SLOTWEAVE: DEPENDENCY-AWARE ADAPTIVE SLOT SCHEDULING FOR DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Masked diffusion large language models (dLLMs) enable parallel token generation, but independently decoding dependent positions can produce incoherent outputs. Slot-based diffusion decoding addresses this issue through autoregressive infilling within fixed-size slots and selecting decodable parallel slots by confidence. However, fixed slot boundaries can separate strongly coupled tokens, while confidence-based selection does not explicitly prevent mutually dependent slots from being decoded together. We propose SlotWeave, a training-free dependency-aware adaptive scheduler for slot-based dLLM decoding. SlotWeave estimates dependencies from attention weights obtained during the decoding forward pass. It first adaptively partitions remaining masked positions into contiguous slots by merging highly coupled neighbors. Then SlotWeave constructs a slot-level dependency graph and greedily selects a non-conflicting set of slots for parallel autoregressive infilling. Experiments on mathematical reasoning and code generation benchmarks show accuracy gains over the fixed slot baseline. For example, on MATH500 at primitive slot size $s = 8$, it improves accuracy from 49.4% to 53.2% with an approximately 10% reduction in throughput. Ablation studies further support the design choices of our method.

1 INTRODUCTION

Autoregressive generation has become the dominant paradigm for current large language models (LLMs) (Jaech et al., 2024; Grattafiori et al., 2024; Guo et al., 2025; Yang et al., 2025), which follows an intrinsically sequential inference process. This strict left-to-right dependency fundamentally limits decoding parallelism. Masked diffusion large language models (dLLMs) offer a compelling alternative to the bottleneck. Rather than producing tokens one at a time, dLLMs iteratively unmask a partially masked sequence and can update multiple positions within a single forward pass (Austin et al., 2021a; Sahoo et al., 2024; Shi et al., 2024; Nie et al., 2026; Ye et al., 2025). In principle, this flexibility enables parallel generation and substantially reduces the number of decoding steps required to produce a complete sequence.

Despite this appeal, realizing the full potential of diffusion language modeling in practice remains an open challenge. One crucial constraint is that parallel decoding typically assumes token-wise conditional independency (Huang et al., 2022; Li et al., 2026; Zhong et al., 2026). When multiple positions are decoded independently from their marginals, the resulting product distribution may mismatch the true joint distribution. For example, if a masked pair of tokens could plausibly form either `New York` or `Los Angeles`, independent token-level decisions may combine fragments of both, yielding an incoherent hybrid such as `New Angeles`. Hence, parallel decoding tokens that are mutually dependent can lead to inconsistent responses.

Recently, striking a trade-off between autoregressive and diffusion generation has emerged as a promising direction for combining the strengths of both paradigms. Hybrid methods such as block diffusion (Han et al., 2023; Arriola et al., 2025; Wu et al., 2026b;a) partition the response into multiple blocks, enforcing autoregressive order across blocks while retaining diffusion generation within each one. While token-wise dependence is preserved at the block level, the tighter dependencies among nearby tokens remain unaddressed. As a counterpart, slot-based diffusion (Li et al., 2026; Leang et al., 2026) is proposed, partitioning the response into units termed slots, such that autoregressive generation operates within each slot while diffusion generation operates across them.

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.

More precisely, at each decoding iteration, slot-based diffusion alternates between decodable slots selection and autoregressive infilling within the selected slots.

Yet slot-based decoding must address two coupled design questions: *where to place slot boundaries and which slots to decode in parallel*. The current framework has limitations in both respects. **(1) Dependency-agnostic slot boundaries.** Fixed-size partitioning does not account for token dependencies and can separate strongly coupled positions into different slots. Although autoregressive infilling captures dependencies within each slot, cross-boundary dependencies remain unaddressed when the corresponding slots are decoded independently in parallel. **(2) Confidence may be an unreliable proxy for independence.** Existing methods such as ReFusion (Li et al., 2026) select slots based solely on confidence. However, two high-confidence slots can still be mutually dependent, making their independent parallel decoding prone to incoherence. Conversely, slots with only moderate confidence may be weakly dependent and therefore suitable candidates for parallel decoding.

To address these issues, we propose SlotWeave, a training-free dependency-aware adaptive slot scheduler for slot-based dLLM decoding. At each decoding iteration, SlotWeave reuses the attention weights (Vaswani et al., 2017) already computed during the decoding forward pass to estimate dependencies among the remaining masked positions. And it adaptively partitions the remaining masked region into contiguous slots. Boundaries with strong cross-dependence are merged, while weakly dependent regions remain separate and available for parallel decoding. Given this adaptive partition, SlotWeave constructs a slot-level dependency graph, where an edge indicates strong estimated inter-slot dependencies. A greedy independent set procedure selects slots without detected conflicts for parallel infilling. The partition and graph are refreshed as generation progresses.

Our experiments on mathematical reasoning and code generation benchmarks show that SlotWeave improves accuracy over the state-of-the-art slot-based diffusion model ReFusion under matched primitive slot sizes and generation settings. For example, on MATH500 at primitive slot size $s = 8$, it improves accuracy from 49.4% to 53.2% with an approximately 10% reduction in throughput. The improvements are most pronounced with fine-grained primitive slots, reaching up to 10.0 points on mathematical reasoning and 15.3 points on code generation. These results support explicitly accounting for dependencies when constructing and selecting slots, while ablation studies further demonstrate the contributions of different design choices of the proposed method.

In summary, our contributions are threefold:

- We identify two limitations of current slot-based dLLM decoding: fixed boundaries can separate strongly coupled tokens, while confidence-based selection does not explicitly account for dependencies among slots.
- We propose SlotWeave, a training-free dependency-aware slot scheduler that utilizes attention-derived dependency estimates to construct adaptive slots and select mutually compatible ones for parallel autoregressive infilling.
- We conduct extensive experiments on mathematical reasoning and code generation. The proposed method shows reasonable improvement compared to the baseline, and ablation studies demonstrate the effectiveness of the design choices.

2 Background

2.1 Masked Diffusion Large Language Models

Masked diffusion large language models generate text by iteratively replacing `[MASK]` tokens with vocabulary tokens. Let x_t denote the partially masked response of length L at decoding iteration t, and let $M_t \subseteq \{1, \ldots, L\}$ denote the remaining masked positions. Conditioned on x_t and the prompt, the model predicts a token distribution at each masked position, allowing multiple positions to be decoded in a single forward pass.

For a selected set of positions $\{i_1, \ldots, i_k\} \subseteq M_t$, independent parallel decoding approximates their joint conditional distribution by a product of token-wise marginals:

$$p(x^{i_1}, \ldots, x^{i_k} \mid x_t) \approx \prod_{j=1}^{k} p(x^{i_j} \mid x_t), \tag{1}$$

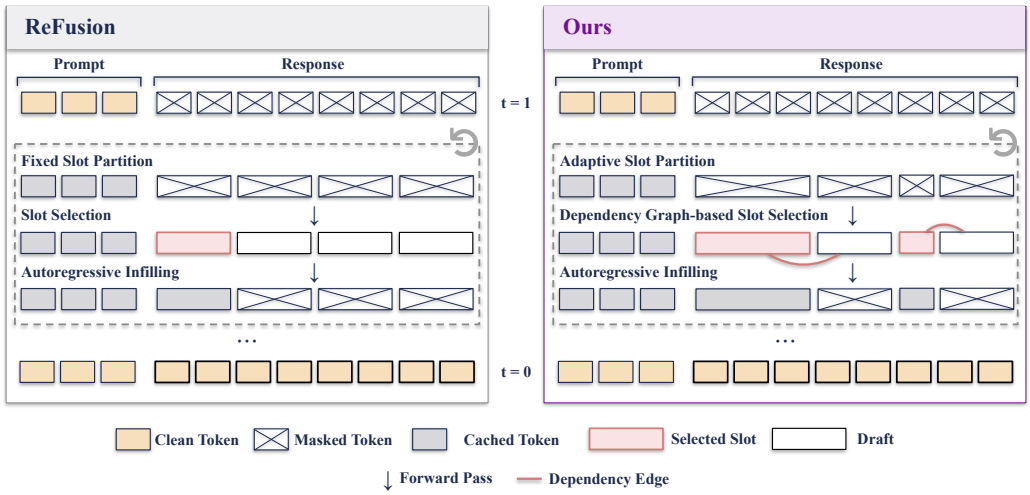


Figure 1: **ReFusion versus our dependency-aware adaptive slot scheduler SlotWeave. Left:** ReFusion uses fixed-size slots and selects them based on confidence. **Right:** SlotWeave merges strongly coupled neighboring slots and then selects a non-conflict set from a dependency graph, before autoregressive infilling.

where x^i denotes the completed token at position i, and conditioning on the prompt is omitted for brevity. This approximation can produce incoherent token combinations when the selected positions are mutually dependent.

2.2 Slot-Level Diffusion with Autoregressive Infilling

Slot-based decoding combines parallel generation across slots with autoregressive generation within each slot. In the state-of-the-art method ReFusion (Li et al., 2026), the response is partitioned into consecutive fixed-size slots, which we refer to as *primitive slots*. At each decoding iteration, a forward pass proposes draft token predictions for the remaining masked slots and provides confidence scores for slot selection. The selected slots are then processed in parallel through autoregressive infilling, which verifies draft tokens and completes the remaining positions within each slot.

Autoregressive infilling conditions each token on previously accepted tokens within the same slot, thereby accounting for dependencies that independent token-level decoding would otherwise ignore. For KV-cache reuse, ReFusion places decoded slots before the remaining masked slots in the causal buffer while retaining their original position identifiers, and applies causal attention during the forward pass.

This design handles dependencies within slots, but fixed boundaries can leave strongly coupled positions in separate slots, and confidence-based selection does not explicitly account for their cross-slot dependencies. Our method replaces fixed partitioning and confidence-only selection with adaptive slot construction and dependency-aware parallel selection, which will be described with details in the next section. Figure 1 illustrates the distinction between both methods.

3 SlotWeave

3.1 Overview

SlotWeave replaces the confidence-based, fixed-size slot scheduler of ReFusion with a dependency-aware adaptive scheduler, while the underlying model and autoregressive infilling procedure remain unchanged.

At each decoding iteration, SlotWeave first estimates dependencies among the remaining masked positions using attention weights from the standard ReFusion slots selection pass. It then constructs

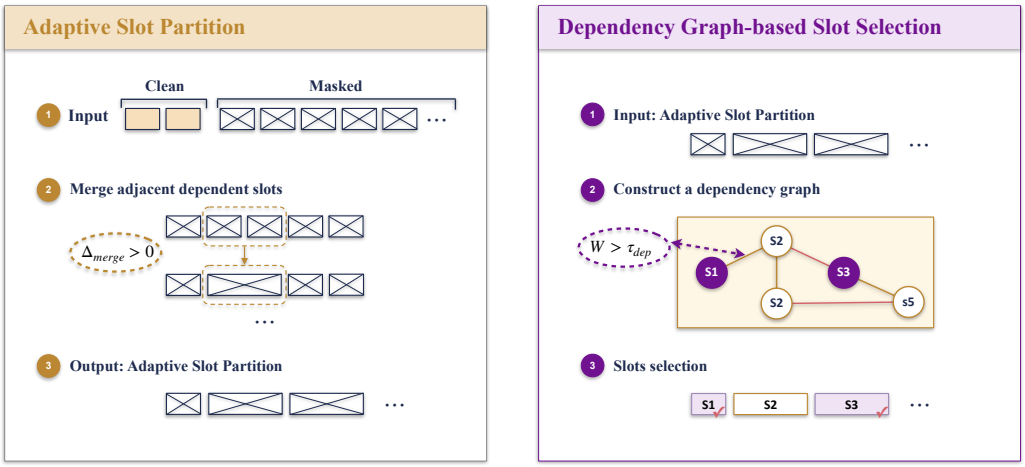


Figure 2: **SlotWeave scheduling within one decoding iteration. Left:** remaining primitive slots are scanned left to right and merged when cross-boundary coupling dominates, yielding variable-length adaptive slots. **Right:** resulting adaptive slots become vertices of a dependency graph, and a non-conflicting subset is selected for the current parallel decoding.

adaptive slots by merging strongly coupled neighboring slots, allowing their dependencies to be handled through within-slot autoregressive infilling. Given the resulting partition, SlotWeave constructs a slot-level dependency graph, where an edge indicates that the estimated dependency between two slots exceeds a scheduled threshold. A greedy independent set procedure selects non-conflicting slots for parallel decoding. The dependency estimates, partition, and graph are refreshed at each iteration. Figure 2 illustrates the two scheduling stages.

3.2 Token Dependency Estimation

At decoding iteration t, SlotWeave estimates dependencies among the remaining masked positions M_t, by reusing attention weights from the standard ReFusion slots selection pass. As described in Section 2.2, ReFusion places decoded slots before the remaining masked slots in the causal buffer to reuse the KV-cache, while retaining their original position identifiers. We write $j \preceq_t i$ when position j appears no later than position i in the current causal buffer, and define the visible position set as $\mathcal{V}_t(i) = \{k : k \preceq_t i\}$. This set includes visible prompt and response positions.

Let $Q^{(t)}_{\ell,h}(i)$ and $K^{(t)}_{\ell,h}(j)$ denote the rotary-positioned (Su et al., 2024) query and key vectors at positions i and j, respectively, for transformer layer ℓ and attention head h. The attention weight is

$$A^{(t)}_{\ell,h}(i,j) = \left[\mathrm{softmax}\left(\left\{\langle Q^{(t)}_{\ell,h}(i), K^{(t)}_{\ell,h}(k)\rangle\right\}_{k\in\mathcal{V}_t(i)}\right)\right]_j. \tag{2}$$

Here, softmax is normalized over the visible positions $\mathcal{V}_t(i)$ and we omit the standard factor for notational simplicity. $A^{(t)}_{\ell,h}(i,j)$ is set to 0 for $j \notin \mathcal{V}_t(i)$. The normalization includes all visible positions, whereas the dependency estimates below use only pairs of remaining masked positions $i, j \in M_t$.

We average the attention weights over the selected layers and attention heads:

$$a^{(t)}_{i\leftarrow j} = \frac{1}{|\mathcal{L}_{\mathrm{attn}}||\mathcal{H}_{\mathrm{attn}}|}\sum_{\ell\in\mathcal{L}_{\mathrm{attn}}}\sum_{h\in\mathcal{H}_{\mathrm{attn}}} A^{(t)}_{\ell,h}(i,j). \tag{3}$$

We use all attention heads in the selected layers. Unless otherwise specified, $\mathcal{L}_{\mathrm{attn}}$ contains the last fraction ρ of transformer layers, with $\rho = 0.8$ in the main experiments. Appendix B.5 specifies the exact layer indices and provides more details.

For two distinct masked positions $i, j \in M_t$, we define the token-level dependency score

$$D_{ij}^{(t)} = \max\left\{a_{i\leftarrow j}^{(t)}, a_{j\leftarrow i}^{(t)}\right\}. \tag{4}$$

Under the causal order, at least one of the two directions is nonzero. We symmetrize the attention scores to obtain an undirected measure of potential coupling. A high score suggests that one position may benefit from conditioning on the other, which is unavailable when the two are decoded independently in parallel (Huang et al., 2022; Kim et al., 2026). We therefore use this score as a heuristic for scheduling, rather than as a direct measure of conditional dependence. For the remainder of this section, we omit the superscript (t) when the decoding iteration is unambiguous.

3.3 Adaptive Slot Partition

SlotWeave constructs adaptive slots by merging strongly coupled neighbors. At each iteration, the procedure starts from the remaining primitive slots in their original left-to-right order. Adjacency and contiguity refer to this compacted sequence, from which decoded slots have been removed. Those decoded positions remain in the conditioning context but are not included in the merged slot.

To assess whether two neighboring slots should be merged, we compare their cross-slot and intra-slot dependencies. For two disjoint slots A and B, we define the average cross-slot dependency $\bar{D}(A, B)$ and the average intra-slot dependency $\bar{D}(A)$ as

$$\bar{D}(A, B) = \frac{1}{|A||B|}\sum_{i\in A}\sum_{j\in B} D_{ij}, \qquad \bar{D}(A) = \frac{1}{|A|(|A|-1)}\sum_{i,j\in A, i\neq j} D_{ij}. \tag{5}$$

The intra-slot dependency is defined for $|A| \geq 2$. Both quantities are normalized by the number of token pairs to account for differences in slot size.

Assuming A as the current accumulated slot and B as the next primitive slot, we define the merge gap as

$$\Delta_{\text{merge}}(A, B) = \bar{D}(A, B) - \min\{\bar{D}(A), \bar{D}(B)\}. \tag{6}$$

We merge the two slots when $\Delta_{\text{merge}}(A, B) > 0$. This criterion removes a boundary when the cross-slot dependency exceeds the weaker of the two intra-slot dependencies. Intuitively, if a slot is more strongly coupled to its neighbor than internally, the boundary may separate positions that are better decoded autoregressively within a single slot. Merging the two slots allows these cross-boundary dependencies to be handled through within-slot autoregressive infilling. Also, the criterion adapts to the local dependency scale without introducing an additional merge threshold hyperparameter.

We apply this criterion in a left-to-right greedy scan. The first remaining primitive slot initializes the accumulated slot A. If the next primitive slot B satisfies $\Delta_{\text{merge}}(A, B) > 0$, we replace A with $A \cup B$ and continue with the updated accumulated slot. Otherwise, we finalize A and initialize a new accumulated slot with B. After the last primitive slot has been processed, the resulting adaptive slots form a partition of M_t. The left panel of Figure 2 illustrates this adaptive slot partitioning process, and the corresponding procedure is given by Algorithm 1 in Appendix A.1.

3.4 Dependency-Aware Slot Selection

Adaptive merging groups strongly coupled neighboring slots, but dependencies may remain between distinct adaptive slots, including non-adjacent ones. SlotWeave therefore constructs a dependency graph to identify pairs of slots that should not be decoded independently in parallel.

Let $\mathcal{S} = \{S_1, \ldots, S_k\}$ denote the adaptive partition and $\mathcal{V} = \{1, \ldots, k\}$ its slot indices. For distinct slots S_a and S_b, define their dependency weight as

$$W_{ab} = \bar{D}(S_a, S_b). \tag{7}$$

The dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ contains an edge between two slots when their dependency weight exceeds a threshold τ_{dep}:

$$\mathcal{E} = \{\{a, b\} : 1 \leq a < b \leq k,\ W_{ab} > \tau_{\text{dep}}\}. \tag{8}$$

Here, W_{ab} measures cross-slot dependency, whereas τ_{dep} determines how conservatively conflicts are detected. For a given partition and dependency estimate, a smaller threshold produces more

conflict edges and more conservative parallel selection, whereas a larger threshold permits more slot pairs to be selected together.

Early in generation, limited decoded context can make dependencies between masked slots difficult to assess (Asano et al., 2026), motivating a conservative conflict criterion. As more decoded context becomes available, we gradually relax this criterion to allow greater parallelism. Specifically, we linearly increase the dependency threshold with generation progress:

$$\tau_{\mathrm{dep}}(r) = \tau_{\mathrm{start}} + (\tau_{\mathrm{end}} - \tau_{\mathrm{start}})r, \qquad r = \frac{n_{\mathrm{acc}}}{n_{\mathrm{pad}}} \in [0, 1], \tag{9}$$

where n_{acc} is the number of accepted response tokens and n_{pad} is the padded generation length. More details about the τ_{dep} parameter settings are provided in Appendix B.6.

Given the dependency graph, we select slots using a greedy independent set procedure guided by a utility score u_a. By default, we use the intra-slot dependency, $u_a = \bar{D}(S_a)$, to prioritize slots with stronger internal coupling. Alternatively, u_a can also be the confidence as used in ReFusion. Section 4.3 compares these two utilities.

The procedure starts with all partitioned slots as candidates. At each step, it selects the highest utility candidate and removes the selected slot and its graph neighbors from the candidate set. It repeats until no candidates remain, yielding an independent set such that the selected slots share no conflict edges. The right panel of Figure 2 illustrates this dependency-aware slot selection process, and the corresponding procedure is given by Algorithm 2 in Appendix A.1. All the selected slots undergo parallel autoregressive infilling, after which their constituent primitive slots are removed from the masked sequence. At the next iteration, SlotWeave re-estimates dependencies and reconstructs both the adaptive partition and the dependency graph. The full decoding loop is given by Algorithm 3. Appendix A.2 further analyzes the algorithm complexity, and Appendix A.3 discusses a theoretical partition-and-coloring view of the two stages.

4 Experiments

We evaluate SlotWeave on mathematical reasoning and code generation to assess the effectiveness of dependency-aware adaptive slot scheduling. We first compare it with ReFusion across primitive slot sizes, then examine the contributions of adaptive merging, threshold scheduling, and dependency-aware selection through ablation studies. Finally, we compare intra-slot dependency and slot confidence as selection utilities.

4.1 Experimental setup

Benchmarks and metrics. We consider five benchmarks: GSM8K (Cobbe et al., 2021), MATH500 (Hendrycks et al., 2021), and OlympiadBench (He et al., 2024) for mathematical reasoning, and HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021b) for code generation. We report exact-match accuracy on GSM8K, answer verification accuracy on MATH500 and OlympiadBench, and pass@1 on HumanEval and MBPP. All scores are reported as percentages. Additional dataset and evaluation details are provided in Appendix B.1.

Baseline and model. We use ReFusion-8B (Li et al., 2026) as our base model, which is the only available slot-based diffusion language model to the best of our knowledge. We compare SlotWeave with ReFusion's original decoding policy, which uses fixed-size slots and confidence-based slot selection. Both methods use the same ReFusion-8B checkpoint without additional training and share the autoregressive infilling procedure.

Generation settings. All evaluations are zero-shot, using the chat template, greedy decoding, and one completion per problem. We use a generation budget of 512 tokens on every benchmark. The main comparison sweeps the primitive slot size over $s \in \{2, 4, 8, 16, 32\}$. Unless otherwise specified, SlotWeave uses $\rho = 0.8$ and linearly increases τ_{dep} from 0 to 0.005. Complete hyperparameter settings are provided in Appendix B.2.

Table 1: Performance comparison on reasoning and code-generation benchmarks.

Method	$s = 2$	$s = 4$	$s = 8$	$s = 16$	$s = 32$
Mathematical reasoning					
			GSM8K		
ReFusion	77.0%	81.3%	83.7%	84.7%	84.9%
SlotWeave	85.3% +8.3%	84.7% +3.4%	85.0% +1.3%	84.9% +0.2%	85.0% +0.1%
			MATH500		
ReFusion	41.8%	48.4%	49.4%	52.4%	53.8%
SlotWeave	51.8% +10.0%	52.8% +4.4%	53.2% +3.8%	53.6% +1.2%	53.8% +0.0%
			OlympiadBench		
ReFusion	16.5%	18.8%	20.3%	20.6%	21.8%
SlotWeave	21.5% +5.0%	21.1% +2.3%	21.5% +1.2%	20.8% +0.2%	22.1% +0.3%
Code generation					
			HumanEval		
ReFusion	58.5%	70.1%	75.0%	73.8%	76.8%
SlotWeave	73.8% +15.3%	77.4% +7.3%	77.4% +2.4%	76.8% +3.0%	76.8% +0.0%
			MBPP		
ReFusion	50.0%	58.8%	63.6%	67.4%	67.4%
SlotWeave	60.4% +10.4%	66.4% +7.6%	67.0% +3.4%	67.6% +0.2%	67.4% +0.0%

4.2 Main results

Table 1 compares SlotWeave with ReFusion across five primitive slot sizes, $s \in \{2, 4, 8, 16, 32\}$, using the shared settings described in Section 4.1. SlotWeave matches or improves upon ReFusion on all five benchmarks at every evaluated slot size, with the largest gains in the fine-grained regime.

Strong gains with fine-grained slots. At $s = 2$, SlotWeave improves over ReFusion by 8.3 percentage points on GSM8K, 10.0 on MATH500, 5.0 on OlympiadBench, 15.3 on HumanEval, and 10.4 on MBPP. Substantial gains persist at $s = 8$ across the five benchmarks. For example, SlotWeave improves accuracy from 49.4% to 53.2% on MATH500, with an approximately 10% reduction in tokens per second (Appendix B.3). These results are consistent with our motivation that fine-grained fixed partitions can separate coupled positions into different slots, making dependency-aware partitioning and selection particularly valuable.

Reduced sensitivity to primitive slot size. ReFusion's performance generally degrades as the primitive slot size decreases, whereas SlotWeave is less sensitive. For example, SlotWeave's GSM8K accuracy ranges from 84.7% to 85.3%, compared with 77.0% to 84.9% for ReFusion. Notably, on GSM8K and HumanEval, SlotWeave with small primitive slots outperforms ReFusion across all evaluated slot-size settings. These results suggest that SlotWeave effectively exploits the flexibility of fine-grained partitioning. It selectively merges coupled neighbors while retaining separate slots elsewhere, rather than relying on uniformly large fixed slots to achieve strong accuracy.

Diminishing gains with larger slots. The improvements generally narrow as the primitive slot size increases. At $s = 32$, the two methods differ by at most 0.3 percentage points on every benchmark. This pattern is consistent with larger primitive slots already handling more dependencies through within-slot autoregressive infilling. However, uniformly enlarging slots also places weakly coupled positions within the same autoregressive sequence, reducing opportunities for parallel decoding across slots. SlotWeave instead uses estimated dependencies to schedule adaptive slot merging and selection, preserving the fine-grained parallelism.

We provide further experimental results in the appendices, including throughput comparisons in Appendix B.3, subject-level results on the full MATH test set in Appendix C, the investigation of the interplay between SlotWeave and block diffusion in Appendix D, and some qualitative case studies in Appendix E.

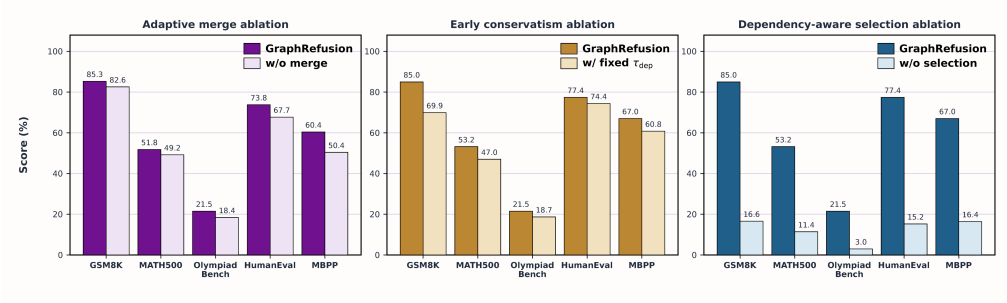


Figure 3: **Component ablations.** Left to right: full SlotWeave versus *w/o merge*, *w/ fixed* τ_{dep}, and *w/o selection*.

4.3 Ablation studies

We examine the contributions of SlotWeave's scheduling components and the choice of selection utility. Figure 3 evaluates adaptive merging, early stage conservatism, and dependency-aware selection by disabling each component separately. Figure 4 compares intra-slot dependency with slot confidence as the selection utility. All experiments use the same generation settings as the main experiments. Component ablations use the intra-slot dependency utility, with adaptive merging evaluated at $s = 2$ and the other two components at $s = 8$. Sensitivity analyses for attention-layer coverage and the dependency threshold are provided in Appendices B.5 and B.6, respectively.

Adaptive slot merging. The left panel of Figure 3 compares the full method with *w/o merge*, which retains the fixed slots while keeping dependency-aware selection unchanged. Removing merging reduces scores on every benchmark, with drops ranging from 2.6 to 10.0 percentage points. The largest reductions occur on code generation: HumanEval falls from 73.8% to 67.7%, and MBPP from 60.4% to 50.4%. These results show that dependency-aware selection alone does not recover the full benefit of adaptive merging, which supports handling strongly coupled neighboring positions through autoregressive infilling within a merged slot, rather than retaining every primitive boundary.

Early stage conservatism. The middle panel of Figure 3 compares the progress-dependent threshold which increases τ_{dep} from 0 to 0.005, with *fixed* τ_{dep} which uses 0.005 throughout generation. The fixed threshold variant performs worse on every benchmark, with reductions ranging from 2.8 to 15.1 percentage points. GSM8K exhibits the largest drop, from 85.0% to 69.9%. This comparison supports enforcing a conservative conflict criterion early in decoding rather than applying the final threshold from the outset. The results are consistent with our motivation that more conservative parallel selection helps avoid premature commitments across potentially dependent slots.

Dependency-aware slot selection. The right panel of Figure 3 compares the full method with *w/o selection*, which retains adaptive merging while removing all conflict edges. Consequently, all remaining adaptive slots are selected for parallel autoregressive infilling, regardless of their estimated dependencies. This causes substantial degradation across all five benchmarks. Specifically, GSM8K falls from 85.0% to 16.6%, and HumanEval from 77.4% to 15.2%, with similarly large reductions on the other benchmarks. These results demonstrate that adaptive merging alone is insufficient to support unrestricted parallel decoding. Constraining parallel selection remains important even after strongly coupled neighboring slots have been merged.

Selection utility. Figure 4 compares two selection utilities: intra-slot dependency with the confidence signals used in ReFusion, while keeping all other SlotWeave settings fixed. Intra-slot dependency consistently performs better in the fine-grained regime, exceeding confidence by 2.7–5.2 percentage points at $s = 2$ and 1.2–6.4 points at $s = 4$ across the five benchmarks. The advantages from intra-slot dependency support prioritizing internally coupled slots within the dependency-aware selection procedure. As the primitive slot size increases, the performance gap between the two utilities narrows. We therefore use intra-slot dependency as the default selection utility.

Together, these ablations support the complementary roles of SlotWeave's design choices: adaptive merging determines which positions receive within-slot autoregressive conditioning, dependency-aware selection constrains parallel selection, the progress-dependent threshold enforces conservatism early in generation, and intra-slot dependency guides selection priority.

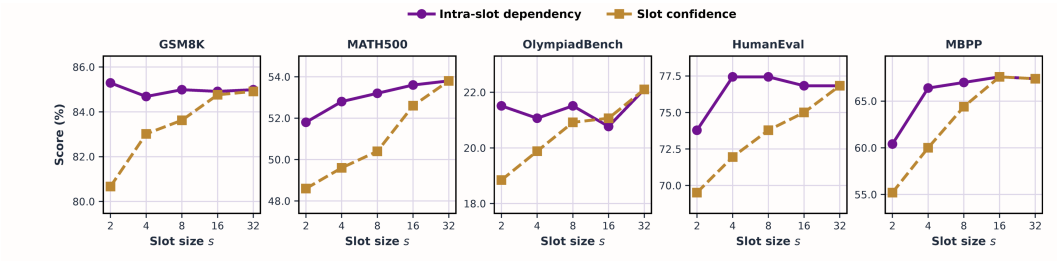


Figure 4: **Selection-utility comparison.** Solid purple: *Intra-slot dependency.* Dashed gold: *Slot confidence.* The two utilities coincide at large s. Intra-slot dependency is stronger at small s.

5 Related work

Hybrid autoregressive–diffusion models. Masked diffusion language models generate text through iterative denoising and allow flexible generation orders (Sahoo et al., 2024; Shi et al., 2024; Nie et al., 2026; Ye et al., 2025). Their main appeal is parallel decoding but they face the challenge from joint marginals mismatch. Block diffusion and related approaches combine autoregressive and diffusion-style generation (Han et al., 2023; Arriola et al., 2025; Wu et al., 2026b;a). ReFusion (Li et al., 2026) is proposed as a counterpart such that autoregressive generation operates within each slot while diffusion generation operates across them.

Adaptive-size partitioning in block diffusion. A related line of work studies how to replace fixed-size blocks with adaptive partitions in block-wise diffusion decoding. AdaBlock-dLLM (Lu et al., 2026) challenges the conventional fixed block-size assumption by identifying a volatility band in the confidence dynamics of masked tokens. It uses this signal to adjust block boundaries during inference. Similarly, WavefrontDiffusion (Yang et al., 2026) expands a dynamic active region based on the confidence signal. Swordsman (Zhang et al., 2026) instead identifies semantic or syntactic constituent boundaries through entropy shifts between adjacent tokens. GeoBlock (Wan et al., 2026) infers block granularity from attention-derived dependency geometry. Instead of using training-free methods, CtrlDiff (Huang & Tang, 2025) uses a learning-based approach to predict the block size.

Dependency-aware decoding for diffusion language models. Another related direction incorporates token dependencies directly into the decoding policy of vanilla diffusion language models. DAPD (Kim et al., 2026) extracts an attention-induced dependency graph over masked tokens and formulates parallel token selection as a graph coloring problem. DAWN (Luo et al., 2026) similarly argues that confidence alone is insufficient for parallel decoding and uses dependency information to select positions that are reliable to unmask in parallel. The other dependency-aware decoding methods for dLLMs include DOS (Zhou et al., 2026a) and Attn-sampler (Zhou et al., 2026b).

6 Conclusion

We presented SlotWeave, a training-free scheduler that combines adaptive slot partitioning with dependency-aware graph selection for slot-based diffusion decoding. Across mathematical reasoning and code generation benchmarks, SlotWeave consistently matches or improves upon ReFusion at matched primitive slot sizes, with the largest gains in the fine-grained regime. Ablations support the complementary roles of adaptive merging, dependency-aware selection, and early stage conservatism, as well as the choice of intra-slot dependency as the selection utility.

Limitations. Despite these improvements, SlotWeave has several limitations. First, the attention-derived dependency score is only a proxy for the true conditional dependency among tokens. Attention weights may capture multiple linguistic or positional relationships and do not necessarily correspond exactly to the influence that one token exerts on another. Also, the current scheduler uses greedy partitioning and selection. These procedures are efficient and easy to integrate into existing inference pipelines, but they are not globally optimal. More sophisticated designs remain to be explored.

AI USE STATEMENT

We used large language models to polish the manuscript, including correcting grammatical errors and improving clarity and organization. These tools also assisted with retrieval and discovery, including identifying related work. LLMs were not involved in the ideation, research methodology, or experimental design.

REFERENCES

Marianne Arriola, Aaron Gokaslan, Justin Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *International Conference on Learning Representations*, volume 2025, pp. 50726–50753, 2025.

Hikaru Asano, Tadashi Kozuno, Kuniaki Saito, and Yukino Baba. Where-to-unmask: Ground-truth-guided unmasking order learning for masked diffusion language models. *arXiv preprint arXiv:2602.09501*, 2026.

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in neural information processing systems*, 34:17981–17993, 2021a.

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

John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory*. Springer Publishing Company, Incorporated, 2008.

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

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Xiaochuang Han, Sachin Kumar, and Yulia Tsvetkov. Ssd-lm: Semi-autoregressive simplex-based diffusion language model for text generation and modular control. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11575–11596, 2023.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3828–3850, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Chihan Huang and Hao Tang. Ctrldiff: Boosting large diffusion language models with dynamic block prediction and controllable generation. *arXiv preprint arXiv:2505.14455*, 2025.

Fei Huang, Tianhua Tao, Hao Zhou, Lei Li, and Minlie Huang. On the learning of non-autoregressive transformers. In *International conference on machine learning*, pp. 9356–9376. PMLR, 2022.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. What does bert learn about the structure of language? In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 3651–3657, 2019.

Tommy R Jensen and Bjarne Toft. *Graph coloring problems*. John Wiley & Sons, 2011.

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

Joshua Ong Jun Leang, Yu Zhao, Mihaela CÄ Stoian, Wenda Li, Shay B Cohen, and Eleonora Giunchiglia. Can i have your order? monte-carlo tree search for slot filling ordering in diffusion language models. *arXiv preprint arXiv:2602.12586*, 2026.

Jia-Nan Li, Jian Guan, Wei Wu, and Chongxuan Li. Refusion: A diffusion large language model with parallel autoregressive decoding. In *International Conference on Learning Representations*, volume 2026, pp. 53846–53869, 2026.

Guanxi Lu, Hao Chen, Yuto Karashima, Zhican Wang, Daichi Fujiki, and Hongxiang Fan. Adablock-dllm: Semantic-aware diffusion llm inference via adaptive block size. In *International Conference on Learning Representations*, volume 2026, pp. 140018–140033, 2026.

Lizhuo Luo, Zhuoran Shi, Jiajun Luo, Zhi Wang, Shen Ren, Wenya Wang, and Tianwei Zhang. Dawn: Dependency-aware fast inference for diffusion llms. *arXiv preprint arXiv:2602.06953*, 2026.

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

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

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis Titsias. Simplified and generalized masked diffusion for discrete data. *Advances in neural information processing systems*, 37: 103131–103167, 2024.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical nlp pipeline. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4593–4601, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Lipeng Wan, Junjie Ma, Jianhui Gu, Zeyang Liu, Xuyang Lu, and Xuguang Lan. Geoblock: Inferring block granularity from dependency geometry in diffusion language models. *arXiv preprint arXiv:2603.26675*, 2026.

Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo Molchanov, Ping Luo, Song Han, and Enze Xie. Fast-dllm v2: Efficient block-diffusion llm. In *International Conference on Learning Representations*, volume 2026, pp. 128353–128370, 2026a.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. In *International Conference on Learning Representations*, volume 2026, pp. 57027–57051, 2026b.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Haojin Yang, Rui Hu, Zequn Sun, Rui Zhou, Yujun Cai, and Yiwei Wang. Wavefrontdiffusion: Dynamic decoding schedule for improved reasoning. In *International Conference on Learning Representations*, volume 2026, pp. 19902–19926, 2026.

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

Yu Zhang, Xinchen Li, Jialei Zhou, Hongnan Ma, Zhongwei Wan, Yiwei Shi, Duoqian Miao, Qi Zhang, and Longbing Cao. Swordsman: Entropy-driven adaptive block partition for efficient diffusion language models. *arXiv preprint arXiv:2602.04399*, 2026.

Yangyang Zhong, Yanmei Gu, Zhengqing Zang, Xiaomeng Li, Yuqi Ding, Xibei Jia, Yuting Shen, Zhenzhong Lan, Liwang Zhu, Weiping Liu, et al. Parallelism and generation order in masked diffusion language models: Limits today, potential tomorrow. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 7178–7218, 2026.

Xueyu Zhou, Yangrong Hu, and Jian Huang. Dos: Dependency-oriented sampler for masked diffusion language models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17404–17419, 2026a.

Yuyan Zhou, Kai Syun Hou, Weiyu Chen, and James Kwok. Attention-based sampler for diffusion language models. *arXiv preprint arXiv:2604.08564*, 2026b.

A Algorithmic Details

A.1 Pseudocode

Algorithms 1 and 2 are the two scheduling stages from Sections 3.3 and 3.4, respectively. Algorithm 3 is the decoding loop that calls them at every iteration.

Algorithm 1 Adaptive Slot Partition

Require: Remaining ordered primitive slots $\widehat{\mathcal{S}} = (S_1, \ldots, S_m)$, dependency scores $\bar{D}(\cdot)$ and $\bar{D}(\cdot, \cdot)$
Ensure: Adaptive partition $\mathcal{S}$

$\mathcal{S} \leftarrow \emptyset$, $i \leftarrow 1$ *// initialize the output partition*
while $i \leq m$ **do** *// scan remaining primitive slots from left to right*
  $A \leftarrow S_i$, $j \leftarrow i + 1$ *// initialize the accumulated slot*
  while $j \leq m$ **do**
    $\Delta_{\text{merge}}(A, S_j) \leftarrow \bar{D}(A, S_j) - \min\{\bar{D}(A), \bar{D}(S_j)\}$ *// Equation (6)*
    if $\Delta_{\text{merge}}(A, S_j) > 0$ **then**
      $A \leftarrow A \cup S_j$, $j \leftarrow j + 1$
    else
      break *// finalize the current accumulated slot*
    end if
  end while
  Append A to $\mathcal{S}$
  $i \leftarrow j$
end while
return $\mathcal{S}$

Algorithm 2 Dependency-Aware Slot Selection

Require: Adaptive partition $\mathcal{S} = \{S_1, \ldots, S_k\}$, utilities $u_1, \ldots, u_k$, threshold τ_{dep}, and $\bar{D}(\cdot, \cdot)$
Ensure: A non-conflicting selected index set $\mathcal{K} \subseteq \{1, \ldots, k\}$

$\mathcal{V} \leftarrow \{1, \ldots, k\}$, $\mathcal{E} \leftarrow \emptyset$
for all $1 \leq a < b \leq k$ **do**
  $W_{ab} \leftarrow \bar{D}(S_a, S_b)$ *// Equation (7)*
  if $W_{ab} > \tau_{\text{dep}}$ **then** *// Equation (8)*
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\{a, b\}\}$
  end if
end for
Construct the dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
$\mathcal{K} \leftarrow \emptyset$, $Cand \leftarrow \mathcal{V}$
while $Cand \neq \emptyset$ **do** *// greedy independent set*
  $a^\star \leftarrow \arg\max_{a \in Cand} u_a$, with ties broken by smaller index
  $\mathcal{K} \leftarrow \mathcal{K} \cup \{a^\star\}$
  $Cand \leftarrow Cand \setminus (\{a^\star\} \cup \mathcal{N}_{\mathcal{G}}(a^\star))$ *// remove the selected slot and its neighbors*
end while
return $\mathcal{K}$

Algorithm 3 SlotWeave Decoding

Require: Remaining primitive slots $\widehat{\mathcal{S}}$, accepted length n_{acc}, padded length n_{pad}, thresholds $\tau_{\mathrm{start}}, \tau_{\mathrm{end}}$, and utility $u(\cdot)$
Ensure: A completed response
while $\widehat{\mathcal{S}} \neq \emptyset$ **do** *// one decoding iteration*
  Estimate D_{ij} and $\bar{D}$ from the ReFusion slot selection pass on $\widehat{\mathcal{S}}$
  $r \leftarrow n_{\mathrm{acc}}/n_{\mathrm{pad}}$ *// generation progress*
  $\tau_{\mathrm{dep}} \leftarrow \tau_{\mathrm{start}} + (\tau_{\mathrm{end}} - \tau_{\mathrm{start}})r$ *// Equation (9)*
  $\mathcal{S} \leftarrow$ ADAPTIVE-SLOT-PARTITION$(\widehat{\mathcal{S}}, \bar{D})$ *// Algorithm 1*
  Compute u_a for each $S_a \in \mathcal{S}$ *// intra-slot dependency or slot confidence*
  $\mathcal{K} \leftarrow$ DEPENDENCY-AWARE-SLOT-SELECTION$(\mathcal{S}, \{u_a\}, \tau_{\mathrm{dep}}, \bar{D})$ *// Algorithm 2*
  Autoregressively infill $\{S_a : a \in \mathcal{K}\}$ as in ReFusion
  Update n_{acc} by the number of newly accepted tokens
  Delete from $\widehat{\mathcal{S}}$ every primitive contained in $\{S_a : a \in \mathcal{K}\}$
end while

Algorithm 1 implements the adaptive partition with a left-to-right greedy scan in Section 3.3. Starting from the remaining primitive slots, it grows an accumulated slot A while the next primitive slot B satisfies $\Delta_{\mathrm{merge}}(A, B) > 0$, and otherwise finalizes A. The output is an ordered partition of M_t into contiguous adaptive slots.

Algorithm 2 implements the dependency-aware selection stage in Section 3.4. It constructs the dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ by adding an edge when $W_{ab} > \tau_{\mathrm{dep}}$, then returns a greedy independent set under the utility u_a. By default, $u_a = \bar{D}(S_a)$ is the intra-slot dependency and the alternative slot confidence utility is compared in Section 4.3.

Algorithm 3 is the complete SlotWeave loop. Every decoding iteration re-estimates dependencies from the ReFusion slot selection pass, increases $\tau_{\mathrm{dep}}(r)$ with generation progress as in Equation (9), and then calls Algorithms 1 and 2. After the selected slots are infilled, their constituent primitive slots are removed and the remainder is compacted in left-to-right order. The next iteration therefore reconstructs both the adaptive partition and the dependency graph, with a larger r and a less conservative threshold. The autoregressive infilling routine remains that of ReFusion.

A.2 Complexity Analysis

At an iteration with m remaining primitive slots, Algorithm 1 performs at most $m - 1$ merging tests, since each test either merges the next primitive slot into the accumulated slot or finalizes the current boundary. The partition stage therefore requires $O(m)$ slot-level checks per iteration. Algorithm 2 then reconstructs the dependency graph under the current threshold $\tau_{\mathrm{dep}}(r)$, which takes $O(k^2)$ pairwise checks for $k \leq m$ adaptive slots, while greedy independent set selection requires at most $O(k)$ steps. Algorithm 3 invokes both subroutines once per decoding iteration. Hence, each iteration adds linear partitioning cost and quadratic graph construction cost in the number of remaining candidate slots, without an additional model forward pass beyond the standard ReFusion slot selection pass.

A.3 Partition-and-Coloring View

The two scheduling stages can also be interpreted as a heuristic for a two-stage graph partitioning and coloring problem (Jensen & Toft, 2011). This interpretation is useful for understanding the roles of adaptive merging and dependency-aware selection, but SlotWeave does not explicitly solve the resulting combinatorial optimization problem.

Let $\mathcal{G}^\star = (\mathcal{V}^\star, \mathcal{E}^\star)$ be an unobserved dependency graph over the N response positions, where $\mathcal{V}^\star = \{1, \ldots, N\}$ indexes these positions from left to right. A feasible partition $\Pi = (S_1, \ldots, S_k)$ is an ordered partition of these positions into contiguous slots, where $k = |\Pi|$ denotes its number of slots. Each partition induces a quotient graph (Bondy & Murty, 2008):

$$\mathcal{G}^\star_\Pi = (\mathcal{V}_\Pi, \mathcal{E}^\star_\Pi), \qquad \mathcal{V}_\Pi = \{1, \ldots, k\}, \tag{10}$$

$$\mathcal{E}_{\Pi}^{\star} = \{\{a, b\} : a, b \in \mathcal{V}_{\Pi},\ a \neq b,\ \exists\, i \in S_a,\ j \in S_b \text{ with } \{i, j\} \in \mathcal{E}^{\star}\}\,. \tag{11}$$

Vertices of $\mathcal{G}_{\Pi}^{\star}$ therefore represent supernodes formed by grouping response positions into slots. To leave as much room as possible for parallel generation, the first stage maximizes the number of slots in a contiguous partition, subject to adjacent slots sharing no quotient edge:

$$\max_{\Pi}\ k \quad \text{s.t.} \quad \{a, a+1\} \notin \mathcal{E}_{\Pi}^{\star}, \quad a = 1, \ldots, k-1. \tag{12}$$

Given this partition, every decoding round selects a set of slots that share no quotient edge and decodes them in parallel. A complete schedule assigns every slot to one such selected set. Hence, this assignment can be viewed as a coloring c, where $c(a)$ is the round of slot S_a. Minimizing the number of decoding rounds on a fixed quotient graph is equivalent to minimizing the number of colors:

$$\min_{c} |\mathrm{im}(c)| \quad \text{s.t.} \quad c(a) \neq c(b) \text{ for all } \{a, b\} \in \mathcal{E}_{\Pi}^{\star}. \tag{13}$$

where $\mathrm{im}(c) = \{c(a) : a \in \mathcal{V}_{\Pi}\}$ is the set of rounds used by c.

The graph $\mathcal{G}^{\star}$ is not available at inference time. We instead use the attention-derived score D_{ij} as a tractable proxy for the latent dependency between positions i and j, and aggregate it into the slot-level scores in Equation (5). Adaptive slot partition then makes a local structural approximation to Equation (12): it removes a boundary whenever $\Delta_{\mathrm{merge}}(A, B) > 0$, i.e., when the cross-slot dependency exceeds the weaker intra-slot dependency. This left-to-right greedy scan favors contiguous adaptive slots that retain strongly coupled neighbors.

Dependency-aware slot selection then approximates this selection on the induced proxy graph. It constructs the dependency graph $\mathcal{G}$ by adding an edge when $W_{ab} > \tau_{\mathrm{dep}}$, and greedily selects a non-conflicting independent set under the utility u_a. The selected independent set is the set of slots decoded in the current round. SlotWeave is not claimed to solve either stage optimally. It provides an efficient, training-free heuristic that uses attention weights to approximate dependency structure, merges strongly coupled neighboring slots, and selects non-conflicting slots for parallel autoregressive infilling.

B EXPERIMENTAL DETAILS

B.1 IMPLEMENTATION DETAILS

SlotWeave is a training-free inference scheduler. We use ReFusion-8B (Li et al., 2026) as the base model and decode from its official checkpoint without additional training. Both ReFusion and SlotWeave share this checkpoint and the same autoregressive infilling procedure. In ReFusion, a slot selection forward pass produces draft token predictions for the remaining masked slots. The selected slots are then processed using autoregressive infilling. During infilling, a draft prefix is accepted sequentially: a draft token is accepted if its causally conditioned probability, given the tokens previously accepted within the same slot, exceeds the token acceptance threshold. Accepted tokens are written back with KV cache reuse. SlotWeave replaces only the original scheduler of fixed-size slots and confidence-based selection with Algorithm 3. It keeps τ_{token} and the infilling routine unchanged. SlotWeave estimates dependencies from the last fraction ρ of transformer layers, linearly increases τ_{dep} with generation progress as in Equation (9), and ranks remaining slots by a utility score, by default the intra-slot dependency $\bar{D}(S_a)$.

All experiments are run on a server with 8 NVIDIA GeForce RTX 3090 GPUs. And the evaluations are zero-shot and use the chat template. We use greedy decoding with temperature 0 and generate one completion per problem. For the performance metrics, we report exact-match accuracy on GSM8K (Cobbe et al., 2021), answer verification accuracy on MATH500 (Hendrycks et al., 2021) and OlympiadBench (He et al., 2024), and pass@1 on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021b). All scores are reported as percentages. For OlympiadBench we use the English text-only open-ended competition-mathematics subset `OE_TO_maths_en_COMP` (674 problems). Following ReFusion, slot-based decoding is applied together with block diffusion: the generation window is partitioned into consecutive blocks that are decoded from left to right, and the slot-level selection-and-infilling procedure runs inside each block. The generation budget is 512 tokens and the main comparison uses $b = 4$ serial blocks, so each block contains 128 tokens. A block of this length is typically sufficient to complete an answer on most of the evaluated datasets. The main comparison sweeps $s \in \{2, 4, 8, 16, 32\}$. Additionally, a sweep over the numbers of blocks $b \in \{1, 2, 4, 8\}$ at different primitive slot sizes, which studies the interplay between slot-based scheduling and block diffusion, is reported in Appendix D.

B.2 HYPERPARAMETER SETTING

During inference, ReFusion exposes three knobs: the slot selection threshold τ_{slot}, the token acceptance threshold τ_{token}, and the primitive slot size s. SlotWeave keeps s and τ_{token}, and adds the layer-coverage ratio ρ, the threshold endpoints $(\tau_{\text{start}}, \tau_{\text{end}})$, and the selection utility.

Table 2: Hyperparameter settings for different benchmarks. The primitive slot size s is swept. τ_{slot} is used only by ReFusion for slot selection. ρ and τ_{end} are used only by SlotWeave.

Benchmark	τ_{slot}	τ_{token}	ρ	τ_{end}
GSM8K	0.9	0.4	0.8	0.005
MATH500	0.8	0.6	0.8	0.005
OlympiadBench	0.8	0.6	0.8	0.005
HumanEval	0.9	0.4	0.8	0.005
MBPP	0.9	0.3	0.8	0.005

Table 2 lists the values used in the main experiments. τ_{slot} is ReFusion's confidence threshold for slot selection and is not used by SlotWeave. The slot selection and token acceptance thresholds follow the official ReFusion settings for GSM8K, MATH500, HumanEval, and MBPP. OlympiadBench has no official ReFusion preset, so we reuse the MATH500 thresholds ($\tau_{\text{slot}} = 0.8$, $\tau_{\text{token}} = 0.6$). In the main experiments, SlotWeave uses $\rho = 0.8$ and linearly increases τ_{dep} from $\tau_{\text{start}} = 0$ to $\tau_{\text{end}} = 0.005$, with intra-slot dependency as the default utility. The slot confidence variant changes only the utility, replacing $\bar{D}(S_a)$ with the confidence signal used by ReFusion. The *w/o merge* ablation disables adaptive merging at $s = 2$ while keeping dependency-aware selection unchanged. The *fixed* τ_{dep} ablation sets $\tau_{\text{start}} = \tau_{\text{end}} = 0.005$ at $s = 8$. The *w/o selection* ablation sets

$\tau_{\text{start}} = \tau_{\text{end}} = 1$ at $s = 8$, which removes all dependency edges. Section B.5 varies only ρ at $s = 8$. Section B.6 varies only τ_{end} at the same setting.

B.3 Comparisons of Throughput

Table 3 reports score, tokens per forward pass (TPF), and tokens per second (TPS) at $s = 8$. TPF denotes the average number of tokens accepted per forward pass, including both the slot-selection pass and the autoregressive-infilling pass. TPS is the average number of generated tokens per second measured over the complete decoding process. Both metrics were measured on a single NVIDIA RTX 3090, with larger values indicating higher throughput. The scores match Table 1 with gains up to 3.8 points. These gains are accompanied by a moderate reduction in decoding throughput. Relative to ReFusion, SlotWeave retains 91.2%, 83.4%, 89.2%, 85.1%, and 88.6% of its TPF on GSM8K, MATH500, OlympiadBench, HumanEval, and MBPP, respectively. The corresponding TPS ratios are 93.0%, 90.0%, 89.4%, 83.8%, and 89.7%.

Table 3: Comparisons of score (%), tokens per forward pass, and tokens per second. The ratio columns are SlotWeave divided by ReFusion.

	ReFusion			SlotWeave			Ratio (%)	
Benchmark	Score	TPF	TPS	Score	TPF	TPS	TPF	TPS
GSM8K	83.7	2.96	47.25	85.0	2.70	43.96	91.2	93.0
MATH500	49.4	3.14	48.37	53.2	2.62	43.51	83.4	90.0
OlympiadBench	20.3	2.41	43.62	21.5	2.15	38.98	89.2	89.4
HumanEval	75.0	3.49	58.08	77.4	2.97	48.65	85.1	83.8
MBPP	63.6	3.08	49.09	67.0	2.73	44.02	88.6	89.7

The lower TPF is primarily a consequence of SlotWeave's more conservative slot-selection policy. Dependency-aware scheduling prevents mutually dependent slots from being accepted in the same parallel batch, which reduces the number of tokens committed per forward pass. This restriction also contributes to the method's accuracy gains: by delaying potentially conflicting slots, SlotWeave reduces errors caused by independently decoding positions that should condition on one another.

The TPF values already include the forward pass used to estimate token dependencies. Hence, the TPF ratios measure the combined algorithmic effect of dependency estimation and stricter slot selection. TPS additionally reflects the wall-clock cost of computing and processing the dependency information. In most cases, the TPS ratio is close to, or even slightly higher than, the TPF ratio. The differences are +1.8, +6.5, +0.1, -1.3, and +1.0 percentage points for GSM8K, MATH500, OlympiadBench, HumanEval, and MBPP, respectively. This comparison suggests that dependency estimation introduces relatively little computational overhead by reusing the results from the selection forward pass.

B.4 Effects of Serialization

The improvements of SlotWeave raise an interesting question: to what extent do they arise from adaptive dependency-aware scheduling, as opposed to simply decoding fewer slots per round? To examine this issue, we construct two serialized variants of ReFusion. In both variants, exactly one slot is decoded at each round, while the autoregressive infilling procedure within the selected slot remains unchanged. The first variant always selects the leftmost remaining slot, hence imposing a strict left-to-right order. The second selects one remaining slot uniformly at random. We report $s = 2$, where Table 1 shows the largest accuracy gap between SlotWeave and ReFusion.

Table 4: Accuracy (%) of serialized ReFusion. Each round decodes exactly one slot, either left to right or in random order.

Benchmark	Left-to-right	Random
GSM8K	84.7	27.7
MATH500	53.6	21.2
OlympiadBench	21.1	5.6
HumanEval	75.6	7.9
MBPP	66.8	1.8

As shown in Table 4, left-to-right serialization substantially improves ReFusion, achieving 84.7%, 53.6%, 21.1%, 75.6%, and 66.8% on GSM8K, MATH500, OlympiadBench, HumanEval, and MBPP, respectively. These results demonstrate that enforcing a left-to-right order is itself a strong intervention. In particular, it prevents a later slot from being decoded before earlier dependent slots have been resolved. However, the left-to-right serialized baseline should not be interpreted as equivalent to SlotWeave. Serialization eliminates nearly all parallelism, whereas SlotWeave attempts to preserve parallel decoding whenever the estimated dependencies permit it. Moreover, the serialized baseline imposes a globally fixed order, while SlotWeave dynamically determines which slots can be decoded together based on the current dependency structure.

The random-order variant performs substantially worse. This degradation indicates that serialization alone is not sufficient and the order in which slots are decoded is also important. A random schedule may decode a slot before the context on which it depends has been established, producing errors that subsequent rounds cannot generally correct because previously accepted tokens are not revised.

Table 5: Average number of decoded slots. Serial schedules represent the ReFusion's variants in Appendix B.4.

Benchmark	Serial	ReFusion	SlotWeave
GSM8K	1.00	1.66	1.54
MATH500	1.00	2.25	1.54
OlympiadBench	1.00	2.01	1.57
HumanEval	1.00	2.10	1.36
MBPP	1.00	1.78	1.33

Table 5 further compares the average number of slots decoded per round by SlotWeave, ReFusion and its serial variants at $s = 2$. As mentioned above, the serialized variants decode exactly one slot per round. ReFusion decodes between 1.66 and 2.25 slots per round across the five benchmarks, whereas SlotWeave decodes between 1.33 and 1.57. SlotWeave therefore retains more parallelism than the serialized baselines while using a narrower and more dependency-aware schedule than ReFusion.

Taken together, the comparison with SlotWeave reveals a task-dependent accuracy–parallelism trade-off. At $s = 2$, left-to-right serialization achieves higher accuracy on MATH500, HumanEval, and MBPP, while SlotWeave performs better on GSM8K and OlympiadBench. However, SlotWeave decodes up to 1.57 times as many slots per round as the serialized baseline. Hence, the serialized baseline provides a useful accuracy reference but does not isolate the full contribution of SlotWeave's adaptive dependency-aware scheduling mechanism. SlotWeave's advantage is that it retains additional slot-level parallelism while remaining substantially more accurate than the confidence-based schedule of ReFusion.

Overall, the results suggest that SlotWeave derives its improvements from a combination of dependency-consistent ordering and selective parallelism. Strict left-to-right serialization can achieve strong scores, but only by reducing the parallelism. Random serialization confirms that the order of decoding matters, rather than serialization alone. SlotWeave provides a compromise: it tends to preserve the accuracy benefits of ordered decoding while still decoding multiple slots in a single round when their estimated dependencies are sufficiently weak.

B.5 ATTENTION-LAYER COVERAGE

SlotWeave estimates dependencies using a suffix of the decoder layers rather than the entire network. This choice is motivated by prior studies suggesting that lower layers capture surface and local information, whereas higher layers encode longer-range syntactic and semantic dependencies (Tenney et al., 2019; Jawahar et al., 2019). Let N be the number of transformer layers, indexed from 0 to $N-1$. Given a coverage ratio ρ, SlotWeave keeps

$$n = \max\big(1,\ N - \lfloor N(1-\rho) \rfloor\big) \tag{14}$$

layers and sets

$$\mathcal{L}_{\text{attn}} = \{N-n, \ldots, N-1\}. \tag{15}$$

ReFusion-8B has $N = 36$ decoder layers. Consequently, the default setting $\rho = 0.8$ uses $n = 29$ layers, corresponding to layers 7–35. Table 6 lists the layers used in the coverage sweep, with the results reported in Table 7.

Table 6: Decoder layers included in $\mathcal{L}_{\text{attn}}$ for ReFusion-8B ($N = 36$, indices 0–35).

ρ	0.3	0.4	0.5	0.6	0.7	0.8
n	11	15	18	22	26	29
Layers	25–35	21–35	18–35	14–35	10–35	7–35

The sweep covers the five benchmarks in Table 1. We vary ρ at $s = 8$, while keeping all other settings the same as the main experiments. On GSM8K, MATH500 and MBPP, accuracy generally improves as n grows, i.e., as more layers enter $\mathcal{L}_{\text{attn}}$. A plausible explanation is that averaging over a larger suffix yields a richer token-to-token dependency estimate. GSM8K and MATH500 peak at $\rho \in \{0.7, 0.8\}$. MBPP saturates at 67.0% from $\rho = 0.5$ onward. However, the trend is not universal. On OlympiadBench, the best score is 22.6% at $\rho = 0.4$, whereas the default $\rho = 0.8$ drops to 21.5%. HumanEval likewise peaks at an intermediate ratio (78.0% at $\rho = 0.6$ versus 77.4% at $\rho = 0.8$). These results indicate that the optimal attention-layer coverage may be task dependent. Nevertheless, $\rho = 0.8$ provides a stable shared setting across the main benchmarks and is therefore used in the primary experiments.

Table 7: Sensitivity to attention-layer coverage ρ. Bold marks the best value in each row (ties included).

Benchmark	ρ=0.3	ρ=0.4	ρ=0.5	ρ=0.6	ρ=0.7	ρ=0.8
GSM8K	84.0	84.1	84.1	84.8	**85.0**	**85.0**
MATH500	52.2	51.4	51.6	52.0	**53.2**	**53.2**
OlympiadBench	22.0	**22.6**	22.0	22.3	21.7	21.5
HumanEval	76.2	75.6	77.4	**78.0**	77.4	77.4
MBPP	66.0	66.4	**67.0**	**67.0**	**67.0**	**67.0**

B.6 DEPENDENCY THRESHOLD

The dependency threshold τ_{dep} controls how conservatively SlotWeave identifies dependency between adaptive slots. A smaller threshold creates more dependency edges and therefore restricts parallel selection. A larger threshold permits more slots to be selected together.

Figure 3 already isolates early stage conservatism (τ_{dep} increasing from 0 to 0.005 versus the *fixed* τ_{dep} control) and the *w/o selection* extreme that removes all dependency edges. Table 8 instead varies only the numeric endpoint τ_{end} at $s = 8$, with all other settings the same as the main experiments. The sweep covers the five benchmarks in Table 1.

The default value $\tau_{\text{end}} = 0.005$ achieves the best or tied-best result on every benchmark in this sweep. Increasing the endpoint permits more aggressive parallel selection but generally reduces accuracy. The degradation becomes particularly pronounced for GSM8K and MATH500 at

$\tau_{\text{end}} \geq 0.05$. These results support the use of a small endpoint and suggest that increasing parallelism beyond the default level can introduce conflicts between slots whose dependencies are not adequately resolved.

Table 8: Sensitivity to the scheduled threshold endpoint τ_{end}. Bold marks the best value in each row (ties included). The default 0.005 is best or tied-best on every benchmark.

Benchmark	τ_{end}=0.005	τ_{end}=0.02	τ_{end}=0.05	τ_{end}=0.1	τ_{end}=0.2
GSM8K	**85.0**	78.0	69.7	54.1	42.4
MATH500	**53.2**	47.0	41.4	31.0	23.2
OlympiadBench	**21.5**	18.0	13.8	10.5	7.3
HumanEval	**77.4**	**77.4**	74.4	70.1	52.4
MBPP	**67.0**	66.8	64.8	57.6	42.0

C SUBJECT-LEVEL RESULTS ON THE FULL MATH TEST SET

MATH500 in Table 1 is a 500-problem subset of the Hendrycks MATH benchmark (Hendrycks et al., 2021). To evaluate whether the trends observed on MATH500 extend to a larger evaluation set, we repeat the same slot size sweep on the full 5000-problem MATH test set. We use the same answer verification protocol as in the main experiments and report results for the overall test set and its seven subject categories: Prealgebra, Algebra, Intermediate Algebra, Counting and Probability, Number Theory, Geometry, and Precalculus.

Table 9 compares SlotWeave and ReFusion across primitive slot sizes. The overall results follow the same pattern as MATH500. SlotWeave provides its largest improvement in the fine-grained regime, increasing accuracy from 43.5% to 52.1% at $s = 2$, a gain of 8.6 percentage points. The improvements are not uniform across subjects. At $s = 2$, the largest gains occur on Algebra ($+12.3$ points), Number Theory ($+11.9$ points), and Prealgebra ($+10.6$ points). SlotWeave also improves over ReFusion on Intermediate Algebra ($+5.8$ points), Counting and Probability ($+7.2$ points), and Precalculus ($+5.8$ points), while the gain on Geometry is smaller ($+2.3$ points). These results indicate that the benefit of dependency-aware scheduling may be task dependent.

Overall, the full test set results corroborate the main MATH500 trend. Dependency-aware scheduling is most beneficial with fine-grained primitive slots, and the subject-level results further suggest that the improvement is particularly pronounced for algebraic and number-oriented problems rather than being uniform across all categories.

Table 9: Performance comparison on the full MATH test set.

Method	$s = 2$	$s = 4$	$s = 8$	$s = 16$	$s = 32$
Overall					
ReFusion	43.5%	48.4%	51.6%	53.0%	53.7%
SlotWeave	52.1% +8.6%	53.7% +5.3%	53.8% +2.2%	53.7% +0.7%	53.9% +0.2%
Prealgebra					
ReFusion	58.7%	65.7%	68.0%	68.9%	69.7%
SlotWeave	69.3% +10.6%	69.9% +4.2%	69.9% +1.9%	69.8% +0.9%	69.8% +0.1%
Algebra					
ReFusion	59.2%	66.0%	70.1%	72.5%	73.0%
SlotWeave	71.5% +12.3%	72.7% +6.7%	73.6% +3.5%	73.5% +1.0%	73.5% +0.5%
Intermediate Algebra					
ReFusion	24.0%	28.5%	31.8%	32.2%	33.3%
SlotWeave	29.8% +5.8%	33.4% +4.9%	33.1% +1.3%	32.7% +0.5%	33.4% +0.1%
Counting & Probability					
ReFusion	40.7%	45.1%	45.8%	48.3%	48.7%
SlotWeave	47.9% +7.2%	49.4% +4.3%	47.7% +1.9%	48.3% +0.0%	49.2% +0.5%
Number Theory					
ReFusion	35.0%	40.2%	44.3%	47.2%	48.0%
SlotWeave	46.9% +11.9%	48.3% +8.1%	47.8% +3.5%	48.7% +1.5%	48.7% +0.7%
Geometry					
ReFusion	41.8%	41.8%	44.9%	44.1%	44.7%
SlotWeave	44.1% +2.3%	45.9% +4.1%	46.3% +1.4%	45.7% +1.6%	44.9% +0.2%
Precalculus					
ReFusion	29.5%	32.2%	36.6%	37.4%	37.9%
SlotWeave	35.3% +5.8%	36.1% +3.9%	36.8% +0.2%	36.3% -1.1%	37.4% -0.5%

D SLOTWEAVE WITHIN BLOCK DIFFUSION

D.1 EMPIRICAL OBSERVATIONS

Table 1 already reports the slot size sweep at a fixed number of serial blocks, $b = 4$. This appendix instead asks how slot-based decoding interacts with block diffusion when the primitive slot size is held fixed. We therefore vary the number of serial blocks $b \in \{1, 2, 4, 8\}$ at different primitive slot sizes on GSM8K, MATH500, OlympiadBench, HumanEval, and MBPP. The generation length remains 512 tokens. Consequently, the block length is $512/b$: 512 tokens for $b = 1$, 256 tokens for $b = 2$, 128 tokens for $b = 4$, and 64 tokens for $b = 8$. Blocks are decoded from left to right. Within each block, ReFusion applies its fixed-slot, confidence-based scheduling policy, whereas SlotWeave performs adaptive slot construction followed by dependency-aware slot selection. Figures 5 and 6 show the scores against b for mathematical reasoning and code generation, respectively. The purple curve is SlotWeave with the intra-slot dependency utility, and the blue curve is ReFusion.

On mathematical reasoning, SlotWeave improves relative to $b = 1$ and then saturates. On GSM8K at $s = 2$, it rises from 82.3% at $b = 1$ to 85.4% at $b = 2$ and then plateaus near the 85.3% reported for $b = 4$ in Table 1. The lift is smaller at larger s. MATH500 shows a clearer change with b that is not confined to the smallest slots. At $s = 4$, SlotWeave improves from 47.6% at $b = 1$ to 53.4% at $b = 8$, while reasonable lifts also appear at $s = 8$ and $s = 16$. OlympiadBench follows the same contest math pattern.

Overall, SlotWeave benefits from introducing a small number of serial block boundaries on the mathematical tasks. The gains are not monotonic in b. Performance often improves from $b = 1$ to $b = 2$ or $b = 4$ and then changes little. This suggests that serial blocking and dependency-aware slot scheduling contribute in different ways: block diffusion imposes a coarse left-to-right ordering, whereas SlotWeave addresses local dependencies within each block. On the other hand, ReFusion does not exhibit a comparable improvement and even slightly declines at some settings.

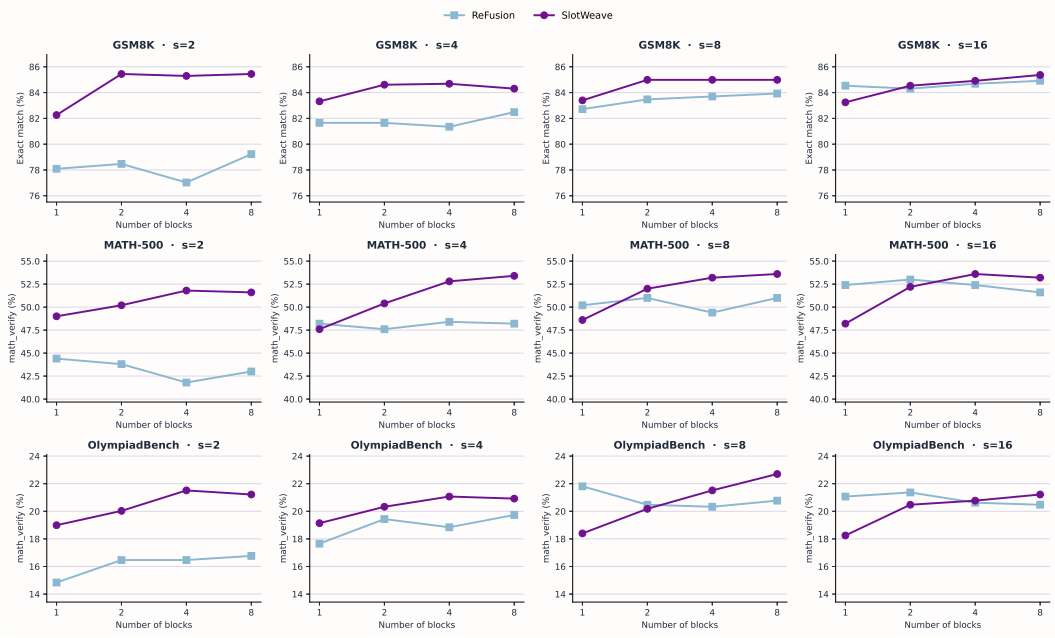


Figure 5: **Effect of the Number of Serial Blocks on Mathematical Reasoning Accuracy.** Each row is one dataset, GSM8K, MATH500, then OlympiadBench. Each column is a primitive slot size s. The curves compare ReFusion with SlotWeave.

Code generation performance is substantially less sensitive to b. On MBPP, SlotWeave moves by at most 0.8 points across $b \in \{1, 2, 4, 8\}$ at every evaluated s, and the gap versus ReFusion at $s = 2$ remains about 10 points. On HumanEval, the SlotWeave curves are similarly flat. These results

suggest that the benefit of SlotWeave on code generation tasks is primarily associated with local dependency-aware scheduling rather than with additional serial block boundaries. The ReFusion curves exhibit a similarly weak dependence on b, but ReFusion continues to lag behind SlotWeave in most of the settings.

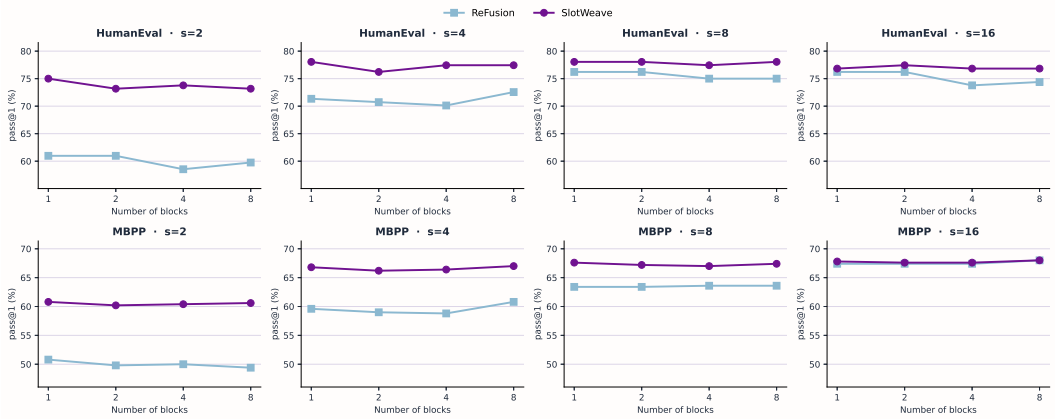


Figure 6: **Effect of the Number of Serial Blocks on Code-Generation Performance.** Each row is one dataset, HumanEval then MBPP. Each column is a primitive slot size s. The curves compare ReFusion with SlotWeave.

D.2 Discussion

The two axes play different roles in the decoder. Specifically, the number of blocks, b, determines the degree of coarse-grained autoregressive serialization across the generation window. A later block is not decoded until the preceding block has been written into the prefix. With a fixed generation budget of 512 tokens, increasing b shortens each block and introduces more cross-block ordering constraints.

By contrast, the primitive slot size s controls the granularity of scheduling within each block. SlotWeave estimates dependencies, merges neighboring primitive slots, constructs a slot-level dependency graph, and selects compatible slots for parallel infilling inside the current block. Hence, s and the SlotWeave scheduler primarily govern local coupling, whereas b governs coarse ordering across larger response regions.

The results on mathematical reasoning are consistent with these distinct roles. A small number of serial block boundaries can provide useful coarse ordering, after which additional boundaries often yield diminishing returns. For example, SlotWeave improves substantially when moving away from $b = 1$ to $b = 4$ or $b = 8$ on several mathematical tasks, but the curves frequently flatten at $b = 4$ or $b = 8$. The results also show that block diffusion and dependency-aware scheduling are not interchangeable. In particular, increasing b does not consistently recover ReFusion's deficit at small slot sizes. ReFusion may still select mutually dependent slots together within a block, even when the response is divided into multiple serial blocks. SlotWeave addresses this within-block issue through adaptive merging and dependency-aware selection.

The code generation results show a different pattern. Performance is relatively insensitive to the number of serial blocks b, while SlotWeave's advantage at small primitive slot sizes s is already present at $b = 1$. One possible explanation is that these datasets consist primarily of short function-synthesis tasks, for which the relevant dependencies often lie locally and should be handled by dependency-aware scheduling. Additional serial block boundaries may therefore provide limited benefit.

E CASE STUDIES

The following case studies, intended as qualitative illustrations of possible generation modes, are drawn from the $s = 2$ setting used in the main experiments. In every displayed example, ReFusion produces an incorrect answer and SlotWeave produces the correct one. Token backgrounds are colored by decode time, from earlier (dark) to later (light). For readability, some space and newline tokens are omitted from the visualizations. These examples illustrate how ReFusion can decode locally plausible but dependency-inconsistent tokens: it may commit a locally confident slot before the tokens it depends on have been resolved or decode mutually dependent fragments independently across slot boundaries, whereas SlotWeave merges strongly coupled slots together or defers a conflicting slot until the relevant dependencies have been resolved.

E.1 GSM8K

GSM8K solutions often contain a sequence of intermediate arithmetic operations. The final answer must therefore remain consistent with quantities computed earlier in the solution. In both examples below, ReFusion produces intermediate calculations that are largely correct but generates an inconsistent final answer.

In Figure 7a, Tyson runs 5000 meters a day and should run $1/5$ more, for a June total of 180000. ReFusion writes the intermediate facts correctly, including the extra 1000 meters, the new daily distance 6000, and the 30 days in June, but then reports 1800000. The decode time coloring shows why: the final numeral is filled early (dark), while the supporting products are completed later (light). Confidence-based selection can therefore lock in a plausible long integer before 6000×30 has been resolved. SlotWeave decodes in essentially left-to-right order: it first forms $5000 + (1/5) \times 5000 = 6000$, then $6000 \times 30 = 180000$, and only afterwards writes the answer. The answer slot remains dependent on the product that determines it.

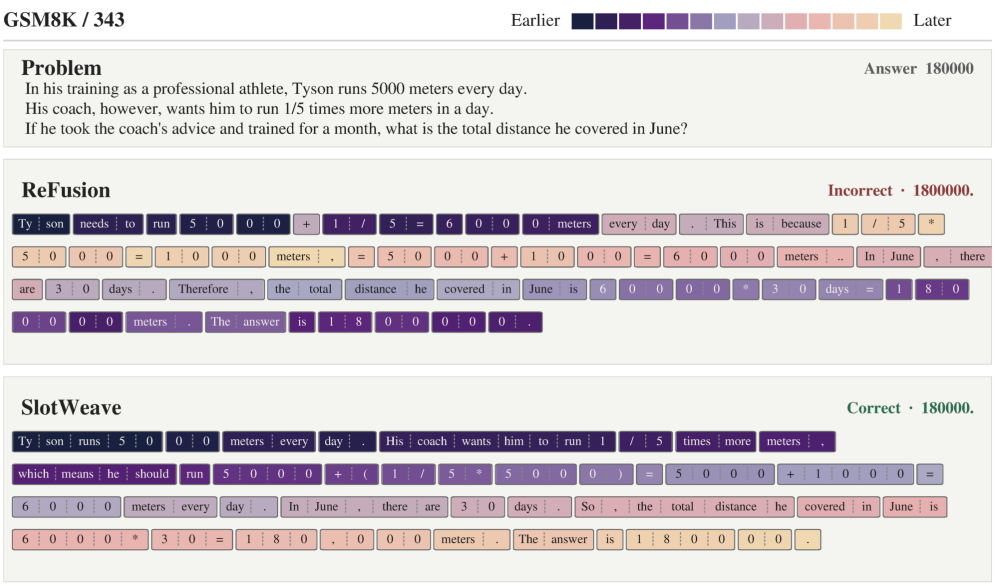
Figure 7b shows a similar copy inconsistency in ReFusion. The playlist problem asks for $20 \times 300 \times 10 = 60000$ hours. Even though ReFusion computes $20 \times 300 = 6000$ and $6000 \times 10 = 60000$ in the body, it still concludes with an inconsistent answer 660000. SlotWeave preserves the relevant units and delays the final answer until it is consistent with the completed arithmetic expression.

E.2 HUMANEVAL

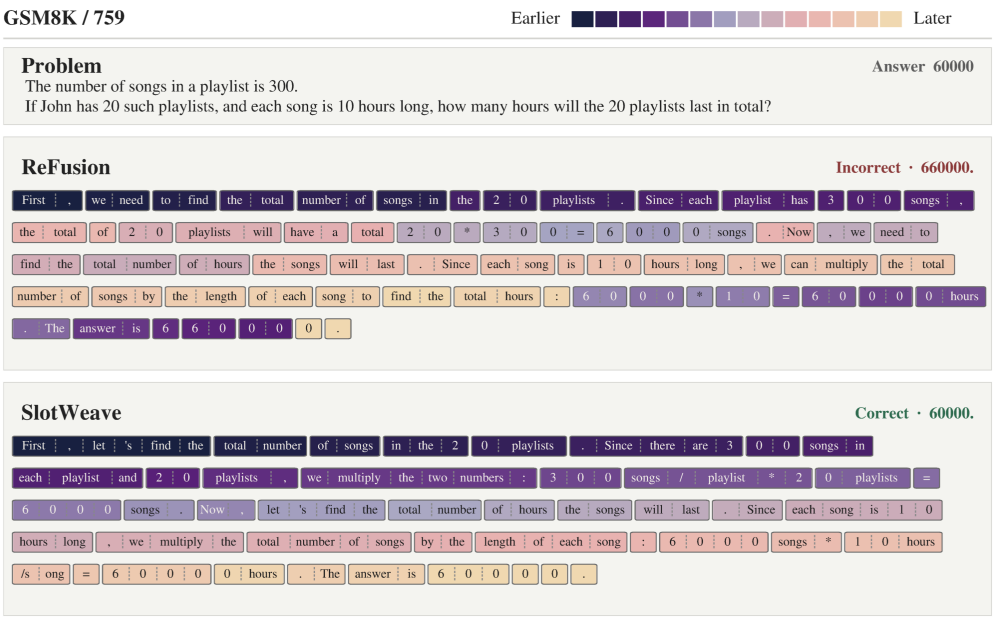
The HumanEval examples illustrate structural dependencies in code. In these cases, correctness depends on maintaining consistency among loop headers, variable references, delimiters, and template fragments.

In Figure 8a, `get_row` must walk a ragged 2D list and return coordinates of x, sorted by increasing row and decreasing column. The loop header generated by ReFusion is malformed: `for i in range(len(len(lst)):`. The expression contains an unnecessary nested `len` call and an unmatched parenthesis, making the loop syntactically invalid. This error may be related to the division of the loop header across slot boundaries. The tokens forming `range(len(lst))` are structurally dependent: the loop range, the list object, and the matching parentheses must form a single valid expression. When these fragments are decoded independently, a locally plausible continuation such as an additional `len(` can be selected without preserving the syntax of the complete header. By contrast, SlotWeave maintains a consistent relationship between the outer loop variable, the row being traversed, the inner loop variable, and the corresponding element value. It therefore produces a valid loop structure and uses the row and column indices consistently in the subsequent sorting operation. This example illustrates how dependency-aware slot construction and selection can reduce conflicts among tokens that jointly determine the structure of a code expression.

Figure 8b requires a rigid template: the code must replace the character `i` in each word with the number of odd digits in a corresponding input. ReFusion counts odd digits correctly, but it omits the word `the` from the generated output. This error may result from decoding the template fragments in parallel. SlotWeave maintains the relevant template structure and produces the complete output. This example illustrates that dependency-aware scheduling can help preserve not only computationally important tokens but also structurally necessary tokens in fixed-format outputs.

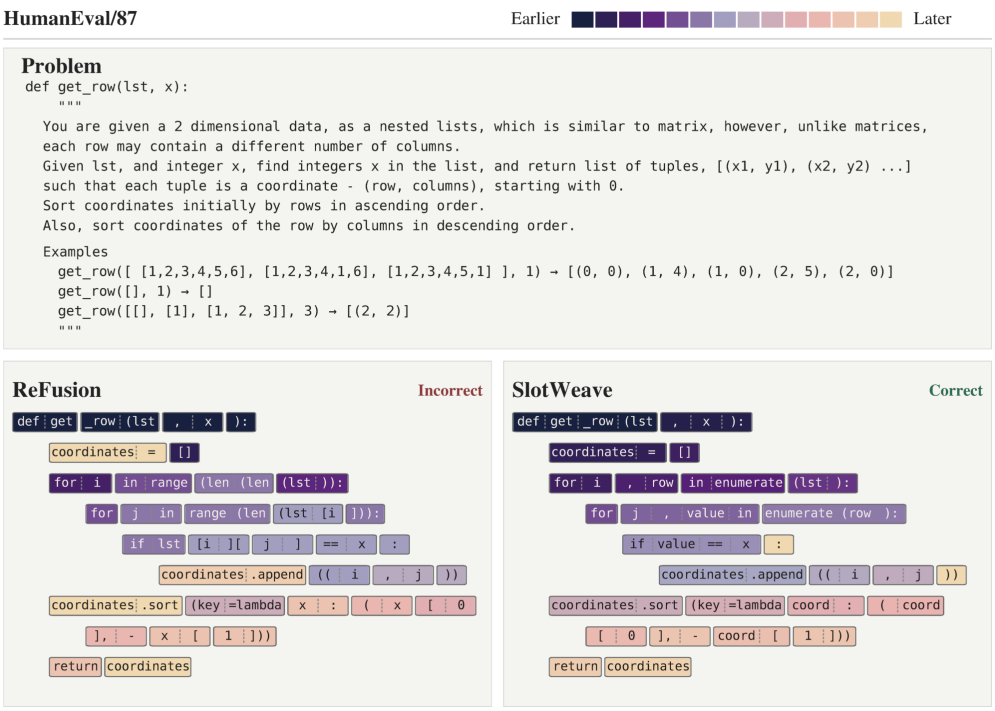


(a) GSM8K problem 343.

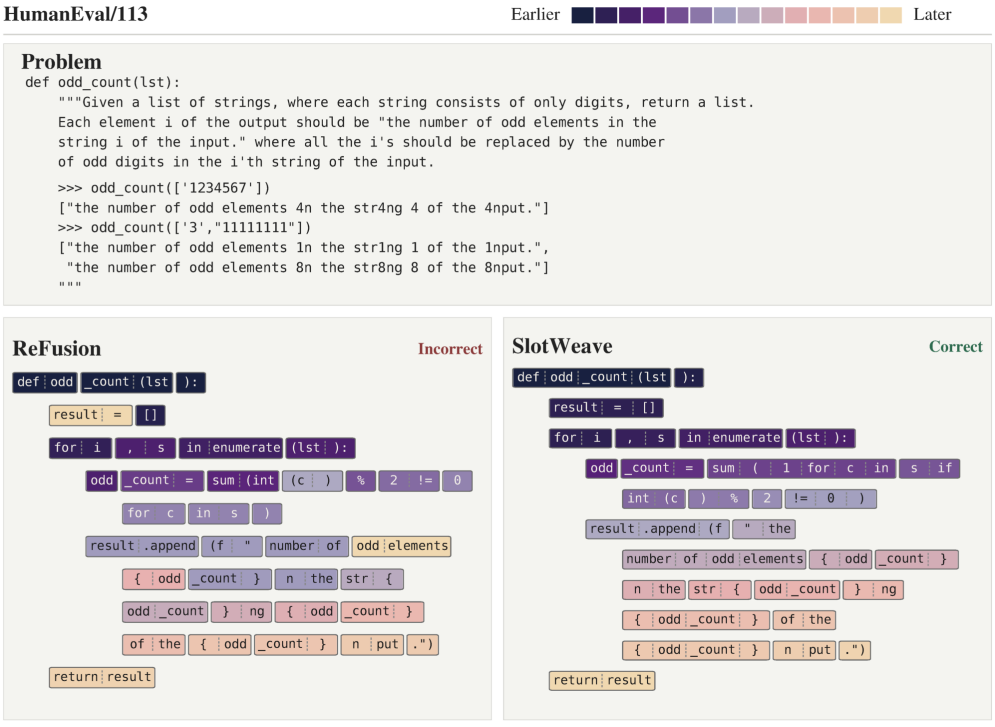


(b) GSM8K problem 759.

Figure 7: **GSM8K case studies.** Token backgrounds are colored by decode time, from earlier (dark) to later (light).



(a) HumanEval problem 87.



(b) HumanEval problem 113.

Figure 8: **HumanEval case studies.** Token backgrounds are colored by decode time, from earlier (dark) to later (light).