99.50	99.00	98.50	97.00	98.50
R=1, T_{rep}=2	89.25	86.50	83.00	80.00	84.69	98.25	98.00	98.50	97.00	97.94
R=1, T_{rep}=4	89.25	86.50	83.00	80.00	84.69	99.00	99.00	98.50	97.00	98.38

G Implementation Details

Adaptation Corpora. Each backbone was adapted separately to the four target domains using the public corpora listed in Table 11. The table reports the train and held-out split sizes, sequence length, and corresponding evaluation benchmark for each domain. Samples whose combined prompt and target exceeded the per-domain token budget in Table 11 were dropped rather than truncated. The same data split was used for LLaDA-8B, Dream-7B, and the autoregressive Qwen2.5-7B reference. The held-out split was used only for periodic evaluation during training, while the checkpoint from the final epoch was used for all reported results.

Table 11: Adaptation corpora, one per domain. Len denotes the domain-specific maximum training sequence length for the combined prompt and target.

Domain	Adaptation corpus	Train	Held-out	Len	Evaluated on
Tool calling (JSON)	xLAM-function-calling-60k	57,000	3,000	1024	BFCL v4
Code (Python)	Magicoder-Evol-Instruct-110K	108,959	2,224	1024	HumanEval(+), MBPP(+)
Text-to-SQL	Gretel synthetic text-to-SQL	98,000	2,000	512	Spider dev
Molecules (SMILES)	ZINC-250k	244,465	4,990	256	ZINC-250k held-out

Training Recipe. The same QLoRA configuration was used across all four domains, both diffusion backbones, and the autoregressive reference. Each base model was loaded with 4-bit NF4 quantization and double quantization using a bfloat16 compute dtype. LoRA adapters were trained with rank 32, $\alpha = 64$, and dropout 0.05, with adapters applied to all linear projections. Training was performed for 3 epochs using AdamW with a peak learning rate of 2×10^{-4}, cosine decay, and a warmup ratio of 0.1. We used an effective batch size of 32 and gradient checkpointing.

Decoding. All methods used greedy decoding. We retained the backbone-specific token selection rule throughout all experiments: low-confidence remasking for LLaDA-8B and `maskgit_plus` for Dream-7B. Maximum generation lengths were 512 tokens for BFCL and EvalPlus, 256 for Spider, and 128 for SMILES.

Parsers. Table 12 summarizes the parser used for each structured format and how its failure location is obtained. JSON, Python, and SQL expose this information directly through their parser interfaces. RDKit does not provide a location through its Python API for every failure, so we extract it from the `stderr` diagnostic when the message contains `check for mistakes around position`. If no usable location is available, PGSR leaves the output unchanged. This occurs for some SMILES failures, including valence errors, for which RDKit provides no repair offset.

Table 12: Parsers, error-position fields, and versions used throughout the experiments.

Grammar	Parser	Version	Error position
JSON	`json` (standard library)	Python 3.10	`JSONDecodeError.pos`
Python	`ast.parse` (standard library)	Python 3.10	`SyntaxError.lineno`, `.offset`
SQL	SQLGlot (SQLite dialect)	30.15.0	`ParseError.errors[0]["col"]`
SMILES	RDKit	2026.03.5	offset parsed from `stderr`

Software and Hardware. All training and evaluation were conducted on a single NVIDIA RTX PRO 6000 Blackwell GPU with 96 GB of memory, on a system equipped with an Intel Xeon Gold 6526Y CPU. Experiments used Python 3.10 with CUDA 12.8. FCD was computed using `fcd` version 1.2.2, code-generation performance was evaluated with the official `evalplus` evaluator, and BFCL accuracy was measured using the official v4 AST checker.

H Source Code

The corresponding code will be made publicly available after the publication of this paper.