EFFICIENT AND PARALLEL INFERENCE UNDER CFG CONSTRAINTS FOR DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Controlling language model outputs is essential for ensuring structural validity, reliability, and downstream usability, and diffusion language models are no exception. Recent advances in diffusion language model decoding have extended output control beyond regular constraints to context-free grammar (CFG) constraints. However, the prior CFG-constrained decoder can be up to four times slower than unconstrained decoding, forcing a trade-off between the correctness benefits of CFG constraints and the decoding efficiency of diffusion models. A key source of this overhead is sequential validity checking, which limits parallel token commitment and adds repeated validation costs. We propose EPIC, a CFG-constrained decoding framework designed to improve this efficiency-correctness trade-off. In order to reduce decoding overhead without sacrificing syntactic correctness, EPIC combines lexing memoization, relaxed compatible subset selection for parallel commit, and validation using Earley-style parsing instead of deterministic automata. This design enables multiple compatible tokens to be committed together while avoiding repeated lexing and expensive validation. Experiments on three benchmarks using four models show that EPIC improves the efficiency-correctness trade-off, bringing runtime close to unconstrained levels while maintaining comparable syntactic and functional correctness. Relative to the prior CFG-constrained decoder, EPIC achieves a best-case inference-time reduction of 67.2%. Our implementation is available at `https://anonymous.4open.science/r/EPIC-83DE/`.

1 INTRODUCTION

Driven by the success of diffusion models in vision, there has been growing interest in applying diffusion-based methods to language modeling (Ho et al., 2020; Sahoo et al., 2024; Nie et al., 2025). Diffusion language models (DLMs) generate text through iterative denoising rather than autoregressive (AR) next-token prediction, and masked diffusion language models (MDLMs) instantiate this paradigm by repeatedly refining partially masked sequences. Recent progress has shown that MDLMs can scale to large language models (LLMs), making DLMs a promising alternative to AR generation (Sahoo et al., 2024; Nie et al., 2025). Unlike AR models, which generate tokens strictly from left to right, DLMs can update multiple positions in parallel, offering a distinctive latency advantage (Kim et al., 2026).

Despite this advantage, unconstrained DLM decoding remains less reliable for structured outputs such as code, JSON, and molecular strings. Proposed tokens can violate inter-token dependencies and degrade sequence-level consistency since multiple positions may be filled independently (Kang et al., 2026; Kim et al., 2026). Furthermore, the nonsequential nature of DLM generation complicates syntax checking during decoding. Unlike autoregressive decoding, DLM decoding must reason about partially filled sequences whose unresolved masks may appear between generated tokens.

Recent work has begun to develop constrained decoding for DLMs motivated by these challenges. Suresh et al. (2025) studied regex constraints, while Mündler et al. (2026) extended constrained DLM decoding to context-free grammars (CFGs) through completion-based rejection sampling. More recently, Zhang et al. (2026) proposed LAVE, which uses lookahead generation to verify candidate updates under CFG constraints. These approaches demonstrate that CFG constraints can substantially improve structured output validity, but enforcing them introduces additional inference

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.

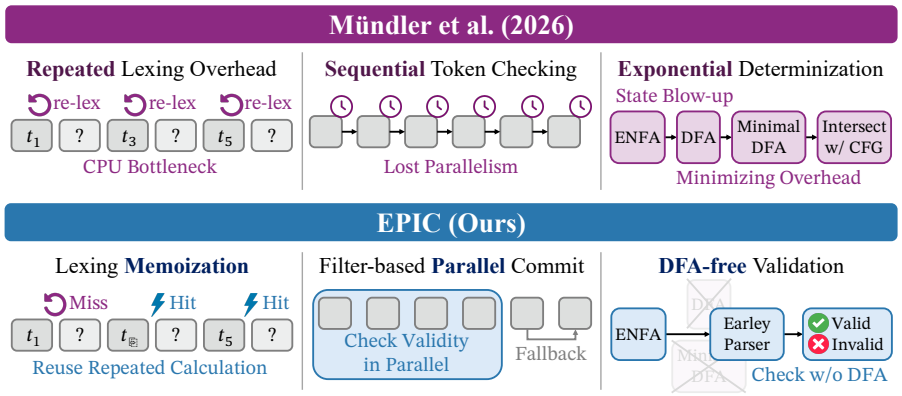


Figure 1: Overview of the bottlenecks in rejection-sampling-based CFG-constrained diffusion decoding and the corresponding components of EPIC.

cost during DLM decoding. This creates a central trade-off between structural correctness and the parallel efficiency of diffusion decoding.

We build on the rejection-sampling decoder of Mündler et al. (2026), which accepts a candidate update only when the remaining masked positions can still be completed into a sequence accepted by the target grammar. While effective, this procedure can be substantially slower than unconstrained decoding because each validity query requires repeated lexical and automaton processing. Moreover, candidate updates are validated sequentially, limiting parallel token commitment.

To improve this trade-off, we propose EPIC, a CFG-constrained decoding framework that reduces the major sources of overhead in this rejection-sampling pipeline, as illustrated in Figure 1. EPIC reuses lexical computations across similar partial outputs, performs DFA-free CFG validation directly on the lexed graph, and selects compatible candidate subsets for parallel commitment before final CFG verification. Together, these components reduce repeated lexing and validation costs while recovering parallel token commitment.

Our main contributions are as follows.

- We identify three major overheads in rejection-sampling-based CFG-constrained diffusion decoding, namely repeated lexing, sequential candidate validation, and DFA construction.
- We propose EPIC, a CFG-constrained decoding framework for DLMs that combines lexing memoization, relaxed compatible subset selection, and DFA-free validation to reduce these costs and recover parallel token commitment.
- We evaluate EPIC against two CFG-constrained DLM decoders across four models and three structured-generation benchmarks, showing an improved trade-off between inference efficiency and syntactic and functional correctness.

2 Related Work

We assume basic familiarity with regular languages, finite automata, context-free grammars (CFGs), lexing, and CFG parsing, and refer to Appendix A for formal background. We focus here on constrained decoding for diffusion language models (DLMs) and defer broader discussions of DLMs, AR constrained decoding, and formal-language-based generation to Appendix B.

Constrained decoding for DLMs has recently progressed from regular-language to context-free constraints. Suresh et al. (2025) proposed DINGO for regular-language constraints, while Mündler et al. (2026) introduced CFG-constrained decoding through completion-based rejection sampling. Given a candidate update to a partially masked output, the latter constructs an automaton representing its possible completions and accepts the candidate when this completion language is compatible with the target CFG. This formulation enables CFG constraints for nonsequential DLM generation, but requires repeated lexing, automaton determinization and minimization, and CFG compatibil-

ity checking. Moreover, candidate updates are checked and committed sequentially, reducing the parallelism available in diffusion decoding.

More recently, Zhang et al. (2026) proposed LAVE, which approaches CFG-constrained DLM decoding through model-guided lookahead. For each proposed token, LAVE samples several fillings of the remaining masked positions from the distributions predicted by the DLM, thereby constructing complete prefixes that can be checked by a standard grammar parser. The candidate is accepted when at least one lookahead prefix is grammatically extendable. LAVE additionally caches verified prefixes to recover from repeatedly rejected proposals. This provides an alternative to automata-based CFG completion checking, but its proposal and verification procedure still commits candidate tokens one at a time, with multiple lookahead prefixes considered during each verification.

Our work focuses on reducing the computational overhead of CFG completion checking while recovering multi-token commitment. Building on the completion-based formulation of Mündler et al. (2026), EPIC reuses lexical computations, avoids DFA construction during exact CFG validation, and uses a regular cover to identify candidate subsets that can be verified and committed together.

3 Problem Formulation

Let $G = (V, \Sigma_{\text{lex}}, P, S)$ be a context-free grammar, where Σ_{lex} is the lexeme vocabulary, the set of grammar terminals. The target constraint language is therefore $L(G) \subseteq \Sigma_{\text{lex}}^*$. This lexeme vocabulary should not be confused with either the byte-level alphabet Σ_{byte}, used to represent raw strings, or the tokenizer vocabulary Γ used by the diffusion language model. A partial model output is represented as $x \in (\Gamma \cup \{\texttt{[MASK]}\})^n$, where `[MASK]` denotes a mask.

Validity checking requires a lexical interface since the model operates over tokenizer-level sequences whereas the CFG is defined over lexeme sequences. We let Λ denote the lexical map that converts a token-level partial output x into a regular or graph representation of compatible lexeme sequences. We denote this representation by $\mathcal{A}_x$, with $L(\mathcal{A}_x) \subseteq \Sigma_{\text{lex}}^*$. For notational convenience, we write $C_x := L(\mathcal{A}_x)$ for the lexeme-sequence completion language induced by x, and $M(x) := \{i \in [n] \mid x_i = \texttt{[MASK]}\}$ for the set of masked positions.

At each diffusion step, the model predicts distributions over tokens for masked positions. Given a reveal budget k, the decoder selects a subset of masked positions $I = \{i_1, \ldots, i_k\} \subseteq M(x)$ and samples tokenizer tokens for these positions in parallel, $t_i \sim p_\theta(\cdot \mid x), \quad i \in I$. Equivalently, the model proposes updates $\Delta = \{(i, t_i) \mid i \in I\}$. Applying this update to x gives a new partial output defined position-wise by

$$(x \oplus \Delta)_j = \begin{cases} t_j & \text{if } (j, t_j) \in \Delta, \\ x_j & \text{otherwise.} \end{cases}$$

The central question in CFG-constrained DLM decoding is whether this token-level update preserves the possibility of completing the remaining masks into a lexeme sequence accepted by the grammar. We say that a partial output x is completable with respect to G if $L(G) \cap C_x \neq \emptyset$. A proposed update Δ is valid when

$$\textsc{Valid}_G(x, \Delta) = \mathbf{1}\left[L(G) \cap C_{x \oplus \Delta} \neq \emptyset\right].$$

This definition gives a rule for constrained diffusion decoding. If $\textsc{Valid}_G(x, \Delta) = 1$, the decoder may commit the proposed tokens. Otherwise, the proposal must be rejected. The completion-based baseline evaluates this condition by constructing an automaton for $C_{x \oplus \Delta}$ and checking its compatibility with the CFG. Our goal is to compute the same validity predicate more efficiently in the repeated and parallel setting induced by DLM decoding.

The multi-token nature of Δ is crucial. In unconstrained DLM decoding, all tokens in I can be committed simultaneously after one model forward pass. Under CFG constraints, however, a naive decoder must test the proposed tokens sequentially, $(i_1, t_{i_1}), (i_2, t_{i_2}), \ldots, (i_k, t_{i_k})$, because the validity of one token may depend on which other tokens have already been accepted. This sequential processing weakens the main advantage of DLMs. The problem we address is therefore efficient validity checking for multiple token updates, while preserving the CFG-completable output criterion over lexeme sequences.

4 PROPOSED METHOD

We propose **EPIC**, a decoding framework for **E**fficient and **P**arallel **I**nference under CFG **C**onstraints. Our method consists of three main components, namely lexing memoization, relaxed compatible subset selection for parallel commit, and DFA-free validation with Earley-style parsing. Each component addresses a distinct source of overhead in prior work. An overview of our framework is illustrated in Figure 2.

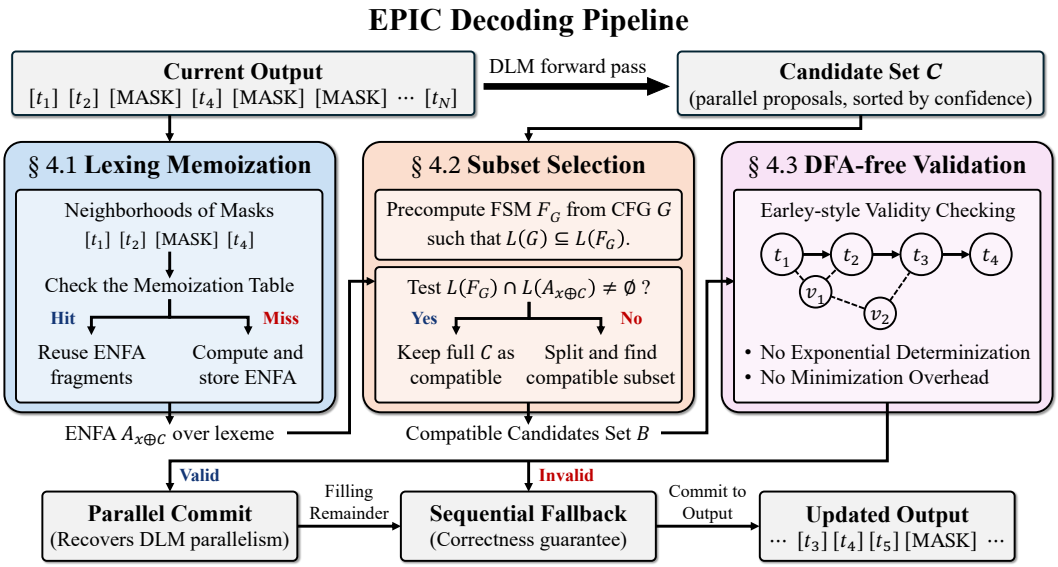


Figure 2: Overview of EPIC. EPIC combines lexing memoization, regular-cover batch selection, and DFA-free graph-parser validation to reduce the overhead of CFG-constrained decoding.

4.1 LEXING MEMOIZATION

The first component exploits locality in lexing. Although masks introduce lexing ambiguity, each ambiguity is determined by a local text segment and its adjacent mask boundaries. During diffusion decoding, an update modifies only a small part of the sequence, while the remaining mask positions stay fixed. As a result, consecutive validation queries often contain many unchanged local lexing contexts. We therefore memoize possible prefix and suffix lexings for segments adjacent to masks. When the same local context appears again, we reuse the stored lexing result during partial-language construction. This avoids repeated lexing of unchanged regions and reduces the cost of constructing the partial-output automaton or graph.

4.2 RELAXED COMPATIBLE SUBSET SELECTION FOR PARALLEL COMMIT

The second component is relaxed compatible subset selection, which addresses the sequential commit problem that significantly limits the parallelism of DLMs. When the number of denoising steps is small, a DLM reveals many tokens in a single step. This is a key advantage of DLMs over AR language models, but it can also make structured generation more difficult because independently proposed tokens may be mutually incompatible. Prior work performs sequential compatibility checking to ensure compatibility among independently sampled tokens. As a result, when the number of denoising steps is small, existing methods incur a larger slowdown relative to unconstrained decoding. The most direct solution would be to check the validity of multiple tokens simultaneously. This approach, however, does not reveal which tokens are compatible when the check fails. Finding the truly compatible subset would require a number of checks exponential in the number of tokens masked at the same time.

We avoid this cost by first selecting a subset that is compatible under a relaxed condition. Our implementation builds a regular cover by flattening the production rules of the given CFG. For a CFG G,

a regular cover is a finite automaton $\mathcal{F}_G$ that satisfies $L(G) \subseteq L(\mathcal{F}_G)$. Given a candidate batch C, we intersect this regular cover with the completion automaton induced by $x \oplus C$ and perform an emptiness check. This relaxed check is cheaper than exact CFG validation and safely rejects candidate subsets that are incompatible even under the relaxed condition. If the full batch is not cover-compatible, we recursively split the candidates by confidence and recover a smaller relaxed-compatible subset. A relaxed-compatible subset is not committed immediately. It is subsequently verified by the exact CFG completability checker, described in the next subsection. If exact verification rejects the selected subset, we recursively shrink it again and then fill the remaining reveal budget with the standard sequential constrained sampler.

4.3 DFA-free Validation with Earley-style Parsing

The third component is DFA-free validation. Prior work represented a lexed partial output as an epsilon-NFA (ENFA) and then determinized and minimized it before CFG validation. After this conversion, validation checked whether the automaton admitted some path whose sequence of lexemes could be generated by the CFG. Prior work implemented this search as dynamic programming over CFG nonterminals and automaton pairs, analogous to CYK-style recognition (Younger, 1967; Kasami, 1966) on a finite automaton. However, this pipeline incurs the cost of ENFA-to-DFA conversion at each decoding step, and this bottom-up formulation provides limited top-down guidance during the search.

In contrast, for exact validation, we avoid ENFA-to-DFA conversion by performing witness checking directly on the ENFA using an Earley-style graph parser. The parser combines bottom-up evidence from ENFA transitions with top-down predictions from the CFG. Intuitively, the search meets in the middle. The ENFA provides possible lexeme transitions, while the grammar restricts which partial derivations are relevant to the start symbol. Epsilon transitions are traversed without advancing the grammar state, while matching terminal-labeled transitions advance the corresponding parser item. Although the worst-case complexity is not substantially different from that of a CYK-style implementation, the procedure avoids repeated determinization and minimization and introduces top-down grammar guidance into the witness search. Thus, DFA-free validation preserves the completable output criterion while removing the per-step overhead of ENFA-to-DFA conversion. Detailed algorithms for the three main components of EPIC are provided in Appendix C, and Appendix D shows that these components preserve the same completable-output criterion used by the baseline decoder.

4.4 Failure recovery

Following prior work, we use the same automatic recovery procedure when decoding reaches a fixed resampling budget. This budget counts the number of rejected proposals over the entire decoding process. Even when the budget is exhausted, the current partial output is known to be completable, since every accepted update has been verified with a witness. Therefore, a valid witness completion always exists for the remaining masks. Under the recovery procedure of prior work, model sampling is terminated early and the output is completed by extracting a valid completion from a witness. This recovery step is not sampled from the model distribution, but provides a valid fallback when further rejection sampling is unlikely to succeed.

5 Experimental Results and Analysis

We evaluate EPIC to show that it reduces the inference cost of CFG-constrained decoding for DLMs while maintaining syntactic and functional correctness. Our experiments follow the evaluation protocol of prior work and cover three structured generation tasks: C++ code generation, JSON generation under instance-specific schemas, and SMILES generation. We organize the evaluation around three questions. First, we compare the relative inference time of EPIC and the prior CFG-constrained baseline across different denoising-step settings, using unconstrained decoding as the normalization reference. Second, we analyze where runtime is spent and how each component of EPIC reduces the runtime cost. Third, we perform an ablation study over lexing memoization, DFA-free graph validity checking, and relaxed compatible subset selection to isolate their individual and combined effects. We provide additional experimental details in Appendix E and additional results in Appendix F.

5.1 Experimental Settings

We evaluate the same four instruction-tuned DLMs as prior work, namely Dream, DreamCoder, LLaDA, and DiffuCoder. The models include both general-purpose DLMs and code-oriented variants. Following prior CFG-constrained decoding work, we use three structured-output benchmarks, C++ code generation, JSON generation with instance-specific schemas, and SMILES generation. The C++ task is based on the HumanEval-X C++ split (Zheng et al., 2023), while the JSON and SMILES tasks use the benchmarks from the prior evaluation suite (Mündler et al., 2026). We primarily compare EPIC with the prior completion-based CFG-constrained baseline, using unconstrained diffusion decoding as the reference for normalizing inference time. *Con.* denotes the prior CFG-constrained decoder, and *EPIC* denotes our constrained decoding framework. All constrained methods use the same task-specific grammars and lexical specifications.

For the main comparison, we measure inference time across denoising steps in $\{16, 32, 64, 128, 256\}$ with a maximum generation length of 256 tokens, and report the results as relative inference time normalized by unconstrained decoding. Smaller denoising step settings reveal more tokens per step and therefore place greater stress on the sequential validity checking bottleneck. We use greedy decoding and repeat runtime measurements with three seeds, $\{42, 43, 44\}$. We additionally report syntactic and functional correctness after automatic recovery to evaluate whether the efficiency gains of EPIC come at the cost of output correctness.

Table 1: Inference time relative to unconstrained decoding. For each method, we report the relative inference time normalized by unconstrained diffusion decoding, where 100% indicates the same runtime as unconstrained decoding. Con. denotes the prior CFG-constrained decoder, and EPIC denotes our method. The shaded column highlights EPIC.

Model	Steps	C++		JSON		SMILES		Average	
		Con.	EPIC	Con.	EPIC	Con.	EPIC	Con.	EPIC
Dream (7B)	16	143.86	98.44	176.32	109.36	102.09	103.43	140.76	103.74
	32	147.89	114.85	123.79	101.22	100.45	101.07	124.04	105.71
	64	114.41	108.12	108.19	101.36	100.67	101.18	107.76	103.55
	128	106.03	102.65	104.42	100.92	100.30	100.29	103.58	101.29
	256	102.12	102.92	102.39	100.82	98.96	99.21	101.16	100.98
DreamCoder (7B)	16	388.60	127.34	178.21	89.19	101.14	103.14	222.65	106.56
	32	269.76	140.51	132.63	89.52	100.72	101.34	167.70	110.46
	64	182.52	128.04	118.02	96.74	100.41	100.65	133.65	108.48
	128	133.57	110.46	106.71	98.21	100.25	100.26	113.51	102.98
	256	117.23	106.50	101.26	98.84	100.03	100.01	106.17	101.78
LLaDA (8B)	16	112.19	74.84	123.15	74.90	112.40	94.95	115.91	81.56
	32	95.49	64.07	111.99	77.51	102.77	92.22	103.42	77.93
	64	93.26	76.78	104.67	87.55	98.76	95.49	98.90	86.61
	128	100.52	93.27	102.42	95.21	98.17	96.03	100.37	94.84
	256	102.30	100.71	100.09	98.25	98.94	98.55	100.44	99.17
DiffuCoder (7B)	16	187.50	108.07	165.58	108.72	103.93	104.96	152.34	107.25
	32	182.20	123.18	125.93	96.37	103.93	102.58	137.35	107.38
	64	138.87	102.33	111.23	97.45	103.36	101.01	117.82	100.26
	128	121.15	103.98	102.38	96.46	108.05	104.62	110.53	101.69
	256	118.54	107.80	95.17	92.97	103.17	102.10	105.63	100.96
	Average	147.90	104.74	119.73	95.58	101.93	100.15	123.18	100.16

5.2 Main Results

Tables 1 and 2 summarize the main results. Table 1 reports relative inference time for four models and three datasets, normalized by unconstrained decoding, while Table 2 reports syntactic and functional correctness after automatic recovery. Thus, a value of 100 indicates the same runtime as unconstrained decoding, while smaller values indicate faster decoding. Across C++ and JSON, EPIC outperforms the prior CFG-constrained baseline in 39 of the 40 settings. In several cases, EPIC achieves relative inference time below 100. This does not imply negative constraint checking cost.

Table 2: Syntactic and functional correctness after automatic recovery. S. and F. denote syntactic and functional correctness, respectively, and Drm., Drm.C., L., and Diff.C denote Dream, DreamCoder, LLaDA, and DiffuCoder. The best result for each setting is shown in bold, and the shaded columns indicate EPIC. All values are percentages.

Model	Step	C++						JSON						SMILES					
		Uncon.		Con.		EPIC		Uncon.		Con.		EPIC		Uncon.		Con.		EPIC	
		S.	F.	S.	F.	S.	F.	S.	F.	S.	F.	S.	F.	S.	F.	S.	F.	S.	F.
Drm.	16	35.4	10.4	**100**	**11.6**	**100**	**11.6**	32.4	13.2	**100**	**23.7**	**100**	**23.7**	61.7	**0.6**	**100**	**0.6**	**100**	**0.6**
	32	67.7	15.2	**100**	**16.5**	**100**	**16.5**	77.6	43.4	**100**	**46.0**	**100**	**46.0**	70.7	**3.0**	**100**	**3.0**	**100**	**3.0**
	64	80.5	18.1	**99.6**	18.5	**99.6**	**18.9**	92.3	53.7	**100**	**54.8**	**100**	**54.8**	72.5	3.0	**100**	**3.6**	**100**	**3.6**
	128	89.0	13.8	**100**	**14.6**	99.8	14.4	90.4	51.5	**100**	**53.3**	**100**	**53.3**	73.1	3.6	**100**	**4.2**	**100**	**4.2**
	256	82.9	12.8	**100**	13.4	**100**	**14.0**	91.2	51.8	**100**	**53.7**	**100**	**53.7**	72.5	3.0	**100**	**4.2**	**100**	**4.2**
Drm.C.	16	17.7	6.7	**99.8**	8.5	**99.8**	**9.1**	79.0	41.9	**100**	**44.1**	**100**	44.0	78.4	**4.2**	**100**	**4.2**	**100**	**4.2**
	32	54.3	19.5	**99.4**	20.7	**99.4**	**21.3**	87.5	52.2	**100**	**53.3**	**100**	**53.3**	79.0	3.6	**100**	**4.2**	**100**	**4.2**
	64	72.6	26.8	99.2	**27.2**	**100**	26.6	90.8	**53.7**	**100**	53.3	**100**	53.3	74.9	**2.4**	**100**	**2.4**	**100**	**2.4**
	128	73.8	28.0	**100**	**28.7**	**100**	**28.7**	93.0	55.9	**100**	**56.6**	**100**	**56.6**	74.3	**2.4**	**100**	**2.4**	**100**	**2.4**
	256	84.1	**28.0**	**100**	27.4	**100**	27.4	94.1	56.3	**100**	**56.6**	**100**	**56.6**	74.3	**2.4**	**100**	**2.4**	**100**	**2.4**
L.	16	5.5	0.6	99.6	4.1	**100**	**4.5**	57.4	39.0	**100**	**44.1**	**100**	**44.1**	41.3	1.2	**100**	**1.8**	**100**	**1.8**
	32	9.1	3.0	99.2	5.3	**99.4**	**6.3**	75.4	44.1	**100**	**50.0**	**100**	**50.0**	60.5	0.6	**100**	**1.2**	**100**	**1.2**
	64	42.7	10.4	99.8	**11.0**	**100**	**11.0**	79.8	44.5	**100**	**50.0**	**100**	**50.0**	65.9	1.2	**100**	**1.8**	**100**	**1.8**
	128	82.9	**15.9**	**100**	**15.9**	**100**	**15.9**	87.9	48.2	**100**	**52.9**	**100**	**52.9**	65.9	**1.8**	**100**	**1.8**	**100**	**1.8**
	256	96.3	**25.6**	**100**	25.0	**100**	25.0	91.2	49.6	**100**	**52.9**	**100**	**52.9**	65.3	**2.4**	**100**	**2.4**	**100**	**2.4**
Diff.C	16	25.0	12.8	**99.8**	14.0	**99.8**	**14.6**	76.8	42.3	**100**	**44.1**	**100**	**44.1**	70.7	0.0	**100**	**0.6**	**100**	**0.6**
	32	36.0	15.2	**99.8**	**18.3**	**99.8**	17.7	87.9	47.8	**100**	**49.6**	**100**	49.3	71.9	0.0	**100**	0.0	**100**	0.0
	64	53.7	17.7	99.8	**18.3**	**100**	**18.3**	84.6	45.2	**100**	**47.4**	**100**	**47.4**	62.3	**0.6**	**100**	**0.6**	**100**	**0.6**
	128	60.4	25.6	**100**	**29.9**	99.8	29.7	77.9	42.3	**100**	**43.8**	**100**	**43.8**	49.7	**1.8**	**100**	**1.8**	**100**	**1.8**
	256	93.9	43.9	**100**	45.5	**100**	**45.7**	77.9	41.9	**100**	**43.4**	**100**	**43.4**	47.9	**2.4**	**100**	**2.4**	**100**	**2.4**

Rather, constrained decoding can terminate early once a complete valid output and EOS have been formed, whereas the unconstrained diffusion sampler executes the full configured schedule. This effect is most visible on JSON and LLaDA. The improvement of EPIC is especially pronounced on C++, where the grammar is more complex than those used for JSON and SMILES and requires more production rules and automaton states.

On SMILES, the relative-time difference between the baseline and EPIC is generally small, except for LLaDA. This is because the fixed CFG and lexical specification used for SMILES are much smaller than those for the other tasks, so the baseline already stays close to unconstrained decoding. As a result, there is less room for our optimizations to reduce per-check cost. The LLaDA case is different because of its block-wise decoding schedule. LLaDA divides the canvas into blocks and fills a fixed number of masks within each block, whereas Dream, DreamCoder, and DiffuCoder determine the number and positions of unmasked tokens dynamically from a global timestep schedule. This makes the candidate sets of LLaDA more localized and stable, which can make compatible subset selection more effective. Thus, even though graph parsing and caching provide limited gains on SMILES, LLaDA can still benefit from a reduction in sequential exact checks. We provide further analysis in Section 5.3.

The benefit of EPIC is larger when the number of denoising steps is small. This matches our motivation, since smaller step counts require the decoder to unmask more tokens per step and therefore amplify the sequential validation bottleneck. The effect is particularly clear for DreamCoder. For example, on DreamCoder with C++ at 16 denoising steps, the prior CFG-constrained baseline requires 388.60% of the unconstrained decoding time, whereas EPIC requires only 127.34%. Equivalently, the overhead beyond unconstrained decoding is reduced from 288.60% to 27.34%.

Table 2 shows that the substantial runtime reductions of EPIC do not come at the cost of syntactic correctness. After automatic recovery, both constrained methods achieve nearly perfect syntactic correctness across all models, datasets, and denoising-step settings. The few residual failures occur when witness extraction does not finish within the configured timeout. EPIC closely matches the

Table 3: Runtime breakdown of the baseline and EPIC. We report the time spent in lexing, ENFA-to-DFA conversion, and validation. Validation includes both CFG checks and regular cover checks. For graph parser variants, DFA time comes from regular cover selection.

Model	Method	C++			JSON			SMILES		
		Lex. (s)	DFA (s)	Val. (s)	Lex. (s)	DFA (s)	Val. (s)	Lex. (s)	DFA (s)	Val. (s)
Dream	Con.	50.8	47.2	49.8	24.4	27.1	2.0	0.3	1.0	0.7
	EPIC	12.0	11.8	17.1	5.9	5.6	2.1	0.1	0.2	0.5
DreamCoder	Con.	96.9	216.8	179.8	17.3	31.0	1.7	0.3	0.3	0.1
	EPIC	33.4	68.5	36.7	4.3	6.4	1.8	0.1	0.0	0.4
LLaDA	Con.	47.1	20.0	56.6	21.7	7.5	1.0	2.6	3.0	33.1
	EPIC	21.6	3.7	11.6	4.2	1.0	0.9	1.5	0.4	7.1
DiffuCoder	Con.	83.4	102.8	146.2	26.2	40.4	2.5	1.2	1.8	18.4
	EPIC	18.3	18.1	25.0	5.3	8.2	2.6	0.2	0.2	1.6

Table 4: Component analysis of EPIC. Memoization reports reused lexing results and hit rate, while parallelism reports the average selected subset size and regular cover selection success rate.

Model	C++				JSON				SMILES			
	Memoization		Parallelism		Memoization		Parallelism		Memoization		Parallelism	
	# Hit	Rate	# Tok.	Rate	# Hit	Rate	# Tok.	Rate	# Hit	Rate	# Tok.	Rate
Dream	141,128	67.2	3.4	97.4	757,257	88.9	2.7	93.9	5,116	43.3	2.2	97.1
DreamCoder	429,345	75.2	3.7	93.5	1,178,326	95.3	4.9	95.0	4,590	44.5	1.9	91.6
LLaDA	62,347	53.1	5.1	94.5	269,782	83.5	7.0	97.1	13,884	46.5	3.6	87.8
DiffuCoder	300,019	72.3	3.7	90.9	970,458	90.9	3.9	95.2	31,442	65.0	2.9	92.1

prior CFG-constrained baseline, with differences limited to at most 0.2 percentage points when EPIC is lower. This indicates that recovering multi-token commitment does not meaningfully weaken the structural correctness provided by CFG-constrained decoding.

Functional correctness shows the same overall pattern. Despite changing the token commitment order and committing multiple compatible tokens together, EPIC remains closely comparable to the sequential CFG-constrained baseline across all settings. The largest decrease relative to the baseline is only 0.6 percentage points, while EPIC also improves functional correctness in several settings. Together with the runtime results in Table 1, these results show that EPIC improves decoding efficiency while largely preserving both syntactic and functional correctness.

5.3 Further Analysis

We further analyze EPIC to verify that its empirical gains come from the intended sources of efficiency improvement. All results in this subsection are measured at 32 denoising steps. We first examine the runtime breakdown in Table 3. The table compares the baseline and EPIC in terms of lexing time, DFA-construction time, and validation time. EPIC reduces overhead in almost all components. Lexing memoization yields roughly a $2\times$ to $4\times$ reduction in lexing time, with the largest gains appearing on C++ and JSON, where lexing accounts for a substantial fraction of the baseline overhead. The DFA-construction time also decreases substantially. Although EPIC does not construct a DFA for exact validation of each partial output, it still constructs automata during relaxed compatible subset selection. This cost remains small because regular-cover checks are used only for parallel commit, while the sequential fallback runs without DFA construction. Validation time is also reduced, especially on C++ and LLaDA. C++ benefits from DFA-free validation because its grammar is more complex than the grammars used for JSON and SMILES. LLaDA benefits more strongly from validation reduction since its fixed-number unmasking policy makes parallel commit more stable than in the other models.

Table 5: Ablation results for EPIC. Starting from the baseline (Con.), we separately add ① lexing memoization, ② DFA-free validation with Earley-style parsing, and ③ relaxed compatible subset selection for parallel commit. EPIC enables all three components.

Model	C++					JSON					SMILES				
	Con.	①	②	③	EPIC	Con.	①	②	③	EPIC	Con.	①	②	③	EPIC
Dream	900	843	863	900	**700**	1203	1095	1237	1162	**983**	**516**	541	540	545	519
DreamCoder	1582	1385	1261	1378	**824**	1089	939	1109	986	**735**	**523**	540	540	543	**523**
LLaDA	791	772	738	601	**531**	1554	1450	1619	1179	**1076**	636	655	628	609	**570**
DiffuCoder	1278	1152	1085	1119	**862**	1314	1172	1307	1195	**1005**	584	598	582	601	**576**

We next report component-level statistics of EPIC in Table 4. Memoization achieves hit rates above 50% in most settings, and C++ and JSON show particularly large numbers of reused lexing results. These high hit rates indicate that diffusion decoding repeatedly visits similar local lexing contexts and that the memoization table captures meaningful reuse. The parallelism statistics show that regular-cover selection succeeds in about 94% of selection attempts. They also show that EPIC selects more than two tokens for parallel commitment on average across all models and datasets. The effect is most pronounced for LLaDA. Unlike the other models, LLaDA reveals a fixed number of tokens in each decoding block, which provides more stable candidates for relaxed compatible subset selection. On JSON, LLaDA commits an average of 7.0 tokens in parallel, demonstrating that EPIC can recover substantial diffusion parallelism under CFG constraints.

5.4 Ablation Results

Table 5 shows the ablation results at 32 denoising steps. We evaluate each component of EPIC by adding it separately to the baseline. The results further confirm the observations from the runtime breakdown analysis. The components of EPIC provide distinct sources of speedup, and their effects become clearer in the ablation study. The gains are especially pronounced on C++. All three components improve or match inference time for every model. This consistent improvement indicates that EPIC is particularly effective when the constraint language is complex enough to make repeated lexing, automaton construction, and exact CFG validation costly. On JSON, DFA-free validation alone sometimes increases runtime. This is likely because JSON grammars are simpler than the C++ grammar, so the baseline intersection-based validation already incurs relatively small overhead. In this setting, the additional graph-parser overhead can outweigh the reduction from avoiding DFA construction. On SMILES, the gains are limited because the baseline constraint overhead is already small. The full EPIC configuration generally achieves the best inference time by combining the complementary benefits of the three components. Pairwise combinations further show incremental improvements over single component variants in most settings, as reported in the full ablation results in Appendix F.3.

6 Conclusions

We presented EPIC, an efficient CFG-constrained decoding framework for diffusion language models. EPIC reduces the main overheads of prior constrained diffusion decoding through lexing memoization, relaxed compatible subset selection for parallel commit, and DFA-free validation with Earley-style graph parsing. These components preserve the completable output criterion while reducing repeated lexing, avoiding per-step DFA construction for exact validation, and recovering parallel token commitment. Experiments on C++, JSON, and SMILES with four DLMs show that EPIC substantially reduces inference time while maintaining correctness comparable to the baseline. At 16 denoising steps, where parallelism matters most, EPIC achieves a $3.1\times$ speedup over the prior CFG-constrained decoder on DreamCoder for C++ and reduces the relative inference time from 388.6% to 127.3% of unconstrained decoding. Component analysis further shows memoization hit rates above 50% in most settings and an average selected subset size above two tokens across all models and datasets. Future work could further reduce overhead through incremental graph parsing, tighter modeling of finite token budgets, and extensions beyond CFGs to lightweight context-sensitive constraints such as type or schema-dependent semantic checks.

AI USE STATEMENT

In this work, we used generative AI tools to assist with the implementation of experimental code. We have not used generative AI tools for other tasks requiring disclosure, and the rest of the required-disclosure tasks are not applicable to this work. Additionally, we used generative AI tools to edit the manuscript for readability and language polishing. We have reviewed all AI-assisted work. AI-assisted experimental code was manually reviewed and tested for correctness, and all AI-assisted manuscript edits were reviewed by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, volume 34, pp. 17981–17993, 2021.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding LLMs the right way: Fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 3658–3673, 2024.

Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 10932–10952, 2023.

Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. DiffuCoder: Understanding and improving masked diffusion models for code generation. In *Proceedings of the 14th International Conference on Learning Representations*, pp. 15723–15750, 2026.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in neural information processing systems*, volume 33, pp. 6840–6851, 2020.

John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz (eds.), *Theory of Machines and Computations*, pp. 189–196, 1971.

John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

Wonjun Kang, Kevin Galim, Seunghyuk Oh, Minjae Lee, Yuchen Zeng, Shuibai Zhang, Coleman Hooper, Yuezhou Hu, Hyung Koo, Nam Ik Cho, and Kangwook Lee. ParallelBench: Understanding the trade-offs of parallel decoding in diffusion LLMs. In *Proceedings of the 14th International Conference on Learning Representations*, volume 2026, pp. 89900–89938, 2026.

Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*, 1966.

Bumjun Kim, Dongjae Jeon, Moongyu Jeon, and Albert No. Dependency-aware parallel decoding via attention for diffusion llms. arXiv preprint arXiv:2603.12996, 2026.

Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. In *Proceedings of the 1st Conference on Language Modeling*, 2024.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 32819–32848, 2024.

Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In *Robustness in Language and Speech Technology*, pp. 153–163. Springer Netherlands, 2001.

Niels Mündler, Jasper Dekoninck, and Martin Vechev. Constrained decoding of diffusion LLMs with context-free grammars. In *Proceedings of the 14th International Conference on Learning Representations*, pp. 99362–99408, 2026.

Mark-Jan Nederhof. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44, 2000.

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. In *Advances in Neural Information Processing Systems*, volume 38, pp. 50608–50646, 2025.

NousResearch. JSON mode eval. https://huggingface.co/datasets/NousResearch/json-mode-eval, 2024.

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

Fernando C. N. Pereira and Rebecca N. Wright. Finite-state approximation of phrase structure grammars. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pp. 246–255, 1991.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 130136–130184, 2024.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, 2021.

Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

Tarun Suresh, Debangshu Banerjee, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. DINGO: Constrained inference for diffusion LLMs. In *Advances in Neural Information Processing Systems*, volume 38, pp. 160518–160551, 2025.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. SynCode: LLM generation with grammar augmentation. *Transactions on Machine Learning Research*, 2025.

David Weininger. SMILES, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

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

Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream-Coder 7B: An open diffusion language model for code. arXiv preprint arXiv:2509.01142, 2025.

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

Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

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.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5673–5684, 2023.

A PRELIMINARIES

This section reviews the formal language theory and compiler theory background used throughout the paper. We focus on the concepts necessary to define CFG-constrained decoding, analyze the computational bottlenecks, and analyze the correctness of our proposed procedures.

A.1 REGULAR LANGUAGES AND FINITE AUTOMATA

A regular language is a set of strings that can be recognized by a finite automaton, or described by a regular expression (Sipser, 1997; Hopcroft & Ullman, 1979). A DFA is defined as a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. An NFA generalizes a DFA by allowing multiple outgoing transitions for the same input symbol and may also include ε-transitions. Every NFA can be converted to an equivalent DFA via subset construction, although the resulting DFA may contain exponentially many states in the worst-case (Sipser, 1997; Hopcroft & Ullman, 1979).

As in prior work, we use NFAs to represent the partial outputs with masked regions. In prior CFG-constrained decoding pipelines, this NFA is determinized and then minimized before CFG compatibility checking. For a DFA with state set Q over alphabet Σ, Hopcroft's algorithm runs in $O(|\Sigma||Q| \log |Q|)$ time in the worst-case (Hopcroft, 1971). Since the automaton is reconstructed from many partial outputs during decoding, repeated determinization and minimization become a non-negligible source of overhead.

A.2 CONTEXT-FREE GRAMMARS

A CFG is defined as a tuple $G = (V, \Sigma, P, S)$, where V is a finite set of nonterminals, Σ is a finite set of terminals, P is a set of productions of the form $A \to \alpha$ with $A \in V$ and $\alpha \in (V \cup \Sigma)^*$, and $S \in V$ is the start symbol. The language generated by G is denoted by $L(G)$. Context-free languages (CFLs) are more expressive than regular languages, and can express recursive structures such as balanced parentheses, nested expressions, and programming languages.

Prior work uses a closure property that the intersection of a CFL and a regular language is context-free (Sipser, 1997). This property is used to formulate validity checking as an emptiness test over the intersection between the target CFL and the possible completions of the partial output. In such a construction, the intersection grammar has nonterminals indexed by CFG nonterminals and pairs of automaton states, which introduces a cubic dependence on the number of automaton states. This leads to a validity-checking cost of $O\left(|P||Q|^3 + |P||\Sigma||Q|^2\right)$.

Our relaxed check follows the classical idea of finite-state or regular approximation of CFG (Pereira & Wright, 1991; Nederhof, 2000; Mohri & Nederhof, 2001). We view the CFG as a recursive transition network, where each nonterminal corresponds to a subautomaton with an entry state and an exit state. We then flatten this network into a finite automaton by replacing each recursive call to a nonterminal with a transition to that nonterminal's entry state, and by connecting its exit state to the possible continuation states. After flattening, the automaton no longer records which call site entered a nonterminal. Consequently, the automaton may accept additional lexeme sequences, but every CFG derivation is still preserved as a valid path. The resulting automaton is therefore a regular cover $\mathcal{F}_G$ satisfying

$$L(G) \subseteq L(\mathcal{F}_G).$$

A.3 LEXERS AND PARSERS

Practical parsers rarely operate on raw character sequences directly. A typical compiler front end first applies lexing, which maps a character-level string into a sequence of tokens or lexemes, and then applies parsing to verify whether the token sequence conforms to a grammar. Additional context-sensitive checks, such as type checking or name resolution, are often performed after parsing.

This separation is important for constrained decoding since language models generate tokens at the token level, whereas CFGs are usually defined over lexemes. Partial outputs with masked regions

make lexing ambiguous, since a generated token may belong to a lexeme on its left, a lexeme on its right, or a lexeme spanning a masked region.

Earley parsing is a general CFG parsing algorithm that combines top-down prediction with bottom-up completion (Earley, 1970). Unlike purely bottom-up methods such as CYK, Earley-style parsing can use top-down grammar predictions to restrict which partial derivations are considered during parsing. Our DFA-free validity checker adopts this principle for witness checking over graph-structured partial outputs.

B Additional Related Work

This section provides additional related work that complements Section 2. We discuss diffusion language models, constrained decoding for autoregressive language models, and constrained decoding methods for diffusion language models.

B.1 Diffusion Language Models

Diffusion language models (DLMs) provide an alternative to autoregressive text generation (Austin et al., 2021; Lou et al., 2024; Sahoo et al., 2024; Nie et al., 2025). Instead of generating tokens from left to right, DLMs iteratively refine a partially observed sequence, often starting from a sequence of masked tokens. At each denoising step, the model predicts tokens for one or more masked positions conditioned on the current partially filled sequence.

Masked diffusion language models (MDLMs) instantiate this idea using a discrete masking process (Sahoo et al., 2024; Nie et al., 2025). During generation, a model repeatedly replaces masked positions with sampled tokens until the sequence is fully specified. This generation process enables nonsequential and potentially parallel token prediction, since multiple masked positions can be updated within the same denoising step. The degree of parallelism is typically controlled by the number of denoising steps or the number of tokens committed per step. Fewer steps allow more tokens to be generated per forward pass, improving speed but often increasing the risk of inconsistent or low-quality outputs (Kim et al., 2026).

This parallel generation makes DLMs attractive for efficient decoding. At the same time, it creates new challenges for structured generation. Since multiple tokens may be sampled independently before being committed, their joint compatibility with syntactic constraints is not guaranteed. This issue motivates constrained decoding methods that can preserve the parallelism of DLMs while enforcing formal language constraints.

B.2 Constrained Decoding for AR Models

Constrained decoding aims to ensure that model outputs conform to a predefined set of constraints. In many applications, this set is specified by a formal language, such as a regular expression, a JSON schema, or a context-free grammar. Constrained decoding has been widely studied for autoregressive language models, where generation proceeds from left to right and the validity of each next token can be checked against the prefix (Scholak et al., 2021; Geng et al., 2023; Park et al., 2025).

For regular constraints, valid next tokens can be computed using finite automata (Koo et al., 2024; Willard & Louf, 2023). For context-free constraints, the decoder must account for nested and recursive structures, which requires parsing-based or grammar-based methods (Geng et al., 2023; Beurer-Kellner et al., 2024; Ugare et al., 2025). Existing approaches differ in whether they mask invalid tokens before sampling, reject invalid samples after sampling, or combine both strategies (Willard & Louf, 2023; Beurer-Kellner et al., 2024; Ugare et al., 2025). These methods are effective for autoregressive generation because the prefix grows monotonically from left to right. The DLM setting breaks this assumption. A partial output may contain multiple masks, and updates can occur at arbitrary positions. As a result, prefix-based constrained decoding algorithms cannot be directly applied. This motivates validity checking procedures that reason about whether a partially filled sequence can still be completed into a valid string.

Algorithm 1 EPIC Decoding under CFG Constraints

Require: Prompt p, diffusion LM M, CFG G, lexical map Λ, answer length L, steps T
Ensure: Output tokens x

Compile Λ and initialize the lexing memoization table.
Construct a regular cover $\mathcal{F}_G$ by flattening the production rules of G.
Initialize x with p followed by L copies of `[MASK]`.
for all diffusion blocks **do**
    for $t = 1, \ldots, T$ **do**
    ▷ Parallel proposal from the diffusion model
        Run M on x and score proposals for the currently eligible masked positions.
        Compute the reveal budget n_t for this denoising step.
        Form C from up to n_t highest-confidence eligible non-EOS proposals.
    ▷ Select a relaxed-compatible batch and verify it exactly before committing
        $B \leftarrow \textsc{RelaxedSubsetSelect}(x, C, \mathcal{F}_G, G, \Lambda)$.
        Commit all candidates in B to x.
    ▷ Fall back to sequential rejection sampling if verification fails or commits remain
        for $j = 1, \ldots, n_t - |B|$ **do**
            Select the next highest-confidence candidate c not already committed.
            while $\textsc{DFAFreeValidate}(x \oplus c, G, \Lambda)$ is false **do**
                Reject c, suppress its logit, and select the next candidate.
            end while
            Commit the accepted candidate c to x.
        end for
    end for
end for
return x

B.3 Constrained Decoding for DLMs

Several recent works have begun to study constrained decoding for diffusion language models. Suresh et al. (2025) proposed DINGO, a constrained inference method for DLMs under regular-language constraints. Their method showed that formal constraints can be integrated into non-autoregressive diffusion decoding. However, regular constraints cannot capture many structured output settings that require nested or recursive structure, such as nested JSON objects, arithmetic expressions, and programming-language syntax.

The closest prior work to ours is Mündler et al. (2026), which introduced CFG-constrained decoding for diffusion LLMs. Their framework formulates validity checking as an infilling-based completion and reduces it to checking the non-emptiness of the intersection between a CFG and a regular language representing possible completions of the partial output. This approach enables CFG constraints for diffusion language models and multi-region infilling, but it requires repeated lexing, automaton construction, DFA minimization, and intersection-based emptiness checking during decoding. Their paper explicitly constructs an NFA for the partial output, converts it to a DFA, minimizes it, and then performs CFG intersection-based checking.

C Detailed Algorithms

We provide implementation-level pseudocode for EPIC. These algorithms clarify how EPIC preserves the completable-output criterion of prior CFG-constrained decoding while changing how this criterion is evaluated during diffusion decoding. Algorithm 1 presents the main decoding loop, including proposal generation, relaxed subset selection, exact verification, and token commitment. Algorithm 2 describes how lexical representations are stored and reused across partial outputs. Algorithm 3 specifies the Earley-style graph parser used for DFA-free validity checking. Algorithm 4 gives the compatible subset selection for multiple-token proposals under the relaxed regular cover.

C.1 Overall Decoding Procedure

Algorithm 1 summarizes the overall decoding procedure of EPIC. At each denoising step, the model scores the currently masked positions and proposes candidate tokens for multiple positions. The role of the constrained decoder is to decide which of these updates can be committed without losing the possibility of completing the sequence into a valid string in $L(G)$. The main difference from the baseline lies in how this validity decision is organized. The baseline effectively checks candidate tokens one at a time using exact CFG validation. This sequentializes the decoding process and weakens the main advantage of diffusion language models. In contrast, EPIC first calls RelaxedSubsetSelect to search for a compatible subset that may be committed together. This subset selection step is intentionally conservative. It may return a smaller or empty batch, but any nonempty batch must still pass exact verification before commitment.

Algorithm 2 Lexing Memoization for Partial Outputs

Require: Partial output x, lexical map Λ, lexing cache $\mathcal{C}_{\text{lex}}$
Ensure: ε-NFA $\mathcal{A}_x$ representing possible lexeme sequences
Merge consecutive generated tokens in x into text spans separated by masked regions.
Initialize an empty ε-NFA $\mathcal{A}_x$.
for all fixed text spans w in order **do**
    if $(w, \text{boundary}) \in \mathcal{C}_{\text{lex}}$ **then**
        Retrieve the cached lexing result for w.
    else
        Compute the lexing result for w using Λ.
        Store the computed result in $\mathcal{C}_{\text{lex}}$.
    end if
    Add span-level transitions to $\mathcal{A}_x$ using the retrieved or computed lexing result.
end for
Complete the construction by adding transitions for masked gaps and lexemes spanning them.
return $\mathcal{A}_x$

For candidates that are not committed through the batch filter, the algorithm falls back to the exact rejection loop. This fallback is important for two reasons. First, it preserves the behavior of rejection-based constrained decoding when parallel commitment is not possible. Second, it separates efficiency from correctness. Relaxed compatible subset selection is used only to propose a subset that is likely to pass exact validation, thereby reducing the number of sequential checks. The final accept-or-reject decision is still made by DFAFreeValidate. Thus, every token committed by Algorithm 1 is still justified by the completable-output condition.

C.2 Lexing Memoization

Algorithm 2 describes the lexing memoization procedure. Since the CFG is defined over lexemes while the diffusion model operates over a tokenizer vocabulary, validity checking first requires an ε-NFA representation of lexeme sequences compatible with the current masked string. The automaton construction follows the partial-output lexing procedure, which handles fixed text spans, masked gaps, and lexemes that may span masked gaps.

EPIC applies memoization to the lexing of text spans. For each span, the procedure first checks whether the same span has already been lexed under the same boundary condition. If the result is cached, it is reused during the construction of $\mathcal{A}_x$. If the entry is missing, the span is lexed using the lexical map Λ, and the result is inserted into $\mathcal{C}_{\text{lex}}$.

The output of the procedure is an ε-NFA $\mathcal{A}_x$. Paths in this automaton correspond to lexeme sequences that are compatible with the current partial output. Lexing memoization does not approximate or change this represented language. It only changes the construction procedure by avoiding repeated computation for local contexts that have already been processed. Thus, lexing memoization reduces the cost of constructing $\mathcal{A}_x$ without changing the completable-output check.

Algorithm 3 DFA-Free Validation with Earley-style Parsing

Require: Partial output x, CFG $G = (V, \Sigma, P, S)$, lexical map Λ
Ensure: Whether x can be completed into a string in $L(G)$

$\mathcal{A}_x \leftarrow \textsc{MemoizedLexing}(x, \Lambda, \mathcal{C}_{\text{lex}})$.
Initialize a worklist with Earley-style items at the start nodes of $\mathcal{A}_x$.
Initialize an empty visited set for parser items.
while the worklist is not empty **do**
    Pop an item from the worklist.
    if the item has already been visited **then**
        Continue to the next item.
    end if
    Mark the item as visited.
    ▷ Traverse ε-transitions without advancing the grammar state
    for all ε-edges from the item's current graph node to q' **do**
        Add the same parser item at graph node q' to the worklist.
    end for
    if the item expects a nonterminal **then**
    ▷ Prediction
        Add predicted items for productions of the expected nonterminal to the worklist.
    else if the item expects a terminal **then**
    ▷ Scanning over graph edges
        Follow matching terminal-labeled edges in $\mathcal{A}_x$ and add advanced items to the worklist.
    else
    ▷ Completion
        Advance items that were waiting for the completed nonterminal at the same graph node.
        Add the advanced items to the worklist.
    end if
    if an accepting item for the start symbol reaches an accepting node of $\mathcal{A}_x$ **then**
        return true
    end if
end while
return false

C.3 DFA-Free Validation with Earley-style Parsing

Algorithm 3 describes the DFA-free validity checker used by EPIC. After lexing memoization constructs the graph $\mathcal{A}_x$, the goal is to determine whether the graph contains at least one path whose terminal sequence belongs to $L(G)$. Equivalently, the procedure checks whether the partial output still has a completion satisfying the target CFG.

The baseline performs this check through an automata-theoretic pipeline. It first converts the graph representation of the partial output into a DFA, often applies minimization, and then checks the non-emptiness of the intersection between the DFA language and the CFG language. This approach is correct, but it introduces substantial repeated overhead. In particular, determinization and minimization are performed many times during decoding.

EPIC avoids repeated DFA construction by running the witness search directly on $\mathcal{A}_x$. The validator generalizes Earley-style parsing from a single input string to an ε-NFA whose paths represent possible lexeme sequences. Each parser item records both a CFG progress state and graph-node information, so that it represents a partial derivation over a path segment of $\mathcal{A}_x$. Prediction introduces productions that can satisfy the next expected nonterminal, ε-transitions move an item between graph nodes without advancing its grammar position, scanning advances items along compatible terminal-labeled edges, and completion connects a recognized constituent to items that were waiting for it at the corresponding graph node. The algorithm accepts once it finds a completed start-symbol item that connects a start node of $\mathcal{A}_x$ to an accepting node.

This DFA-free formulation preserves the exact witness-search problem. The algorithm does not ask whether one fixed string is generated by G. Instead, it asks whether any path in the graph is generated by G. This is precisely the same condition captured by the CFG and DFA intersection

test. This change should be understood as a practical optimization rather than an improvement in the worst-case asymptotic complexity of validity checking. It reduces overhead by searching for the witness directly on $\mathcal{A}_x$ and avoiding construction of an equivalent minimal DFA at every validation.

Algorithm 4 Relaxed Compatible Subset Selection for Parallel Commit

Require: Partial output x, candidates C, regular cover $\mathcal{F}_G$, CFG G, lexical map Λ
Ensure: Exactly verified candidate subset B with $|B| \geq 2$, or $\emptyset$

function RELAXEDSELECT(x, C)
  if $|C| \leq 1$ **then**
    return $\emptyset$
  end if
  $x_C \leftarrow x \oplus C$
  $\mathcal{A}_{x_C} \leftarrow$ MEMOIZEDLEXING($x_C, \Lambda, \mathcal{C}_{\text{lex}}$)
  if $L(\mathcal{F}_G) \cap L(\mathcal{A}_{x_C}) \neq \emptyset$ **then**
    return C ▷ Keep the full subset as relaxed-compatible
  end if
  Split C into high-confidence halves C_L and C_R.
  $B_L \leftarrow$ RELAXEDSELECT(x, C_L)
  $B_R \leftarrow$ RELAXEDSELECT($x \oplus B_L, C_R$)
  return $B_L \cup B_R$
end function
function EXACTSHRINK(x, B)
  if $|B| \leq 1$ **then**
    return $\emptyset$
  end if
  if DFAFREEVALIDATE($x \oplus B, G, \Lambda$) is true **then**
    return B
  end if
  Split B into high-confidence halves B_L and B_R.
  $B'_L \leftarrow$ EXACTSHRINK(x, B_L)
  $B'_R \leftarrow$ RELAXEDSELECT($x \oplus B'_L, B_R$)
  $B''_R \leftarrow$ EXACTSHRINK($x \oplus B'_L, B'_R$)
  return $B'_L \cup B''_R$
end function
Sort C by descending confidence.
$B \leftarrow$ RELAXEDSELECT(x, C)
return EXACTSHRINK(x, B)

C.4 RELAXED COMPATIBLE SUBSET SELECTION FOR PARALLEL COMMIT

Algorithm 4 describes the candidate subset selection procedure used to recover parallel token commitment. For a candidate set C, we write $x \oplus C$ for the partial output obtained by applying all candidate updates in C to x. Checking and committing every proposed token sequentially weakens the parallelism of DLM decoding. Although a joint CFG check over all candidates is possible, a failed check does not identify which candidates caused the failure. Finding the largest compatible subset by exhaustive search would require a number of checks exponential in the number of proposed tokens.

EPIC avoids this cost by selecting a subset under a cheaper relaxed condition before exact verification. The relaxed condition is defined by a regular cover $\mathcal{F}_G$ of the CFG language. The automaton $\mathcal{F}_G$ can be viewed as a finite-state relaxation of a recursive transition network representation of G. It ignores the return sites of recursive calls, which may introduce additional paths but preserves every CFG derivation. Therefore, $L(G) \subseteq L(\mathcal{F}_G)$. This inclusion makes rejection under the regular cover sound. If a candidate batch is rejected by $\mathcal{F}_G$, then applying the whole batch cannot yield a CFG-completable partial output. If a candidate batch is accepted by $\mathcal{F}_G$, it may still be invalid under the original CFG, so exact verification is required.

The recursive structure of Algorithm 4 uses the regular cover as a cheap group test. The algorithm first tests the whole candidate set under the relaxed condition. When the whole set fails the cover check, it splits the candidates by confidence and searches for smaller high-confidence subsets that pass the relaxed check. This allows the decoder to find a jointly plausible subset without enumerating all subsets. The procedure is biased toward high-confidence candidates because those are more likely to be selected by the unconstrained diffusion decoder as well. The final exact check is essential because the regular cover may introduce false positives. Before a nonempty subset is committed, the algorithm verifies the partial output obtained after applying the subset using the exact CFG validity checker. If a relaxed-compatible subset fails exact checking, the algorithm recursively shrinks it using the exact CFG validator. Thus, before any nonempty subset is committed, the partial output obtained after applying the subset has been verified by the exact CFG validity checker. The selector returns only nontrivial batches. Singleton candidates are left to the standard sequential rejection sampler, which verifies them with the exact CFG checker. After committing the verified subset, the decoder uses the standard sequential rejection loop to make the remaining commits. This fallback preserves correctness while allowing EPIC to recover parallel commitment whenever a sufficiently large compatible subset is found.

D Correctness Analysis of Proposed Methods

This section analyzes why EPIC preserves the same completable-output criterion as the baseline decoder. The analysis consists of three parts. First, lexing memoization reuses local lexing results without changing the partial-output language. Second, DFA-free validation checks the same non-emptiness condition as the CFL and the regular language intersection test, but searches directly over the partial-output graph. Third, the finite automaton used in relaxed subset selection is a regular cover of the CFG language.

D.1 Lexing Memoization

Lexing memoization preserves the partial-output language. For any partial output x, Algorithm 2 constructs an NFA $\mathcal{A}_x$ that recognizes the same set of lexeme sequences as the non-memoized lexing procedure.

The reason is that memoization changes only how local lexing alternatives are obtained. For each local context, the algorithm either retrieves a previously computed result from the cache or computes the result using the same lexical map Λ. Since cache entries are keyed by the corresponding local prefix and suffix configuration, a cache hit refers to the same lexing subproblem. The final graph is assembled from the same local alternatives and the same ϵ-connections across masked regions. Thus, memoization reduces construction cost but does not approximate or change the represented lexeme sequence language.

D.2 DFA-Free Validation

DFA-free validation checks exact CFG completability over the partial-output graph. For a partial-output graph $\mathcal{A}_x$ and CFG G, Algorithm 3 searches for a path in $\mathcal{A}_x$ whose label sequence belongs to $L(G)$. Thus, it targets the same condition

$$L(G) \cap L(\mathcal{A}_x) \neq \emptyset.$$

The validator can be viewed as an Earley-style recognizer generalized from string positions to graph nodes. Prediction introduces productions that may satisfy the next expected nonterminal, ε-transitions move items between graph nodes without consuming a grammar terminal, scanning advances items along matching terminal-labeled edges, and completion connects recognized constituents to items waiting for them. An accepting item therefore corresponds to a CFG derivation along some path of $\mathcal{A}_x$. Conversely, any such derivation can be explored by the same prediction, scanning, and completion operations. The algorithm therefore changes the search strategy, not the validity criterion.

D.3 Regular Cover

The flattened automaton is a regular cover of the CFG language. Let $\mathcal{F}_G$ be the finite automaton obtained by flattening the recursive transition network induced by a CFG G. Then the automaton over-approximates the CFG language:

$$L(G) \subseteq L(\mathcal{F}_G).$$

This construction follows the standard RTN-based regular superset approximation of CFGs (Nederhof, 2000). A CFG can be viewed as a recursive transition network by introducing entry and exit states for each nonterminal and rule-position states for each production. Flattening joins these components into a single finite automaton and replaces the recursive call-return mechanism with ordinary transitions. This removes return-site matching and can therefore introduce additional paths, but it does not remove any path corresponding to a valid CFG derivation.

The inclusion follows by a structural induction over CFG derivation trees. For each subtree rooted at a nonterminal A, the corresponding RTN contains a path from the entry state of A to the exit state of A whose labels are exactly the yield of that subtree. For a production $A \to X_1 \cdots X_m$, terminal symbols are matched by the corresponding labeled rule-position transitions, and nonterminal children are matched by the induction hypothesis together with the flattened call and return transitions. Since flattening keeps all such transitions and only forgets which call site a return must match, the yield of the whole derivation tree labels an accepting path in $\mathcal{F}_G$. Thus every string generated by G is accepted by $\mathcal{F}_G$.

E Experimental Details

We follow the benchmark suite, constraints, and prompting protocol of prior CFG-constrained decoding work for DLMs. All main experiments use a maximum generation length of 256 tokens and temperature 0.0 with greedy decoding. For the main comparison, we evaluate denoising steps in $\{16, 32, 64, 128, 256\}$ and repeat runtime measurements with seeds $\{42, 43, 44\}$, although greedy decoding makes the generated outputs seed independent. For each seed, we normalize the runtime of each constrained method by the unconstrained runtime under the same model, task, and decoding-step setting, and then average the relative runtimes across seeds. For the ablation study, we fix the denoising step count to 32 and evaluate all subsets of the three acceleration components of EPIC.

E.1 Implementation and Decoding Settings

We implement EPIC on top of the public implementation of prior CFG-constrained decoding for diffusion language models (Mündler et al., 2026). The evaluation pipeline, dataset wrappers, model wrappers, and profiling scripts are written in Python, while the formal-language backend uses Rust components for finite automata, CFG operations, regular-expression compilation, automaton construction, and intersection-related routines.

We compare three decoding modes. Unconstrained decoding disables grammar constraints. The baseline enables CFG constraints but disables the three acceleration components. EPIC enables lexing memoization, DFA-free validation, and relaxed compatible subset selection. For the main comparison, we evaluate all three modes across the three tasks, four models, five denoising-step settings, and three runtime seeds. Smaller step counts reveal more tokens per denoising step and therefore place more pressure on the sequential checking bottleneck. The ablation study is performed at 32 denoising steps and varies only which of the three acceleration components are enabled. For each run, we record generated outputs, total wall-clock time, and component-level profiling statistics used in the runtime analysis.

Wall-clock time is measured around each generation call, excluding model and tokenizer loading, common grammar and lexer preprocessing, and downstream correctness evaluation. Regular-cover construction is performed lazily within the decoding path and is therefore included in EPIC's runtime. For C++ and SMILES, which use fixed task-level grammars, the constructed cover is reused across instances. For JSON, each instance uses a schema-specific grammar, and the corresponding cover is constructed separately for each instance.

E.2 Datasets, Prompts, and Metrics

We evaluate three structured-generation tasks: C++ code generation, JSON generation under instance-specific schemas, and SMILES generation. Each prompt follows the prior constrained-diffusion implementation and contains a system instruction, a user input, and an assistant-side target prefix. The assistant-side prefix places the model inside the target format before generation begins, so that the generated continuation aligns with the grammar used by the constrained decoder.

For C++, we use the C++ split of HumanEval-X (Zheng et al., 2023). Each instance provides a programming problem, a target function declaration, and test cases. We use the C++ CFG and lexical specification from the prior implementation. Syntactic correctness is measured by CFG-based syntax checking, and functional correctness is measured by compiling the generated program and running the provided tests.

For JSON, we use the JSON benchmark from the prior CFG-constrained diffusion evaluation suite (Mündler et al., 2026), which extends JSON-Mode-Eval (NousResearch, 2024). Each instance contains a natural-language input and an instance-specific JSON schema. The schema is converted into an instance-specific CFG and lexical map using the prior implementation. Syntactic correctness is measured by schema-CFG acceptance, and functional correctness is measured by comparing the normalized generated JSON object with the reference output.

For SMILES (Weininger, 1988), we use the benchmark introduced by prior CFG-constrained diffusion work (Mündler et al., 2026). Each instance asks the model to generate a SMILES string from a molecule description. We use the SMILES CFG and lexical specification from the prior implementation. Unlike C++ and JSON, this setting does not allow arbitrary whitespace between lexed tokens. Syntactic correctness is measured by validity under the SMILES grammar and lexer, and functional correctness is measured by molecular equivalence to the reference SMILES string after canonicalization.

E.3 Models

We evaluate four instruction-tuned diffusion language models following prior CFG-constrained decoding work: Dream (Ye et al., 2025), DreamCoder (Xie et al., 2025), LLaDA (Nie et al., 2025), and DiffuCoder (Gong et al., 2026). Dream and LLaDA are general-purpose DLMs, while DreamCoder and DiffuCoder are code-oriented DLMs. The evaluated checkpoints are Dream[1], DreamCoder[2], LLaDA[3], and DiffuCoder[4]. All models are evaluated with their corresponding tokenizer and wrapper. Dream, DreamCoder, and DiffuCoder use the Dream-style generation interface. LLaDA uses a separate wrapper because its chat template and generation interface differ. These wrapper-level differences affect prompt formatting but not the task instructions, constraints, or evaluation protocol.

E.4 Experimental Environments

All experiments were conducted on a workstation with an NVIDIA RTX A6000 GPU and an AMD Ryzen Threadripper 3960X CPU. The software environment consisted of Rocky Linux 8.10, Python 3.11.15, Rust 1.93.1, and PyTorch 2.12.0 built against CUDA 13.0. For model and dataset handling, we used `transformers` 4.52.2, `datasets` 3.6.0, and `accelerate` 1.13.0. Rust extension modules were built with `maturin` 1.13.3. For task-specific evaluation, we used `partialsmiles` 2.0 and `rdkit` 2026.3.2.

F Additional Experimental Results

This section provides additional results for inference time, correctness, and ablation analyses. We first report the full wall-clock inference times across all models, datasets, and denoising-step settings, complementing the relative inference-time results in Table 1. We then report syntactic and

[1] `Dream-org/Dream-v0-Instruct-7B`.
[2] `Dream-org/Dream-Coder-v0-Instruct-7B`.
[3] `GSAI-ML/LLaDA-8B-Instruct`.
[4] `apple/DiffuCoder-7B-Instruct`.

Table 6: Full inference time results. We report wall-clock inference time in seconds for unconstrained decoding (Uncon.), the prior CFG-constrained baseline (Con.), and EPIC. These timings correspond to the relative inference times reported in Table 1. The results cover all models, tasks, and denoising-step settings used in the main comparison. The shaded column highlights EPIC.

Model	Steps	C++			JSON			SMILES		
		Uncon.	Con.	EPIC	Uncon.	Con.	EPIC	Uncon.	Con.	EPIC
Dream 7B	16	571	656	450	549	969	601	232	237	240
	32	609	900	700	972	1203	983	514	516	519
	64	1089	1245	1177	1822	1971	1847	1015	1022	1028
	128	2044	2167	2098	3476	3630	3508	1998	2004	2003
	256	3958	4042	4071	6812	6975	6867	4043	4001	4011
DreamCoder 7B	16	594	1876	609	517	921	461	240	242	247
	32	586	1582	824	821	1089	735	519	523	526
	64	1068	1949	1368	1530	1806	1480	1012	1016	1019
	128	2024	2703	2236	2833	3023	2782	2000	2005	2005
	256	3806	4462	4054	5479	5548	5415	3983	3984	3984
LLaDA 8B	16	520	578	386	923	1137	690	361	405	343
	32	828	791	531	1388	1554	1076	619	636	570
	64	1336	1246	1026	2332	2441	2042	1156	1141	1104
	128	2382	2394	2221	4233	4335	4030	2238	2197	2149
	256	4359	4459	4389	8023	8030	7882	4384	4337	4320
DiffuCoder 7B	16	628	1114	657	605	1002	657	349	362	366
	32	701	1278	862	1044	1314	1005	562	584	576
	64	1137	1579	1163	1902	2116	1854	1010	1043	1020
	128	1897	2298	1973	3601	3687	3474	1952	2109	2042
	256	3108	3684	3351	7046	6706	6551	3736	3854	3814

functional correctness, showing that EPIC maintains correctness comparable to the prior CFG-constrained baseline while substantially reducing inference time. Finally, we provide the full ablation results over all subsets of lexing memoization, DFA-free validation, and relaxed compatible subset selection, complementing the main ablation table in Section 5.4.

F.1 Full results on Inference Time

Table 6 provides the raw wall-clock inference times corresponding to the relative inference times reported in Table 1. The table includes unconstrained decoding, the prior CFG-constrained baseline, and EPIC for every model, task, and denoising-step setting. These raw values are reported to make the normalization transparent and to show the absolute runtime scale across tasks. As in the main results, all times are measured in seconds and averaged over three runtime seeds.

The raw times also show how the absolute decoding cost scales with the number of denoising steps. Although the increase is not exactly linear, runtime generally grows with the step count for all methods because each additional step requires another model forward pass. This trend is visible even for unconstrained decoding, indicating that the overall runtime is still dominated by repeated model evaluation at larger step counts. In this regime, EPIC stays close to unconstrained decoding, whereas the baseline constrained decoder often remains noticeably slower due to additional per-step validity-checking overhead. Thus, the absolute timings complement the relative inference-time results in the main text. In many settings, EPIC brings CFG-constrained decoding much closer to the runtime scale of unconstrained decoding.

F.2 Full Results on Correctness

Tables 7 and 8 report the full syntactic and functional correctness results. We use U. for unconstrained decoding, C. for the prior CFG-constrained baseline, and E. for EPIC. The superscript $-$ denotes the result before applying the final witness-based recovery procedure, while the version without $-$ includes this recovery.

Table 7: Syntactic correctness across all models, denoising steps, and tasks. U. denotes unconstrained decoding, C. denotes the prior CFG-constrained baseline, and E. denotes EPIC. The superscript − reports correctness before witness-based recovery, while the version without − includes recovery. All values are percentages.

Model	Steps	C++					JSON					SMILES				
		U.	C.$^-$	C.	E.$^-$	E.	U.	C.$^-$	C.	E.$^-$	E.	U.	C.$^-$	C.	E.$^-$	E.
Dream	16	35.4	54.3	**100**	53.0	**100**	32.4	50.4	**100**	50.0	**100**	61.7	95.2	**100**	95.2	**100**
	32	67.7	80.5	**100**	80.5	**100**	77.6	87.1	**100**	87.1	**100**	70.7	98.2	**100**	98.2	**100**
	64	80.5	90.2	**99.6**	90.9	**99.6**	92.3	95.6	**100**	96.0	**100**	72.5	99.4	**100**	98.8	**100**
	128	89.0	95.1	**100**	95.1	99.8	90.4	96.3	**100**	96.3	**100**	73.1	**100**	**100**	**100**	**100**
	256	82.9	94.5	**100**	94.5	**100**	91.2	96.1	**100**	96.3	**100**	72.5	**100**	**100**	**100**	**100**
DreamCoder	16	17.7	33.5	**99.8**	39.0	**99.8**	79.0	85.3	**100**	85.7	**100**	78.4	97.0	**100**	97.0	**100**
	32	54.3	66.5	**99.4**	65.9	**99.4**	87.5	91.9	**100**	92.3	**100**	79.0	97.0	**100**	97.6	**100**
	64	72.6	77.4	99.2	78.7	**100**	90.8	91.9	**100**	92.3	**100**	74.9	98.8	**100**	98.8	**100**
	128	73.8	80.5	**100**	81.1	**100**	93.0	94.1	**100**	94.1	**100**	74.3	98.8	**100**	98.8	**100**
	256	84.1	85.0	**100**	86.0	**100**	94.1	94.9	**100**	94.9	**100**	74.3	98.8	**100**	98.8	**100**
LLaDA	16	5.5	34.1	99.6	32.9	**100**	57.4	78.3	**100**	78.3	**100**	41.3	90.4	**100**	89.2	**100**
	32	9.1	31.7	99.2	31.1	**99.4**	75.4	89.0	**100**	88.6	**100**	60.5	90.4	**100**	89.8	**100**
	64	42.7	67.1	99.8	67.1	**100**	79.8	92.3	**100**	92.6	**100**	65.9	94.0	**100**	93.4	**100**
	128	82.9	92.7	**100**	92.7	**100**	87.9	96.3	**100**	96.3	**100**	65.9	91.6	**100**	91.6	**100**
	256	96.3	99.4	**100**	99.4	**100**	91.2	96.3	**100**	96.3	**100**	65.3	95.8	**100**	95.6	**100**
DiffuCoder	16	25.0	40.9	**99.8**	39.0	**99.8**	76.8	83.5	**100**	83.5	**100**	70.7	94.0	**100**	93.4	**100**
	32	36.0	50.0	**99.8**	51.8	**99.8**	87.9	90.8	**100**	90.4	**100**	71.9	98.2	**100**	98.2	**100**
	64	53.7	67.1	99.8	67.7	**100**	84.6	86.8	**100**	86.8	**100**	62.3	98.8	**100**	98.8	**100**
	128	60.4	72.6	**100**	74.4	99.8	77.9	79.0	**100**	79.0	**100**	49.7	99.2	**100**	99.4	**100**
	256	93.9	91.3	**100**	93.7	**100**	77.9	79.4	**100**	79.4	**100**	47.9	97.0	**100**	97.0	**100**

F.2.1 Syntactic Correctness

Table 7 shows that both constrained methods substantially improve syntactic correctness over unconstrained decoding. Even before recovery, both C.$^-$ and E.$^-$ consistently improve over unconstrained decoding, showing that rejection-based CFG checking already prevents syntactic errors. EPIC generally achieves pre-recovery syntactic correctness comparable to the prior CFG-constrained baseline. The numbers are not always identical since EPIC may commit several tokens in parallel, and therefore can fill positions in a different order from the sequential baseline. This changes the subsequent partial outputs observed by the model, which can lead to differences in either direction. In some settings, EPIC is slightly higher than the baseline before recovery, while in others it is slightly lower. These differences are expected because relaxed subset selection changes the commit order for efficiency, but every committed batch is still verified by the exact CFG checker.

After applying the recovery procedure, both constrained methods achieve near-perfect syntactic correctness in almost all settings. This confirms that the efficiency improvements of EPIC do not weaken the final CFG-based acceptance criterion. Rather, EPIC mainly changes how compatible candidates are selected and verified.

F.2.2 Functional Correctness

Table 8 reports task-level functional correctness. Constrained decoding often improves functional correctness over unconstrained decoding, especially on C++ and JSON, since syntactically invalid outputs cannot satisfy the downstream evaluator. As the number of denoising steps increases, functional correctness also tends to improve. This is consistent with the usual diffusion decoding trade-off. With more steps, the decoder unmasks fewer tokens at each step, making each update more conditioned on previously committed context and generally improving generation quality.

However, the improvement in syntactic correctness does not always translate into a proportional improvement in functional correctness. This behavior is also observed in prior work and reflects a common limitation of grammar-only constrained decoding. A grammar can enforce formal syntac-

Table 8: Functional correctness across all models, denoising steps, and tasks. For C++, functional correctness is measured by passing the test cases. For JSON and SMILES, it is measured by task-specific normalized equivalence to the reference output. U. denotes unconstrained decoding, C. denotes the prior CFG-constrained baseline, and E. denotes EPIC. The superscript − reports results before witness-based recovery, while the version without − includes recovery. All values are percentages.

Model	Steps	C++					JSON					SMILES				
		U.	C.$^-$	C.	E.$^-$	E.	U.	C.$^-$	C.	E.$^-$	E.	U.	C.$^-$	C.	E.$^-$	E.
Dream	16	10.4	**11.6**	**11.6**	**11.6**	**11.6**	13.2	18.0	**23.7**	18.0	**23.7**	**0.6**	**0.6**	**0.6**	**0.6**	**0.6**
	32	15.2	**16.5**	**16.5**	**16.5**	**16.5**	43.4	45.2	**46.0**	45.2	**46.0**	**3.0**	**3.0**	**3.0**	**3.0**	**3.0**
	64	18.1	18.5	18.5	**18.9**	**18.9**	53.7	**54.8**	**54.8**	**54.8**	**54.8**	3.0	**3.6**	**3.6**	**3.6**	**3.6**
	128	13.8	**14.6**	**14.6**	14.4	14.4	51.5	**53.3**	**53.3**	**53.3**	**53.3**	3.6	**4.2**	**4.2**	**4.2**	**4.2**
	256	12.8	13.4	13.4	**14.0**	**14.0**	51.8	**53.7**	**53.7**	**53.7**	**53.7**	3.0	**4.2**	**4.2**	**4.2**	**4.2**
DreamCoder	16	6.7	7.9	8.5	**9.1**	**9.1**	41.9	43.8	**44.1**	43.4	44.0	**4.2**	**4.2**	**4.2**	**4.2**	**4.2**
	32	19.5	20.1	20.7	20.7	**21.3**	52.2	52.6	**53.3**	52.6	**53.3**	3.6	**4.2**	**4.2**	**4.2**	**4.2**
	64	26.8	26.8	**27.2**	26.2	26.6	**53.7**	53.3	53.3	53.3	53.3	**2.4**	**2.4**	**2.4**	**2.4**	**2.4**
	128	28.0	**28.7**	**28.7**	**28.7**	**28.7**	55.9	**56.6**	**56.6**	**56.6**	**56.6**	**2.4**	**2.4**	**2.4**	**2.4**	**2.4**
	256	**28.0**	27.4	27.4	27.4	27.4	56.3	**56.6**	**56.6**	**56.6**	**56.6**	**2.4**	**2.4**	**2.4**	**2.4**	**2.4**
LLaDA	16	0.6	2.8	4.1	3.3	**4.5**	39.0	43.8	**44.1**	43.8	**44.1**	1.2	**1.8**	**1.8**	**1.8**	**1.8**
	32	3.0	5.1	5.3	5.5	**6.3**	44.1	49.6	**50.0**	49.6	**50.0**	0.6	**1.2**	**1.2**	**1.2**	**1.2**
	64	10.4	**11.0**	**11.0**	**11.0**	**11.0**	44.5	49.6	**50.0**	49.6	**50.0**	1.2	**1.8**	**1.8**	**1.8**	**1.8**
	128	**15.9**	**15.9**	**15.9**	**15.9**	**15.9**	48.2	**52.9**	**52.9**	**52.9**	**52.9**	**1.8**	**1.8**	**1.8**	**1.8**	**1.8**
	256	**25.6**	25.0	25.0	25.0	25.0	49.6	52.2	**52.9**	52.2	**52.9**	**2.4**	**2.4**	**2.4**	**2.4**	**2.4**
DiffuCoder	16	12.8	14.0	14.0	**14.6**	**14.6**	42.3	43.8	**44.1**	43.8	**44.1**	0.0	**0.6**	**0.6**	**0.6**	**0.6**
	32	15.2	**18.3**	**18.3**	17.7	17.7	47.8	48.9	**49.6**	48.5	49.3	0.0	0.0	0.0	0.0	0.0
	64	17.7	**18.3**	**18.3**	**18.3**	**18.3**	45.2	47.1	**47.4**	47.1	**47.4**	**0.6**	**0.6**	**0.6**	**0.6**	**0.6**
	128	25.6	27.4	**29.9**	27.4	29.7	42.3	43.4	**43.8**	43.4	**43.8**	**1.8**	**1.8**	**1.8**	**1.8**	**1.8**
	256	43.9	44.3	45.5	44.5	**45.7**	41.9	**43.4**	**43.4**	**43.4**	**43.4**	**2.4**	**2.4**	**2.4**	**2.4**	**2.4**

tic validity, but it does not verify task semantics. The witness-based recovery procedure has only a marginal effect on functional correctness in most settings. Although recovery raises syntactic correctness to nearly 100%, the recovered completion is selected to satisfy the grammar rather than to optimize the task-level objective. Therefore, recovery is useful as a syntactic fallback, but it should not be expected to substantially improve semantic or functional quality.

The gap between syntactic and functional correctness is particularly visible on SMILES. Although the grammar can enforce the surface syntax of SMILES strings, it does not fully characterize task-level molecular correctness or equivalence to the target molecule. In addition, the underlying CFG is only an approximation to the full set of functionally valid molecular strings. As a result, syntactic correctness can approach 100% while functional correctness remains low.

F.3 Full Ablation Results

Table 9 reports the full ablation results at 32 denoising steps. The table includes all individual components, all pairwise combinations, and the full EPIC configuration. Overall, the results show that the three components provide complementary sources of speedup. Lexing memoization mainly reduces repeated lexical analysis, DFA-free validation removes the exact-checking dependence on repeated DFA construction, and relaxed compatible subset selection reduces the number of sequential exact checks by recovering parallel commitment.

The effect of each component depends on the task and model. On C++, all three components are consistently useful because the grammar and lexical specification are more complex, making repeated lexing, automaton construction, and CFG validation expensive. DFA-free validation is particularly effective for DreamCoder and DiffuCoder on C++, while relaxed subset selection gives large gains for LLaDA, whose block-wise decoding schedule provides stable candidate batches. On JSON, lexing memoization and relaxed subset selection are generally more effective than DFA-free validation alone. This is consistent with the fact that JSON validation is relatively cheap once the

Table 9: Full ablation results for EPIC. Starting from the prior CFG-constrained decoder (CON.), we separately add ① lexing memoization, ② DFA-free validation with Earley-style parsing, and ③ relaxed compatible subset selection for parallel commit. EPIC enables all three components.

Model	Method	C++	JSON	SMILES
Dream	Con.	900	1203	**516**
	①	843	1095	541
	②	863	1237	540
	③	900	1162	545
	①+②	758	1071	540
	①+③	795	1043	545
	②+③	844	1160	545
	①+②+③	**700**	**983**	519
DreamCoder	Con.	1582	1089	**523**
	①	1385	939	540
	②	1261	1109	540
	③	1378	986	543
	①+②	1006	907	540
	①+③	1056	774	543
	②+③	1154	974	543
	①+②+③	**824**	**735**	**523**
LLaDA	Con.	791	1554	636
	①	772	1450	655
	②	738	1619	628
	③	601	1179	609
	①+②	704	1442	627
	①+③	577	1129	608
	②+③	571	1176	595
	①+②+③	**531**	**1076**	**570**
DiffuCoder	Con.	1278	1314	584
	①	1152	1172	598
	②	1085	1307	582
	③	1119	1195	601
	①+②	918	1129	581
	①+③	947	1054	599
	②+③	996	1174	597
	①+②+③	**862**	**1005**	**576**

partial-output representation has been constructed, so avoiding DFA construction alone does not always compensate for the overhead of graph parsing. On SMILES, the differences are small because the baseline constraint overhead is already low. In some cases, the added component can slightly increase runtime.

Pairwise combinations usually improve over the corresponding single-component variants, indicating that the optimizations address different parts of the pipeline. For example, combining memoization with DFA-free validation substantially reduces C++ runtime, while combining memoization with relaxed subset selection is especially helpful on JSON. The full configuration, which enables all three components, gives the best runtime in most settings and is never substantially worse than the best ablation. These results support the design of EPIC as a combination of complementary optimizations rather than a speedup from a single dominant component.

F.4 DFA-FREE VALIDATOR ANALYSIS

We further analyze the cost of the DFA-free exact validator independently of model inference. For a lexeme-path graph with N nodes and

$$R = \sum_{A \to \alpha \in P} (|\alpha| + 1)$$

Table 10: Per-check cost of DFA-free validation across JSON schema grammars with different numbers of production rules. Total time includes lexeme-path graph construction and exact graph parsing.

CFG size	# Checks	Total (ms)	Parsing (ms)	Graph (ms)	# States
< 30 rules	10,129	0.728	0.039	0.686	19.5
30–59 rules	11,397	0.858	0.053	0.802	26.8
> 59 rules	5,079	1.442	0.124	1.316	224.3

Table 11: Per-check cost of DFA-free validation at different mask densities, defined as the fraction of masked positions in the current partial output. Total time includes lexeme-path graph construction and exact graph parsing.

Mask density	# Checks	Total (ms)	Parsing (ms)	Graph (ms)	# States
$[0, .25)$	16,233	0.891	0.058	0.831	56.3
$[.25, .50)$	7,079	0.980	0.068	0.909	84.3
$[.50, .75)$	2,240	1.011	0.067	0.941	51.8
$[.75, 1]$	1,053	0.755	0.056	0.696	14.3

dotted grammar positions, the number of distinct parser items is bounded by $O(RN^2)$. Each item is identified by its grammar position, origin graph node, and current graph node, and duplicate items are removed before being added to the worklist. This bound characterizes the parser state space rather than the overall worst-case parsing complexity, which remains comparable to general CFG recognition.

Table 10 reports per-check runtime as the grammar size increases. The total cost includes construction of the lexeme-path graph and exact graph-based CFG validation. Validation becomes more expensive for larger grammars and larger graphs, but the exact parser itself remains a small fraction of the total validator-side cost. Even for the largest grammar-size bin, exact graph parsing takes 0.124 ms per check on average, compared with 1.316 ms for graph construction.

Mask density shows a different pattern. As shown in Table 11, validation cost is not monotonic in the fraction of masked positions. Highly masked sequences contain little fixed text, whereas intermediate mask densities can induce more fragmented fixed regions and larger lexeme-path graphs. This suggests that graph structure and lexical ambiguity, rather than mask density alone, determine the practical cost of exact validation.

F.5 Regular Cover Analysis

We next characterize the preprocessing cost and operational tightness of the regular cover used for parallel subset selection. The cover is constructed by flattening the recursive transition network induced by the CFG into an ENFA. This cover is determinized and minimized once for each grammar and then cached. Unlike the baseline validity checker, this conversion is not repeated for individual candidate updates. Table 12 reports the construction cost and automaton size for the instance-specific JSON schema grammars. The size of the intermediate ENFA increases with grammar complexity, while the minimized cover DFA remains small across all schema-size bins. Construction is also inexpensive, taking approximately 0.4–1.3 ms on average.

Table 12: Construction cost and automaton size of regular covers for instance-specific JSON schema grammars. Build time includes cover construction, determinization, and minimization.

CFG size	# Unique Grammars	# Production Rules	# ENFA states	# DFA states	Build Time (ms)
< 30 rules	26	25.8	107.3	8.5	0.432
30–59 rules	143	36.2	147.1	9.7	0.574
> 59 rules	68	79.4	318.3	8.0	1.334

Table 13: Operational tightness of the regular cover on JSON. FP rate denotes the fraction of initially selected cover-compatible subsets rejected by the subsequent exact CFG check. Average parallel commit size is measured after exact verification. Instances for which no nontrivial cover-compatible subset is found are omitted.

Instance Group	# Inst.	FP Rate	Parallel Commit
No FP observed	244	0.0%	2.73
FP observed	22	50.5%	1.53

We further examine how closely the regular cover approximates the CFG during decoding. Since the cover defines a language superset, a cover-compatible subset may still be rejected by the subsequent exact CFG check. We measure this operational tightness using the false-positive rate of initially selected cover-compatible subsets. Table 13 groups JSON instances according to whether any such false positive is observed during decoding. Most instances exhibit no false positives. For instances where false positives occur, the average parallel commit size decreases from 2.73 to 1.53. This indicates that a looser cover primarily reduces the amount of parallelism recovered by subset selection, while correctness is preserved by the subsequent exact CFG verification.

F.6 Output Fidelity to the Constrained Baseline

Although EPIC preserves the same CFG-completability criterion as the baseline, parallel commitment can change the order in which partial outputs are presented to the model and may therefore alter the generated sequence. We further examine this effect by comparing the outputs of EPIC and the prior constrained baseline on C++. Table 14 reports functional accuracy, overlap between the sets of correctly solved instances, and exact output agreement. Let C_{base} and C_{EPIC} denote the sets of instances that are functionally correct under the baseline and EPIC, respectively. We compute their Jaccard similarity as

$$J(C_{\text{base}}, C_{EPIC}) = \frac{|C_{\text{base}} \cap C_{EPIC}|}{|C_{\text{base}} \cup C_{EPIC}|}.$$

The mean functional accuracy remains nearly unchanged, from 18.72% for the baseline to 18.86% for EPIC. The correct-instance sets have a Jaccard similarity of 94.20%, and 97.58% of the instances solved by the baseline are also solved by EPIC. In addition, 73.13% of generated outputs exactly match the baseline text. These results indicate that parallel commitment largely preserves the functional behavior of the original constrained decoder while reducing its decoding cost.

Table 14: Output fidelity of EPIC relative to the prior constrained baseline on C++. Correct-set Jaccard measures the overlap between the sets of functionally correct instances produced by the two methods.

Metric	Value
Baseline Functional Accuracy	18.72%
EPIC Functional Accuracy	18.86%
Correct-set Jaccard	94.20%
Baseline-to-EPIC Correct Overlap	97.58%
EPIC-to-Baseline Correct Overlap	95.86%
Exact Text Match	73.13%

F.7 Comparison with LAVE

We additionally compare EPIC with LAVE (Zhang et al., 2026), a recent CFG-constrained decoding method for DLMs. We report this comparison separately from the main results because LAVE uses a different decoding protocol. In particular, LAVE adopts semi-autoregressive decoding with a fixed block size of 32 for all models, rather than preserving the native decoding configuration of each model. Using smaller blocks limits the amount of masked context considered within each block and thereby also helps reduce the cost of repeated lookahead verification. In contrast, our main

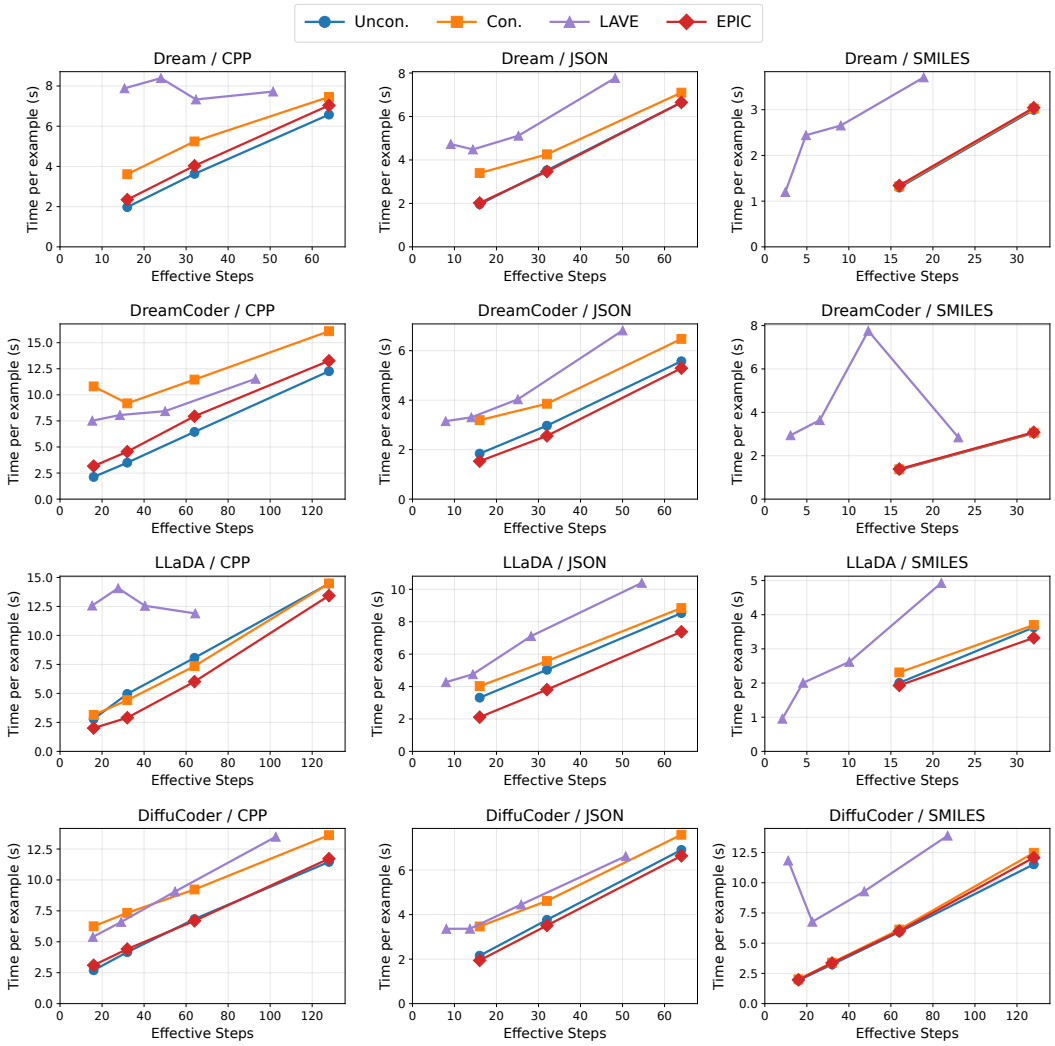


Figure 3: Runtime of unconstrained decoding, Con., LAVE, and EPIC across effective decoding steps.

experiments follow the decoding schedules used by the respective models and the prior completion-based CFG decoder. Consequently, the same nominal number of denoising steps does not necessarily correspond to the same amount of generation progress across the two protocols, making a direct comparison in the main tables potentially misleading.

To provide a comparison under conditions that are as similar as possible, we integrate LAVE into the same evaluation pipeline and match the model checkpoints, tokenizers, datasets, prompts, maximum generation length, seeds, hardware, and evaluators. We retain LAVE's semi-autoregressive decoding procedure with a block size of 32, temperature 0.2, $N = 10$ lookahead samples, and a retry budget of $\tau = 5$. Since LAVE generates the output semi-autoregressively in blocks, its nominal step budget is not directly comparable to the sequence-level step budgets used for Uncon., Con., and EPIC. In particular, LAVE may terminate before visiting all blocks, in which case part of its nominal step budget is never used. We therefore compare the methods using an upper bound on the number of executed denoising steps. For Uncon., Con., and EPIC, this is simply the configured number of steps. For LAVE, we estimate this upper bound as the average number of visited blocks multiplied by the configured number of steps per block.

Figure 3 compares wall-clock runtime at comparable effective decoding progress. Across most model and task combinations, EPIC remains close to unconstrained decoding and is substantially

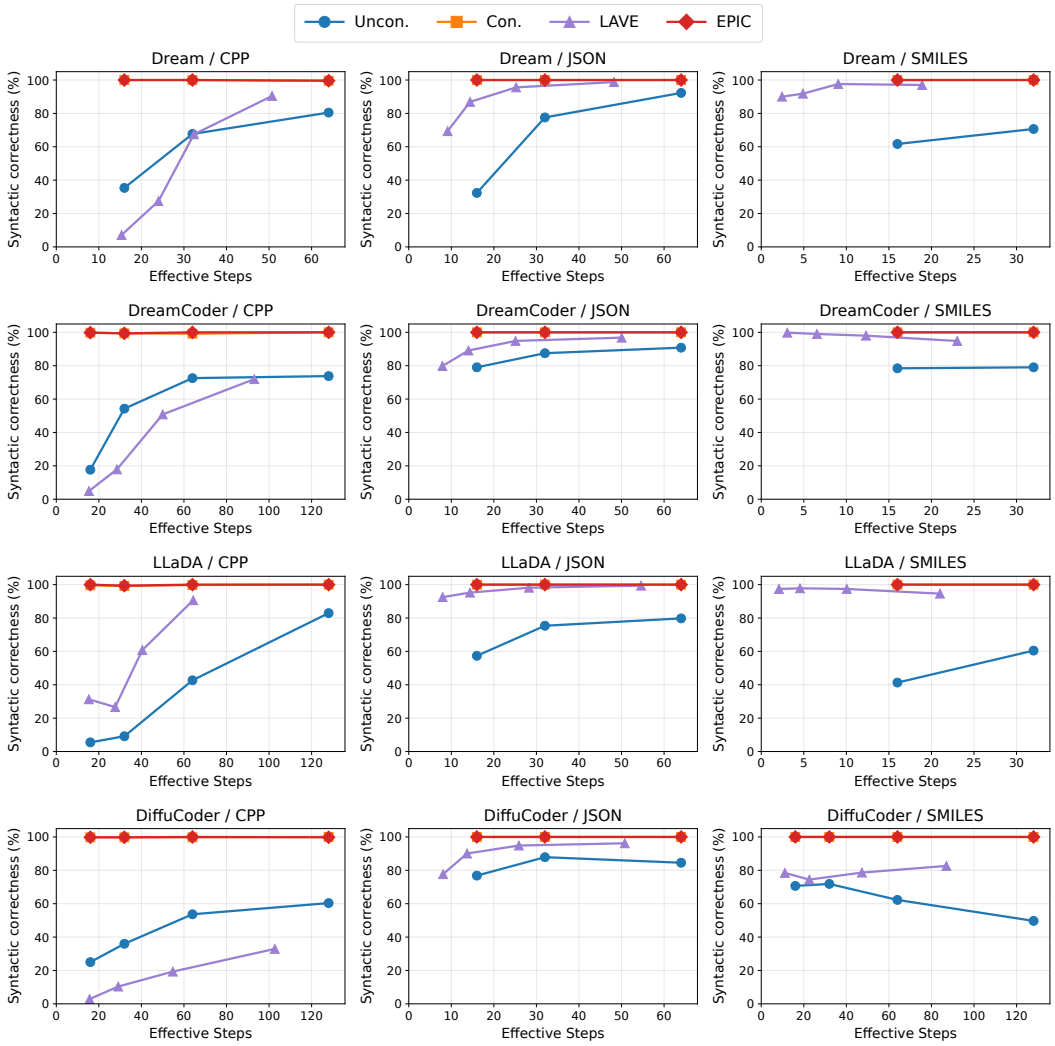


Figure 4: Syntactic correctness of unconstrained decoding, Con., LAVE, and EPIC across effective decoding steps.

faster than LAVE. The difference is particularly visible on C++ and JSON, where repeated lookahead verification introduces a larger runtime cost. On SMILES, where the grammar is substantially smaller, the difference is less consistent and there is less room for either constrained decoder to reduce overhead relative to the underlying generation cost.

Figures 4 and 5 compare correctness under the same view. EPIC maintains near-perfect syntactic correctness over a broad range of effective steps, while LAVE is more sensitive to decoding progress, particularly on C++. For functional correctness, the two methods are comparable in some settings, but LAVE does not show a consistent advantage despite its larger runtime. Overall, these results support the same efficiency-correctness trade-off observed in the main experiments. EPIC substantially reduces constrained decoding cost while maintaining strong syntactic and functional correctness.

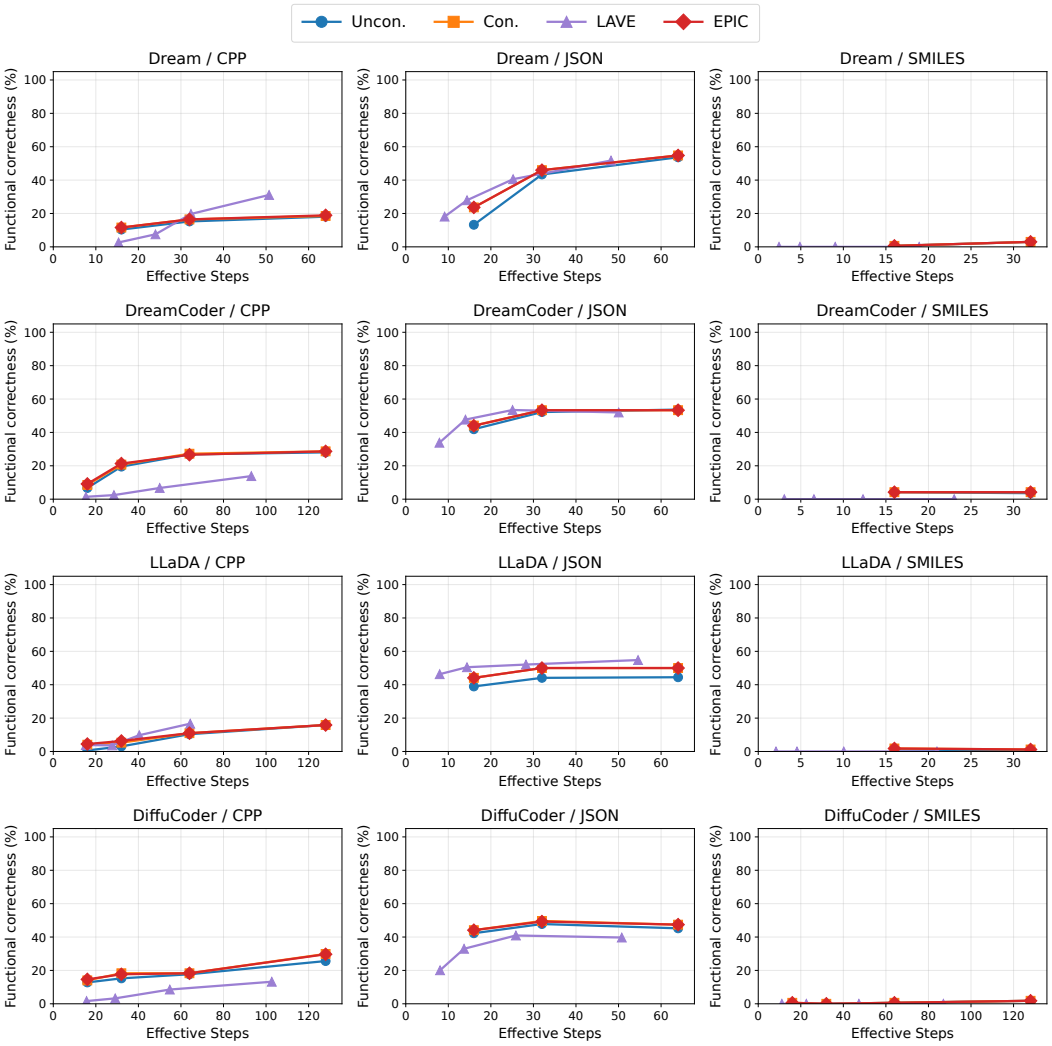


Figure 5: Functional correctness of unconstrained decoding, Con., LAVE, and EPIC across effective decoding steps.