CONFIDENCE-ADAPTIVE PERMUTATION-TREE SELF SPECULATIVE DECODING FOR DIFFUSION LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion-based Large Language Models (dLLMs) offer a promising alternative to autoregressive models by enabling parallel decoding and bidirectional context modeling. However, existing speculative decoding methods for dLLMs rely on linear or heuristic verification structures that fail to capture the full space of valid generation orders within a block, leading to premature rejection of correct tokens and suboptimal acceptance rates. In this work, we propose **CAPSpec** (**C**onfidence-**A**daptive **P**erm-tree **Spec**ulative Decoding), a training-free framework targeting block-based diffusion models. CAPSpec exploits the key insight that, within a fixed-length block, the n remaining masked positions admit $n!$ possible decoding orders, forming a full permutation tree whose size grows factorially. By first constructing this complete permutation tree over a block's decoding trajectories and then applying **Adaptive Confidence Partitioning (ACP)**—which clusters candidate positions by their draft-stage confidence and prunes low-value branches before verification—CAPSpec compresses the factorial-sized search space into a compact tree that retains all high-probability trajectories. A single batched forward pass over the pruned tree evaluates the retained trajectories and identifies a unique accepted path. Extensive experiments on LLaDA-8B-Instruct across mathematics, code, and dialogue benchmarks demonstrate that CAPSpec achieves up to $3.08\times$ wall-clock speedup over step-by-step baselines, establishing a new state of the art for training-free speculative decoding in block-based diffusion language models.

1 INTRODUCTION

Autoregressive large language models (ARMs) dominate natural language generation but suffer from high latency due to their inherent sequential, token-by-token decoding. In contrast, diffusion-based LLMs (dLLMs) like LLaDA and Dream Nie et al. (2025); Ye et al. (2025) formulate generation as parallel iterative denoising over masked sequences. By leveraging bidirectional context, dLLMs offer a promising alternative for reducing decoding latency, a potential further realized by recent parallel decoding techniques such as confidence-based dynamic unlocking Wu et al. (2025b); Wei et al. (2025). However, modern inference systems such as vLLM's PagedAttention Kwon et al. (2023) and SGLang's RadixAttention Zheng et al. (2023) are designed for causal KV-cache reuse and do not directly accommodate the bidirectional attention and iterative refinement of dLLMs. Although dKV-Cache Ma et al. (2025) and fast-dLLM Wu et al. (2025b) introduce cache-aware acceleration, redundant computation and inefficient hardware utilization remain unresolved.

Speculative decoding effectively mitigates memory bottlenecks in ARM inference by decoupling generation into a lightweight drafting stage and a parallel verification stage. Recent self-speculative methods further reduce overhead, where Medusa Cai et al. (2024) employs additional prediction heads and the EAGLE series Li et al. (2024a) Li et al. (2024b) Li et al. (2025b) reuses target model hidden states. However, applying this to dLLMs is non-trivial, as their iterative, confidence-based unmasking diverges from the strict left-to-right autoregressive pattern, necessitating structurally distinct drafting and verification mechanisms. While adapting traditional speculative decoding to diffusion models is difficult, recent paradigms leverage their inherent parallelism to generate efficient drafts. DFlash Chen et al. (2026) employs a lightweight block diffusion drafter conditioned on target ARM semantic features, achieving a $5.1\times$ speedup in SGLang on Qwen3-8B. Similarly, Self-Speculative Decoding

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.

(SSD) Gao et al. (2025) exploits parallel draft generation via hybrid verification trees that combine linear and out-of-order structures.

Optimizing tree-based verification mechanisms is crucial for addressing the bottlenecks of computational waste and low acceptance rates in draft token verification. SpecInfer Miao et al. (2023) leverages a single forward pass to parallelly verify pre-defined Top-k candidate trees, thereby maximizing hardware utilization. Building on this, DYSPEC Xiong et al. (2024) proposes a dynamic tree construction algorithm based on greedy strategies. By online-optimizing the tree structure, DYSPEC significantly improves token acceptance rates while maintaining efficient parallel verification.

As shown in Figure 2, increasing the number of tokens decoded per denoising step reduces accuracy across the evaluated SDAR and LLaDA model families on GSM8K, while one-token-per-step decoding preserves the highest generation quality, consistent with the conditional-independence limitation of parallel decoding Wu et al. (2025b). These results highlight the need to balance accuracy and decoding speed in block-based dLLMs.

Based on this observation, we propose **CAPSpec** (**C**onfidence-**A**daptive **P**erm-tree **Spec**ulative Decoding), a one-token-per-step speculative decoding method that accelerates dLLM generation. Under single-position unmasking, the n remaining masked positions within a block give rise to $n!$ possible decoding orders—a factorial-sized space that existing linear or heuristic verification structures explore only partially, causing premature rejection of valid tokens. First, **Permutation Tree Construction** enumerates the complete set of block decoding trajectories as a full permutation tree, ensuring that every valid generation order is represented. Second, **Adaptive Confidence Partitioning (ACP)** clusters the remaining positions by their draft-stage confidence scores and prunes low-value branches *before* the verification forward pass, compressing the factorial-sized tree into a compact structure that retains all high-probability trajectories while bounding computational overhead. The pruned tree is verified in a single batched forward pass to select an accepted decoding path. Experiments on LLaDA-8B-Instruct demonstrate up to $3.08\times$ wall-clock speedup over step-by-step baselines.

Our contributions are summarized as follows:

- We identify a fundamental limitation of existing speculative decoding for block-based dLLMs: linear and heuristic verification structures explore only a fraction of the $n!$ valid decoding orders within a block, leading to premature token rejection and low acceptance rates.
- We propose **Permutation Tree Construction**, which builds the complete permutation tree over all block decoding trajectories via confidence-ordered BFS expansion, ensuring that every valid generation order is pre-covered by the full tree before pruning.
- We propose **Adaptive Confidence Partitioning (ACP)**, a depth-aware strategy that clusters shallow-layer candidates by draft-stage confidence and applies aggressive top-k pruning at deeper layers, compressing the factorial search space into a more compact verification tree.
- We integrate these components into **CAPSpec**, achieving up to $3.08\times$ speedup on LLaDA-8B-Instruct across mathematics, code, and dialogue benchmarks.

2 Related Work

2.1 Diffusion Large Language Models

Diffusion Large Language Models (dLLMs) generate tokens via iterative denoising, offering a parallel alternative to ARMs that captures bidirectional context. Large-scale discrete diffusion models like LLaDA Nie et al. (2025) achieve performance comparable to billion-parameter AR models such as LLaMA-3. However, pure parallel diffusion faces challenges including fixed generation lengths, inefficient Key-Value (KV) cache usage, and high latency from multiple denoising steps. To address this, block diffusion Arriola et al. (2025), SDAR Cheng et al. (2025), and Fast-dLLM v2 (Wu et al., 2025a) employ semi-autoregressive block generation and cache. These hybrid architectures combine parallelism with AR structures, enabling variable-length generation and efficient caching. Despite advances in inference optimization, post-training alignment (e.g., diffu-GRPO, d1 Zhao et al. (2025)), and multimodal extensions like LLaDA-V You et al. (2025), dLLMs still lag behind top-tier AR

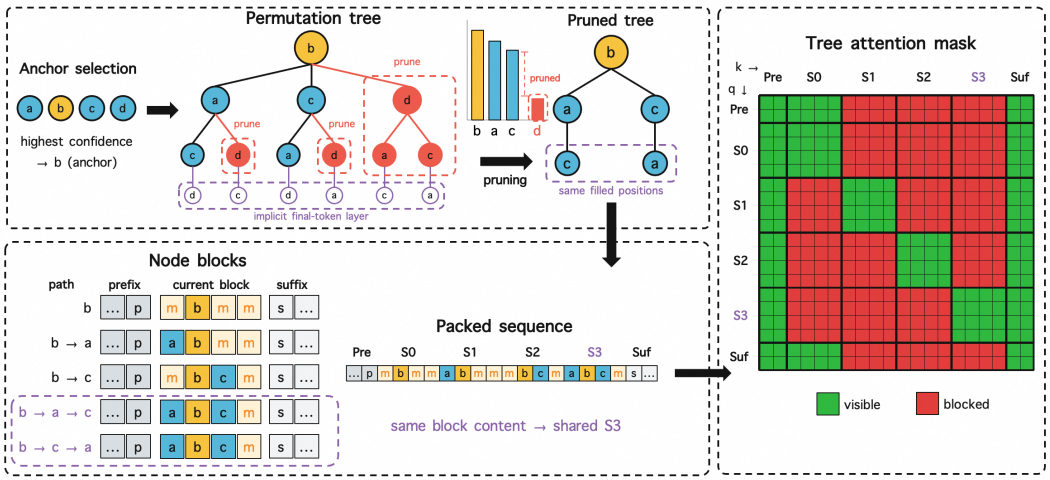


Figure 1: Overview of CAPSpec pruning and deduplicated packed verification. Starting from the block-size-four permutation tree rooted at anchor b, shallow ACP removes low-confidence sibling subtrees while retaining candidate trajectories. Logically distinct states with identical filled blocks then share one physical segment, and a tree-aware attention mask enables their verification in a single parallel target-model forward pass.

models in long-sequence handling, infrastructure maturity, and performance ceilings. Particularly in complex reasoning, dLLMs require more computational steps, limiting their deployment in latency-sensitive scenarios.

2.2 Speculative Decoding for dLLMs

Recent work employs diffusion models for speculative drafting. FailFast Pan et al. (2025a) adapts draft length via a confidence threshold τ, extending up to 70 tokens in high-confidence regions for tuning-free acceleration. However, coarse generation yields low acceptance rates (34%–40%), relying on long drafts in simple regions to amortize verification overhead, which limits benefits in complex reasoning. DiffuSpec Li et al. (2025a) addresses causal inconsistency and fixed lengths via causal consistency path search (CPS), filtering left-to-right paths from token lattices. Yet, its reliance on beam search and external proxies like n-grams expands the search space, potentially offsetting parallel speedups. SSD Gao et al. (2025) leverages dLLMs for self-drafting, using hierarchical trees to batch-verify tokens for a $3.46\times$ speedup. However, its greedy linear verification is sensitive to token order. While multi-branch trees improve acceptance, they drastically increase batch size and memory, limiting gains.

3 Preliminaries

3.1 Block-Based dLLM Decoding Strategies

Block-based dLLMs such as the LLaDA family divide the generation region into fixed-size blocks and denoise each block over a prescribed number of steps. The number of tokens committed at each step is jointly determined by the block size and the number of denoising steps. When the number of steps equals the block size, the decoder commits one token per step. When the number of steps is smaller than the block size, the decoding schedule must commit more than one token per step on average, and the usual evenly distributed schedule reveals multiple tokens in each step. Thus, for a fixed block size, reducing the number of steps increases the per-step decoding count and exposes greater parallelism.

For block-based dLLMs, revealing more than one token in a denoising step reduces the number of sequential model evaluations, but those tokens cannot benefit from the intermediate contexts

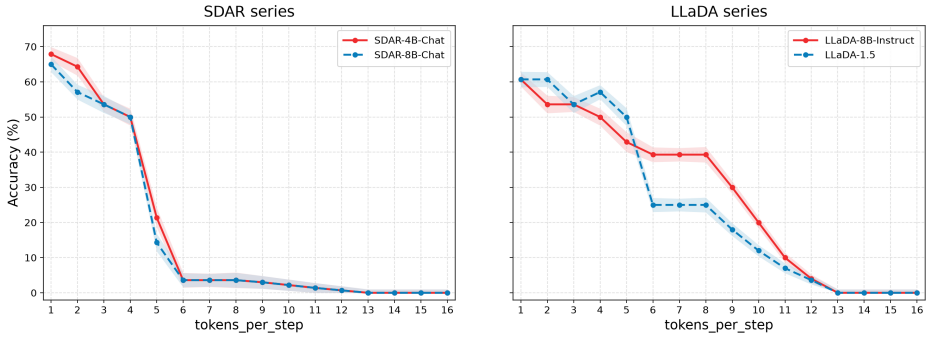


Figure 2: Task accuracy of a block-based dLLM as the number of tokens decoded per denoising step increases. Although larger per-step decoding counts expose more parallelism, they reduce accuracy on the evaluated tasks, motivating speculative acceleration built on the one-token-per-step greedy decoding baseline.

that one-token decoding would create between reveals. As shown in Figure 2, our evaluation results show that task accuracy decreases as the number of tokens decoded per step increases. This quality–parallelism trade-off makes the one-token-per-step setting a meaningful high-quality basis for studying speculative decoding: multiple draft tokens are filtered through parent-conditioned verification rather than committed through an unchecked parallel update.

3.2 Step Generation Formulation

We study speculative acceleration relative to one-token-per-step greedy decoding. This regime exposes every newly decoded token to all subsequent predictions and therefore provides a high-quality target trajectory for block-based dLLMs. Let x_t denote the sequence at time step t, with t decreasing from T to 1, and let $\mathcal{M}_t = \{n \mid x_t^{(n)} = [\text{MASK}]\}$ be its unresolved positions. A single model forward pass simultaneously produces the logits $f_\theta(x_t)^{(n)}$ for every $n \in \mathcal{M}_t$. These logits define the probability vector $P_t^{(n)} = \text{Softmax}\big(f_\theta(x_t)^{(n)}\big) \in \mathbb{R}^{|V|}$, the greedy candidate token $V_t^{(n)} = \text{Argmax}(P_t^{(n)})$, and its confidence $C_t^{(n)} = \|P_t^{(n)}\|_\infty$ at every masked position.

Let k_t denote the number of tokens decoded at step t, and let $\mathcal{U}_t \subseteq \mathcal{M}_t$ contain the k_t positions with highest confidence, so $|\mathcal{U}_t| = k_t$. Although candidates and confidences are predicted for all masked positions in parallel, only positions in $\mathcal{U}_t$ are committed through the state update

$$x_{t-1}^{(n)} = \begin{cases} V_t^{(n)}, & n \in \mathcal{U}_t, \\ x_t^{(n)}, & n \notin \mathcal{U}_t. \end{cases} \tag{1}$$

In the decoding strategy studied in this work, $k_t = 1$. Hence, $\mathcal{U}_t = \{n_t\}$ with $n_t = \arg\max_{n \in \mathcal{M}_t} C_t^{(n)}$, using a fixed deterministic rule for confidence ties. Equation (1) therefore commits only the single highest-confidence candidate, while every other masked position remains unresolved for the next forward pass. Repeating this update yields a sequential greedy trajectory in which each new prediction is conditioned on all previously revealed tokens.

3.3 Diffusion Model Limitation

Predictive difficulty is nonuniform across a generation trajectory. Evidence from Large Reasoning Models shows that simple intermediate steps can be delegated to speculative execution, whereas a smaller number of planning-intensive steps require stronger inference-time computation Pan et al. (2025b). This easy–difficult asymmetry also motivates confidence-adaptive allocation in dLLMs: high-confidence positions can support broader speculative branching, while low-confidence positions require a more selective verification budget.

The parallel prediction mechanism of dLLMs often induces *out-of-order generation*. Due to the Curse of Parallel Decoding Wu et al. (2025b), the conditional independence assumption $p(\boldsymbol{x}_{t-1}^{(i)} \mid \boldsymbol{x}_t) \cdot p(\boldsymbol{x}_{t-1}^{(j)} \mid \boldsymbol{x}_t) \neq p(\boldsymbol{x}_{t-1}^{i,j} \mid \boldsymbol{x}_t)$ neglects inter-token dependencies, potentially yielding semantically inconsistent outputs. As the denoising process progresses, evolving context shifts confidence scores, causing the actual generation order to deviate from the initial draft ranking. For example, when generating a phrase like "at the farmers," the draft may propose tokens in confidence order ["at", "farmers", "the"], but after verifying "at", the target model may rank "the" higher than "farmers" due to contextual shifts. Because either continuation may be favored after a context update, strict linear verification fixed to the initial ranking can reject otherwise reusable draft tokens.

4 METHOD

4.1 FULL-TRAJECTORY PERMUTATION TREE

Under the $k_t = 1$ confidence-greedy policy in Equation (1), each realized state uniquely determines the next revealed position, but the initial forward pass cannot know the future order because every committed token changes the context and remaining confidences. Completing a block of size m therefore requires m sequential denoising forwards. The initial pass already predicts each masked position's draft token and confidence, which CAPSpec reuses to construct self-speculative candidates for joint packed verification. A speculative iteration thus uses one drafting and one verification evaluation, reducing sequential forwards when multiple tokens are accepted.

Each draft token is evaluated under its corresponding previously unmasked state while verification preserves the reveal order. Because the context-dependent future order is unknown at draft time, CAPSpec tests alternatives through a rooted permutation tree: the current block state is the root, each edge reveals one masked position, and each root-to-leaf path is a candidate trajectory whose verified prefix can be committed. Figure 1 shows a size-four block where, after committing anchor b, the tree enumerates every order of a, c, and d. The purple final-token layer is not materialized because terminal-node logits recover it as the bonus token. This structure preserves intermediate conditioning while testing reveal orders in parallel.

For $m \geq 1$ unresolved positions, the full tree is confidence-agnostic: any unselected position may be revealed next. At depth k, exactly k positions are filled, giving

$$N_k = \frac{m!}{(m-k)!}, \quad 0 \leq k \leq m, \qquad N_{\text{full}} = \sum_{k=0}^{m} \frac{m!}{(m-k)!} = m! \sum_{j=0}^{m} \frac{1}{j!} \approx e\, m!. \tag{2}$$

Each depth-m leaf is a fully filled block and each root-to-leaf path a complete trajectory. Appendix A gives the full definition and derivation.

4.2 ADAPTIVE CONFIDENCE PARTITIONING

Although the full permutation tree contains every reveal order and can serve directly as the verification tree, its factorial growth in Equation (2) can push packed verification beyond the hardware-efficient compute region, making verification overhead outweigh the saved forwards. Every admissible step-by-step order appears as a complete root-to-leaf path, while parent-conditioned verification evaluates prefixes at their corresponding intermediate states and accepts matching drafted edges. CAPSpec therefore restricts and adaptively prunes candidates before verification.

It selects the K highest-confidence unresolved positions and orders them as $\mathcal{K}_t = \{n_0, \ldots, n_{K-1} \mid C_t^{(n_0)} \geq \cdots \geq C_t^{(n_{K-1})} \geq C_t^{(j)},\ \forall j \notin \mathcal{K}_t\}$. CAPSpec commits $a_t = n_0$ as the anchor and defines $\mathcal{R}_t = \mathcal{K}_t \setminus \{a_t\}$ with $m = |\mathcal{R}_t| = K - 1 \geq 1$. The m-position tree from Section 4.1 is then instantiated over the still-masked positions in $\mathcal{R}_t$.

At shallow levels, a confidence gap may separate locally competitive siblings from low-confidence candidates. Retaining every sibling would otherwise preserve entire low-confidence subtrees. DB-SCAN keeps the highest-confidence sibling group and removes lower-confidence groups with their descendant subtrees before deep-layer pruning. In Figure 1, this discards the d-first branch while retaining the a-first and c-first alternatives.

Each retained node corresponds to a sequence-state segment for packed verification. A depth-k node follows the ordered, no-replacement prefix $\boldsymbol{n}_{1:k} = (n_{i_1}, \ldots, n_{i_k})$ and is identified by $v_{t,\boldsymbol{n}_{1:k}} \triangleq \big(\boldsymbol{n}_{1:k}, \tilde{x}_{t,\boldsymbol{n}_{1:k}}\big)$, with snapshot

$$\tilde{x}^{(j)}_{t,\boldsymbol{n}_{1:k}} = \begin{cases} V^{(j)}_t, & j \in \{a_t, n_{i_1}, \ldots, n_{i_k}\}, \\ x^{(j)}_t, & \text{otherwise.} \end{cases} \tag{3}$$

The prefix remains as path memory because different reveal orders can yield the same snapshot but different parent-conditioned acceptance decisions. The fully filled level is omitted: expansion stops at depth $m-1$, and terminal-node verification logits recover the last unresolved position as a bonus token. Thus, the explicit tree contains $N_{\text{explicit}} = \sum_{k=0}^{m-1} m!/(m-k)! = m! \sum_{j=1}^{m} 1/j! \approx (e-1)m!$ nodes while recovering a complete trajectory.

ACP scores each child u of a retained non-leaf parent by the original single-position confidence from Section 3.2: with $\delta(u)$ denoting its newly added position, $\rho_t(u) = C_t^{(\delta(u))}$ for $u \in \mathrm{Ch}(v)$. This precomputed confidence is reused across permutation prefixes. Since context updates can reorder similarly confident positions, fixing a linear path to the initial ranking may terminate verification prematurely. This can lower acceptance by rejecting reusable candidates after a local rank swap. Grouping similar-confidence siblings preserves competitive reveal orders while pruning groups separated by a large gap, as motivated in Section 3.3.

For a shallow parent with $q_v = |\,\mathrm{Ch}(v)| \geq 3$, let $\rho_{(1)} \geq \cdots \geq \rho_{(q_v)}$ be the sorted child confidences. ACP sets $\epsilon_v = \alpha \max_{1 \leq j < q_v} \big(\rho_{(j)} - \rho_{(j+1)}\big)$, where α controls the clustering radius, and forms

$$\mathcal{G}_t(v) = \mathrm{DBSCAN}(\{\rho_t(u) : u \in \mathrm{Ch}(v)\}; \epsilon_v) = \{G_v^{(1)}, \ldots, G_v^{(r_v)}\}, \tag{4}$$

ordered by decreasing maximum confidence. Below switch depth d_{sw}, ACP retains the highest-confidence group, $\mathrm{Ch}_{\text{ACP}}(v) = G_v^{(1)}$. Appendix B gives the default parameters, equivalent gap-partition implementation, and boundary cases.

4.3 Depth-Aware ACP Post-Pruning

Shallow DBSCAN pruning removes clearly inferior branches, but the surviving branching factor still compounds across successive levels, so deep expansion remains exponential in depth. Moreover, a deep node can contribute accepted tokens only when every preceding edge on its root-to-node path is accepted, making useful deep continuations progressively less likely. CAPSpec therefore preserves multiple high-quality alternatives near the root through DBSCAN and switches to an aggressive depth-dependent top-k policy for deeper levels. The retained width decreases with depth and eventually degenerates to top-1, preventing verification capacity from being spent on low-value deep branches.

At child depth $d \geq d_{\text{sw}}$, the current implementation retains the $k(d) = \max\{1, k_{\text{deep}} - (d - d_{\text{sw}})\}$ highest-confidence children. With $d_{\text{sw}} = 3$ and $k_{\text{deep}} = 2$ by default, it keeps two children at depth 3 and one child at every depth $d \geq 4$. Starting from the anchor root, the policy processes retained parents with a FIFO queue and enqueues only the children selected by the applicable shallow or deep rule. Removing a child therefore also removes all of its descendants, preserving a connected pruned tree while concentrating deeper capacity on the strongest continuations.

4.4 Deduplicated Packed Verification

After applying the construction budget in Appendix C and ACP pruning, let $\mathcal{V}_t^{\text{ACP}} = \{u_0, \ldots, u_{N_t^{\log}-1}\}$ denote the retained logical states, where $u_0 = v_{t,\emptyset}$ is the anchor root. Each node materializes an all-mask segment $\tilde{\boldsymbol{x}}_{t,u}$ with fixed draft tokens at its applied positions. Nodes with the same applied-position set therefore produce the same filled block. The packer keeps one representative per distinct block, $\{r_0, \ldots, r_{N_t^{\text{seg}}-1}\} \subseteq \mathcal{V}_t^{\text{ACP}}$, where $N_t^{\text{seg}} \leq N_t^{\log}$, and records the many-to-one map $\pi_t : \mathcal{V}_t^{\text{ACP}} \to \{0, \ldots, N_t^{\text{seg}}-1\}$. The packed input is $\boldsymbol{z}_t = \boldsymbol{x}_t^{\text{pre}} \oplus \tilde{\boldsymbol{x}}_{t,r_0} \oplus \cdots \oplus \tilde{\boldsymbol{x}}_{t,r_{N_t^{\text{seg}}-1}} \oplus \boldsymbol{x}_t^{\text{suf}}$, with length $L_t = \ell_t^{\text{pre}} + N_t^{\text{seg}} b_t + \ell_t^{\text{suf}}$ for shared prefix and suffix lengths ℓ_t^{pre} and ℓ_t^{suf} and block size b_t.

At a fixed depth, several retained logical prefixes may apply the same positions in different orders. Their path identities must remain distinct because edge acceptance is parent-conditioned, but their materialized blocks are identical because every applied position carries the same precomputed draft token. The packer therefore counts unique applied-position sets, not logical histories: it evaluates one physical segment per distinct filled block, then uses π_t to route those logits back to all corresponding logical parents.

For each non-root node u, verification reads $\widehat{P}^{(\delta(u))}_{t,p(u)}$ at the newly introduced position $\delta(u)$ from the physical segment $\pi_t(p(u))$ of its logical parent. Duplicate states thus share physical logits without merging their logical edges or parent chains. With the same deterministic Argmax tie-break in drafting and verification, an edge is accepted when $A_t(u) = \mathbf{1}\Big[V_t^{(\delta(u))} = \text{Argmax}\Big(\widehat{P}^{(\delta(u))}_{t,p(u)}\Big)\Big]$.

CAPSpec follows accepted children from the anchor to the last accepted node; Appendix A.1 gives the selection rule and uniqueness proof. The terminal node's verification logits select the highest-confidence bonus token among its unapplied positions, avoiding another tree level.

Figure 1 shows that paths $b \to a \to c$ and $b \to c \to a$ retain distinct logical histories but share the filled block represented by segment $S3$. Both logical nodes remain in the tree, while only one physical segment is packed and evaluated. This sharing is exact because each segment query attends only to the shared prefix and suffix and its own segment, so identical filled blocks receive identical model inputs and logits. The map π_t supplies the physical logits, while acceptance and path traversal remain on the logical tree.

The boolean mask has shape $1 \times 1 \times L_t \times L_t$ and marks an entry visible under three conditions: the key lies in the shared prefix or suffix, the query and key lie in the same segment, or a shared prefix/suffix query attends to the anchor segment $\pi_t(u_0)$. These rules preserve each node state while blocking leakage across sibling segments. Converted to a $0/-\infty$ bias, the mask produces logits for all unique retained parent states in one packed target-model forward pass rather than one invocation per logical node.

5 EXPERIMENTS

5.1 SETUP

Models. We evaluate dLLMs that support block-wise decoding, including iLLaDA-8B-Instruct, LLaDA-8B-Instruct Nie et al. (2025), LLaDA-1.5 Zhu et al. (2025), SDAR-8B-Chat, and SDAR-4B-Chat Cheng et al. (2025). Unless otherwise noted, all experiments are conducted on 8× NVIDIA A100 GPUs.

Datasets. We evaluate on three categories of tasks: *Mathematics*: GSM8K Cobbe et al. (2021); *Code*: HumanEval Chen et al. (2021), MBPP Austin et al. (2021); *Dialogue*: Alpaca Taori et al. (2023). All benchmarks are evaluated under zero-shot settings.

Metrics. We report throughput in tokens per second (TPS), draft acceptance rate, the number of tree nodes constructed for verification, and the number of accepted nodes. The `<eos>` and `<eot>` termination tokens remain enabled so that each model can end its output correctly.

Baselines. We use each diffusion model's native one-token-per-step generation (Step) as the reference baseline. Following the expansion construction of SSD Gao et al. (2025), we instantiate fixed-width variants that retain the top one or top two candidates at each level and denote them Top-K1 (K1) and Top-K2 (K2), respectively; these labels indicate only the retained width rather than new baseline methods. Accordingly, K1 and K2 are the only speculative baselines in our evaluation that directly extend native one-token-per-step generation.

Hyperparameters. We set the generation length to 512 and the block size to 8. Within each block, the speculative draft tree adapts automatically to the number of masked positions that remain unresolved. The DBSCAN radius scale is $\alpha = 0.4$. We set `switch_depth`$= 3$ to control the transition to aggressive layer-wise pruning: when depth 3 is present, ACP applies top-2 pruning at that depth, while every depth from 4 onward uses top-1 pruning.

Table 1: Main throughput results across mathematics, code, and dialogue benchmarks. Step denotes native one-token-per-step decoding; K1 and K2 denote unlimited-budget fixed-width Top-k drafting with $k = 1$ and $k = 2$; and ACP denotes CAPSpec with Adaptive Confidence Partitioning. Values are throughput in tokens per second (TPS); green annotations report the ACP speedup over Step.

Model	**GSM8K**				**HumanEval**				**MBPP**				**Alpaca**			
	Step	K1	K2	ACP	Step	K1	K2	ACP	Step	K1	K2	ACP	Step	K1	K2	ACP
LLaDA-8B-Instruct	10.01	**32.93**	29.03	30.82 3.08×	15.57	27.03	23.78	**27.65** 1.78×	15.80	28.29	24.90	**28.88** 1.83×	12.55	21.36	18.80	**22.02** 1.76×
LLaDA-1.5	10.62	**32.18**	28.32	30.84 2.90×	15.60	26.01	22.89	**26.61** 1.71×	16.80	28.24	24.85	**28.83** 1.72×	11.51	21.26	18.71	**21.92** 1.90×
iLLaDA-8B-Instruct	11.33	**30.15**	26.85	28.72 2.54×	13.27	22.73	20.23	**23.25** 1.75×	13.27	26.71	23.78	**27.27** 2.05×	13.30	19.83	17.65	**20.45** 1.54×
SDAR-4B-Chat	12.08	21.86	21.70	**22.44** 1.86×	11.96	22.38	22.15	**22.89** 1.91×	12.24	18.33	18.15	**18.71** 1.53×	11.91	14.68	14.54	**15.13** 1.27×
SDAR-8B-Chat	12.15	20.10	19.95	**20.65** 1.70×	11.20	20.35	20.15	**20.81** 1.86×	12.74	18.21	18.03	**18.59** 1.46×	11.84	15.55	15.40	**16.03** 1.35×

Table 2: Post-pruning tree statistics and throughput for LLaDA-8B-Instruct. Tree and accepted-node counts are averages, and utilization is their ratio. Bold marks the highest TPS within each dataset.

Dataset	**Tree Nodes**			**Accepted Nodes**			**Utilization**			**TPS**		
	K1	K2	ACP	K1	K2	ACP	K1	K2	ACP	K1	K2	ACP
GSM8K	7.4	112.5	38.6	5.6	5.8	5.6	75.7%	5.2%	14.5%	**32.93**	29.03	30.82
HumanEval	7.3	110.2	36.8	4.3	4.5	5.0	58.9%	4.1%	13.6%	27.03	23.78	**27.65**
MBPP	7.3	109.8	37.1	4.4	4.6	5.1	60.3%	4.2%	13.7%	28.29	24.90	**28.88**
Alpaca	7.2	108.7	35.4	4.1	4.3	4.8	56.9%	4.0%	13.6%	21.36	18.80	**22.02**

Node-Count Control. For the main throughput comparison, all strategies use an effectively unlimited construction budget (`budget_K`$= 9999$), followed by post-pruning and block deduplication.

5.2 Main Results

As shown in Table 1, ACP consistently improves throughput over native one-token-per-step decoding across all 20 model–dataset combinations, with speedups ranging from $1.27\times$ to $3.08\times$. ACP achieves the highest TPS in 17 of the 20 comparisons. The exceptions are the three LLaDA-family models on GSM8K, where K1 is 4.3%–6.8% faster than ACP; ACP remains best for both SDAR models on GSM8K and for every evaluated model on HumanEval, MBPP, and Alpaca.

ACP also exceeds K2 in all 20 comparisons by 3.1%–17.2%, while K2 remains below K1 throughout. These results indicate that adaptive shallow branching is beneficial across code and dialogue tasks and for the SDAR models on mathematics, whereas a single-path draft can be more efficient for the LLaDA-family models on GSM8K.

5.3 Tree-Structure Analysis

Table 2 examines how post-pruning tree size relates to accepted nodes and throughput under the same unlimited construction budget used in the main comparison. K2 retains 108.7–112.5 nodes, approximately $15\times$ as many as K1, yet gains only 0.2 accepted nodes on each dataset and yields 4.0%–5.2% node utilization. ACP retains 35.4–38.6 nodes—65.7%–67.4% fewer than K2—while accepting 4.8–5.6 nodes. Relative to K1, ACP accepts 0.7 additional nodes on HumanEval, MBPP, and Alpaca, accompanied by 2.1%–3.1% higher TPS. On GSM8K, both strategies accept 5.6 nodes on average, while ACP retains a larger tree and has 6.4% lower TPS; together, these measurements associate ACP's higher throughput with additional accepted nodes and show that extra branches without an acceptance gain do not improve throughput in this setting.

5.4 Clustering and ACP Pruning-Depth Analysis

Figure 3 examines the throughput sensitivity of ACP to the DBSCAN radius scale α and the clustering switch depth. As α increases from 0.1 to 0.4, TPS decreases only gradually across all four benchmarks, whereas the decline becomes more pronounced beyond 0.4, placing the selected $\alpha = 0.4$ near the observed transition. A switch depth of 3 gives the highest TPS on HumanEval, MBPP, and Alpaca, while GSM8K peaks at depth 2 and declines as the switch is delayed. These trends support $\alpha = 0.4$ and switch depth 3 as a cross-task operating point rather than a per-dataset optimum.

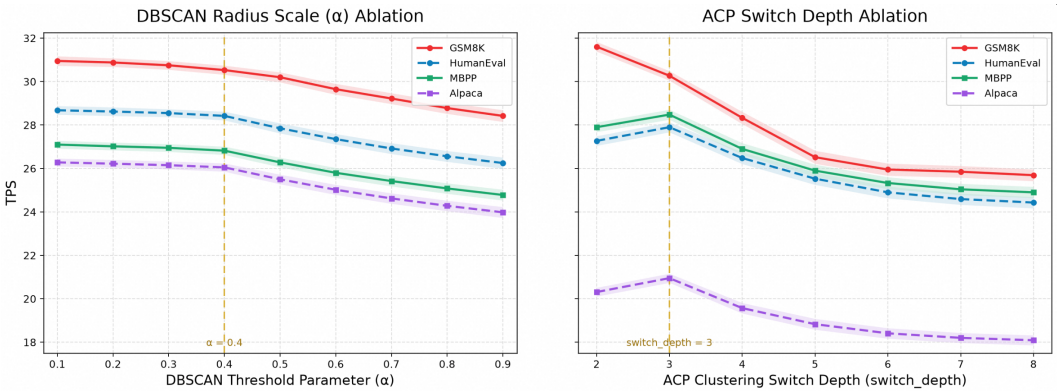


Figure 3: Throughput sensitivity to ACP hyperparameters across GSM8K, HumanEval, MBPP, and Alpaca. The left panel varies the DBSCAN radius scale α, and the right panel varies the clustering switch depth; dashed vertical lines mark the settings used in the main experiments.

6 Conclusion

In this work, we introduced **Confidence-Adaptive Permutation-Tree Speculative Decoding (CAPSpec)**, a training-free self-speculative decoding framework for block-based diffusion LLMs. Rather than committing to a fixed draft width, CAPSpec constructs a full-trajectory permutation tree over each block and prunes it with *Adaptive Confidence Partitioning (ACP)*, which applies mild DBSCAN-based clustering at shallow depths and aggressive depth-decaying top-k contraction at deeper levels. The retained trajectories are then validated through deduplicated packed verification, where logically distinct states that share identical filled blocks collapse into a single physical segment and a tree-aware attention mask verifies all candidates in one parallel target-model forward pass. Across five diffusion LLMs and four benchmarks spanning mathematics, code, and dialogue, CAPSpec delivers $1.27\times$–$3.08\times$ throughput speedups over native step decoding and achieves the highest TPS in 17 of 20 comparisons against fixed-width Top-k drafting. These results indicate that confidence-adaptive tree pruning combined with deduplicated verification is an effective, drafter-free route to accelerating diffusion language models.

Limitations

Model scale. Our evaluation is limited to models with at most 8B parameters. At larger scales, such as 13B or 70B, the absolute cost of the verification forward pass increases, and throughput may respond differently to tree size; in particular, the relative overhead of ACP's approximately 38-node trees compared with K1's approximately 7-node trees may change. Evaluating this scaling behavior is left to future work.

Tree-construction implementation. The breadth-first construction of the full permutation tree and DBSCAN clustering are currently implemented in pure Python on the CPU. At a block length of 8, this overhead is negligible (< 1 ms), but the implementation may limit scalability to larger blocks. We have not yet evaluated GPU-accelerated tree construction and clustering.

AI use statement

Generative AI tools were used to assist with LaTeX format conversion. The authors reviewed all AI-assisted changes and take responsibility for the final content of this work.

ACKNOWLEDGMENTS

We gratefully acknowledge access to the NVIDIA A100 GPU resources used in this study. We also thank the diffusion language modeling community for releasing the open-source models and software that supported our experiments.

REFERENCES

Marianne Arriola, Aaron Kerem Gokaslan, Justin T. Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *The Thirteenth International Conference on Learning Representations*, 2025.

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

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Jian Chen, Yesheng Liang, and Zhijian Liu. DFlash: Block diffusion for flash speculative decoding. *arXiv preprint arXiv:2602.06036*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Niki Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Shuang Cheng, Yihan Bian, Dawei Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, and Bowen Zhou. Sdar: A synergistic diffusion-autoregression paradigm for scalable sequence generation. *arXiv preprint arXiv:2510.06303*, 2025.

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

Yifeng Gao, Ziang Ji, Yuxuan Wang, Biqing Qi, Hanlin Xu, and Linfeng Zhang. Self speculative decoding for diffusion large language models. *arXiv preprint arXiv:2510.04147*, 2025.

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 ACM Symposium on Operating Systems Principles*, 2023.

Guanghao Li, Zhihui Fu, Min Fang, Qibin Zhao, Ming Tang, Chun Yuan, and Jun Wang. DiffuSpec: Unlocking diffusion language models for speculative decoding. *arXiv preprint arXiv:2510.02358*, 2025a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 7421–7432. Association for Computational Linguistics, 2024b. doi: 10.18653/v1/2024.emnlp-main.422. URL https://aclanthology.org/2024.emnlp-main.422/.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *arXiv preprint arXiv:2503.01840*, 2025b.

Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. dKV-Cache: The cache for diffusion language models. *arXiv preprint arXiv:2505.15781*, 2025.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating generative large language model serving with tree-based speculative inference and verification. *arXiv preprint arXiv:2305.09781*, 2023.

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. *arXiv preprint arXiv:2502.09992*, 2025.

Rui Pan, Zhuofu Chen, Hongyi Liu, Arvind Krishnamurthy, and Ravi Netravali. Fail fast, win big: Rethinking the drafting strategy in speculative decoding via diffusion LLMs. *arXiv preprint arXiv:2512.20573*, 2025a.

Rui Pan, Yinwei Dai, Zhihao Zhang, Gabriele Oliaro, Zhihao Jia, and Ravi Netravali. Specreason: Fast and accurate inference-time compute via speculative reasoning, 2025b. Preprint. Under review.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following LLaMA model, 2023.

Qingyan Wei, Yaojie Zhang, Zhiyuan Liu, Puyu Zeng, Yuxuan Wang, Biqing Qi, Dongrui Liu, and Linfeng Zhang. Accelerating diffusion large language models with SlowFast sampling: The three golden principles. *arXiv preprint arXiv:2506.10848*, 2025.

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. *arXiv preprint arXiv:2509.26328*, 2025a.

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. *arXiv preprint arXiv:2505.22618*, 2025b.

Yunfan Xiong, Ruoyu Zhang, Yanzeng Li, Tianhao Wu, and Lei Zou. DySpec: Faster speculative decoding with dynamic token tree structure. *arXiv preprint arXiv:2410.11744*, 2024.

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.

Zebin You, Shen Nie, Xiaolu Zhang, Jun Hu, Jun Zhou, Zhiwu Lu, Ji-Rong Wen, and Chongxuan Li. Llada-v: Large language diffusion models with visual instruction tuning. *arXiv preprint arXiv:2505.16933*, 2025.

Siyan Zhao, Devaansh Gupta, Qinqing Zheng, and Aditya Grover. d1: Scaling reasoning in diffusion large language models via reinforcement learning. *arXiv preprint arXiv:2504.12216*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2023.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Hu, Jun Zhou, Jianfei Chen, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. LLaDA 1.5: Variance-reduced preference optimization for large language diffusion models. *arXiv preprint arXiv:2505.19223*, 2025.

A Complete Permutation-Tree Definition and Size Derivation

Given the confidence-ordered candidate set $\mathcal{K}_t$ and the remaining set $\mathcal{R}_t$ defined in Section 4.2, CAPSpec commits $a_t = n_0$ as the anchor and permutes the $m = K - 1$ remaining positions. Let k denote the number of selections after the anchor. For $0 \le k \le m$, define the ordered no-replacement prefixes

$$\mathcal{O}_t^{(k)} = \{\boldsymbol{n}_{1:k} = (n_{i_1}, \dots, n_{i_k}) \mid n_{i_\ell} \in \mathcal{R}_t,\ i_\ell \neq i_q \text{ for all } \ell \neq q\}, \qquad \mathcal{O}_t^{(0)} = \{\emptyset\}. \tag{5}$$

Here, $i_\ell \in \{1, \dots, K-1\}$ is the original index in $\mathcal{K}_t$ of the position chosen at selection step ℓ; ℓ and q range over $\{1, \dots, k\}$. The inequality $i_\ell \neq i_q$ prevents any position from being selected twice. Therefore,

$$|\mathcal{O}_t^{(k)}| = m(m-1)\cdots(m-k+1) = \frac{m!}{(m-k)!}. \tag{6}$$

For prefix $\boldsymbol{n}_{1:k}$, the corresponding hypothetical state is

$$\tilde{x}^{(j)}_{t,\boldsymbol{n}_{1:k}} = \begin{cases} V_t^{(j)}, & j \in \{a_t\} \cup \{n_{i_1}, \dots, n_{i_k}\}, \\ x_t^{(j)}, & \text{otherwise.} \end{cases} \tag{7}$$

All $V_t^{(j)}$ are computed once by the initial draft forward pass. Tree construction changes only which draft tokens are instantiated and their reveal order. Two prefixes containing the same positions in different orders can yield identical snapshots under Equation (7); nevertheless, their parent chains and newly added edge positions differ. We therefore bind path memory and state snapshot in the node identity

$$v_{t,\boldsymbol{n}_{1:k}} \triangleq \big(\boldsymbol{n}_{1:k}, \tilde{x}_{t,\boldsymbol{n}_{1:k}}\big). \tag{8}$$

This prevents distinct parent-conditioned verification paths from being merged merely because their filled token snapshots coincide.

The complete mathematical tree $\mathcal{T}_t^{\text{full}} = (\mathcal{V}_t^{\text{full}}, \mathcal{E}_t^{\text{full}})$ uses the anchor state $v_{t,\emptyset}$ as its root, with

$$\mathcal{V}_t^{\text{full}} = \left\{ v_{t,\boldsymbol{n}_{1:k}} \;\middle|\; 0 \le k \le m,\ \boldsymbol{n}_{1:k} \in \mathcal{O}_t^{(k)} \right\}, \tag{9}$$

and edges from each prefix to every valid one-position extension:

$$\mathcal{E}_t^{\text{full}} = \left\{ \big(v_{t,\boldsymbol{n}_{1:k-1}}, v_{t,\boldsymbol{n}_{1:k}}\big) \;\middle|\; 1 \le k \le m,\ \boldsymbol{n}_{1:k} \in \mathcal{O}_t^{(k)} \right\}. \tag{10}$$

For each edge, $\boldsymbol{n}_{1:k} = (\boldsymbol{n}_{1:k-1}, n_{i_k})$, so the child adds exactly one previously unselected position while preserving the full reveal history.

At depth k, every prefix is a distinct node; Equation (6) therefore gives

$$N_k = \frac{m!}{(m-k)!}, \qquad 0 \le k \le m. \tag{11}$$

The root count is $N_0 = 1$, and for $k \ge 1$ the sequence is $N_1 = m$, $N_2 = m(m-1)$, and so forth. At depth m, the tree has $N_m = m!$ fully filled leaves, one for every complete reveal order. Summing all levels of the complete root-to-leaf tree gives

$$\begin{aligned} N_{\text{full}} &= \sum_{k=0}^{m} \frac{m!}{(m-k)!} = m! \sum_{j=0}^{m} \frac{1}{j!} \\ &\approx e\, m!. \end{aligned} \tag{12}$$

The implementation does not materialize the final level. It stops at $k = m - 1$, when each terminal node has one unresolved candidate position, and obtains that position from the terminal node's verification logits as a bonus token. The explicitly constructed candidate tree therefore contains

$$N_{\text{explicit}} = \sum_{k=0}^{m-1} \frac{m!}{(m-k)!} = m! \sum_{j=1}^{m} \frac{1}{j!} \approx (e-1)m!. \tag{13}$$

Thus the full trajectory space contains complete root-to-leaf paths, while CAPSpec avoids constructing the redundant fully filled level before the shallow and deep pruning stages in Sections 4.2 and 4.3; the budgeted breadth-first construction is detailed in Appendix C.

A.1 Uniqueness of the Committed Path

The anchor node u_0 is accepted by construction because its token is selected by the initial draft forward pass. At each visited node, CAPSpec orders retained children by descending draft confidence and selects the first child whose edge satisfies the edge-acceptance rule defined in Section 4.4:

$$u_{r+1} = \text{First}_{\rho_t \downarrow}\big\{u \in \text{Ch}_{\text{ACP}}(u_r) : A_t(u) = 1\big\}. \tag{14}$$

The walk stops when this set is empty. Distinct sibling confidences induce a strict order; when confidences are equal, the stable child-storage order provides a deterministic tie-break.

Proposition. The walk in Equation (14) produces a unique connected path from the anchor node to the last accepted node.

Proof. At a visited node u_r, let $\mathcal{S}_r = \{u \in \text{Ch}_{\text{ACP}}(u_r) : A_t(u) = 1\}$ be its accepted children. If $\mathcal{S}_r$ is empty, the walk stops uniquely. Otherwise, the confidence order together with the stable tie-break defines exactly one first element of $\mathcal{S}_r$, and hence exactly one successor u_{r+1}. Starting from the unique anchor u_0, induction over r yields one and only one connected anchor-to-node path, even if accepted edges exist elsewhere in the tree. ■

The draft tokens applied by the terminal node are written back to the sequence. Among positions still unapplied at that node, the verification logits select the highest-confidence prediction as one bonus token, so CAPSpec does not expand an additional tree level solely to obtain that token.

B ACP Clustering Parameters and Boundary Cases

For the shallow clustering rule in Section 4.2, the default radius scale is $\alpha = 0.4$ and $\text{min_samples} = 1$. Under this setting, DBSCAN is equivalent to a sorted gap partition: consecutive children remain in the same group unless their confidence difference exceeds ϵ_v. A parent with one child retains that child directly, and equal-confidence siblings are all retained. For two children, the implementation uses the absolute-gap threshold $\tau_g = 0.1$: both children are retained when their confidence difference is at most τ_g; otherwise, only the higher-confidence child enters the retained first group. These boundary rules apply before the depth-aware policy in Section 4.3 takes over.

C Roofline-Aware Tree-Budget Derivation

Materializing the factorial tree before pruning can exceed the efficient operating region of a verification forward pass. CAPSpec therefore imposes a hardware-aware logical-node budget during the breadth-first construction in Section 4.1. Let b_t be the current block length and ℓ_t the shared prefix-and-suffix context length. A roofline profile for device d provides an efficient packed-token limit $\Lambda_{\text{roof}}(d)$, leaving the residual capacity

$$\Lambda_{\text{tree}}(d, \ell_t) = \max\{0, \Lambda_{\text{roof}}(d) - \ell_t\}. \tag{15}$$

The corresponding context-aware logical-node budget is

$$Q_t = \max\left\{1, \min\left(Q_{\text{user}}, Q_{\text{large}}(b_t), \left\lfloor \frac{\Lambda_{\text{tree}}(d, \ell_t)}{b_t} \right\rfloor\right)\right\}, \tag{16}$$

where Q_{user} is the configured hard limit and $Q_{\text{large}}(b_t)$ is an additional safeguard for large blocks.

For q retained logical candidate nodes, deduplication produces $s \leq q$ physical segments, so the realized packed length is $\ell_t + sb_t$ and is upper-bounded by the conservative estimate $\ell_t + qb_t$. The current implementation uses a static A100-profiled surrogate for Equation (16): `packed_len_cap`$= 400$ gives the counter cap $\lfloor 400/b_t \rfloor$, and the same counter is capped at 16 when $b_t > 10$. This implementation counter includes one virtual all-mask root that is excluded from packed verification. Hence, if its resolved value is $K_t^{\mathrm{code}} \geq 2$, at most $K_t^{\mathrm{code}} - 1$ logical candidate nodes and no more physical segments are packed before and after deduplication, respectively. The code does not probe the device or subtract the live context length at runtime; those factors and the unused virtual-root allowance make the static bound conservative.

The tree builder starts from the anchor and expands nodes with a strict FIFO breadth-first traversal until the resolved implementation counter is reached. Within each sibling set, children are created in descending order of their raw position confidence. Consequently, all available shallow nodes are constructed before deeper nodes, and a budget boundary inside one level favors its higher-confidence siblings. A deep candidate can be committed only if every edge on its ancestor path is accepted, so allocating the budget to upper levels exposes more reachable paths than spending it on isolated deep continuations. The pruning policy in Sections 4.2 and 4.3 then traverses this budgeted tree in the same top-down order and applies its shallow clustering or deep top-k rule at each surviving parent. The resulting verification tree $\mathcal{T}_t^{\mathrm{ACP}} = (\mathcal{V}_t^{\mathrm{ACP}}, \mathcal{E}_t^{\mathrm{ACP}})$ is connected and satisfies

$$|\mathcal{V}_t^{\mathrm{ACP}}| \leq Q_t. \tag{17}$$

D Algorithm Details

Algorithm 1 summarizes CAPSpec at the level of its main computational stages. Implementation-specific limits and pruning thresholds are encapsulated by the budget and ACP policies defined in the preceding sections. Each iteration drafts and ranks the unresolved positions, commits an anchor, constructs and prunes a permutation tree, deduplicates its logical states for packed verification, and commits the selected path together with a bonus token.

Algorithm 1 Blockwise CAPSpec Generation

Require: dLLM f_θ, prompt $\boldsymbol{x}^{\mathrm{pre}}$, generation configuration Γ
Ensure: Generated sequence $\boldsymbol{x}$

$\boldsymbol{x} \leftarrow \textsc{Initialize}(\boldsymbol{x}^{\mathrm{pre}}, \Gamma)$
for all $\mathcal{B} \in \textsc{BlockSchedule}(\Gamma)$ **do**
  while $\mathcal{B}$ contains unresolved positions **do**
    $(\mathcal{K}_t, V_t, C_t) \leftarrow \textsc{DraftAndRank}(f_\theta, \boldsymbol{x}, \mathcal{B})$
    $(a_t, \mathcal{R}_t, \boldsymbol{x}) \leftarrow \textsc{CommitAnchor}(\mathcal{K}_t, V_t, \boldsymbol{x})$
    if $\mathcal{R}_t = \emptyset$ **then**
      continue
    end if
    $\mathcal{T}_t^{\mathrm{full}} \leftarrow \textsc{BuildTree}(a_t, \mathcal{R}_t, C_t, \Gamma)$
    $\mathcal{T}_t^{\mathrm{ACP}} \leftarrow \textsc{ACPPrune}(\mathcal{T}_t^{\mathrm{full}}, C_t, \Gamma)$
    $(\widehat{\boldsymbol{L}}_t, \pi_t) \leftarrow \textsc{PackAndVerify}(f_\theta, \boldsymbol{x}, \mathcal{T}_t^{\mathrm{ACP}}, V_t)$
    $A_t \leftarrow \textsc{EvaluateEdges}(\mathcal{T}_t^{\mathrm{ACP}}, \widehat{\boldsymbol{L}}_t, \pi_t, V_t)$
    $u^* \leftarrow \textsc{FirstAcceptedPath}(\mathcal{T}_t^{\mathrm{ACP}}, A_t)$
    $\boldsymbol{x} \leftarrow \textsc{CommitPathAndBonus}(\boldsymbol{x}, u^*, V_t, \widehat{\boldsymbol{L}}_t, \pi_t)$
  end while
  if the completed block contains a termination token **then**
    break
  end if
end for
return $\boldsymbol{x}$