DCONSTRAIN: COMPILING CONTEXT-PARAMETERIZED TRANSITIONS FOR DIFFUSION LANGUAGE MODEL CONSTRAINED DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion language models (DLMs) enable parallel token generation, but structured-output constraints are difficult to enforce because token choices at unresolved positions can depend on one another. Automata-based DLM constrained decoding resolves this interdependence through joint inference, which requires exact token-labeled successors for many hypothetical parser configurations rather than validity information for a single realized prefix. Existing finite-state pipelines bind runtime parsing contexts into explicit states and materialize vocabulary-level transitions, duplicating reusable grammar logic across contexts. We present **DConstrain**, which instead compiles *context-parameterized transition programs*: token consumption is specialized to local grammar states and tokenizer tokens, while the runtime stack and parsing environment remain unbound. Warm compilation executes these reusable programs only for reached configurations, recovering the exact successors required by DLM joint inference. Token-effect grouping further shares successor construction across equivalent tokens. On BFCL with LLaDA2.2, DConstrain compiles 100% of schemas under arbitrary property order versus 71.3% for an optimized finite-state baseline, with a $9.43\times$ paired cold-compilation speedup. It also reduces constrained-decoding overhead by $6.00\times$ while preserving 100% schema validity and matched task accuracy.

1 INTRODUCTION

Diffusion language models (DLMs) offer an alternative to autoregressive (AR) generation by iteratively predicting multiple tokens in parallel (Sahoo et al., 2024; Nie et al., 2025; Ye et al., 2025). Applying DLMs to tool calling, API interaction, and structured data extraction requires reliable adherence to output constraints (Patil et al., 2025; Geng et al., 2025). However, parallel generation fundamentally complicates constrained decoding. In AR decoding, each token can be validated against a single generated prefix (Dong et al., 2025). In DLMs, multiple positions remain unresolved and are predicted jointly, so the validity of one token may depend on choices made at other positions. Consequently, local validity no longer implies joint validity: tokens that each admit a valid completion in isolation may become incompatible when committed together.

Recent automata-based methods address this interdependence through joint inference over a canvas-specific grammar graph, whose paths connect token choices that are compatible across positions (Suresh et al., 2025; Dang & Ermon, 2026). This allows grammar constraints to guide parallel decoding before tokens are committed. However, constructing the canvas-specific grammar graph can be prohibitively expensive for complex structured constraints, limiting the practical benefits of joint inference.

The difficulty lies in the *successor-construction interface* required by joint inference. AR constrained decoding follows the generated prefix, constructs a validity mask, and advances the parser after selecting a token (Dong et al., 2025). DLM joint inference instead requires exact token-labeled successors for multiple hypothetical parser configurations before committing token choices. Each configuration carries a runtime context comprising a parser stack and a parsing environment. Existing automata-based methods expose these successors through explicit finite-state representations (Suresh et al., 2025; Dang & Ermon, 2026); DINGO additionally precomputes token transi-

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.

tions across the vocabulary (Suresh et al., 2025). Under nested constraints and arbitrary property order, representing runtime contexts as explicit states can duplicate shared grammar logic, while vocabulary-level precomputation further multiplies this work.

This exposes a *missing compiler abstraction* for DLM constrained decoding: a reusable representation that provides exact context-specific successors while preserving sharing across runtime contexts. The central design question is therefore not merely whether to use a DFA or PDA, but *what object to compile*. Our key insight is *a different specialization boundary*: compile local grammar–token behavior while leaving the runtime stack and parsing environment as parameters. For example, returning from a nested rule follows the same stack operation even when different runtime contexts yield different successors.

We introduce **DConstrain**, whose core representation is a *context-parameterized transition program*. Each program specializes token consumption to a local grammar state and tokenizer token while keeping the runtime stack and parsing environment as parameters. Programs can therefore be reused across nested calling contexts and field-history combinations while producing exact context-specific successors. *Pushdown factorization* exposes this separation, and partial evaluation resolves context-independent work while retaining context-dependent operations for runtime execution.

This representation supports *two-stage compilation*: cold compilation constructs reusable programs, and warm compilation executes them only for configurations reached by the current decoding canvas, constructing the exact *canvas-specific grammar graph*. Building on these programs, *token-effect grouping* shares successor construction across tokens with equivalent effects at the same local grammar state and represents their parallel edges as group-level edges. With objective-specific score aggregation and token restoration, we prove that grouping preserves the Viterbi optimum and the full token-labeled path distribution of the corresponding ungrouped graph.

Our contributions are summarized as follows:

- We identify and formalize the successor-construction interface required by automata-based DLM joint inference: exact token-labeled successors for multiple hypothetical parser configurations.
- We introduce context-parameterized transition programs, realized through pushdown factorization and two-stage compilation, to share grammar logic across runtime contexts while instantiating exact successors only for configurations reached by the current decoding canvas.
- Building on this representation, we introduce token-effect grouping to share successor construction and compress the edges processed by joint inference. We prove equivalence to ungrouped inference under objective-specific score aggregation and token restoration, preserving the Viterbi optimum for max-product inference and the full token-labeled path distribution for sum-product inference.
- Across structured-output workloads and multiple DLM tokenizers, DConstrain substantially improves compilation coverage and latency, particularly under arbitrary property order and deep nesting, while reducing constrained-decoding overhead and preserving structured-output validity and task accuracy.

2 Related Work

Autoregressive Constrained Decoding. Grammar-constrained decoding is widely used to guarantee structured outputs in autoregressive LLMs. Prior work ranges from incremental parsing and semantic constraints (Scholak et al., 2021; Poesia et al., 2022) to efficient tokenizer-aware mask generation (Willard & Louf, 2023; Moskal et al., 2025; Dong et al., 2025). In AR generation, token selection uses a validity mask for the realized prefix, after which the parser advances with the selected token. XGrammar 2 (Li et al., 2026) further optimizes mask construction through JIT compilation and cross-grammar cache reuse. DConstrain targets a different *successor-construction interface*: DLM joint inference requires $(c, v) \mapsto T_{\mathcal{G}}(c, v)$, exposing exact successors for multiple hypothetical parser configurations before token commitment. A validity mask identifies admissible tokens but does not expose the successor configurations needed to connect choices across unresolved

positions. The distinction therefore concerns the information supplied to joint inference, rather than the use of a particular parser or JIT strategy.

DLM Constrained Decoding. Verification-based methods validate model proposals against structural constraints (Mündler et al., 2026; Zhang et al., 2026), with LAVE using model-informed lookahead. EPIC (Jin & Han, 2026) combines memoized lexing, Earley-style parsing, and parallel token commitment. These approaches avoid full finite-state transition materialization through proposal validation and incremental parsing. Automata-based methods (Suresh et al., 2025; Dang & Ermon, 2026) instead couple token choices through joint inference over finite automata. DINGO constructs a DFA and precomputes vocabulary-level token transitions, while Dang & Ermon (2026) develop efficient chain and tree inference over a given finite automaton. Finite-state pipelines materialize context-specific token transitions, whereas direct pushdown interpretation executes token-consumption logic for each reached configuration. DConstrain compiles reusable local grammar–token logic while leaving the stack and parsing environment unbound, supplying the exact successors required by downstream joint inference through runtime execution.

3 PRELIMINARIES

We write a parser configuration as $c = (q, s, e)$, where q is the local grammar state, s the parser stack, and e the parsing environment. The token-transition relation $\mathsf{T}_{\mathcal{G}}(c, v)$ returns the set of successor configurations after consuming token v. Appendices A.1 and A.2 provide detailed parser semantics and define the model predictions P_t.

Automata-Based DLM Constrained Decoding. Automata-based methods apply $\mathsf{T}_{\mathcal{G}}$ across a decoding window to coordinate token choices at multiple unresolved positions (Suresh et al., 2025; Dang & Ermon, 2026). Fix a denoising iteration t and a window of H positions. Let $\mathcal{V}_i^t \subseteq \mathcal{V}$ denote the candidate tokens at position i, with singleton sets at fixed positions, and let $\mathcal{C}_0$ contain the parser configurations compatible with the fixed prefix $\boldsymbol{u}$. The token-labeled transitions and reachable configurations are

$$\begin{aligned} E_i &= \big\{(c, v, c') \mid c \in \mathcal{C}_{i-1},\ v \in \mathcal{V}_i^t,\ c' \in \mathsf{T}_{\mathcal{G}}(c, v)\big\}, \\ \mathcal{C}_i &= \big\{c' \mid \exists c, v :\ (c, v, c') \in E_i\big\}, \qquad i = 1, \ldots, H. \end{aligned} \tag{1}$$

The position-indexed layers $\{\mathcal{C}_i\}_{i=0}^{H}$ and transitions $\{E_i\}_{i=1}^{H}$ form a layered *canvas-specific grammar graph* $\mathcal{F}_t$. A full-window path is admissible when its endpoint satisfies the chosen terminal acceptance or continuation condition. With token weights from P_t, max-product dynamic programming finds a highest-weight admissible path following the Viterbi principle (Viterbi, 1967), while sum-product computes path-weight sums that support probabilistic inference (Goodman, 1999; Dang & Ermon, 2026). These objectives are defined on the specified graph.

Constructing $\mathcal{F}_t$ therefore requires the exact successor set for each relevant configuration–token pair, not merely a validity mask. The source–token–successor associations determine which token choices can be connected across positions.

Problem Formulation. In AR constrained decoding, token selection uses a validity mask for the realized prefix, after which the parser advances with the selected token; the prefix may itself admit multiple parser configurations (Dong et al., 2025). DLM joint inference additionally requires successors induced by alternative token choices before commitment. An exhaustive construction without query reuse or token grouping performs

$$N_{\text{query}}^{\text{naive}} = \sum_{i=1}^{H} |\mathcal{C}_{i-1}|\,|\mathcal{V}_i^t| \tag{2}$$

configuration–token queries. This counts successor queries, not returned edges or an unavoidable computational lower bound.

Finite-state pipelines can shift this work to *cold compilation* by precomputing token-level transitions, as in DINGO (Suresh et al., 2025). For a materialized finite state space $Q_{\mathcal{G}}$, the full state–token domain contains $|Q_{\mathcal{G}}||\mathcal{V}|$ pairs. A dense table stores an entry for every pair, although sparse representations have different storage costs. Expanding runtime contexts into explicit finite states can

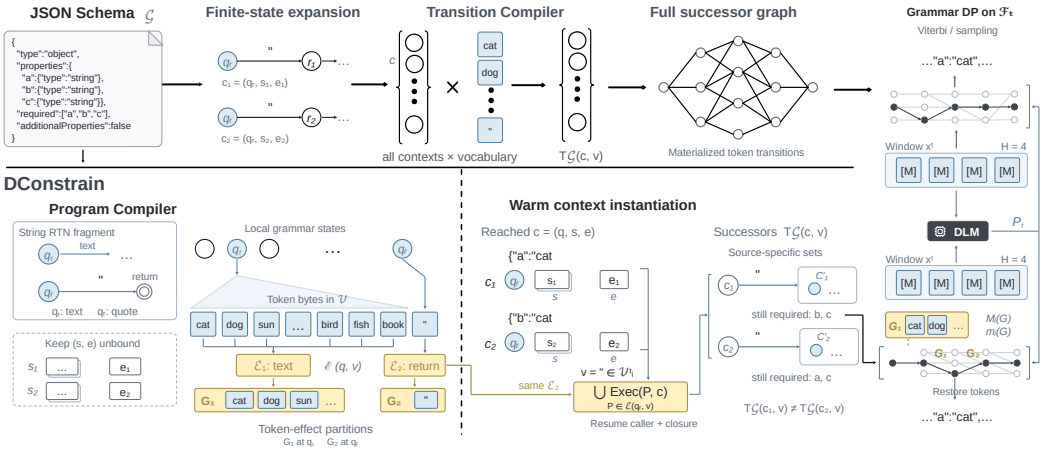


Figure 1: DConstrain's compilation and inference pipeline. Finite-state baselines materialize context-specific token transitions; DConstrain compiles context-parameterized programs with stacks and environments unbound. Warm compilation recovers source-specific successors for the canvas-specific grammar graph. In the three-field example, a shared closing-quote program yields different successor sets under different field histories. Local token-effect grouping preserves weighted inference through objective-specific score aggregation and token restoration. The token boundaries, graph topology, and four-token windows are schematic.

nevertheless duplicate shared transition logic and precompute successors for contexts never reached by a given decoding canvas.

We therefore seek a reusable compiled artifact $\mathcal{A}_{\mathcal{G}}$ that provides the *exact successor-construction interface*

$$\mathrm{Succ}(\mathcal{A}_{\mathcal{G}}, c, v) = \mathsf{T}_{\mathcal{G}}(c, v) \tag{3}$$

for every relevant configuration–token pair, under a fixed supported grammar, tokenizer, and parser protocol. With identical initial configurations, candidate sets, and boundary conditions, this source-specific equality preserves the logical canvas-specific grammar graph. Our objective is to reduce cold-compilation cost, compiled storage, and online successor-construction work without eagerly materializing the full context-expanded transition space. The compiled representation may share transition logic, but must preserve the source–token–successor associations required by joint inference.

4 Context-Parameterized Transition Programs

DConstrain realizes the exact successor-construction interface in Eq. (3) through *context-parameterized transition programs*. Its *two-stage compilation* separates reusable token-consumption logic from context-specific successor construction: cold compilation uses *pushdown factorization* to build shared programs with runtime stack and parsing-environment contexts unbound; warm compilation executes these programs only for configurations reached by the current decoding canvas, constructing the *canvas-specific grammar graph* on demand. *Token-effect grouping* further shares successor construction across equivalent tokens and compresses the edges processed by joint inference, preserving the ungrouped graph's weighted inference under objective-specific score aggregation and token restoration. Compiler–runtime co-design reduces exposed runtime overhead. Figure 1 summarizes the pipeline. Appendices A.3 and A.5 provide the correctness argument and complexity analysis.

4.1 Pushdown Transition Factorization

Given a supported schema $\mathcal{G}$, DConstrain represents local grammar logic as a recursive transition network (RTN), keeping the parser stack and parsing environment as runtime context. Cold compilation specializes token consumption to each local grammar state–token pair (q, v) while leaving

(s, e) unbound:

$$(q, v) \xrightarrow{\text{ColdCompile}} \mathcal{E}(q, v) = \{P_k\}_{k=1}^{K_{q,v}}. \tag{4}$$

Each program $P_k = (I_{k,1}; \ldots; I_{k,m_k})$ records a grammar alternative as an ordered instruction sequence. Cold compilation resolves context-independent byte matching and grammar operations, retaining stack and parsing-environment instructions for runtime execution. Configurations sharing q therefore reuse $\mathcal{E}(q, v)$ rather than separately compiling their transitions, even when their concrete successors differ.

Cold compilation resolves return destinations determined by local symbolic execution. When further token consumption or token-boundary processing depends on a caller selected by the incoming stack, it records a residual containing the token input and execution state needed to resume parsing. Warm compilation resolves the residual under the concrete runtime context. Shared stack operations thus remain parameterized, while execution requiring an unknown caller is deferred rather than approximated.

Parsing-environment factorization similarly shares transition logic across earlier parsing choices. Under arbitrary property order, the environment tracks field history, while programs retain guards and updates for field availability and required-field completion. These programs are shared across field-history combinations, preserving operation order and scope without expanding those histories into local grammar states.

4.2 Warm Compilation

At position i, warm compilation binds each reached configuration $c = (q, s, e) \in \mathcal{C}_{i-1}$ to the shared programs for candidate tokens $v \in \mathcal{V}_i^t$, recovering the successor set

$$\widehat{\mathsf{T}}_{\mathcal{G}}(c, v) = \bigcup_{P \in \mathcal{E}(q,v)} \text{Exec}(P, c). \tag{5}$$

Execution applies ordered stack and parsing-environment operations, resolves residuals, and performs token-boundary processing. These successors drive the layer-by-layer construction in Eq. (1), without precomputing transitions for contexts unreachable under the current prefix and candidate sets.

Graph nodes are indexed by (i, c), and edges retain their source configurations and token labels. Shared storage preserves the association among q, s, and e within each configuration rather than combining these components independently.

For completed queries, Appendix A.3 establishes $\widehat{\mathsf{T}}_{\mathcal{G}}(c, v) = \mathsf{T}_{\mathcal{G}}(c, v)$ under the reference parser semantics. With all required queries completed and identical initial configurations, candidate sets, and boundary conditions, this source-specific equality preserves the logical canvas-specific grammar graph required by joint inference.

4.3 Token-Effect Grouping

A reached configuration may still require successor queries for many candidate tokens. DConstrain groups tokens with identical compiled grammar effects at the same local grammar state:

$$v_1 \sim_q v_2 \quad \Longleftrightarrow \quad \text{Canon}(\mathcal{E}(q, v_1)) = \text{Canon}(\mathcal{E}(q, v_2)). \tag{6}$$

Cold compilation records the partition $\Pi_q = \mathcal{V}/\!\sim_q$ after constructing the token effects. Canonicalization retains instruction order, operands, stack and parsing-environment operations, and residual execution information. Equal canonical programs therefore yield the same successors under every compatible runtime context.

At position i, warm compilation computes successors once for each reached configuration $c = (q, s, e)$ and active group $G \in \Pi_q$:

$$\widehat{\mathsf{T}}_{\mathcal{G}}(c, G) = \widehat{\mathsf{T}}_{\mathcal{G}}(c, v_G), \qquad v_G \in G \cap \mathcal{V}_i^t. \tag{7}$$

Within each layer, parallel token-labeled edges sharing the same source, successor, and token-effect group are represented by one group-level edge. This reduces both successor construction and the

explicit edges processed by joint inference, while retaining individual token identities through group membership.

Given nonnegative token weights $p_i(v)$ from P_t, define

$$M_i(G) = \sum_{v \in G \cap \mathcal{V}_i^t} p_i(v), \qquad m_i(G) = \max_{v \in G \cap \mathcal{V}_i^t} p_i(v). \tag{8}$$

Sum-product inference uses $M_i(G)$ and restores an admitted token with probability $p_i(v)/M_i(G)$ for positive-mass groups. Max-product inference uses $m_i(G)$ and restores a group maximizer. Appendix A.4 proves that these weighted reductions preserve the Viterbi optimum and, with conditional token restoration, the full token-labeled path distribution of the corresponding ungrouped graph.

Solver-agnostic inference interface. Preserving the source-specific successor relation and both weighted-inference operators allows the same compiled representation to support different automata-based solvers. Appendix D.2 evaluates Viterbi and the inference algorithm of Dang & Ermon (2026).

Runtime integration. To limit exposed runtime overhead, DConstrain supports overlapping warm compilation with model forward computation. Compact compressed sparse row (CSR) representations and specialized GPU operators enable efficient joint inference over the canvas-specific grammar graph. Appendix B.3 details the compiler–runtime integration and the optimizations enabled in each experimental variant.

5 Experiments

Workloads and models. We evaluate on BFCL (Patil et al., 2025), JSONSchemaBench (Geng et al., 2025), and ComplexFuncBench (Zhong et al., 2025). BFCL and ComplexFuncBench provide tool declarations for both compilation and task-level evaluation, while JSONSchemaBench stress-tests compilation with diverse and complex schemas. For end-to-end evaluation, we select two recent DLM families with substantially different decoding granularities: LLaDA2.2 (Bie et al., 2026) with a block size of 32 and DiffusionGemma (DiffusionGemma Team et al., 2026) with a block size of 256. To evaluate constrained decoding under realistic serving conditions, we use the SGLang (Zheng et al., 2024) implementation of LLaDA2.2 and the vLLM (Kwon et al., 2023) implementation of DiffusionGemma rather than standalone research inference loops. Cold-compilation experiments require only the corresponding tokenizers.

Comparisons. For cold compilation, we compare with DINGO's DFA compiler (Suresh et al., 2025), an NFA adaptation, and our engineering-optimized DINGO-Opt under matched schemas and tokenizers. Runtime comparisons use unconstrained decoding and DINGO-Opt with matched model and decoding settings on optimized serving engines, where constraint-processing overhead is directly relevant.

5.1 Main Results

Table 1 evaluates cold compilation under fixed and arbitrary property order, with nesting depth limited to four for comparability with finite-state baselines. Any-order compilation preserves all schema-valid property orders; Appendix C.1 shows that this distinction also affects task accuracy. Engineering details appear in Appendix B.3. Table 2 compares DConstrain, DINGO-Opt, and unconstrained decoding on BFCL and JSB under matched model and decoding settings. Constrained runs use depth four and fixed property order unless marked any-order. Constrained results cover cases successfully compiled by both methods in each setting; runtime latencies are means excluding cold compilation and loading (Appendix B.2).

Cold-compilation efficiency. DConstrain consistently reduces compilation cost across both tokenizers and all datasets. Under fixed order and depth four, it achieves 3.88–$4.83\times$ paired-mean speedups over DINGO-Opt, despite DINGO-Opt already incorporating the C++ pipeline and CSR representation of Appendix B.3. The gap widens when arbitrary property order is enabled: DConstrain retains 100% compilation coverage on BFCL and CFB, whereas DINGO-Opt loses substantial

Table 1: Cold-compilation performance on BFCL, JSONSchemaBench (JSB), and Complex-FuncBench (CFB). Success@180s is the fraction of all attempted schemas compiled successfully within 180 s; P50/P95 use each method's successful cases. DINGO-Opt* retains DINGO's finite-state strategy with the same engineering optimizations as DConstrain (Appendix B.3). Speedup† is the row method's mean latency divided by DConstrain's on paired successful cases; timeouts count only toward Success@180s.

	LLaDA2.2				DiffusionGemma			
Method	**Success** (%)	**P50** (s)	**P95** (s)	**Speedup**†	**Success** (%)	**P50** (s)	**P95** (s)	**Speedup**†
BFCL — Fixed order								
DINGO	90.3	54.30	132.90	92.9×	91.7	38.45	115.41	140.4×
NFA	82.5	61.30	148.50	113.4×	86.0	45.52	122.17	167.4×
DINGO-Opt*	**100.0**	2.21	5.64	3.92×	**100.0**	1.20	3.29	3.88×
DConstrain	**100.0**	**0.58**	**1.48**	—	**100.0**	**0.31**	**0.82**	—
BFCL — Any order								
DINGO-Opt*	71.3	5.85	12.25	9.43×	85.8	3.76	14.16	10.39×
DConstrain	**100.0**	**0.66**	**1.23**	—	**100.0**	**0.41**	**0.90**	—
JSB — Fixed order								
DINGO	67.7	38.00	129.11	85.0×	67.7	31.97	96.34	122.9×
NFA	63.7	45.26	128.40	102.8×	71.3	30.71	113.85	114.0×
DINGO-Opt*	74.8	2.51	70.50	4.83×	77.6	1.65	37.66	4.53×
DConstrain	**81.4**	**0.60**	**14.14**	—	**83.3**	**0.37**	**8.27**	—
CFB — Fixed order								
DINGO	34.8	115.18	159.24	121.9×	44.7	84.73	124.04	174.6×
NFA	32.5	131.29	175.14	132.0×	74.0	116.63	171.69	221.9×
DINGO-Opt*	**100.0**	4.66	7.34	4.02×	**100.0**	2.21	3.02	3.96×
DConstrain	**100.0**	**1.17**	**1.78**	—	**100.0**	**0.56**	**0.76**	—
CFB — Any order								
DINGO-Opt*	9.9	7.63	16.69	8.12×	13.7	2.90	9.40	3.54×
DConstrain	**100.0**	**0.99**	**1.41**	—	**100.0**	**0.80**	**1.16**	—

coverage, while remaining 3.54–10.39× faster on cases completed by both methods. Thus, removing artificial ordering restrictions exposes the scalability advantage of factorized compilation in both coverage and latency.

Warm compilation and inference efficiency. DConstrain preserves 100% schema validity and matches DINGO-Opt's BFCL task accuracy while substantially reducing grammar-inference and end-to-end latency (Table 2). Although context-specific graph construction is deferred to runtime, Figure 2 shows that most warm compilation overlaps with model forward computation, leaving only a small fraction exposed on the critical path. Solver comparisons and verification-based baselines appear in Appendices D.2 and D.3.

5.2 Ablation Study

Table 3 isolates three mechanisms in DConstrain on BFCL with the LLaDA2.2 tokenizer and arbitrary property order. *w/o Two-Stage Env* binds parsing-environment values during cold compilation rather than deferring their specialization to runtime; *w/o Local-State Reuse* disables memoization of transition programs for repeated occurrences of the same local grammar state, while retaining the factorized runtime environment and continuation stack; and *w/o Token Grouping* processes tokens with equivalent transition effects independently. All variants preserve the same grammar semantics and decoding setup.

The results show that these mechanisms remove different forms of redundancy. Deferred environment specialization provides the largest cold-compilation benefit, with eager binding raising mean compilation latency to 14.61× that of the full system. Local-state reuse avoids recompiling identical local transition logic, while token-effect grouping primarily reduces vocabulary-level work in

Table 2: Runtime quality and latency on BFCL and JSONSchemaBench (JSB). Constrained results use paired successful compilations within each setting; Native is unconstrained. E2E is mean request latency; Solve/call is mean within-request grammar-inference time per invocation.

Dataset	Method	LLaDA2.2				DiffusionGemma			
		Valid. (%) ↑	Acc. (%) ↑	E2E (ms) ↓	Solve/call (ms) ↓	Valid. (%) ↑	Acc. (%) ↑	E2E (ms) ↓	Solve/call (ms) ↓
BFCL	Native	93.1	79.4	85.42	—	69.2	56.7	1948.23	—
	DINGO-Opt*	100.0	80.4	264.23	32.45	100.0	35.8	12874.32	216.72
	DConstrain	**100.0**	**80.4**	**115.21**	**3.16**	**100.0**	**35.8**	**2521.42**	**9.95**
BFCL (any-order)	DINGO-Opt*	100.0	83.4	—	—	100.0	74.1	145529	3195.39
	DConstrain	100.0	83.4	122.35	2.81	100.0	74.1	3145.64	29.44
JSB	Native	73.7	—	169.42	—	55.3	—	1894.82	—
	DINGO-Opt*	100.0	—	2282.87	172.57	100.0	—	54323.33	853.25
	DConstrain	**100.0**	—	**327.61**	**11.78**	**100.0**	—	**5388.21**	**34.17**

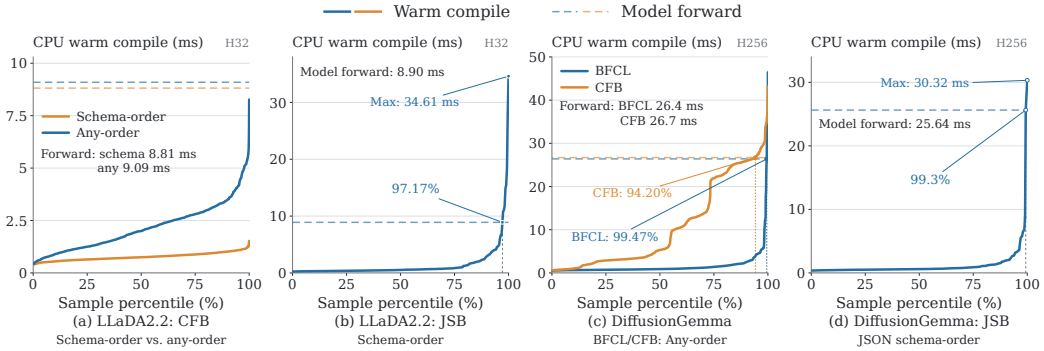


Figure 2: Warm-compilation latency under asynchronous execution. CPU warm compilation overlaps with GPU model execution; only warm compilation exceeding the mean model-forward duration (dashed lines) is exposed on the critical path.

downstream grammar inference, with its removal raising mean solve latency to $6.81\times$ that of the full system. Appendix D.1 further compares DConstrain with direct on-demand PDA interpretation to isolate the benefit of compiled transition programs beyond avoiding finite-state expansion.

5.3 Analysis

DConstrain is designed to avoid materializing runtime parsing contexts into explicit finite states. We therefore analyze the two sources of context growth most directly targeted by the factorized representation: alternative object-property orders, which enlarge the parsing environment, and nested structures, which enlarge the continuation stack.

Factorized representation. Table 4 compares the stored representations of finite-state baselines and DConstrain. Finite-state compilation materializes context-expanded states and token-level transitions, whereas DConstrain stores local RTN structure and symbolic transition programs while leaving parsing environments and continuation stacks to runtime. This yields substantially smaller recorded payloads across all three datasets. Because the representations use different state and edge units, these counts are descriptive rather than directly comparable; deferred graph construction is accounted for separately in the runtime experiments.

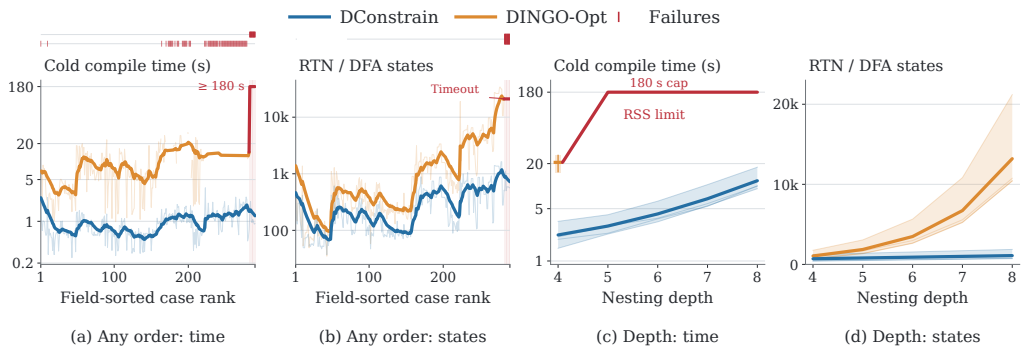
Scaling with property order and nesting depth. Figure 3 varies the two runtime-context dimensions factorized by DConstrain: arbitrary property order enlarges the parsing environment, while deeper nesting enlarges the continuation stack. Finite-state compilation materializes these contexts explicitly, whereas DConstrain keeps the local grammar logic symbolic and carries them at runtime.

Table 3: Compiler ablations on BFCL with arbitrary property order. Values are means; parentheses show ratios to the full system.

Variant	Cold (s) ↓	Solve (ms) ↓	Memory (MiB) ↓
Full (Any Order)	**0.72**	**3.72**	**5.94**
w/o Two-Stage Env	10.52 (14.61×)	3.83 (1.03×)	20.97 (3.53×)
w/o Local-State Reuse	1.87 (2.60×)	3.75 (1.01×)	12.98 (2.19×)
w/o Token Grouping	0.96 (1.33×)	25.32 (6.81×)	24.42 (4.11×)

Table 4: Compiled representation statistics with the LLaDA2.2 tokenizer.

Dataset	Method	States	Edges (10^3)		Recorded payload (MB)	
		Mean	P50	P95	P50	P95
BFCL	DINGO	228.74	524.77	1 226.27	54.20	114.91
	NFA	233.74	352.71	1 052.08	31.95	93.55
	DConstrain	184.00	3.45	14.63	**4.74**	**6.31**
JSONSchemaBench	DINGO	182.42	350.97	1 224.94	36.86	118.83
	NFA	214.94	353.62	1 054.28	36.99	130.63
	DConstrain	163.09	3.60	264.38	**5.03**	**31.35**
ComplexFuncBench	DINGO	404.39	1 053.46	1 405.18	90.61	107.51
	NFA	409.39	1 053.28	1 405.17	121.60	139.51
	DConstrain	309.13	13.08	20.62	**5.11**	**5.94**



(a) Any order: time (b) Any order: states (c) Depth: time (d) Depth: states

Figure 3: Scaling with runtime-context complexity on a fixed 300-case BFCL subset. (a,b) Arbitrary-order objects are ranked by maximum parameter-object field count; (c,d) nesting depth varies from 4 to 8.

As either dimension grows, DINGO-Opt increasingly reaches the time or memory limit, including all evaluated cases at depths 5–8, while DConstrain continues to compile successfully. This trend is consistent with the full-pool results in Table 1. Arbitrary property order is also practically relevant: Appendix C.1 shows that fixing a single order can materially reduce DLM task accuracy despite preserving schema validity.

6 Conclusion

In this work, we presented DConstrain, a compiler system for efficient constrained decoding in diffusion language models. By compiling reusable transition programs instead of context-expanded finite states, DConstrain reduces redundant state and transition materialization. Combined with compiler–runtime co-design, DConstrain substantially improves compilation and inference efficiency while preserving structured-output validity and task accuracy.

Ethics Statement

This work studies constrained decoding for diffusion language models and does not involve human subjects, personally identifiable information, or the collection of private user data. Our experiments use existing model checkpoints and established function-calling and JSON-schema benchmarks. The proposed method enforces syntactic and schema-level constraints during generation; such constraints can improve structural validity but do not guarantee the semantic correctness, safety, or appropriateness of generated content. Accordingly, constrained outputs should not be interpreted as independently verified or safe solely because they satisfy a grammar or schema. We report these limitations explicitly and encourage downstream users to retain application-level validation and safety checks.

Reproducibility Statement

We provide the algorithmic definitions, correctness arguments, implementation details, experimental protocols, and additional analyses required to reproduce the main results. The paper specifies the compilation and runtime pipeline, including pushdown factorization, parameterized transitions, token-effect partitioning, warm compilation, graph construction, and downstream inference. We additionally report model and tokenizer settings, decoding configurations, hardware and software environments, dataset construction and filtering rules, timeout and failure accounting, and the sampling protocols used for ablation and scaling analyses. Main cold-compilation experiments are evaluated on the full benchmark pools, while runtime comparisons are computed on schemas successfully compiled by the compared methods; fixed sampled subsets used in subsequent analyses are explicitly identified. Mathematical claims are accompanied by formal definitions and proofs in the appendix. These details are intended to make both the algorithmic behavior and the reported system measurements independently verifiable.

AI Use Statement

Generative AI tools were used in a supportive manner during manuscript preparation, including language polishing, and assistance in developing and checking mathematical derivations and proofs. AI-generated suggestions were not treated as authoritative results. All AI-assisted text and proofs were reviewed, revised where necessary, and independently verified by the authors. The authors take full responsibility for the correctness of the implementation, experiments, mathematical arguments, citations, and claims presented in this work.

References

Tiwei Bie, Jiansheng Cai, Maosong Cao, Xiang Cao, Binbin Chen, Fuyuan Chen, Kun Chen, Yuqi Ding, Lun Du, Zhuocheng Gong, Yanmei Gu, Kaiyuan Guan, Xiaoxu Guo, Kairong Han, Zenan Huang, Zijin Huang, Zhenzhong Lan, Chengxi Li, Jianguo Li, Zhe Li, Ruiqi Liang, Xinyao Lin, Huabin Liu, Lin Liu, Xinyi Liu, Xuejie Liu, Guoshan Lu, Yuan Lu, Yuxin Ma, Chenhui Mao, Xiaojing Qi, Kaida Qiu, Xiaoyu Shi, Yiding Tian, Huanyu Wang, Qi Wang, Rui Wang, Shuaidi Wang, Lanning Wei, Tao Wu, Siyang Xiao, Yipeng Xing, Chenkai Xu, Song Yan, Ying Yan, Jingyi Yang, Yichun Yang, Liangyu Zha, Tianze Zhang, Yifan Zhang, Junbo Zhao, Wenqian Zhao, Da Zheng, Jianyuan Zhong, Jiahao Zhou, Junlin Zhou, Tianyu Zhou, Liwang Zhu, and Yihong Zhuang. LLaDA2.2: Enabling agentic diffusion language models via levenshtein editing. Technical report, Inclusion AI, 2026. URL https://github.com/inclusionAI/LLaDA2.X/blob/main/LLaDA2_2_tech_report.pdf.

Meihua Dang and Stefano Ermon. Constrained decoding for diffusion language models via efficient inference over finite automata, 2026. URL https://arxiv.org/abs/2607.07026. arXiv:2607.07026.

DiffusionGemma Team, Adrien Ali Taïga, James Assiene, Daniele Calandriello, Rahma Chaabouni, João Gante, Tamara von Glehn, Nate Keating, Chris Knutsen, Martin Kukla, Tianlin Liu, Ivan Lobov, Ofir Nabati, João Gabriel Oliveira, Nicolas Perez-Nieves, Nastasia Prutianova, Bobak Shahriari, Jean Tarbouriech, Pavel Tyletski, Çağlar Ünlü, Cindy Wu, Glenn Cameron, Jerome

Connor, Sertan Girgin, Maarten Grootendorst, Alon Levkovitch, Eliya Nachmani, Omar Sanseviero, Piotr Stanczyk, Quentin Berthet, Andrew Campbell, Clément Crepy, Valentin De Bortoli, Arnaud Doucet, Romuald Elie, Alexandre Galashov, Klaus Greff, Alexis Jacq, David Ruhe, Yu-Han Wu, Sebastian Flennerhag, Brendan O'Donoghue, George Scrivener, and Shantanu Thakoor. DiffusionGemma Technical Report, 2026. URL https://arxiv.org/abs/2608.00146. arXiv:2608.00146.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. XGrammar: Flexible and efficient structured generation engine for large language models. In *Proceedings of Machine Learning and Systems*, volume 7. MLSys, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/5c20ca4b0b20b0bd2f1d839dc605e70f-Abstract-Conference.html.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. JSONSchemaBench: A Rigorous Benchmark of Structured Outputs for Language Models, 2025. URL https://arxiv.org/abs/2501.10868. arXiv:2501.10868.

Joshua Goodman. Semiring parsing. *Computational Linguistics*, 25(4):573–606, 1999. URL https://aclanthology.org/J99-4004/.

Hyundong Jin and Yo-Sub Han. EPIC: Efficient and parallel inference under CFG constraints for diffusion language models, 2026. URL https://arxiv.org/abs/2606.00722. arXiv:2606.00722.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Linzhang Li, Yixin Dong, Guanjie Wang, Ziyi Xu, Alexander Jiang, and Tianqi Chen. XGrammar-2: Dynamic and efficient structured generation engine for agentic LLMs, 2026. URL https://arxiv.org/abs/2601.04426. arXiv:2601.04426.

Michał Moskal, Harsha Nori, Hudson Cooper, and Loc Huynh. LLGuidance: Making structured outputs go brrr. Microsoft technical article, 2025. URL https://guidance-ai.github.io/llguidance/llg-go-brrr. June 11, 2025.

Niels Mündler, Jasper Dekoninck, and Martin Vechev. Constrained decoding of diffusion LLMs with context-free grammars. In *International Conference on Learning Representations*, pp. 99362–99408, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/a1622fc2f8e37ea5fc696d1c13861c36-Abstract-Conference.html.

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. doi: 10.52202/085713-1689. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/48b383b24230e0e6e649d9c98dae4d8c-Abstract-Conference.html.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *International Conference on Machine Learning*, pp. 48371–48392, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

Subham S. Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and Effective Masked Diffusion Language Models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 130136–130184, 2024. doi: 10.52202/079017-4135. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/eb0b13cc515724ab8015bc978fdde0ad-Abstract-Conference.html.

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. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.emnlp-main.779. URL https://aclanthology.org/2021.emnlp-main.779/.

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. Curran Associates, Inc., 2025. doi: 10.52202/085713-5362. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/eb17a2030d1bd4a1bd29531bcd626705-Abstract-Conference.html.

A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967. doi: 10.1109/tit.1967.1054010. URL https://doi.org/10.1109/tit.1967.1054010.

Brandon T. Willard and Rémi Louf. Efficient Guided Generation for Large Language Models, 2023. URL https://arxiv.org/abs/2307.09702. arXiv:2307.09702.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7B: Diffusion Large Language Models, 2025. URL https://arxiv.org/abs/2508.15487. arXiv:2508.15487.

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, 2026. URL https://arxiv.org/abs/2602.00612. arXiv:2602.00612.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H. Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583, 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Lucen Zhong, Zhengxiao Du, Xiaohan Zhang, Haiyi Hu, and Jie Tang. ComplexFuncBench: Exploring Multi-Step and Constrained Function Calling under Long-Context Scenario, 2025. URL https://arxiv.org/abs/2501.10132. arXiv:2501.10132.

A Additional Background and Theoretical Analysis

This appendix formalizes the parser interface used in Section 3, establishes correctness of the compiler in Section 4, establishes weighted inference preservation under token-effect grouping, and analyzes compilation and graph-construction costs.

A.1 Grammar Automata and Parser Configurations

Let $\mathcal{G}$ define an output language $\mathcal{L}_{\mathcal{G}}$. Finite-state constraints admit an automaton $\mathcal{M}_{\mathcal{G}} = (Q, \Sigma, \delta_{\mathrm{FA}}, q_0, F)$, with transitions $\delta_{\mathrm{FA}} : Q \times \Sigma \to 2^Q$; a DFA has at most one successor per state–symbol pair, whereas an NFA may have several. For nested grammars, a pushdown representation augments local control with a runtime stack (Dong et al., 2025). Its logical configuration is

$$c = (q, s, e), \tag{9}$$

where s stores continuations and e contains parsing variables. Guards, stack operations, and environment updates allow the same local transition logic to operate across different contexts.

Token-level transitions consume tokenizer tokens rather than individual grammar symbols (Willard & Louf, 2023; Dong et al., 2025). Let $\mathsf{T}^{\mathrm{raw}}_{\mathcal{G}}(c, v)$ contain all successors after consuming v, including required stack, environment, and token-boundary operations. We use the normalized relation

$$\mathsf{T}_{\mathcal{G}}(c, v) := \mathsf{T}^{N}_{\mathcal{G}}(c, v) = N\big(\mathsf{T}^{\mathrm{raw}}_{\mathcal{G}}(c, v)\big)\,, \tag{10}$$

where N removes statically certified dead configurations, as defined below. Relations extend to frontiers by union over their configurations. For finite automata, c reduces to a state q.

A.2 Diffusion Language Models

A partially resolved window $\boldsymbol{x}^t \in (\mathcal{V} \cup \{[\texttt{MASK}]\})^H$ follows a fixed prefix $\boldsymbol{u}$. At denoising iteration t, the model predicts distributions at all positions (Sahoo et al., 2024; Nie et al., 2025):

$$\begin{aligned} P_t &= \mathrm{Model}_\theta(\boldsymbol{u}, \boldsymbol{x}^t), \\ P_t[i, v] &= p_\theta(y_i = v \mid \boldsymbol{u}, \boldsymbol{x}^t). \end{aligned} \tag{11}$$

A decoding policy selects updates for the next iteration. Multiple unresolved positions permit joint grammar-constrained token selection (Suresh et al., 2025; Dang & Ermon, 2026). Let $\mathcal{V}_i^t$ be the admitted tokens at position i; fixed positions can use singleton sets. All comparisons below use identical candidate sets, prefixes, protocols, and boundary conditions.

A.3 Correctness of Factorized Compilation

We expose $e = (d, \eta)$, where d is container depth and η is the choice environment, and write $c = (q, s, d, \eta)$. Execution states also retain input, cursor, and protocol phase. A token-complete state has finished all required boundary processing; an exhausted cursor alone is insufficient. We assume agreement only between corresponding primitive parser operations. All results use set semantics and concern completed computations; resource exhaustion is an error, not an empty result. Byte execution, compilation, and assembly are defined for every finite $w \in \mathbb{B}^*$ over the byte alphabet $\mathbb{B}$. Token compilation specializes $w = \mathrm{bytes}(v)$ with its initialization/finalization. Suffixes need not encode vocabulary tokens and retain the enclosing metadata and phase without fresh token initialization.

Static normalization. The abstract RTN soundly overapproximates concrete execution by relaxing environment guards while preserving call/return structure. For the set $\mathrm{Ret}(q)$ of finite abstract derivations returning from q's current rule, define

$$D(q) = \min_{\pi \in \mathrm{Ret}(q)} |\pi|_{\mathrm{bytes}}, \qquad \min \varnothing = \infty. \tag{12}$$

Rule exits, including complete acceptance as a distinguished root exit, are productive terminals of distance zero. Consequently,

$$\begin{aligned} &D(q) \le D(q') && \text{for zero-byte edges,} \\ &D(q) \le 1 + D(q') && \text{for byte edges,} \\ &D(q) \le D(a) + D(r) && \text{for calls to } a \text{ returning to } r, \\ &D(q) = 0 && \text{at rule exits.} \end{aligned} \tag{13}$$

Min-plus relaxation starts with zero at terminals and infinity elsewhere. Each finite value has a derivation witness, and saturation covers every finite return, computing Eq. (12). Thus $D(q) = \infty$ excludes any concrete return; finite distance need not imply concrete feasibility. Define

$$P(c) = [D(q) < \infty] \wedge \bigwedge_{r \in s} [D(r) < \infty], \qquad N(X) = \{c \in X : P(c)\}, \tag{14}$$

Complete-output accepting configurations satisfy P by their terminal distance and empty stack. Protocol steps obey the same abstract control rules and do not bypass required returns. Infinity is used only after the distance computation completes.

Lemma A.1 (Normalization safety)**.** *For raw byte execution δ_{raw} with its prescribed zero-byte closures,*

$$N(\delta_{\mathrm{raw}}(N(X), w)) = N(\delta_{\mathrm{raw}}(X, w)). \tag{15}$$

The same property holds for closures and complete token execution, including normalization at intermediate stages.

Proof. Every primitive step satisfies $c \to c' \wedge P(c') \Rightarrow P(c)$. For local and byte edges this follows from the distance inequalities. For calls, productivity of the successor implies finite distances for the callee and pushed return address. For returns, the source is an exit, and the popped address and remaining stack have finite distances. Consequently, every finite execution ending in a retained configuration contains only retained configurations. Such a path survives initial and intermediate normalization; the opposite inclusion follows from $N(X) \subseteq X$. □

Write $\delta_N(X, w) = N(\delta_{\mathrm{raw}}(X, w))$. Unlike the raw relation, this semantics excludes token endpoints that are statically known never to complete the output.

Representations and admissibility. A parameterized bundle $B = (\mathbf{a}, \mathbf{R}, \eta_0)$ contains atoms $a_i = (q_i, d_i, \Omega_i)$ and their own stack relations R_i. Its logical interpretation is

$$\mathcal{D}(B) = \bigcup_i \{(q_i, s, d_i, \Omega_i(\eta_0)) : s \in R_i,\ \Omega_i(\eta_0) \neq \bot\}. \tag{16}$$

Here Ω_i is an ordered program of partial choice operations. A public representation is *admissible* when its atom–stack–program associations are preserved; control, frames, operands, and cursor/protocol metadata are well formed in their binding scope; and its decoded frontier is normalized and in entry-closure form. Initialization creates aligned atoms and applies closure and N. The transfers below preserve these invariants; internal seeds carry their specified suspended phases.

Updates modify the current environment; a non-committing guard obeys

$$\mathsf{check}_{\{U_j\}}(\eta) = \begin{cases} \eta, & \exists j :\ U_j(\eta) \neq \bot, \\ \bot, & \text{otherwise.} \end{cases} \tag{17}$$

Each trial starts from the same current environment. Ordered composition satisfies $(U; V)(\eta) = V(U(\eta))$, including failure.

Source-indexed decoding. At each token query's entry, label every source c by its immutable origin $o = c$. Source selectors and stack transformations denote a relation $\mathcal{R}(\widehat{Y})$ between origins and current configurations. Define

$$\mathcal{D}_o(\widehat{Y}) = \{c' : (o, c') \in \mathcal{R}(\widehat{Y})\}, \qquad \mathcal{D}(\widehat{Y}) = \bigcup_o \mathcal{D}_o(\widehat{Y}). \tag{18}$$

Initially $\mathcal{R} = \{(c, c) : c \in \mathcal{D}(\widehat{X})\}$. Transfers keep the origin while changing the current configuration. Stack operations act pointwise on the selected atom's input stack, not just its index. Merging unions these relations; shared suffixes compose them through the same intermediate configuration, never through a Cartesian product. Origins are proof annotations on existing selectors and stack maps, fixed within a query and reset at the next query.

Independent byte execution. The byte executor matches source atoms, appends call frames, partitions returns by their actual top frame, and appends choice operations in order. Its closure explores all prescribed zero-byte alternatives, deduplicates full current states with their source associations, and applies N to each atom's own stack relation. It does not invoke local-effect assembly.

Lemma A.2 (Source-indexed byte simulation)**.** *For every origin o and byte string w, the specified byte executor with matching closure conventions satisfies*

$$\mathcal{D}_o\Big(\widehat{\delta}_{\mathrm{bytes}}(\widehat{Y}, w)\Big) = \delta_N\Big(\mathcal{D}_o(\widehat{Y}), w\Big). \tag{19}$$

Closure and token-finalization satisfy the analogous equations, and full public operations return admissible representations.

Proof. Fix o. Primitive agreement gives the same byte, stack, and choice successors for each of its configurations, retaining o. Extracting an identical, equally bound prefix U preserves $V_i(U(\eta_0)) = (U; V_i)(\eta_0)$ and the origin associations. Merging identical current states unions their origins and shares their future relation. Newly attached origins receive those successors by propagation or reuse of that relation; saturation therefore covers every origin's finite closure paths. Normalization acts per endpoint, so Lemma A.1 applies to each origin. Induction over bytes, followed by the same finalization argument, proves the equation. Transfers preserve scopes and phases; public exit closure and N restore admissibility. □

Local summaries and worklist propagation. For arbitrary $w \in \mathbb{B}^*$, cold compilation scans bytes and expands local closures using items $h = (a, \rho, \Delta, m, \Omega, \nu)$. Here a includes control and execution phase, ρ is the unmatched local return stack, Δ is net depth change, m is its minimum prefix change including zero, and ν retains additional used metadata. Known calls and local returns modify ρ; choice operations append to Ω. Initially $\rho = \Omega = \varepsilon$ and $\Delta = m = 0$. A depth change ζ updates

$$(\Delta, m) \leftarrow (\Delta + \zeta, \min\{m, \Delta + \zeta\}). \tag{20}$$

Every explicit alternative is explored. A return with empty ρ emits a residual before accessing the unknown incoming stack. An ordinary summary $P = (q', \rho, \Delta, m, \Omega, \nu)$ binds as

$$\mathrm{Bind}(P; s, d, \eta) = \begin{cases} \{(q', s \parallel \rho, d + \Delta, \Omega(\eta))\}, & d + m \geq 0,\ \Omega(\eta) \neq \bot, \\ \varnothing, & \text{otherwise.} \end{cases} \tag{21}$$

Frames are appended bottom-to-top. Additional checks retain their primitive tests and operands; a depth-dependent check other than underflow also records its relative evaluation offset, or sufficient trace statistics, in keyed metadata ν. Binding evaluates these tests and preserves the source selector and input-stack argument. Structural steps have no unrecorded choice effects; execution phase determines remaining closure.

The worklist keys by $(a, \rho, \Delta, \Omega, \nu)$, excluding only m. Each new key or strict increase of its best m is enqueued. A popped item is skipped only when its m is stale; otherwise all successors are explored. Ordinary and residual output tables also retain the largest m.

Lemma A.3 (Summary and worklist preservation)**.** *For every concrete input binding, saturation preserves its local reference endpoints, cut before the first unknown-input-stack return, and attributes them to that same source.*

Proof. Along a local trace the stack is $s \parallel \rho$. If h_j denotes its cumulative depth change, then $\Delta = h_k$ and $m = \min_j h_j$; the underflow checks succeed exactly when $d + m \geq 0$. Ordered interpretation gives precisely $\Omega(\eta)$; other retained tests use their recorded operands and offsets. Each summary therefore has the same endpoint and failure domain as its witnessing trace, for that fixed input binding.

For equal keys and $m_1 \leq m_2$, the first success domain is contained in the second, and outputs agree. A common suffix with summary (Δ_U, m_U) yields

$$(\Delta, m); (\Delta_U, m_U) = (\Delta + \Delta_U, \min\{m, \Delta + m_U\}), \tag{22}$$

which is monotone in m. Dominance therefore persists through successors. Every retained item has a trace witness, giving soundness. Conversely, each prefix of any finite trace is retained or dominated by a retained item. Re-enqueuing improvements propagates this coverage to outputs at saturation, giving completeness. The inclusion and propagation argument holds for each input separately, and binding never changes its origin. Applying N preserves this sourcewise equality. □

Residual resolution. Whole-token replay consumes the original token from its original source, without first applying the compiled prefix. Source-indexed byte simulation gives that source's full normalized successor set; union with ordinary results already contained in that set does not change it or its origin.

Suffix restoration instead applies the prefix once, pops the actual source-stack return frame once, and continues in that caller with its retained environment, input, and protocol phase. The compiler's initial residual set is empty; new records appear only after a byte is consumed, with cursor equal to

its zero-based index plus one. Propagation preserves that cursor, and empty input generates only the identity effect. Hence compiler-generated residuals satisfy

$$1 \le j \le |w|, \qquad |w[j:]| < |w|. \tag{23}$$

Entry and caller closures handle zero-byte returns independently.

Lemma A.4 (Suffix assembly preservation)**.** *For every finite byte string w, compiler-generated effects, and origin o, assembly with its required closures satisfies*

$$\mathcal{D}_o\Big(\widehat{A}^{\mathrm{cl}}(\widehat{Y}, w)\Big) = \delta_N\Big(\mathcal{D}_o(\widehat{Y}), w\Big). \tag{24}$$

Proof. Fix o and induct strongly on $|w|$ over all finite byte strings, with compatible input and protocol phases. Empty input uses the identity effect and source-indexed closure simulation. Otherwise, summary and worklist preservation cover every ordinary endpoint and first residual cut from this origin. At a cut, prefix binding, the actual return, and caller closure recover its reference continuation without changing o. The suffix is shorter by Eq. (23), so induction applies whether or not it is a tokenizer token. Its results are composed with the prefix through the same caller configuration. An empty suffix can return an intermediate seed; the enclosing closure still processes its pending returns and checks. Union over alternatives and normalization commutation prove the claim for o. □

Token effects and transition preservation. Let $\mathcal{E}(q, v)$ be the token specialization of the byte-string compiler, including its prescribed initialization/finalization and residual modes. Define

$$\mathrm{Canon}(\mathcal{E}) = \mathrm{Ser}(\mathrm{Red}(\mathcal{E})), \tag{25}$$

where Red applies the proved duplicate and dominance reductions, and Ser completely encodes the resulting alternatives. It retains ordered operations, bindings, and all *ordinary and residual* token-boundary, initialization/finalization, and protocol information used by execution. Residuals additionally retain input, cursor, source bindings, and replay mode. Whole-token selectors retain token identity. Interned operands are compared within the same immutable registry and binding scope; hashing must resolve full key equality. Let Π_q partition tokens by these keys.

Lemma A.5 (Token-effect preservation)**.** *For fixed q and scope, equal canonical effects imply identical normalized successor sets for each admissible concrete source, with the same source attribution.*

Proof. The reductions preserve unions by the summary and worklist lemma. Structural equality identifies the same ordinary bindings, residual computations, and finalization for a fixed source. Their completed outcomes agree and retain that source. This is a sufficient equivalence test, not a complete semantic-equivalence procedure. □

Theorem A.6 (Sourcewise normalized token-transition preservation)**.** *For an admissible public source $\widehat{X}$ and compiler-generated effects, completed factorized token execution returns an admissible representation and, for every $c \in \mathcal{D}(\widehat{X})$, satisfies*

$$\boxed{\mathcal{D}_c\Big(\widehat{\mathsf{T}}(\widehat{X}, v)\Big) = \mathsf{T}_{\mathcal{G}}(c, v).} \tag{26}$$

Consequently,

$$\mathcal{D}\Big(\widehat{\mathsf{T}}(\widehat{X}, v)\Big) = \mathsf{T}_{\mathcal{G}}\Big(\mathcal{D}(\widehat{X}), v\Big). \tag{27}$$

Proof. Fix the entry origin c. Source-indexed byte simulation proves replay; pointwise summary binding, worklist propagation, and suffix induction prove local-effect execution. Both preserve c through closure, normalization, and finalization. Grouping reuses the same per-source relation. Public exit closure restores the established representation invariants. Union over sources gives the aggregate equation. □

Window and complete-output languages. For normalized C_0, allowed sets $\mathbf{A} = (A_0, \dots, A_{H-1})$, and $Y \in \{N, \mathrm{fac}\}$, define

$$\begin{aligned} C_0^Y &= C_0, \\ C_{i+1}^Y &= \bigcup_{c \in C_i^Y} \bigcup_{v \in A_i} \mathsf{T}^Y(c, v), \\ E_i^Y &= \{(c, v, c') : c \in C_i^Y,\ v \in A_i,\ c' \in \mathsf{T}^Y(c, v)\}. \end{aligned} \tag{28}$$

Here $\mathsf{T}^N = \mathsf{T}_{\mathcal{G}}$ and $\mathsf{T}^{\mathrm{fac}}(c, v) = \mathcal{D}_c(\widehat{\mathsf{T}}(\widehat{X}_i, v))$ for a representation of C_i^{fac}. Stored edges use these source–successor pairs, not Cartesian products of frontiers; graph nodes retain their position. Let $\mathcal{L}^Y_{H,B_H}$ contain the token sequences labeling paths from C_0 to a configuration satisfying the common normalized boundary predicate B_H.

Theorem A.7 (Normalized window-language preservation)**.** *Complete on-demand construction preserves $C_i^{\mathrm{fac}} = C_i^N$, $E_i^{\mathrm{fac}} = E_i^N$, and*

$$\boxed{\mathcal{L}^{\mathrm{fac}}_{H,B_H}(C_0; \mathbf{A}) = \mathcal{L}^N_{H,B_H}(C_0; \mathbf{A}).} \tag{29}$$

Proof. Induct on position. Equal frontiers and Eq. (26) give the same successors for each pair (c, v), hence identical triples (c, v, c') and next frontiers. Output admissibility permits the next query. Effect groups reconstruct these same edges when every admitted token label is retained. Equal labeled paths and the same boundary predicate give equal window languages. □

For a fixed token sequence, normalization commutation also gives $X_H^N = N(X_H^{\mathrm{raw}})$. Thus raw-to-normalized boundary agreement holds whenever $B_H^{\mathrm{raw}}(X) = \overline{B}_H(N(X))$. This extra condition is unnecessary when comparing factorized execution directly with the normalized reference.

Let $\mathcal{L}^Y_{\mathrm{complete}}$ contain finite token sequences reaching complete-output acceptance under semantics Y.

Theorem A.8 (Complete-output language preservation)**.** $\mathcal{L}^N_{\mathrm{complete}} = \mathcal{L}^{\mathrm{raw}}_{\mathrm{complete}}$. *Factorized acceptance agrees with both on completed queries.*

Proof. Every complete accepting configuration satisfies P, so backward closure ensures that every raw accepting path survives normalization. Normalization only deletes alternatives, proving the reverse inclusion. The token theorem then establishes factorized agreement. The same argument preserves the existence of a finite legal completion, but not observations of dead threads. □

These statements preserve logical transitions and token languages, not derivation multiplicities, arbitrary path weights, or resource-truncated results.

A.4 Weighted Inference under Token-Effect Grouping

Fix one denoising iteration and a finite layered window graph with initial configuration c_0, horizon H, terminal predicate B_H, and fixed nonnegative token weights $p_i(v)$. Forbidden tokens have zero weight. All comparisons use the same normalized, sourcewise transition relation $\mathsf{T}_{\mathcal{G}}$ and count each logical edge (i, c, v, c') once. Let Π_q partition $\mathcal{V}$ and assume effect equivalence:

$$\mathsf{T}_{\mathcal{G}}(c, v) = \mathsf{T}_{\mathcal{G}}(c, G), \qquad v \in G \in \Pi_{q(c)}. \tag{30}$$

Positions, endpoints, and group membership are retained. An accepting path π has weight $W(\pi) = \prod_{i=1}^{H} p_i(v_i)$; assume $Z = \sum_{\pi \text{ accepting}} W(\pi) > 0$. The following equalities use exact arithmetic.

Define

$$M_i(G) = \sum_{v \in G} p_i(v), \qquad m_i(G) = \max_{v \in G} p_i(v). \tag{31}$$

For $\oplus \in \{+, \max\}$, let $r_i^+ = M_i$ and $r_i^{\max} = m_i$. Disjoint group membership and Eq. (30) give

$$\begin{aligned} A_i^{\oplus}(c, c') &:= \bigoplus_{v:\, c' \in \mathsf{T}_{\mathcal{G}}(c,v)} p_i(v) \\ &= \bigoplus_{G:\, c' \in \mathsf{T}_{\mathcal{G}}(c,G)} r_i^{\oplus}(G). \end{aligned} \tag{32}$$

Both cases use multiplication to compose edge weights, with empty reductions equal to zero. Thus grouping preserves the entire sum-product or max-product transition operator, not only its support.

Lemma A.9 (Sum-product inference preservation)**.** *Group-sum reduction with conditional token restoration preserves Z, the full token-labeled path distribution, and all token marginals under chain or tree inference using the conditionals below.*

Proof. Equation (32) preserves every forward, backward, and interval message. Set $\beta_H = B_H$ and $\beta_{i-1} = A_i^+ \beta_i$. Chain sampling selects a valid group edge and then a token using

$$\Pr(G, c' \mid c) = \frac{M_i(G)\beta_i(c')}{\beta_{i-1}(c)}, \qquad \Pr(v \mid G, c, c') = \frac{p_i(v)}{M_i(G)}. \tag{33}$$

Their product is $p_i(v)\beta_i(c')/\beta_{i-1}(c)$, exactly the expanded edge conditional. Multiplication along a path cancels the messages and gives $W(\pi)/Z$.

For tree inference, write $K_{\ell:r} = A_{\ell+1}^+ \cdots A_r^+$. Sample the terminal state with probability $K_{0:H}(c_0, b)B_H(b)/Z$ and an interval midpoint by

$$\Pr(C_k = z \mid C_\ell = a, C_r = b) = \frac{K_{\ell:k}(a, z)K_{k:r}(z, b)}{K_{\ell:r}(a, b)}. \tag{34}$$

At leaf i with endpoints a, b, choose G with probability $M_i(G)\mathbf{1}[b \in \mathsf{T}_{\mathcal{G}}(a, G)]/A_i^+(a, b)$, then restore v with probability $p_i(v)/M_i(G)$. Associativity preserves the interval operators; cancellation of the interval and leaf normalizers again gives $W(\pi)/Z$. Only positive-mass events are sampled. Equality of full path distributions implies equality of all token marginals. □

Theorem A.10 (Equivalence to Dang–Ermon sampling)**.** *Consider Algorithm 1 of Dang & Ermon (2026), expressed in the equivalent state-space formulation of their Appendix A.2. Use the same finite token-labeled graph, initial and terminal factors, and logical edge-counting convention as above, with per-token factors*

$$g_i(c, v, c') = p_i(v)\mathbf{1}[c' \in \mathsf{T}_{\mathcal{G}}(c, v)]. \tag{35}$$

Under Lemma A.9, the grouped tree sampler has the same partition function and output-token distribution as this instance of Algorithm 1.

Proof. Their state-space transition matrix is

$$\mathsf{M}_i(c, c') = \sum_v g_i(c, v, c') = A_i^+(c, c'). \tag{36}$$

Equation (32) preserves this matrix under group-sum reduction. Thus all bottom-up interval messages, the root boundary distribution, and the conditional midpoint distributions are unchanged.

At a leaf with endpoints c, c', selecting a group and restoring $v \in G$ gives

$$\frac{M_i(G)\mathbf{1}[c' \in \mathsf{T}_{\mathcal{G}}(c, G)]}{A_i^+(c, c')} \frac{p_i(v)}{M_i(G)} = \frac{g_i(c, v, c')}{A_i^+(c, c')}, \tag{37}$$

which is the same leaf token conditional. Only positive-mass events are sampled. Induction over the sampling tree therefore gives the same state-space joint distribution and hence the same output-token distribution. The identical interval operators and boundary factors also give the same partition function. □

Lemma A.11 (Max-product inference preservation)**.** *Group-max reduction preserves the Viterbi optimum under chain or tree composition. All maximizing token sequences can be recovered when all tied token and state-path choices are retained.*

Proof. Equation (32) preserves every max-product message. For a fixed grouped state path,

$$\max_{v_i \in G_i} \prod_{i=1}^{H} p_i(v_i) = \prod_{i=1}^{H} m_i(G_i). \tag{38}$$

Every token replacement is feasible by effect equivalence. Hence each optimal grouped path lifts to an optimal token path by choosing group maximizers, and every positive-weight expanded optimum is recovered this way. Retaining all ties preserves the full maximizing sequence set. A single argmax and backpointer return one optimum; identical selected outputs additionally require the same tie-breaking rule. □

Parsing ambiguity and the probability measure. Let $n(y)$ be the number of accepting paths labeled by token sequence y in the chosen token-level graph, and let $w(y) = \prod_i p_i(y_i)$. Summing path weights gives

$$Q_{\mathrm{path}}(y) = \frac{n(y)w(y)}{\sum_z n(z)w(z)}, \qquad Q_{\mathrm{seq}}(y) = \frac{\mathbf{1}[n(y) > 0]w(y)}{\sum_{z:n(z)>0} w(z)}. \tag{39}$$

The second distribution counts each legal sequence once. Cancelling positive $w(y)$ shows that the two distributions coincide if and only if $n(y)$ is constant on the positive-weight legal support; unambiguity is sufficient. A constant $n(y) = k$ gives the same normalized distribution but scales its partition function by k. For example, equal-weight sequences with path counts two and one have path probabilities $2/3$ and $1/3$, rather than $1/2$ each. Thus language equivalence alone does not establish sum-product equivalence across different parser representations. Grouping preserves the chosen graph's measure, not an assumed absence of parsing ambiguity.

These results require the stated weighted conditionals; greedy selection by group mass is generally not equivalent. Extra token-dependent scores require corresponding weighted reductions and token-restoration rules. Distributional equality does not imply identical outputs under the same random seed or floating-point reduction order.

A.5 Complexity Analysis

We separate compilation and graph construction from model inference, token-probability processing, and grammar dynamic programming.

Context expansion. Let Q_{loc} be the local grammar states, $g > 1$ the number of possible continuation-frame symbols, $d_{\max}$ the stack-depth bound, and B the maximum number of remaining context assignments per local-state–stack pair. Then

$$S_{d_{\max}} \le |Q_{\mathrm{loc}}| B \sum_{\ell=0}^{d_{\max}} g^{\ell} = O(|Q_{\mathrm{loc}}| B g^{d_{\max}}). \tag{40}$$

The stack factor is $d_{\max} + 1$ for $g = 1$. The bound need not be attained. For bounded-depth brackets with g types, however, each opening-only prefix has a distinct matching completion. A minimal complete DFA needs one state per stack plus a sink:

$$S_{\mathrm{DFA}}^{\min} = 1 + \sum_{\ell=0}^{d_{\max}} g^{\ell} = \Theta(g^{d_{\max}}). \tag{41}$$

This is a worst-case DFA representation gap, not a universal runtime bound.

Entry and instantiation counts. A dense context-expanded table has $S_{d_{\max}}|\mathcal{V}|$ entries; factorized compilation has at most $|Q_{\mathrm{loc}}||\mathcal{V}|$ local-state–token entries. These are counts, not constant-time

operations. For reachable frontiers $\mathcal{C}_{i-1}$, define

$$\begin{aligned} R_t &= \sum_{i=1}^{H} |\mathcal{C}_{i-1}|, \\ \Pi_{q,i}^t &= \{G \in \Pi_q : G \cap \mathcal{V}_i^t \neq \varnothing\}, \qquad K = \max_q |\Pi_q|, \\ N_{\text{inst}} &= \sum_{i=1}^{H} \sum_{c \in \mathcal{C}_{i-1}} |\Pi_{q(c),i}^t| \leq \min\{N_{\text{query}}, R_t K\}, \end{aligned} \tag{42}$$

where $N_{\text{query}} = \sum_i \sum_{c \in \mathcal{C}_{i-1}} |\mathcal{V}_i^t|$. Repeated configurations are counted per position. With full-vocabulary candidates, the ungrouped count is $R_t|\mathcal{V}|$.

Construction cost. Let T_{setup} include shared schema lowering and static-distance analysis. Let $C(q, v)$ include a cold entry's queue processing, improvement propagation, and effect/key construction. Let W_j include a top-level instantiation's binding, all residual descendants, closures, suffix compilation, and successor creation. For cold pairs $\mathcal{P}_{\text{cold}} \subseteq Q_{\text{loc}} \times \mathcal{V}$, actual top-level queries $|\mathcal{J}_{\text{inst}}| \leq N_{\text{inst}}$, and additional indexing and label/graph work A_t,

$$T_{\text{construct}} = O\left(T_{\text{setup}} + \sum_{(q,v) \in \mathcal{P}_{\text{cold}}} C(q, v) + \sum_{j \in \mathcal{J}_{\text{inst}}} W_j + A_t\right). \tag{43}$$

Each shared construction is charged once; recursive work included in W_j is not counted again. If $C(q, v) \leq \tau_c$ and $W_j \leq \tau_w$, then

$$T_{\text{construct}} = O(T_{\text{setup}} + \tau_c |Q_{\text{loc}}||\mathcal{V}| + \tau_w R_t K + A_t). \tag{44}$$

Neither per-entry cost is assumed constant for arbitrary grammars. Grouping also does not remove the work needed to discover effects or emit individual token labels.

Scope. For an L-byte entry, if every depth prefix is at least $-\kappa L$, then $-\kappa L \leq m \leq 0$ and each complete key admits at most $\kappa L + 1$ retained values, including its first admission. This bounds re-expansion per key, not the number or size of keys. Similarly, residual progress bounds a recursive chain by L strict decreases, not the total work of its branching tree. Local stacks and pending programs may still grow, R_t may approach $HS_{d_{\max}}$, and K may approach $|\mathcal{V}|$. Cold assets are reusable across requests with the same grammar and binding scope; graph reuse additionally requires compatible candidate sets and boundary/protocol conditions. The analysis therefore characterizes avoided replication without claiming an unconditional exponential decoding speedup.

B Additional Experimental Details

This section provides additional implementation and evaluation settings for the experiments in Section 5.

B.1 Implementation and Decoding Settings

Inference uses NVIDIA H20-3e GPUs with approximately 140 GiB of visible memory per device. Both models run in BF16 with model CUDA Graphs enabled. LLaDA2.2 (Bie et al., 2026) uses SGLang (Zheng et al., 2024) with tensor parallelism across four GPUs, while DiffusionGemma (DiffusionGemma Team et al., 2026) uses vLLM (Kwon et al., 2023) 0.28.0 with one GPU per model instance. DConstrain implements cold and warm compilation in C++ and grammar inference in CUDA.

Cold-compilation experiments run on an Intel Xeon Platinum 8575C CPU. Each schema compilation is single-threaded, with up to 64 schemas compiled concurrently. Cold timing starts after tokenizer and service initialization and ends when the reusable compiled artifact is ready for online use. Warm compilation and model inference are excluded from cold-compilation time.

LLaDA2.2. The adaptive runtime comparison uses 32-token physical blocks, with the first block's available positions determined by the prompt suffix. Generation is greedy with seed zero. MASK-to-token updates use a confidence threshold of 0.5, falling back to the highest-confidence MASK position when none qualifies. The token-to-token threshold is 0.0; DELETE and SPLIT are disabled. A block is committed after all MASK positions are filled and token assignments stabilize. Generation terminates at an accepted EOS or the 2,048-token output limit, with bounded update/refinement rounds; budget exhaustion is counted as failure.

DiffusionGemma. Generation uses 256-token canvases, seed zero, and the native entropy-bound sampler. The entropy budget is 0.1, and temperature decreases linearly from 0.8 to 0.4. Gumbel-max candidates are retained at entropy-selected positions while the remaining positions are re-noised, with native soft self-conditioning preserved. Generation uses at most 48 denoising iterations per canvas and terminates at EOS/EOT or the 2,048-token output limit.

B.2 Compilation and Evaluation Settings

The BFCL pool contains 2,495 cases, obtained by screening all `required` cases in BFCL and excluding cases whose schemas admit no valid instance. The JSONSchemaBench (JSB) pool contains all 5,754 training schemas, and the ComplexFuncBench (CFB) pool contains all 1,000 benchmark cases.

The main experiments and ablation study in the main text use the full corresponding pools. Unless otherwise specified, all other analyses and appendix experiments use fixed subsets of 300 BFCL cases and 300 JSB schemas, and the full 1,000-case CFB pool, as applicable. Runtime comparisons are computed over cases successfully compiled by both compared methods within the designated pool. Failed compilation cases are not replaced.

For cold compilation, the nominal timeout is 180 s per schema. Success rates include all attempted cases, while compilation latency statistics use each method's successfully compiled cases. Reported speedups divide the baseline's arithmetic mean latency by DConstrain's over paired successful cases. Timeouts and failures affect success rates but are excluded from these latency ratios.

Schema validity is evaluated independently of solver acceptance. BFCL uses the official task evaluator, with joint correctness additionally requiring schema validity. JSONSchemaBench validates outputs against the original schema. Invalid outputs, timeouts, and exhausted decoding budgets are counted as failures.

Runtime timing. End-to-end latency is measured by wall-clock time at the request level, while GPU grammar-inference stages are measured with CUDA events. Warm compilation measures the request-local graph-construction work; These intervals are therefore not additive. Cold compilation and artifact loading are excluded from runtime latency. Runtime latency statistics in Table 2 are arithmetic means over the paired cases successfully compiled by both compared methods.

B.3 Engineering Optimizations

We complement DConstrain's compilation design with engineering optimizations that reduce execution overhead and graph-storage cost.

Native compilation pipeline. Cold and warm compilation execute in a native C++ pipeline, reducing interpreter overhead in grammar processing and transition instantiation. Reusable schema artifacts are stored in packed native arrays, avoiding per-edge Python objects and their allocation overhead.

Compact sparse graph storage. We store canvas-specific grammar graphs in compressed sparse row (CSR) format, using row offsets and contiguous edge arrays. This representation stores only existing transitions rather than allocating dense graph structures. CUDA grammar-inference kernels consume the CSR representation directly, connecting compact graph construction with GPU execution.

DINGO-Opt adopts the same C++ pipeline and CSR representation while retaining finite-state compilation. Direct-PDA-Opt uses edge merging, shared token-membership storage, and grouped GPU reductions while interpreting reached parser configurations on demand, without compiling symbolic transition programs. Its + TG variant additionally applies conservative token grouping before interpretation (Appendix D.1).

Asynchronous execution. Native CPU workers prepare score-independent graph data while the model performs its forward computation. The scheduling interface supports both eager execution and model CUDA Graph replay. Data transfer and readiness tracking connect warm compilation to GPU inference, which begins when both graph data and model scores are available. Score-dependent operations remain on the inference path. This overlap limits exposed warm-compilation latency, while workspace reuse reduces repeated setup costs. We measure end-to-end latency directly rather than summing overlapping components, and report compilation and exposed costs in Section 5.1.

C Additional Analyses of Constraint Semantics and Runtime

C.1 Any-Order Object Decoding

A subtle but important requirement of structured-output decoding is that the compiler should not introduce restrictions beyond those required by the supported schema semantics. For JSON objects, schemas generally specify which properties are allowed or required without prescribing a unique serialization order. For schemas admitting at least two simultaneously present properties whose relative order is unconstrained, fixing a single serialization order yields

$$\mathcal{L}_{\text{fixed}} \subsetneq \mathcal{L}_{\text{schema}}. \tag{45}$$

Although every output in $\mathcal{L}_{\text{fixed}}$ remains schema-valid, valid serializations outside this subset are artificially removed by the decoding system. This distinction is particularly important for DLMs. Unlike autoregressive generation, where fields are generated sequentially from a single realized prefix, DLM decoding jointly reasons over multiple unresolved positions. The model may therefore assign high probability to different valid field orders across the decoding canvas. Imposing a fixed order can eliminate high-probability joint assignments and alter the semantic decision selected under constrained inference. Consequently, achieving 100% schema validity alone does not guarantee that constrained decoding preserves the schema-permitted serialization choices available to the model.

Exact property-order support. DConstrain avoids introducing this implementation-induced ordering restriction. Instead of compiling an object schema into a single predetermined property sequence, the compiler represents the remaining property choices within the parsing environment, allowing schema-valid field orders to be realized without compiling each complete permutation as an independent schema. These alternatives are exposed to the same joint inference procedure used elsewhere in DConstrain. Within the supported grammar fragment, the resulting decoder therefore enforces the schema constraints while preserving arbitrary schema-permitted parameter-field orders. This capability is integrated into the same pushdown and context-parameterized compilation framework described in Section 4.

Impact on task accuracy. Table 5 isolates the effect of property ordering. We compare the unconstrained native model, a prompt-matched unconstrained baseline, fixed-order constrained decoding, and any-order constrained decoding. DConstrain and existing automata-based compilers ultimately expose the same logical token-labeled successor relation required by downstream joint inference. DConstrain therefore changes how this transition structure is compiled and instantiated, not the inference interface itself: the resulting canvas-specific grammar graph can be consumed by the same downstream solvers, including sequential Viterbi and the inference algorithm of Dang & Ermon (2026). In this experiment, we use DConstrain to compile both the fixed-order and any-order constraints while keeping the downstream solver unchanged, so that the comparison isolates the effect of the permitted property-order language rather than differences in compiler implementation. Native + Prompt uses the same task-formatting instruction as the corresponding constrained run. For CFB, all constrained and prompt-matched runs use the same JSON-only instruction, so the observed differences cannot be attributed simply to prompting.

Table 5: Schema validity and task accuracy (%) under fixed- and arbitrary-order constrained decoding. BFCL results use the full set of 2,495 schemas. Accuracy deltas are relative to Native + Prompt. CFB reports schema-gated semantic accuracy over 3,260 tool-call turns from 1,000 dialogues.

	BFCL		CFB	
Method	Validity ↑	Accuracy ↑	Validity ↑	Accuracy ↑
LLaDA2.2				
Native	93.1	79.4	82.4	39.1
Native + Prompt	95.4	81.0	84.1	39.5
Fixed-order + Viterbi	100.0	$80.4_{\downarrow 0.7}$	100.0	$34.5_{\downarrow 4.9}$
Fixed-order + DANG	100.0	80.0	100.0	33.8
Any-order + Viterbi	100.0	$\mathbf{82.8}_{\uparrow 1.7}$	100.0	$\mathbf{39.8}_{\uparrow 0.3}$
Any-order + DANG	100.0	81.7	100.0	39.7
DiffusionGemma				
Native	69.2	56.7	48.6	21.7
Native + Prompt	62.4	59.8	51.0	23.0
Fixed-order + Viterbi	100.0	$35.8_{\downarrow 23.9}$	100.0	$16.0_{\downarrow 7.0}$
Fixed-order + DANG	100.0	35.1	100.0	15.7
Any-order + Viterbi	100.0	$\mathbf{73.8}_{\uparrow 14.0}$	100.0	$\mathbf{27.5}_{\uparrow 4.5}$
Any-order + DANG	100.0	70.9	100.0	25.3

The results reveal a failure mode that schema-validity metrics alone do not capture. With LLaDA2.2 on BFCL, fixed-order constrained decoding reaches 100% validity but reduces accuracy from 81.0% to 80.4%. The effect is larger on CFB, where accuracy falls from 39.5% to 34.5% despite perfect schema validity. Allowing arbitrary schema-valid property orders removes this degradation: Any-order + Viterbi reaches 82.8% accuracy on BFCL and 39.8% on CFB while retaining 100% schema validity, corresponding to improvements of 1.7 and 0.3 percentage points over the prompt-matched unconstrained baselines.

The effect is even more pronounced for DiffusionGemma on BFCL. Fixed-order constrained decoding reaches 100% validity but reduces task accuracy from 59.8% for the prompt-matched native baseline to 35.8%. The enforced schema order differs from the model template's alphabetical order on 2,008 of 2,495 cases, illustrating how a compiler-imposed serialization order can conflict with the model's preferred realization. Any-order decoding instead reaches 73.8% accuracy while retaining 100% schema validity.

These results distinguish output validity from preservation of schema-permitted serialization choices. A constrained decoder may generate only schema-valid outputs while still excluding valid property orders that the schema itself permits. For DLM joint inference, such over-constraining can alter which high-probability assignments remain available and thereby affect downstream task accuracy. DConstrain's any-order support therefore removes an implementation-induced serialization restriction rather than merely making the constraint more permissive.

C.2 Warm Compilation Scaling and Runtime Overlap

DConstrain avoids materializing context-expanded token transitions during cold compilation, but exact joint inference still requires the runtime contexts that are reachable from the current decoding canvas. Warm compilation therefore does not eliminate context-specific successor construction; instead, it instantiates successors on demand from shared transition programs. This section examines whether the resulting deferred work becomes a runtime bottleneck as parsing-environment and stack complexity increase, and how this work is overlapped with model computation during execution.

Scaling with runtime-context complexity. Figure 4 complements the cold-compilation scaling analysis in Figure 3 by measuring the corresponding online cost. We consider the two sources of runtime-context growth targeted by the factorized representation. For arbitrary-order objects, in-

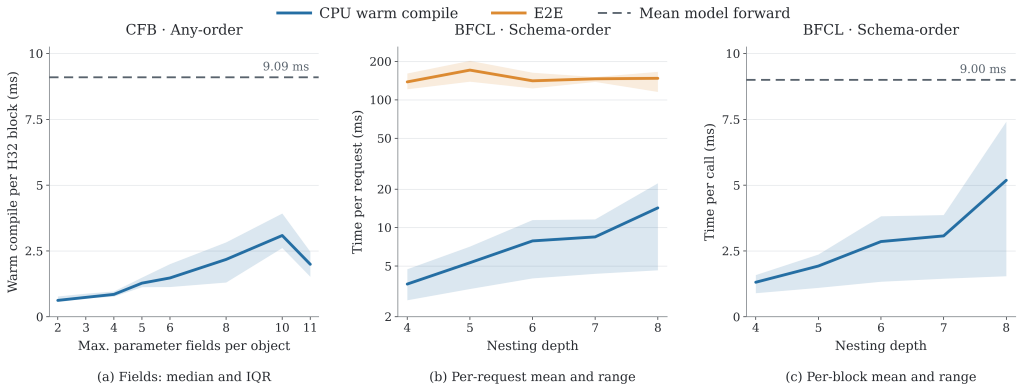


Figure 4: Warm-compilation scaling with runtime-context complexity. (a) CPU warm-compilation time per 32-token block on CFB with arbitrary property order, grouped by the maximum number of parameter fields per object; the line and band report the median and interquartile range. (b) Per-request end-to-end latency and accumulated CPU warm-compilation time on BFCL as nesting depth increases. (c) Mean CPU warm-compilation time per call over the same depth sweep. Bands in (b,c) show the observed range across requests. Dashed lines denote the corresponding mean model-forward time. The results characterize the online work deferred by factorized compilation rather than claiming that reachable runtime contexts are eliminated.

creasing the number of parameter fields increases the number of possible parsing-environment configurations. For nested structures, increasing nesting depth increases the diversity of continuation-stack contexts that may be reached during graph construction.

On the CFB any-order sweep in Figure 4(a), median CPU warm-compilation time per 32-token block increases gradually with the maximum number of parameter fields per object. Even at the largest field counts, the median remains well below the corresponding mean model-forward time of 9.09 ms. Thus, supporting arbitrary schema-valid property order introduces additional online work, but the measured warm-compilation cost remains small relative to the model computation with which it can overlap.

Figures 4(b) and 4(c) examine nesting depth on BFCL. As depth increases from four to eight, warm-compilation work increases, consistent with a larger set of reachable stack contexts. The increase is visible both in the total warm-compilation work accumulated within a request and in the mean cost of an individual warm-compilation call. At depth eight, the mean per-call CPU cost remains below the approximately 9 ms mean model-forward time, while end-to-end request latency remains comparatively stable across the depth sweep. These measurements show that factorization does not remove the runtime contexts required by exact joint inference, but avoids turning their existence into the substantially larger cold-compilation expansion observed for context-expanded finite-state representations.

Warm compilation during asynchronous execution. The scaling experiments above characterize how online work changes with specific sources of context complexity. Figure 2 in the main text reports warm-compilation timings from actual asynchronous execution across the broader evaluation workloads. Native CPU workers prepare score-independent graph data concurrently with GPU model forward computation. Grammar inference starts when both the graph and model scores are ready, so only warm compilation remaining at that point is exposed on the decoding critical path. The figure sorts the measured CPU warm-compilation times and shows the corresponding mean model-forward duration as a reference; the percentage annotations summarize the fraction of warm-compilation calls below that duration.

For LLaDA2.2 on CFB, arbitrary property order increases warm-compilation cost relative to schema order, as expected from the additional parsing-environment alternatives, while the distribution remains below the corresponding mean forward time over the plotted samples. On JSON-SchemaBench, which contains substantially more heterogeneous schemas, the distribution has a

longer tail: 97.17% of warm-compilation calls remain below the 8.90 ms mean model-forward duration, while the largest observed call reaches 34.61 ms.

The same pattern holds with the substantially larger 256-token canvases of DiffusionGemma. On BFCL and CFB with arbitrary property order, 99.47% and 94.20% of warm-compilation calls, respectively, are below the corresponding mean model-forward durations of 26.4 and 26.7 ms. On JSONSchemaBench, 99.3% are below the 25.64 ms mean forward duration, with a maximum observed warm-compilation time of 30.32 ms. The tail therefore occasionally exceeds a model forward, but such cases are rare in these workloads.

Implications for deferred instantiation. Together, Figures 4 and 2 clarify the trade-off introduced by two-stage compilation. DConstrain does not eliminate the runtime contexts required to construct the exact token-labeled canvas-specific grammar graph. Instead, it avoids materializing independent copies of shared transition logic for those contexts during cold compilation and performs context-specific instantiation only when the corresponding configurations become reachable. The resulting online work grows with runtime-context complexity, as expected, but remains modest relative to model-forward computation over the evaluated regimes and is overlapped with model execution by the asynchronous schedule. The request-level means in Table 2 account for this overlap directly.

D Additional Experimental Results

These experiments complement the main runtime and compilation results in Section 5. We examine direct pushdown interpretation, tree-parallelized automata inference, verification-based baselines, compiled representations, and compilation coverage.

D.1 Comparison with Direct Pushdown Interpretation

The ablations in Section 5.2 vary reuse within DConstrain. Here we additionally evaluate a baseline that computes $(c, v) \mapsto \mathsf{T}_{\mathcal{G}}(c, v)$ by directly interpreting the pushdown parser at reached runtime configurations, without materializing a context-expanded DFA or compiling symbolic transition programs. This comparison tests whether on-demand interpretation alone can provide efficient successor construction.

Baseline variants. *Direct-PDA-Opt* already merges output edges with common destinations and uses shared token-membership storage and grouped GPU reductions. *Direct-PDA-Opt + TG* additionally groups tokens *before interpretation*: bytes are equivalent only when their raw transitions agree at every local RTN node, and token-trie paths with identical byte-class sequences are merged. It also reuses unions of token memberships. Calls, returns, and environment updates still execute in the interpreter, and all original token identities and model scores are retained. This conservative grouping uses no symbolic transition programs and differs from DConstrain's local token-effect grouping.

Evaluation setting. We evaluate all variants on the same fixed set of 300 BFCL cases, with LLaDA2.2, BF16, tensor parallelism of four, batch size one, and 32-token blocks. All variants use the same name-first tool-call format, arbitrary parameter-field order, a nesting-depth limit of four, prompt, and MAP decoding. These controlled runs execute graph construction and model forward serially, with a fresh runtime per request. E2E spans the first prefill through runtime and parser cleanup after generation; cold preparation is excluded. Figure 5 uses the original paired runs. Figure 6 reruns DConstrain and the grouped baseline in the same service on the same 300 cases. All 300 cases produce identical output tokens across variants and satisfy their schemas.

Runtime comparison. Pre-interpretation grouping substantially reduces the direct interpreter's online cost, but a large gap remains. In the grouped paired run, mean E2E is 878.10 ms for Direct-PDA-Opt + TG and 136.63 ms for DConstrain, a $6.43\times$ speedup. The warm-compilation curves show that repeated runtime interpretation remains expensive even after conservative grouping. Together with the compiler ablations, these results support compiling and reusing transition programs as a contribution beyond simply avoiding DFA expansion.

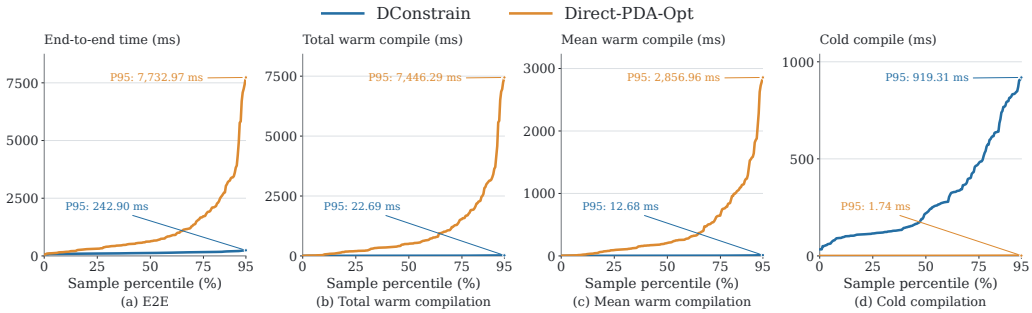


Figure 5: DConstrain versus Direct-PDA-Opt without pre-interpretation token grouping on 300 BFCL cases with LLaDA2.2. Panels report (a) complete-request E2E, (b) accumulated warm-compilation time per request, (c) each request's mean time per warm-compilation call, and (d) cold-compilation time. Each curve shows its latency percentiles from P0 to P95; the upper 5% is omitted without rescaling the percentile axis.

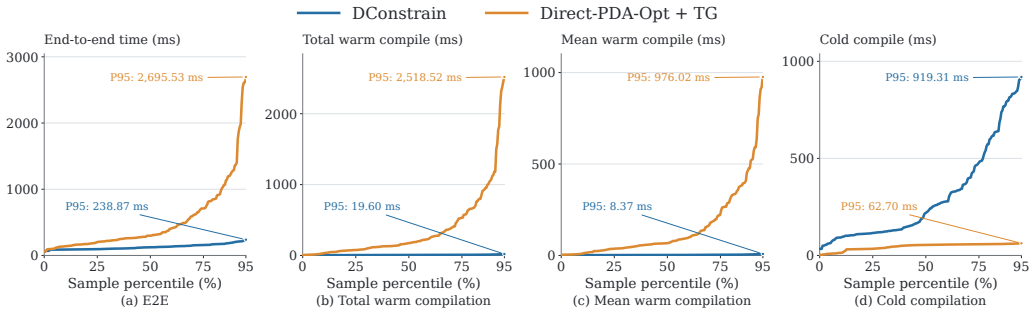


Figure 6: DConstrain versus Direct-PDA-Opt + TG, which adds conservative pre-interpretation token grouping and membership reuse. Both methods are rerun on the same 300 BFCL cases. Panel definitions and the P0–P95 display follow Figure 5.

D.2 Tree-Parallelized Automata Inference

DConstrain changes how the canvas-specific grammar graph is constructed while preserving the token-labeled successor relation consumed by downstream automata-based inference. It is therefore solver-agnostic rather than specialized to the Viterbi solver used in our main experiments. We verify this compatibility by applying both Viterbi inference and the inference algorithm of Dang & Ermon (2026) directly to DConstrain graphs, using both sequential and tree-parallelized implementations.

We further examine whether tree parallelism can reduce the solver cost underlying Table 2. All four solver variants are evaluated on the same 300 BFCL cases using both DINGO and DConstrain graph representations. Table 6 reports results for two cases near the 25th and 95th percentiles of DINGO graph node count, denoted P25 and P95. The selected cases and inference objectives are matched across graph representations.

We consider the log-depth tree construction proposed by Dang and Ermon (Dang & Ermon, 2026) and apply the same reduction to MAP inference for a direct comparison with the Viterbi-style decoding used by DINGO (Suresh et al., 2025). Let L denote the decoding-window length, S the automaton states, and E the grammar-transition structure. Conventional Viterbi inference performs position-wise dynamic programming, where state scores at position i depend on those at position $i-1$, yielding a sequential dependency across the decoding window. Tree-based inference instead composes adjacent transition operators in a binary reduction tree, reducing the sequential depth from $O(L)$ to $O(\log L)$. For MAP inference, the same construction can be performed over the max-product semiring (Goodman, 1999), preserving the Viterbi objective (Viterbi, 1967) while exposing tree-level parallelism.

Table 6: Solver latency (ms) and peak solver GPU memory (GiB). All four solver variants are evaluated on 300 BFCL cases; the table displays cases near the 25th and 95th percentiles of DINGO graph node count. P25/P95 refer to graph size, not latency quantiles. "Ours" denotes DConstrain; "seq." and "tree" denote sequential and tree-parallelized execution. DANG denotes the inference method of Dang & Ermon (2026).

		P25 BFCL schema		P95 BFCL schema	
Graph	Method	Time (ms) ↓	Mem. (GiB) ↓	Time (ms) ↓	Mem. (GiB) ↓
Ours	Viterbi seq.	**3.90**	**0.01**	**30.85**	**0.03**
	DANG seq.	4.70	0.01	48.97	0.09
	Viterbi tree	30.80	1.14	7 824.19	12.55
	DANG tree	60.70	1.56	4 344.17	12.30
DINGO	Viterbi seq.	39.80	0.01	425.08	0.08
	Viterbi tree	7.13	0.21	880.38	12.84
	DANG seq.	331.32	0.02	13 536.26	0.09
	DANG tree	11.40	0.29	3 577.13	17.63

This reduction trades sequential depth for substantially more computation and intermediate state. Under the state-space formulation, sequential message passing requires

$$O\big(L\left(|S|^2 + |E||\mathcal{V}|\right)\big) \tag{46}$$

work, whereas the tree construction requires

$$O\big(L\left(|S|^3 + |E||\mathcal{V}|\right)\big) \tag{47}$$

work while reducing the critical path to $O(\log L)$. Dang and Ermon characterize this trade-off as an $|S|$-fold increase in message-passing FLOPs in exchange for greater GPU parallelism. The memory cost can also increase substantially: each interval in the reduction tree materializes a state-to-state transition summary of size $O(|S|^2)$. Retaining the summaries required by the subsequent top-down pass therefore requires $O(L|S|^2)$ intermediate storage, compared with $O(L|S|)$ backpointer storage for conventional Viterbi reconstruction, or $O(|S|)$ score storage when reconstruction is unnecessary. This quadratic dependence on the automaton state space can become expensive for large structured-output graphs.

Table 6 shows that the benefit of tree parallelism depends strongly on graph size. On the P25 DINGO graph, tree-parallelized Viterbi reduces latency from $39.8\,\mathrm{ms}$ to $7.13\,\mathrm{ms}$, a $5.58\times$ speedup, but remains slower than sequential Viterbi on the DConstrain graph at $3.90\,\mathrm{ms}$. On the P95 DINGO graph, tree-parallelized Viterbi instead increases latency from $425.08\,\mathrm{ms}$ to $880.38\,\mathrm{ms}$, while sequential Viterbi on the DConstrain graph requires only $30.85\,\mathrm{ms}$. Peak solver GPU memory on the DINGO graph increases from $0.01\,\mathrm{GiB}$ to $0.21\,\mathrm{GiB}$ at P25 and from $0.08\,\mathrm{GiB}$ to $12.84\,\mathrm{GiB}$ at P95, approximately $21\times$ and $167\times$, respectively. Tree parallelism also increases both latency and memory for Viterbi on the DConstrain graph in these two cases. These measurements illustrate that reducing sequential depth does not guarantee lower solver cost: as the graph grows, the additional dense intermediate computation and storage can dominate the available parallelism.

D.3 Comparison with Verification-Based Baselines

Table 7 compares DConstrain with LAVE (Zhang et al., 2026), Constrained Diffusion (Mündler et al., 2026), and EPIC (Jin & Han, 2026), representing verification-based alternatives to automata-based joint inference. This comparison contrasts constrained-decoding approaches rather than isolating compiler implementation differences. DConstrain achieves 100% schema validity and the lowest mean end-to-end latency in all four model–dataset settings. DConstrain provides mean-latency speedups of 4.91–$26.05\times$ over LAVE, 12.57–$75.50\times$ over Constrained Diffusion, and 8.34–$45.58\times$ over EPIC. Mean latency and validity include all attempted requests, with invalid outputs, timeouts, and exhausted decoding budgets retained in the denominator. These verification-based results complement the mean-latency comparison with automata-based decoding in Table 2.

Table 7: Mean end-to-end latency (s) and schema validity (%) for LAVE (Zhang et al., 2026), Constrained Diffusion (Mündler et al., 2026), EPIC (Jin & Han, 2026), and DConstrain on BFCL and JSONSchemaBench (JSB), with LLaDA2.2 and DiffusionGemma. Validity includes failures in the denominator. The mean-latency comparison with DINGO-Opt is reported in Table 2; see Appendix B.2 for timing definitions.

	BFCL		JSB	
Method	E2E (s) ↓	Schema (%) ↑	E2E (s) ↓	Schema (%) ↑
LLaDA2.2				
LAVE	3.13	91.7	4.44	65.3
Constrained Diffusion	9.06	36.3	10.90	11.3
EPIC	5.47	35.3	7.25	12.0
DConstrain (ours)	**0.12**	**100.0**	**0.73**	**100.0**
DiffusionGemma				
LAVE	11.08	80.7	18.48	52.0
Constrained Diffusion	28.34	23.3	47.34	6.3
EPIC	22.45	23.7	31.40	7.0
DConstrain (ours)	**2.19**	**100.0**	**3.77**	**100.0**

Validity, completion, and rejection cost. LAVE and Constrained Diffusion verify whether proposed updates remain completable under a supplied grammar (Zhang et al., 2026; Mündler et al., 2026). EPIC (Jin & Han, 2026) reduces verification overhead through lexing memoization, Earley-style graph validation, and parallel commitment of candidate subsets that pass exact CFG verification. With sound verification, completed outputs satisfy that grammar; original-schema validity additionally requires faithful schema-to-grammar translation. This guarantee does not ensure completion within a finite token or time budget. Constrained Diffusion permits arbitrary-length infillings in its feasibility check (Mündler et al., 2026), while LAVE reports length-limit failures and acknowledges that finite lookahead can reject feasible proposals (Zhang et al., 2026). When acceptable updates or completion witnesses are difficult to find, repeated proposals, verification, and recovery add online work. Aggregate failure rates alone cannot distinguish schema-translation errors, verification errors, and budget exhaustion.

Feasibility versus joint inference. A feasibility decision establishes the existence of a valid continuation, but does not quantify its probability mass or identify the highest-scoring joint assignment. Verifying individual proposals or selected batches does not by itself establish fixed-canvas joint MAP optimality or an exact weighted path distribution. EPIC also provides witness-based recovery after exhausting its resampling budget; this fallback produces a grammar-valid completion but is not sampled from the model distribution (Jin & Han, 2026). This limitation is not inherent to rejection sampling: exact rejection of complete samples from a fixed proposal recovers that proposal conditioned on acceptance. DConstrain's canvas-specific grammar graph instead supports downstream max-product or sum-product inference over compatible assignments under the current token weights. Token-effect grouping preserves the ungrouped graph's Viterbi optimum and weighted path distribution (Appendix A.4). These guarantees concern the specified graph and fixed-canvas objective, rather than the distribution of the entire iterative diffusion process.

D.4 Compiled Representations with DiffusionGemma

Table 8 complements the LLaDA2.2 results in Table 4. DConstrain's median recorded memory footprint is 4.10 MB on BFCL, 4.23 MB on JSONSchemaBench, and 4.57 MB on ComplexFuncBench; the corresponding P95 values are 5.39 MB, 28.41 MB, and 5.09 MB. State counts use the three-method successful intersection, while edge and memory quantiles use each method's successful cases.

To complement the any-order compilation results in Table 1, Table 9 compares the compiled representations produced by DINGO-Opt and DConstrain under the native tool-call protocols. All representation statistics are computed on the same successful intersection within each setting. Because

Table 8: Compiled representations with the DiffusionGemma tokenizer on fixed sets of 300 BFCL cases, 300 JSONSchemaBench schemas, and 1,000 ComplexFuncBench cases. Edges are reported in thousands; edge and memory columns show P50 and P95. Mean state counts use the three-method successful intersection (258, 202, and 421 cases, respectively); edge and memory quantiles use each method's successful cases. States denote automaton states for baselines and local RTN nodes for DConstrain; edges are token-level versus symbolic. ‡Memory measures baseline edge-array payloads versus DConstrain's recorded native-snapshot payloads, in decimal MB, rather than total process memory.

Compiled representation and memory footprint

Dataset	Method	States ↓	Edges (10^3) ↓		Memory (MB)‡ ↓	
		Mean	P50	P95	P50	P95
BFCL	DINGO	235.21	884.93	2 070.61	42.49	99.46
	NFA	240.21	593.88	1 774.72	28.53	85.25
	DConstrain	**208.61**	**0.76**	**4.00**	**4.10**	**5.39**
JSONSchemaBench	DINGO	187.28	591.32	2 066.61	28.40	99.24
	NFA	223.56	596.35	3 001.78	28.65	144.16
	DConstrain	**169.92**	**0.85**	**51.80**	**4.23**	**28.41**
ComplexFuncBench	DINGO	460.21	1 776.40	3 252.16	85.32	156.17
	NFA	465.21	2 365.89	3 842.17	113.62	184.51
	DConstrain	**424.23**	**2.35**	**3.70**	**4.57**	**5.09**

Table 9: Compiled representations under native tool-call protocols with arbitrary parameter-field order. Statistics use the common successful intersection within each setting. States denote DFA states for DINGO-Opt and local RTN nodes for DConstrain; edges denote token-level and symbolic transitions, respectively. DConstrain successfully compiles every evaluated case.

Tokenizer	Dataset	Depth	Success	Method	States	Edges (10^3)		Payload (MB)	
					Mean	P50	P95	P50	P95
LLaDA2.2†	BFCL	4	213/300	DINGO-Opt	655.92	49 124.94	161 307.59	2 358.16	7 743.37
				DConstrain	164.98	3.27	8.87	–	–
	CFB	4	99/1000	DINGO-Opt	409.19	77 163.56	169 703.33	3 704.11	8 146.34
				DConstrain	247.52	9.65	15.14	–	–
DiffusionGemma‡	BFCL	4	2141/2495	DINGO-Opt	1 737.94	9 178.19	65 604.92	440.59	3 149.94
				DConstrain	481.19	0.96	3.58	**3.46**	**4.20**
	CFB	4	137/1000	DINGO-Opt	1 379.69	19 669.07	62 942.46	944.23	3 021.61
				DConstrain	693.85	2.28	3.61	**3.66**	**3.86**

DConstrain succeeds on every evaluated case, the intersection size n is determined by the finite-state baseline's compilation coverage. The results show that arbitrary property order leads to very large context-expanded transition structures for DINGO-Opt, whereas DConstrain retains compact local RTN nodes and symbolic transitions.

D.5 Cold Compilation with Additional Tokenizers

To evaluate whether DConstrain's compilation behavior depends on the two tokenizers used in the main experiments, we additionally compile the same structured-output workloads with the Dream 7B and LLaDA 8B tokenizers. These experiments evaluate compilation only and do not run model inference. All settings use arbitrary property order and the same recursive structured-output grammar configuration.

Table 10 shows that DConstrain retains high compilation coverage and sub-second median cold-compilation latency across both additional tokenizers. BFCL and ComplexFuncBench compile suc-

Table 10: Any-order cold compilation with Dream 7B and LLaDA 8B tokenizers. All runs use the same recursive structured-output grammar configuration and nesting bounds of 8/8. Statistics use successful compilations; Success reports coverage. States are mean local RTN node counts, edges are symbolic transitions, and payload is the schema-exclusive native-snapshot footprint in decimal MB. Cold compilation uses 64 concurrent single-threaded workers.

Tokenizer	Dataset	Success	States	Edges (10^3)		Payload (MB)		Cold (s)	
			Mean	P50	P95	P50	P95	P50	P95
Dream 7B	BFCL	300/300	416.25	0.92	4.24	2.77	4.08	0.36	1.05
	JSONSchemaBench	281/300	616.65	1.03	157.53	2.78	24.71	0.38	12.85
	ComplexFuncBench	1000/1000	770.05	2.83	4.39	3.22	3.69	0.67	0.97
LLaDA 8B	BFCL	300/300	416.25	0.85	4.00	2.09	3.03	0.31	0.81
	JSONSchemaBench	281/300	616.65	0.93	138.53	2.08	14.90	0.33	10.78
	ComplexFuncBench	1000/1000	770.05	2.63	4.06	2.45	2.80	0.60	0.84

cessfully in all cases, while JSONSchemaBench succeeds on 281 of 300 evaluated schemas. The identical local-state counts across tokenizers reflect the tokenizer-independent pushdown grammar structure, while token-dependent symbolic transitions, payload size, and compilation latency vary with tokenization. These results indicate that the factorized compilation design is not specific to the LLaDA2.2 or DiffusionGemma tokenizers used in the main experiments.

D.6 Failure Analysis on JSONSchemaBench

We further analyze the remaining DConstrain compilation failures on JSONSchemaBench with the LLaDA2.2 tokenizer. Table 11 categorizes all 1,072 failed schemas out of 5,754, corresponding to an 18.6% failure rate and an 81.4% success rate. Most failures arise from unsupported schema semantics rather than compiler execution failures: semantic-coverage limitations account for 789 cases (13.7% of the full benchmark and 73.6% of all failures). Resource limits account for another 274 cases (4.8% of the benchmark), primarily RSS exhaustion and the 180 s compilation timeout. Only nine cases (0.2%) are attributable to known implementation errors or unresolved compiler behavior.

The dominant failure modes therefore reflect the current supported schema fragment rather than failures of the factorized compilation mechanism itself. In particular, `oneOf`/`allOf` reasoning and nested resource semantics account for a substantial fraction of unsupported cases. Resource failures form the second-largest category, showing that although DConstrain substantially reduces context expansion, sufficiently complex schemas can still exceed finite construction or memory budgets.

Localizing frontend coverage limitations. To separate frontend coverage from factorized execution, we replayed the 178 `oneOf` branch-disjointness rejections and 127 general `allOf` rejections in Table 11 using the archived frontend from the reported experiment. All 305 rejections were reproduced by the schema-to-parser frontend alone, without invoking the factorized token compiler. These failures therefore precede factorized execution; their localization does not establish that downstream compilation would succeed after extending the frontend.

Frontend acceptance versus semantic preservation. We examined the same 305 original schemas using XGrammar (Dong et al., 2025). With `strict_mode=True`, XGrammar 0.2.8 returned grammars for 298 schemas. A bounded search identified 124 distinct schemas for which the native matcher accepted a complete JSON text that violated the original schema. We independently replayed all 124 counterexamples and confirmed their invalidity with `jsonschema` 4.26.0; 104 were also confirmed by `fastjsonschema` 2.22.2. We additionally found valid instances for 86 schemas, establishing their satisfiability. Replaying the same invalid texts in XGrammar 0.1.27 reproduced violations for 123 schemas. Table 12 summarizes these results.

The audit used an empty tokenizer and the native complete-string matching interface, requiring an accepting terminal state. It therefore tests membership in the compiled character language, rather than compilation against the LLaDA2.2 vocabulary or the likelihood of generating these texts during

Table 11: Failure breakdown for DConstrain on JSONSchemaBench with the LLaDA2.2 tokenizer. Percentages are relative to all 5,754 schemas and rounded to one decimal place; the total is calculated from counts before rounding.

Failure cause	Cases	Full set (%)
Schema coverage and semantic limitations		
Unsupported keyword or keyword combination	285	5.0
`oneOf` branch exclusivity not provable	178	3.1
General `allOf` intersection unsupported	127	2.2
Nested resource scopes (`id`/`$id`) unsupported	127	2.2
Nested schema-dialect switching unsupported	28	0.5
Dynamic-property constraints conflict with empty-object subset	38	0.7
Regular expression outside supported dialect	4	0.1
Tuple-style `items` unsupported	1	0.0
`$ref` and outer-type intersection not provable	1	0.0
Resource and construction limits		
State/transition/closure or pattern-size budget exceeded	68	1.2
RSS limit exceeded	134	2.3
Compilation timeout	72	1.3
Implementation and unresolved failures		
`_deduplicate_json_values` implementation error	4	0.1
Duplicate `enum` values rejected	3	0.1
Compiled language unexpectedly empty	2	0.0
Total failures	**1,072**	**18.6**

Table 12: Targeted semantic audit of 305 historical DConstrain rejections. All counts refer to schemas. Counterexamples are complete JSON texts accepted by XGrammar but rejected under the original schema. Rows use DConstrain's first rejection cause, which need not match the constraint violated by the witness. Counterexample counts are lower bounds from a bounded search; absence of a witness does not establish semantic preservation.

Historical rejection	Cases	Compiled XG 0.2.8	Counterexamples XG 0.2.8	Same witnesses XG 0.1.27
`oneOf` disjointness	178	175	43	42
General `allOf`	127	123	81	81
Total	**305**	**298**	**124**	**123**

model inference. Validation followed each schema's declared dialect, defaulting to Draft 2020-12 when unspecified. Optional `format` assertions and network reference retrieval were disabled.

For example, JSB schema `o41599` specifies `oneOf` over integer, string, number, boolean, and null. Both tested XGrammar versions accept `0`, which satisfies both the integer and number branches and thus violates the exactly-one requirement. Similarly, schema `o44455` explicitly requires an object and contains a multi-branch `allOf`, yet both matchers accept `0`. Both examples were confirmed by both validators.

Scope of the conclusions. Frontend acceptance alone is insufficient evidence that the original schema constraints are preserved. This targeted audit does not estimate overall semantic accuracy, establish DConstrain's semantic correctness on all accepted schemas, or imply that every rejected schema is intrinsically difficult. We distinguish execution correctness relative to supported reference-parser semantics from frontend coverage and resource-bounded construction. Incomplete coverage, resource failures, and implementation issues remain limitations of the current system.

Limitations. DConstrain substantially reduces the cost of nested constraints during cold compilation by retaining a symbolic RTN representation rather than expanding stack contexts into finite

states, and this representation can in principle express context-free structure. However, factorization does not eliminate the worst-case complexity of joint inference. During warm compilation, uncertainty over unresolved token choices can propagate across the decoding canvas and induce a rapidly growing set of reachable runtime contexts; in the worst case, this expansion can still be exponential in nesting depth or other sources of branching. DConstrain therefore extends the practical nesting depth substantially relative to finite-state baselines, but it does not provide unbounded or uniformly efficient compilation for arbitrary deeply nested grammars.