Dgrammar: Efficient Constrained Decoding for DLMs

Guan-Ming Chiu
National Taiwan University
gmchiu@arbor.ee.ntu.edu.tw

Jeng-Yue Liu
School of Computer Science
Carnegie Mellon University
buffettl@andrew.cmu.edu

Abstract

Diffusion language models (DLMs) can decode several tokens in one step. However, IG-CD and LAVE verify candidate tokens sequentially within each denoising step, introducing repeated sampling and grammar checks. A parser identifies the first invalid token while preserving the accepted prefix. We propose Dgrammar, a training-free decoder that repairs this frontier token using the current denoising scores. The method combines local repair with a grammar-gated batch schedule, asynchronous mask construction, and bounded grammar-guided completion. On JSONSchemaBench medium with LLaDA-8B, Dgrammar improves schema validity by 8.6 percentage points over LAVE, reduces mean latency by 5.8×, and eliminates timeouts. Validity and latency gains also hold on the other JSB splits and on Dream-7B. Experiments with a SMILES grammar demonstrate that the decoder also supports constraints beyond JSON-Schema.

1 Introduction

Masked diffusion language models (DLMs) such as LLaDA (Nie et al., 2025) and Dream (Ye et al., 2025) have emerged as a parallel alternative to autoregressive (AR) generation. In a single forward pass the model produces a distribution over every masked position, and the decoder can unmask any subset of confident positions at once. This multi-token unmasking is the main systems advantage of DLMs, and recent accelerators build directly on it (Wu et al., 2025; Chen et al., 2026, 2025).

Many applications need output that obeys a formal grammar, such as JSON that conforms to a schema, code, or a chemical notation (Willard and Louf, 2023; Dong et al., 2024). For AR models, a token-level matcher such as XGrammar or llguidance (Dong et al., 2024; Microsoft, 2024) builds a vocabulary mask in microseconds, far below the cost of one forward pass. However, the AR interface assumes a single causal frontier, and a DLM has none until the decoder defines one.

ETH/IG-CD (Mündler et al., 2025) and LAVE (Zhang et al., 2026) verify candidate tokens individually within each denoising step. This design has two costs. (i) *Sequential commitment.* Tokens pass through the verifier one by one, so grammar work grows with every committed token and the sampler's block-level parallelism is unused at the grammar stage. (ii) *Heavy tails.* Each violation triggers further sampling and parser checks, and after repeated rejection the AR fallback spends one forward pass per token and runs the diffusion model in a regime it never saw during training. Both papers report only end-to-end wall time, which hides the per-token constraint cost and the timeout tail.

We observe that a grammar violation in a parallel step is local. The parser consumes the unmasked tokens left to right and stops at the first token it cannot accept, and every token before it is already valid. To exploit this, we propose Dgrammar, a training-free decoder built on *Single-Frontier Repair*. Dgrammar remasks the offending frontier token and selects a replacement using the same denoising scores. The batch size K then follows a *Grammar-Gated Backoff* rule that doubles after clean steps and resets on a violation. Two systems optimizations, asynchronous mask construction and a bounded grammar-guided tail, keep the constraint cost below the forward pass.

We evaluate Dgrammar on all three JSONSchemaBench splits (Geng et al., 2025), JSONBench, and a SMILES context-free grammar with LLaDA-8B-Instruct and Dream-7B. On the medium split with LLaDA-8B on a B200 GPU, Dgrammar raises schema validity from 76.1% (LAVE) to 84.7%, reduces mean latency from 30.6 s to 5.2 s and p95 latency from 120 s to 12.7 s, and lowers the 120 s timeout rate from 12.7% to zero. Functional@1 stays within 1.1 pp of LAVE on JSON-Bench, so our claims concern structural

Proceedings of the 15th International Joint Conference on Natural Language Processing and the 5th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, pages 5051–5061
November 6-10, 2026 ©2026 Association for Computational Linguistics

validity and latency (Table 5). In summary, our contributions are:

- We characterize how constrained DLM decoders modify token selection and profile their per-token constraint overhead (§3, Table 6).
- We propose Dgrammar, whose Single-Frontier Repair preserves multi-token unmasking under grammar constraints. On JSON-SchemaBench medium it improves schema validity by 8.6 pp over LAVE while cutting mean and p95 latency by 5.8× and 9.4× with zero timeouts.
- We evaluate validity and latency across three JSB difficulty splits and two backbones. Further analyses measure validity across seeds, characterize residual failures, and assess transfer to a SMILES context-free grammar (§5).

2 Background and Related Work

Masked DLMs. A masked diffusion model predicts every position from a partially masked sequence. At inference, decoding starts from an all-mask sequence of length L and runs for T steps. At each step the model produces logits for all L positions in one forward pass, and the decoder unmasks K positions chosen by a confidence rule. With $K > 1$ each forward pass commits several tokens instead of one. LLaDA (Nie et al., 2025) and Dream (Ye et al., 2025) reach AR-comparable quality this way. Parallel-decoding accelerators (Wu et al., 2025; Chen et al., 2026, 2025) adapt K per step but remain unconstrained, and Dgrammar's mask-based repair is in principle compatible with such schedules.

Grammar-constrained AR decoding. Token-level matchers (Willard and Louf, 2023; Dong et al., 2024; Microsoft, 2024; Geng et al., 2023; Park et al., 2025) maintain an incremental parser state and return a vocabulary mask of allowed next tokens at every step. Per-token mask construction is in the microsecond range, well below the cost of a forward pass. Park et al. (2024) further analyze how greedy mask truncation alters the model's scoring distribution under grammar constraints, motivating our framing in §3, and Reddy et al. (2026) study draft-conditioned constrained decoding for AR models. These interfaces do not transfer to diffusion because there is no single causal frontier until we define one.

Grammar-constrained DLM decoding. ETH/IG-CD (Mündler et al., 2025) suppresses rejected candidates by checking whether the partial output still admits a grammar-valid completion. LAVE (Zhang et al., 2026) replaces this verifier with a beam-search lookahead and falls back to per-position AR decoding after repeated rejection. It reports 99% syntactic validity on `json-mode-eval`, but on harder splits validity drops sharply (§5). DINGO (Suresh et al., 2025) runs full-sequence dynamic programming at higher per-step cost, while plan-and-fill decoders (Li et al., 2026) emphasize structure-aware planning. Both ETH/IG-CD and LAVE commit tokens sequentially through their verifiers and report only end-to-end runtime.

3 Where Each Decoder Intervenes

We compare how Dgrammar, IG-CD, and LAVE incorporate grammar constraints into token selection. The comparison distinguishes frontier masking, completion-based filtering, and stochastic lookahead under a shared notation.

Let $\ell \in \mathbb{R}^{L\times V}$ be the logits from one forward pass on the current block context $\mathbf{x}$, and let $\tilde{\ell}$ be the per-step selection scores actually used by the sampler: $\tilde{\ell} = \ell$ at temperature 0, and ℓ plus a Gumbel perturbation drawn once per step at temperature $\tau > 0$ (our runs use τ=0.2). Write

$$P_p(t) \;=\; p_\theta(t \mid \mathbf{x}, p) \;=\; \mathrm{softmax}(\ell[p,\cdot])_t$$

for the model's per-position distribution at masked position p. Let $G_p \subseteq V$ be the token set accepted by the incremental matcher at the current prefix frontier, and define the grammar-truncated block-context distribution

$$P_p^G(t) \;=\; \frac{\mathbf{1}[t \in G_p]P_p(t)}{\sum_{u\in G_p} P_p(u)} \quad \text{when } P_p(G_p) > 0.$$

Any grammar-constrained decoder must replace P_p by some grammar-biased scoring rule. The distinction is whether this bias is applied to the same block-context logits or after changing the scoring objective.

Proposition 1 (Local MAP under truncation)**.** *Assume $G_p \neq \emptyset$, the matcher state is restored between retries, and the retry cap is not reached. Then the logit-truncation loop selects, in at most $|V \setminus G_p|$ rejected candidates,*

$$\hat{t}_p^D \;=\; \arg\max_{t\in G_p} \tilde{\ell}[p,t],$$

which equals $\arg\max_{t \in G_p} p_\theta(t \mid \mathbf{x}, p)$ *at temperature* 0*. This result characterizes selection at one frontier under fixed step scores; it does not establish a guarantee for joint sampling.*

Proof. Each retry picks $\arg\max \tilde{\ell}[p]$ under the current suppression set. If the token is in G_p, the loop exits; otherwise its score is set to $-\infty$. After all higher-scoring tokens outside G_p have been suppressed, the best remaining token is the constrained maximizer of the step's scores. □

Where the baselines apply their bias. The other DLM-constrained decoders apply their grammar bias to a different object. ETH/IG-CD performs rejection filtering with a completion test: at temperature 0 its accepted token is $\hat{t}_p^{\text{ETH}} = \arg\max_{t \in C_p(\mathbf{x})} P_p(t)$, the MAP over the feasible set $C_p(\mathbf{x}) = \{t \in V : \text{Comp}(\mathbf{x}, p, t, \mathcal{G}) = 1\}$ of tokens whose insertion leaves a nonempty grammar intersection. This still uses the same forward-pass logits, but the feasibility set depends on the whole partially filled block and the sequential commitment order, not only the current frontier.

LAVE applies a different feasibility test: a candidate token is accepted only if at least one of N lookahead prefixes, sampled from the same forward pass's per-position distributions, is grammar-extensible. Its accepted token is the first sampled candidate that passes this stochastic test, which need not coincide with $\arg\max_t P_p(t)$ or $\arg\max_{t \in G_p} P_p(t)$. If it falls back to per-position AR decoding, the conditioning changes to $Q_p(t) = p_\theta(t \mid \mathbf{x}_{<p})$ (grammar-truncated to Q_p^G), so the relevant distortion is the context-switch mismatch between P_p^G and Q_p^G.

4 Method

Dgrammar runs the standard masked-diffusion loop but changes how grammar feedback is used. It selects several tokens before extending the accepted prefix, repairs only the first invalid frontier token under the same step's scores, and lets parser feedback control how aggressively the next step unmasks. Figure 1 contrasts this with LAVE, and Algorithm 1 gives one denoising step: one forward pass, then an inner loop that fills the step's token quota q in batches of at most K; the outer loop over steps and the tail completion of §4.3 wrap it.

4.1 Single-Frontier Repair

The decoder maintains a left-to-right cursor c into the generated region and a token-level matcher state (Microsoft, 2024). After each batch unmask, ExtendPrefix feeds the contiguous non-mask suffix starting at c to the matcher. If all tokens are consumed, c advances. Otherwise, the first unconsumed position p is the frontier violator, and every earlier position has already been accepted by the matcher.

The repair is local. Let t_p be the rejected token at p. Dgrammar sets $\tilde{\ell}[p, t_p] = -\infty$ in the step's selection scores (§3), returns x_p to `[MASK]`, takes the next argmax of $\tilde{\ell}[p]$, and retries the matcher from the same accepted prefix. This loop reuses the step's scores, so each retry costs one vector argmax plus one matcher call. It stops when the matcher accepts a replacement or when `max_resamples` is reached. Tokens already accepted in the same batch are never re-decoded, so a single violation costs one position, not the whole step.

Prefix validity versus whole-sequence validity. Frontier checking validates only the prefix that the matcher has consumed. Tokens ahead of the cursor remain unchecked until the parser reaches them. The decoder repairs rejected tokens when they reach the frontier. After diffusion, grammar-guided completion attempts to finish any remaining suffix within the generation budget (§4.3). Evaluation applies JSON parsing and schema validation to the complete output (§5).

4.2 Grammar-Gated Backoff

Grammar-Gated Backoff doubles the batch size K after each batch without violations, up to $K_{\max}$, and resets K to one after a violation. We evaluate this heuristic through component ablations (§5.2) and report the realized batch sizes in App. E. Like confidence-adaptive schedules (Wu et al., 2025), the rule adjusts parallelism during decoding, using parser feedback to determine the batch size. App. F gives the per-backbone implementation details.

Grammar constraints are applied only at the current frontier. Before token selection, ComputeMask returns the set of tokens the matcher forbids at cursor c, and their scores in $\tilde{\ell}[c]$ are set to $-\infty$. Non-frontier positions follow the usual diffusion confidence rule: each masked position proposes $\arg\max \tilde{\ell}[p]$, and positions are ranked by that token's probability under the raw logits, $\text{softmax}(\ell[p])$. This frontier-only masking avoids building a mask for every masked position in the block. It still guarantees that the next token the parser consumes is either grammar-valid or be-

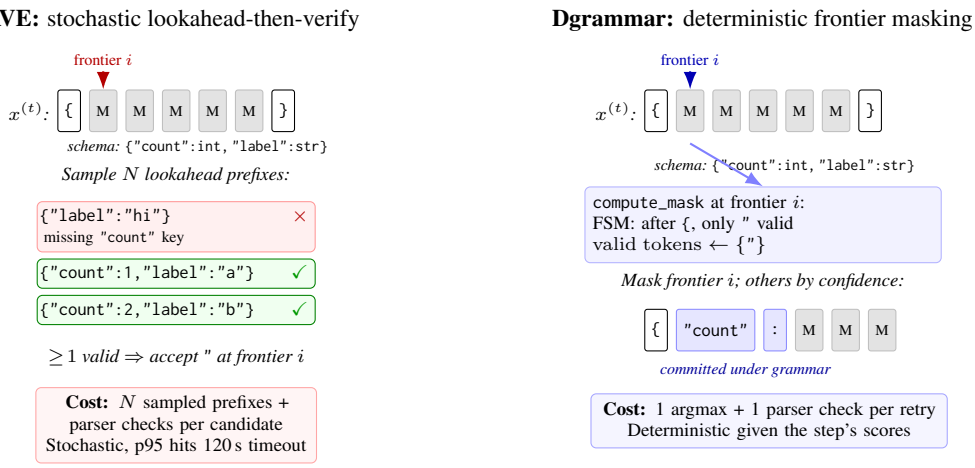


Figure 1: One denoising step under schema `{"count":int, "label":str}`. **LAVE** (Zhang et al., 2026) (left) samples N lookahead prefixes from the current forward pass at each frontier token and accepts the token only when at least one prefix is grammar-extensible; rejected candidates are resampled, and repeated rejection triggers an AR fallback; on hard schemas some instances reach the 120 s per-instance timeout. **Dgrammar** (right) adds the matcher's mask at the frontier, selects the remaining positions by confidence, verifies the unmasked prefix, and repairs the first violator from the same scores; given the step's scores and matcher state the repair is deterministic.

comes a repair target.

4.3 Async Mask Overlap and Grammar-Guided Tail

Two engineering choices keep constraint overhead from dominating the diffusion loop. First, mask construction for the current frontier does not depend on the next forward pass, so we launch COMPUTEMASK on a background thread and join it after the forward pass. We fall back to synchronous mask computation only when a mid-step repair changes the matcher state. In our runs this hides almost all mask construction latency behind GPU compute (Table 6).

Second, if the diffusion loop exits before the matcher reaches an accepting state, we invoke a grammar-guided AR completion once from the current cursor. It is restricted to the unfinished suffix and refreshes the model logits at every emitted token under the grammar mask. In practice this mostly closes short tails such as braces or brackets. The tail is deliberately bounded, and any locally valid but schema-invalid completion is counted as a failure by the final metric.

5 Experiments

5.1 Main JSB Results

Setup. We evaluate on JSONSchemaBench (JSB) medium_test (Geng et al., 2025); on LLaDA, after llguidance skips 75 schemas, the 511 instances with both a Dgrammar and a LAVE record form the headline set (506 on Dream; App. B). All runs use an NVIDIA B200, T=128 steps, generation length 256, block length 32, temperature 0.2, and `seed=0` unless stated otherwise (seed variance is reported in §5.2). We compare Dgrammar with VANILLA unconstrained generation, LAVE (Zhang et al., 2026), and ETH/IG-CD (Mündler et al., 2025) on LLaDA-8B-Instruct (Nie et al., 2025) and Dream-v0-Instruct-7B ("Dream-7B") (Ye et al., 2025). Dgrammar uses one per-backbone setting for `max_resamples` (500 for LLaDA, 2000 for Dream) and per-position AR autocomplete; LAVE uses its published default `max_retry_num_total`=1000.

Metric. We report *schema@1*: an instance succeeds when its final output parses as JSON and `jsonschema.validate` (default settings, no format checker) accepts it against the schema. Every JSB, ablation, JSON-Bench, and overhead number in this paper is produced by one script (`bench/analysis/tables.py`) applied to the stored outputs of every method (Vanilla, IG-CD, LAVE, Dgrammar); the runners' internal validity flags, whose semantics differ across systems, are not used (App. D). SMILES scores come from a separate CFG and RDKit scorer.

Main results. As shown in Table 1, Dgrammar outperforms LAVE in both validity and latency on every split and both backbones. On LLaDA medium, VANILLA reaches only 51.3% schema@1. LAVE improves validity to 76.1%, but

Split	Method	n	sch@1	mean	p50	p90	p95	p99	Timeout
				LLaDA-8B-Instruct					
easy	Vanilla	558	64.2	2.58	2.49	3.23	3.38	4.05	0
	IG-CD†	449	79.1	**2.58**	**2.31**	**3.97**	**5.12**	**9.60**	**0**
	LAVE	558	82.1	24.95	2.76	120.00	120.22	120.60	83
	DGR.	558	**94.3**	3.14	2.70	4.48	5.34	10.98	**0**
medium	Vanilla	511	51.3	3.91	3.68	5.15	5.65	7.59	0
	IG-CD†	383	67.4	9.65	**3.91**	17.97	44.55	106.89	4
	LAVE	511	76.1	30.56	12.34	120.00	120.00	120.58	65
	DGR.	511	**84.7**	**5.24**	4.39	**8.56**	**12.70**	**15.46**	**0**
hard	Vanilla	269	24.9	9.30	7.54	15.34	17.71	25.58	0
	IG-CD†	245	38.8	39.97	**14.82**	120.00	120.00	120.19	50
	LAVE	269	53.2	45.01	22.50	120.00	120.00	120.00	56
	DGR.	269	**54.6**	**24.17**	19.56	**46.59**	**54.60**	**87.10**	**0**
				Dream-7B					
easy	Vanilla	552	65.0	1.20	0.98	2.27	2.46	2.63	0
	IG-CD†	449	87.3	**1.70**	**1.32**	**3.41**	**3.93**	**5.81**	**0**
	LAVE	552	83.5	31.18	10.32	120.00	120.07	120.36	65
	DGR.	552	**93.7**	5.29	3.27	11.13	14.05	24.22	**0**
medium	Vanilla	506	41.5	2.71	2.80	3.81	4.17	5.25	0
	IG-CD†	383	67.4	**5.96**	**3.33**	**8.40**	**12.96**	90.69	3
	LAVE	506	78.3	34.20	15.42	120.00	120.04	120.64	58
	DGR.	506	**82.4**	9.11	6.24	17.62	21.49	**46.60**	**0**
hard	Vanilla	258	12.4	5.99	5.29	9.25	10.96	14.70	0
	IG-CD†	245	40.8	33.71	**8.90**	123.15	131.02	150.40	35
	LAVE	258	56.2	41.01	19.09	120.00	120.00	120.16	43
	DGR.	258	**56.6**	**18.23**	17.78	**37.27**	**47.49**	**51.17**	**0**

Table 1: JSB difficulty spectrum, B200, T=128. DGR. denotes Dgrammar. schema@1 is `jsonschema.validate` on the parsed output (§5). Latencies in seconds, quantiles by linear interpolation; Timeout counts instances whose recorded time reached the 120 s budget ($\geq$ 119 s, since the kill can land slightly before 120 s). Bold marks the best constrained decoder (IG-CD, LAVE, Dgrammar) per column within a split. n is matched-instance count after llguidance skips. †IG-CD (Mündler et al., 2025) compiles only a subset of each split (449/577 easy, 383/586 medium, 245/368 hard, i.e. 22–35% rejected, identical across backbones); its rows are over that compilable subset, so schema@1 is not instance-matched to the other methods. On Dream its soft (`stopit`) timeout cannot interrupt a running kernel, so a few hard instances overshoot the 120 s budget. Hyperparameters in App. A.

at 7.8× higher mean latency and with 65/511 timeouts. Dgrammar raises schema@1 to 84.7%, runs 5.8× faster than LAVE on mean latency and 9.4× faster on p95, and has zero timeouts. ETH/IG-CD (Mündler et al., 2025) is not instance-matched in Table 1. Its CFG pipeline rejects 22–35% of each split (Table 1, †), and its rows are over the subset it compiles.

Strict instance-matched comparison. To remove the compile-rate mismatch, Table 2 recomputes schema@1 on the per-cell intersection of instances that all four methods compile. Five of six cells keep the ordering of Table 1. On LLaDA hard, LAVE leads by 0.6 pp, while in that same cell LAVE times out on 34 instances and Dgrammar on none. The failure modes also stay disjoint. On the intersection, IG-CD produces 49 to 104 schema-invalid outputs per cell, LAVE produces 18 to 55 empty timeout outputs per cell, and Dgrammar produces neither.

Backbone	Split	n	Van.	IG-CD	LAVE	Dgr.
LLaDA	easy	440	63.0	79.5	84.3	**94.8**
LLaDA	medium	351	55.8	68.7	79.8	**86.0**
LLaDA	hard	168	30.4	49.4	**58.9**	58.3
Dream	easy	435	66.4	87.8	85.7	**94.3**
Dream	medium	347	45.8	70.3	78.4	**84.1**
Dream	hard	167	13.8	55.7	59.3	**60.5**

Table 2: schema@1 (%) on the strict per-cell intersection of instances that all four methods compile. Same runs as Table 1. n is the intersection size.

5.2 Ablations and Failure Analysis

Table 3 evaluates Dgrammar's components on the same 511 instances. Async overlap produces identical per-instance validity outcomes in both fall-

Algorithm 1 Dgrammar denoising step

Input: x, cursor c, matcher M, batch K, resamples r, quota q
Launch COMPUTEMASK(M) asynchronously → `pending`
$\ell \leftarrow$ FORWARD(x); $\tilde{\ell} \leftarrow \ell + \text{Gumbel}(\tau)$; $n \leftarrow 0$
while $n < q$ and masked positions remain **do**
  `forbid` $\leftarrow$ join `pending` if valid, else COMPUTEMASK(M)
  $\tilde{\ell}[c, \texttt{forbid}] \leftarrow -\infty$ ▷ frontier mask
  $x_p^* \leftarrow \arg\max \tilde{\ell}[p]$, $\text{conf}_p \leftarrow \text{softmax}(\ell[p])_{x_p^*}$; $P \leftarrow$ top-min$(K, q{-}n)$ masked p by conf
  $x_p \leftarrow x_p^*$ for $p \in P$; $n \leftarrow n + |P|$
  $c, v \leftarrow$ EXTENDPREFIX(M, x, c)
  if $v \neq -1$ **then** ▷ violation at $v = c$
    $\tilde{\ell}[v, x_v] \leftarrow -\infty$; $x_v \leftarrow$ [MASK]; $n \leftarrow n - 1$; $r \leftarrow r + 1$
    while $r <$ `max_resamples` **do**
      $t \leftarrow \arg\max \tilde{\ell}[v]$
      if M.TRYCONSUME($[t]$) $= 1$ **then**
        $x_v \leftarrow t$; $n \leftarrow n + 1$; $c, _ \leftarrow$ EXTENDPREFIX($M, x, v{+}1$); **break**
      end if
      $\tilde{\ell}[v, t] \leftarrow -\infty$; $r \leftarrow r + 1$
    end while
    if $r \geq$ `max_resamples` **then return** to tail completion (§4.3)
  end if
  $K \leftarrow 1$; invalidate `pending`
 else
  $K \leftarrow \min(2K, K_{\max})$
 end if
 if M.ISACCEPTING() **then** fill remaining with EOS; **return**
 end if
end while

back configurations. With greedy block completion, overlap reduces mean and p95 latency by 18% and 37%, respectively. With AR autocomplete, overlap increases mean latency by 0.27 s (5.24 vs. 4.97 s), with similar p95 latency (12.70 vs. 12.60 s). These latency measurements come from single runs and are subject to the allocation variability reported below.

Autocomplete recovers valid outputs on instances that the diffusion loop leaves incomplete. Disabling it under the AR setting reduces schema@1 from 84.7% to 75.7%. It activates on 24.1% of instances and contributes 8.8% of all emitted tokens (8,928 of 100,997).

Forcing $K_{\max}$=1 increases mean latency by 144% under AR autocomplete, while schema@1 changes from 84.7% to 84.0%. The mean diffusion forward count remains similar at 115.3 versus 115.4 per instance, consistent with the sampler's fixed step budget. The selection loop takes 6.7 s per instance with $K_{\max}$=1, compared with 0.6 s for the full method. Frontier masks are recomputed more often, increasing total mask-computation time from 0.5 s to 3.1 s per instance. At 12.8 s mean latency, the $K_{\max}$=1 variant remains 2.4× faster than LAVE (30.6 s). In the full method, mean realized K is 1.88, with a minimum per-instance mean of 1.11. App. E reports the batch-size distribution and instances with the lowest mean batch sizes.

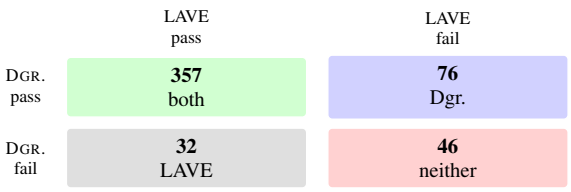


Figure 2: Paired schema@1 outcomes on JSB medium_test (LLaDA-8B-Instruct, n=511). Off-diagonal cells show instances solved by only one decoder.

The effect of `max_resamples` depends on the fallback configuration. Lower budgets reduce both validity and latency under greedy completion. With AR autocomplete, lower budgets leave validity nearly unchanged but increase latency as more work shifts to per-token completion.

Config	schema@1	mean	p95
AR autocomplete (Table 1 setting)			
Dgrammar (full)	84.7%	5.24	12.70
− async overlap	84.7%	4.97	12.60
− AUTOCOMPLETE	75.7%	5.94	11.12
$K_{\max}$=1	84.0%	12.79	37.15
`max_resamples`=100	84.5%	9.55	26.62
`max_resamples`=50	84.0%	11.55	31.11
Greedy block-completion fallback			
Dgrammar (full)	81.0%	6.35	13.97
− async overlap	81.0%	7.71	22.30
− AUTOCOMPLETE	75.7%	4.64	7.45
$K_{\max}$=1	80.4%	10.26	34.99
`max_resamples`=100	77.3%	4.81	9.26
`max_resamples`=50	75.5%	4.74	7.77

Table 3: Component ablation on JSB medium_test (n=511, LLaDA-8B-Instruct, B200, T=128, seed=0). Latencies in seconds. Top block uses per-position AR autocomplete (the Table 1 setting); bottom block uses the greedy block-completion variant for comparison. The two blocks were run on different days and their latencies are comparable within a block only.

Figure 2 illustrates where the validity gains and the remaining failures come from. Among the 511 common JSB medium_test instances, Dgrammar and LAVE agree on 403 cases (357 both pass, 46 both fail). Dgrammar wins 76 cases that LAVE misses, while LAVE wins 32 cases that Dgrammar misses. Dgrammar's wins concentrate in LAVE's

expensive failure mode, namely schemas where stochastic lookahead repeatedly rejects constrained continuations and often reaches the 120 s timeout. On the 65 LAVE timeout instances, Dgrammar never times out and 46 become schema-valid, at a median of 4.7 s, a 25× speedup over the LAVE timeout.

LAVE succeeds on 32 instances where Dgrammar fails, with a mean runtime of 21.8 s and a maximum of 87.3 s, below the timeout budget. All remaining Dgrammar failures occur during JSON parsing, before schema validation. Across the six backbone and split cells, 463 of 468 failures (98.9%) contain unbalanced brackets or quotes, indicating incomplete output. This signature occurs in every failure on the hard splits: 122/122 on LLaDA and 112/112 on Dream (Table 4). These results identify tail closure within the 256-token budget as the dominant residual failure mode.

Under AR autocomplete, lowering `max_resamples` from 500 to 100 and 50 changes schema@1 from 84.7% to 84.5% and 84.0%, respectively. Mean latency increases from 5.24 s to 9.55 s and 11.55 s as more instances exhaust the resample budget and invoke AR completion. Dgrammar improves the aggregate validity–latency trade-off, while LAVE succeeds on some instances where Dgrammar fails.

Seed sensitivity. We reran the headline cell (LLaDA, JSB medium, n=511) with seeds 1 and 2. Schema@1 is 84.7%, 83.8%, and 84.7% for seeds 0, 1, and 2, respectively, with a mean of 84.4 and sample standard deviation of 0.6. Every seed produces zero schema-invalid outputs, and each exceeds LAVE's seed-0 schema@1 by at least 7.7 pp. This comparison measures Dgrammar's variation; LAVE was evaluated with seed 0 only. At the instance level, 16.8% of cases change between pass and fail across seeds, despite the small aggregate variation.

Latency variability. Latency varied across separately allocated B200 instances, even when repeated runs produced identical outputs. Two repeated configurations produced identical outputs on all 132 and 133 shared instances, while mean latency differed by 2.5× and 1.7×, respectively. The three seed runs average 115.4, 115.7, and 116.0 diffusion forward passes per instance. Their mean latencies differ by 2.6× and 3.2× relative to seed 0. We report seed-0 latencies throughout the paper, with each method run separately on the same GPU type. These measurements show substantial variation across allocations, so inter-method ratios within the observed spread require caution.

Method	Split	Fails	Empty	Parse	Schema
LAVE	medium	110	33	77	0
DGR.	medium	89	0	89	0
LAVE	hard	113	40	73	0
DGR.	hard	112	0	112	0

Table 4: Dream-7B failure decomposition on JSB medium (n=506) and hard (n=258). Both decoders have zero schema errors; all failures occur before validation. Dgrammar eliminates empty timeout outputs; all of its remaining failures are parse errors, and every one ends in an unclosed structure or an unterminated string, the signature of truncation at the 256-token budget. The Dream hard regime thus mostly tests generation length rather than constraint quality.

Case study. Instance `o83272` is representative of the timeout wins (App. C). Its schema is a flat object with a bounded-length string, integer and number fields with minimum and maximum bounds, and two `date`-format strings. LAVE hits the 120 s timeout on it and returns no output. Dgrammar completes it in 3.15 s with zero resamples and produces schema-valid output, so under frontier masking every token the matcher exposed was accepted at once.

Empirical intervention rate. Figure 3 compares intervention counts per instance on JSB medium_test (LLaDA-8B). LAVE's lookahead retries have a long tail, with a median of 16, mean of 260, and maximum of 3334. Each retry samples lookahead prefixes again and repeats parser checks. Dgrammar has a median of one selective remask, and 43.4% of instances require no resamples. Its retry count is bounded by `max_resamples`, and each retry uses existing scores without an additional forward pass.

5.3 Transfer and Runtime Overhead

Difficulty spectrum. Across all three JSB splits and both backbones, Dgrammar matches or exceeds LAVE on validity while always running with zero timeouts. On LLaDA the validity advantage is +12.2/+8.6/+1.5 pp on easy/medium/hard, and on Dream the pattern is similar (+10.1/+4.2/+0.4 pp). The gap narrows on hard schemas. However, LAVE's timeout rate stays roughly stable across difficulty (14.9%/12.7%/20.8% on LLaDA), which suggests that its long-tail failure mode depends on

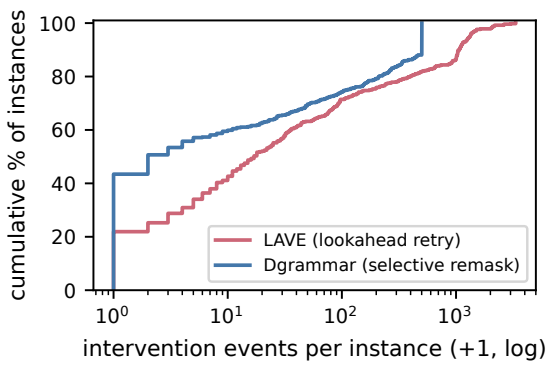


Figure 3: Empirical CDF of intervention events per instance on JSB medium_test (LLaDA-8B-Instruct, n=511). LAVE retries are lookahead-verification draws; Dgrammar resamples are selective remasks (no extra forward pass). Dgrammar's median is 1 vs. LAVE's 16; Dgrammar bounds its tail at `max_resamples`=500.

schema structure rather than on JSB's difficulty labels.

Functional correctness. On the JSON-Bench split (`json-mode-eval-extended`, n=272) used by LAVE (Zhang et al., 2026), Dgrammar's functional@1 is 1.1 pp below LAVE's on LLaDA-8B (51.8 vs. 52.9) and 0.7 pp below on Dream-7B (44.9 vs. 45.6); case-insensitive string matching gives 60.7 vs. 61.0 on LLaDA and 51.8 vs. 51.5 on Dream. Relative to unconstrained VANILLA (46.7/49.6), Dgrammar is +5.1 pp on LLaDA and −4.7 pp on Dream, and LAVE shows the same sign pattern (+6.2/−4.0). These comparisons use a single seed per method, without a significance test. Latency, in contrast, drops 17% (mean) and 70% (p95) on LLaDA and 2.9×/10× (mean/p95) on Dream, with zero timeouts vs. LAVE's one on LLaDA and eight on Dream.

Cross-grammar transfer (SMILES). To test that Dgrammar is not specific to JSON-Schema, we plug in a CFG-mode checker (rustformlang-backed CFG/DFA intersection-emptiness) and run on `eth-sri/smiles-eval` (n=167, Table 5). Dgrammar reaches 100.0% syn@1 on LLaDA-8B, +34.7 pp over VANILLA and above LAVE's 98.7%, and 99.4% on Dream-7B, matching LAVE. Functional@1 stays at 2–4 % for Vanilla and Dgrammar because the SMILES grammar accepts any single atom as a valid `Line`, and exact match against a stereoisomeric reference depends on the model's chemistry knowledge, which a grammar cannot

Dataset	Method	syn@1	fun@1
LLaDA-8B-Instruct			
JSON	Vanilla	96.0	46.7
JSON	LAVE	**99.6**	**52.9**
JSON	DGR.	98.5	51.8
SMILES	Vanilla	65.3	2.4
SMILES	LAVE‡	98.7	n/a
SMILES	DGR.	**100.0**	**3.0**
Dream-7B			
JSON	Vanilla	90.4	**49.6**
JSON	LAVE†	**96.7**	45.6
JSON	DGR.	94.1	44.9
SMILES	Vanilla	80.8	**3.6**
SMILES	LAVE‡	99.4	n/a
SMILES	DGR.	**99.4**	2.4

Table 5: Cross-benchmark validity (%). JSON-Bench (n=272) and SMILES (n=167); B200, same hyperparameters as Table 1 except that Dgrammar uses `max_resamples`=500 on both backbones for JSON-Bench. JSON syn@1 is JSON parseability and fun@1 is exact string equality after re-serialization with `json.dumps(indent=4)`, which preserves key order (App. D); the Dgrammar runs are missing two instances and the Dream LAVE run one, all counted as failures. SMILES syn@1 is CFG acceptance and fun@1 is RDKit canonical equivalence with the reference, scored with the released `score_smiles.py`; each CFG check is capped at 10 s and counted invalid on timeout, which affected one Vanilla Dream output. LAVE runs with the defaults of its released code; the setting in Zhang et al. (2026) (N=10, τ=5) reports fun@1 of 55.0/54.4 on LLaDA/Dream JSON. Bold marks the best value per column within a dataset. †LAVE Dream-7B JSON timeouts = 8; Dgrammar = 0. ‡LAVE SMILES syn@1 is quoted from Zhang et al. (2026) Table 1; that paper does not report SMILES fun@1, so the cell is not available (n/a) and the row is not a matched run.

supply. Zhang et al. (2026) excludes SMILES from its functional metric for the same reason, so no LAVE fun@1 is available. The full pipeline transfers across backbones through a per-model adapter alone (forward signature, mask and EOS token IDs, tokenizer).

Table 6 reports per-operation timing for Dgrammar on JSB medium_test. The grammar check averages 0.05 ms per call, more than two orders of magnitude below the 12.6 ms forward pass. Mask construction averages 5.6 ms but overlaps with the forward pass on every step where the frontier is masked. This saves about 0.5 s per instance on average and reduces the observed mask wait to 0.04 ms. The autocomplete fallback fires on 24.1% of instances; it produces a median of 21 tokens per

Operation	Mean
grammar check (per call)	0.05 ms
mask compute (per call)	5.59 ms
mask wait after async overlap	0.04 ms
mask time saved (per instance)	503 ms
model forward (per call)	12.6 ms
autocomplete fallback fires	24.1%

Table 6: Per-operation overhead for Dgrammar on JSB medium_test (LLaDA-8B, B200, the Table 1 run). Per-call values are means of per-instance means.

fired instance (mean 72.6, max 245) and accounts for 8.8% of all generated tokens.

6 Conclusion

Dgrammar combines multi-token selection with local repair at the grammar frontier. It preserves accepted tokens and reuses the current denoising scores during repair. On JSONSchemaBench medium with LLaDA-8B, Dgrammar raises schema validity from LAVE's 76.1% to 84.7%. It reduces mean latency by 5.8× and p95 latency by 9.4×, with zero timeouts. Program-scale grammars such as Python or C++ and integration with adaptive-K accelerators (Wu et al., 2025; Chen et al., 2026, 2025) remain untested. The implementation, scoring script, and run capsule are available at `https://github.com/guan404ming/dgrammar`.

Limitations

Although Dgrammar improves both validity and latency on the JSB splits and transfers to a SMILES grammar, it has several limitations.

First, our evaluation covers two DLMs and three benchmark families. We have not evaluated program-scale grammars such as Python or C++, where larger grammars may increase mask-construction costs. Repair behavior under deeply nested program grammars also remains untested. Dgrammar inherits llguidance's schema coverage: recursive `$ref+oneOf` JSB schemas, including 75 on medium, cannot be compiled and are excluded from matched sets (App. B).

Second, the baseline comparison is bounded by what could be run. ETH/IG-CD (Mündler et al., 2025) is not instance-matched in Table 1 because its regex backend rejects 22–35% of each split (for example `\/` URI escapes and null-bounded length quantifiers), and Table 2 gives the strict intersection. We omit DINGO (Suresh et al., 2025) because no runnable implementation was released, so an instance-matched comparison is not possible. Analytically, it applies its bias through full-sequence dynamic programming at higher per-step cost, whereas Dgrammar intervenes only at the frontier. LAVE runs with the defaults of its released code. The setting reported in the LAVE paper (N=10, τ=5, averaged over at least four runs) gives fun@1 of 55.0/54.4 on LLaDA/Dream, 2.1 and 8.8 pp above our LAVE runs and above Dgrammar; we did not rerun LAVE with it. Seed variance is reported for the headline cell only (§5.2), and the hard splits mostly test the 256-token generation budget rather than constraint quality.

Third, structural validity does not ensure that an output satisfies the intended task. Valid outputs can still contain empty arrays, empty objects, or uninformative strings, contributing to the gap between schema@1 and functional@1 (Table 5). Incomplete tail closure accounts for most observed failures, motivating further work on bounded completion strategies. Short beam search or learned completion heads may improve these completions. Frontier-only masking also introduces a sampling bias, so Dgrammar does not guarantee sampling from the constrained joint distribution.

References

Zigeng Chen, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. 2025. dParallel: Learnable parallel decoding for dLLMs. *arXiv preprint arXiv:2509.26488*.

Zigeng Chen, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. 2026. DMax: Aggressive parallel decoding for dLLMs. *arXiv preprint arXiv:2604.08302*.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2024. XGrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100*.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. 2025. JSONSchemaBench: A rigorous benchmark of structured outputs for language models. *arXiv preprint arXiv:2501.10868*.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In

Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP).

Miao Li, Hanyang Jiang, Sikai Cheng, Hengyu Fu, Yuhang Cai, Baihe Huang, Tinghan Ye, Xuanzhou Chen, and Pascal Van Hentenryck. 2026. Plan, verify and fill: A structured parallel decoding approach for diffusion language models. *arXiv preprint arXiv:2601.12247*.

Microsoft. 2024. llguidance: Super-fast structured outputs. https://github.com/guidance-ai/llguidance.

Niels Mündler, Jasper Dekoninck, and Martin Vechev. 2025. Constrained decoding of diffusion LLMs with context-free grammars. *arXiv preprint arXiv:2508.10111*.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. 2025. Large language diffusion models. *arXiv preprint arXiv:2502.09992*.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-aligned decoding. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Kanghee Park, Timothy Zhou, and Loris D'Antoni. 2025. Flexible and efficient grammar-constrained decoding. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *PMLR*, pages 48262–48275.

Avinash Reddy, Thayne T. Walker, James S. Ide, and Amrit Singh Bedi. 2026. The hidden cost of structured generation in LLMs: Draft-conditioned constrained decoding. *arXiv preprint arXiv:2603.03305*.

Tarun Suresh, Debangshu Banerjee, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. 2025. DINGO: Constrained inference for diffusion LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Brandon T. Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. 2025. Fast-dLLM: Training-free acceleration of diffusion LLM by enabling KV cache and parallel decoding. *arXiv preprint arXiv:2505.22618*.

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

Yitong Zhang, Yongmin Li, Yuetong Liu, Jia Li, Xiaoran Jia, Zherui Li, and Ge Li. 2026. Lookahead-then-verify: Reliable constrained decoding for diffusion LLMs under context-free grammars. *arXiv preprint arXiv:2602.00612*.

A Hyperparameters

Table 7 lists the hyperparameters shared across all methods and JSB splits.

Hyperparameter	**Value**
Diffusion steps T	128
Random seed	0 (1, 2: §5.2)
Block length	32
Generation length	256
Remasking strategy	low-confidence
Temperature	0.2
Max resamples (LAVE)	1000
Max resamples (Dgrammar)	500 / 2000
$K_{\max}$	8
Per-instance timeout	120 s

Table 7: Shared hyperparameters for the JSB runs; Dgrammar's resample budget is 500 on LLaDA and 2000 on Dream, except 500 on both for JSON-Bench (Table 5).

B Dataset and Instance Filtering

Each cell of Table 1 is scored on the instances for which both the Dgrammar and the LAVE run produced a record. On LLaDA this set is exactly the set that llguidance compiles: 75 of the 586 medium_test schemas (19 of 577 easy, 99 of 368 hard) are rejected at compile time because of recursive `$ref+oneOf` structures, LAVE returns `"Prompt does not conform to grammar"` on them, and Dgrammar skips them through the same guard, so n=558/511/269. On Dream, the LAVE runs additionally lack 6, 5, and 11 records on easy, medium, and hard, respectively. We exclude these records from the matched sets, yielding n=552/506/258; their causes were not investigated. Vanilla has a record for every schema and is scored on the same matched set. IG-CD compiles only a subset of each split (449/383/245) and its Table 1 rows are over that subset; Table 2 restricts all four methods to the instances that all of them cover.

C Case Study: Instance `o83272`

Instance `o83272` is a resource-specification schema on which LAVE times out and Dgrammar succeeds in one generation attempt with zero resamples. The schema is a flat object with `resource` (string, `maxLength` 64), `nodes` and `processors` (integers with a minimum of 0), `ppn` and `percent_allocated` (numbers with bounds), `start_date` and `end_date` (strings with `format`:

date), and a free-text comments field; resource, nodes, and processors are required.

LAVE (timeout, 120 s, no output). The stored record shows that LAVE reached the 120 s budget without committing a complete output. We did not record token-level traces for LAVE, so we do not attribute the timeout to a specific field.

Dgrammar (valid, 0 resamples, 3.15 s). Frontier masking restricts the vocabulary at the cursor to the tokens the matcher permits, and on this instance no selected token was ever rejected: the resample counter stayed at zero and the selective-remasking path never ran. The output is complete and schema-valid; the comments value is elided here:

```
{"resource": "CPU", "nodes": 4, "processors": 16,
 "ppn": 4, "start_date": "2023-10-01",
 "end_date": "2023-10-31", "percent_allocated": 75,
 "comments": "..."}
```

D Scoring Details

Every schema@1 number in this paper is computed by bench/analysis/tables.py from the stored outputs. An output passes if json.loads parses it and jsonschema.validate (version 4.x, default settings) accepts it against the instance schema. Default validation does not enforce format keywords such as uuid or date, and we do not enable a format checker, so a well-typed string with a malformed format still passes for every method. We do not use the runners' internal validity flags, because their semantics differ: the Dgrammar runner records whether EOS was emitted with no [MASK] left, the LLaDA LAVE runner records its own grammar acceptance, and the Dream LAVE runner records a JSON validation result. For JSON-Bench the same script computes syn@1 (the output parses as JSON) and fun@1, which requires exact string equality after parsing and re-serializing both the output and the reference with json.dumps(..., indent=4); this preserves object-key order and compares the re-serialized numeric representation, so 1 and 1.0 differ. The case-insensitive variant lowercases both strings before the same comparison.

E Realized Unmask Batch Size

Figure 4 plots the per-instance mean K and the distribution of selective remasks on the headline cell (LLaDA-8B, JSB medium, n=511, seed 0). The mean realized K is 1.88 (median 1.95, minimum 1.11); 17 of 511 instances have a mean K below 1.5. 43.4% of instances trigger no violation at all, the median instance resamples once, and the three instances with the lowest mean K (1.11, 1.13, 1.15) finish in 3.1, 5.4, and 3.2 s with 332 to 447 resamples and no timeout. The upper bound near 2 comes from the sampler's per-step token quota with T=128 steps and 256 generated tokens, not from the backoff rule; $K_{\max}$=8 leaves headroom for shorter step budgets.

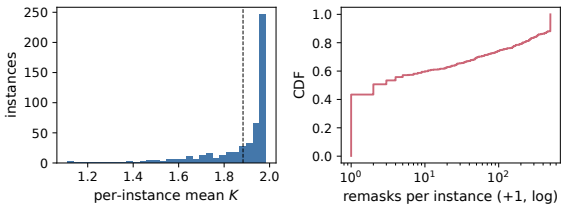


Figure 4: Left: histogram of per-instance mean unmask batch size K (dashed line: mean 1.88). Right: CDF of selective remasks per instance (log scale, shifted by one). LLaDA-8B, JSB medium_test, n=511.

F Engineering Details

Grammar-gated backoff. With $K_{\max}$=8, T=128, generation length 256, and block length 32, the unmask schedule of §4 is also bounded by the per-step token quota of the diffusion sampler, so the realized K stays near 2 (App. E).

Async mask overlap. compute_mask runs on a CPU worker launched right after the previous step's logits commit, so the next GPU forward starts without waiting; observed mask wait then drops from 5.6 ms (compute-bound) to 0.04 ms (sync-bound).

Per-backbone adapter. Model-specific bits sit behind three hooks: forward_fn (on Dream, a one-position causal shift aligning logits with the masking convention), the (mask_id, eos_id) pair, and the tokenizer; no other code paths differ across LLaDA-8B and Dream-7B.