DynaTree: Confidence-Aware Adaptive Tree Speculative Decoding for Efficient LLM Inference

Nuoyan Chen* **Jiamin Liu*** **Zhaocheng Li***
School of Computer Science
Shanghai Jiao Tong University
{cny123222, logic-1.0, lzc050419}@sjtu.edu.cn

Abstract

Autoregressive decoding in large language models (LLMs) is fundamentally sequential and therefore underutilizes modern accelerator parallelism during token generation. Speculative decoding mitigates this bottleneck by letting a lightweight draft model propose multiple tokens that are verified in parallel by the target model; however, linear variants explore only a single draft chain per step and can waste substantial computation when early tokens are rejected, while existing tree-based approaches employ *fixed* structures that cannot adapt to varying draft model confidence. We propose **DynaTree**, a tree-based speculative decoding framework with *confidence-aware adaptive branching* that dynamically adjusts tree structure based on draft model uncertainty through adaptive per-node branching, dynamic depth control, and historical acceptance tuning, combined with probability-threshold pruning under an explicit node budget. Experiments on Pythia models demonstrate that DynaTree achieves 210.8 tokens/sec on WikiText-2 (1.61× speedup, 94.7% acceptance rate), outperforming fixed tree structures by 16.3% and consistently surpassing linear speculative decoding baselines across diverse datasets.

1 Introduction

Large language models (LLMs) are typically deployed with autoregressive decoding, where each output token is generated after conditioning on all previously generated tokens. While transformer inference can exploit parallelism during the prefill stage, the decode stage remains inherently sequential and requires a full forward pass per token, leading to poor hardware utilization and high latency [1, 2].

Speculative decoding alleviates this bottleneck by separating *proposal* and *verification* [3]. A small draft model proposes several candidate tokens, and the target model verifies them in parallel; when the proposal matches the target distribution, multiple tokens can be committed per iteration. Importantly, with rejection sampling, speculative decoding preserves the exact output distribution of the target model [4].

In practice, most speculative decoding systems employ *linear* drafting: the draft model proposes a single chain of K tokens. This design is brittle under draft–target mismatch. A rejection at an early position forces all subsequent drafted tokens to be discarded, wasting both draft computation and target-model verification work. This single-path exploration fundamentally limits achievable speedups, as the system cannot recover from early errors without restarting the drafting process [4].

Tree-based speculation offers a natural remedy. When multiple plausible next tokens compete, exploring several continuations in parallel increases the chance that at least one path aligns with the target model, thereby improving the expected number of accepted tokens per verification step. The

*Equal contribution.

Preprint.

draft expands multiple candidates via top-k branching, and the target verifies the resulting token tree in parallel with a structured, causality-preserving attention mask. This multi-path exploration addresses the fundamental brittleness of linear drafting.

While tree-based drafting addresses the single-path limitation of linear methods, existing approaches typically employ *fixed* tree configurations with predetermined depth and branching factor [5, 6]. This rigid structure cannot adapt to the draft model's varying prediction confidence, creating an *efficiency gap*: high-confidence predictions waste compute exploring unnecessary branches, while uncertain predictions suffer from insufficient exploration. Recent adaptive methods [7–9] adjust draft length or employ learned predictors, yet most focus on linear speculation rather than fundamentally restructuring the tree itself. We hypothesize that *confidence-aware* tree construction—dynamically adjusting branching per node based on draft uncertainty—can bridge this gap while maintaining robust exploration.

We present **DynaTree**, which addresses the efficiency gap through confidence-aware adaptive branching that dynamically adjusts tree structure based on draft model uncertainty. Our three-phase mechanism adapts per-node branching (1–3 branches), implements dynamic depth control via early stopping and deep expansion, and tunes parameters based on historical acceptance rates. Combined with probability-threshold pruning under an explicit node budget, DynaTree verifies candidate paths in a single forward pass using tree attention. Empirically, DynaTree achieves 210.8 tokens/sec on WikiText-2 (1.61$\times$ speedup, 94.7% acceptance rate), outperforming fixed tree structures by 16.3%.

In summary, our contributions are:

- We propose a tree-based speculative decoding framework with **confidence-aware adaptive branching** that dynamically adjusts tree structure based on draft model uncertainty, directly addressing the efficiency gap in static configurations.
- We introduce a three-phase adaptive mechanism combining confidence-based branching, dynamic depth control, and historical parameter tuning. Our ablation study reveals that dynamic depth contributes most to performance gains.
- Experiments demonstrate that DynaTree achieves 1.61$\times$ speedup with 94.7% acceptance rate, outperforming fixed tree baselines by 16.3% across WikiText-2 and PG-19.

2 Related Work

2.1 Speculative Decoding

Speculative decoding accelerates autoregressive generation by decoupling *proposal* and *verification*: a lightweight draft model proposes multiple tokens, and the target model verifies these candidates in parallel while preserving the exact output distribution [3, 4, 10]. The memory-bound nature of LLM inference (approximately 1 FLOP/byte operational intensity) makes parallel verification particularly valuable [1, 2]. Recent work highlights robustness challenges across heterogeneous workloads and long-context inputs [11–13], but the dominant *linear* drafting paradigm suffers from a fundamental inefficiency: when early tokens are rejected, all downstream draft tokens are discarded, wasting computation.

2.2 Tree-Based Speculative Decoding

To overcome single-path inefficiency, tree-based methods draft multiple candidate continuations and verify them in one target-model forward pass using structured attention masks [5, 6, 14, 15]. SpecInfer [5] pioneered token tree verification with expansion-based and merge-based construction mechanisms. OPT-Tree [14] algorithmically searches for tree structures that maximize expected acceptance length, while Medusa [6] augments models with multiple decoding heads to eliminate separate draft models. However, these methods predominantly use *fixed* tree configurations with static depth D and branching factor B, creating an efficiency gap when draft confidence varies [5, 6].

Recent adaptive approaches address this rigidity through dynamic parameter adjustment. CM-ASD [7] modulates drafting length and verification thresholds based on entropy, logit margin, and softmax margin, achieving 4–5$\times$ speedups. AdaEAGLE [8] employs a lightweight MLP to predict optimal draft length per iteration. CAS-Spec [9] introduces dynamic tree cascades with acceptance

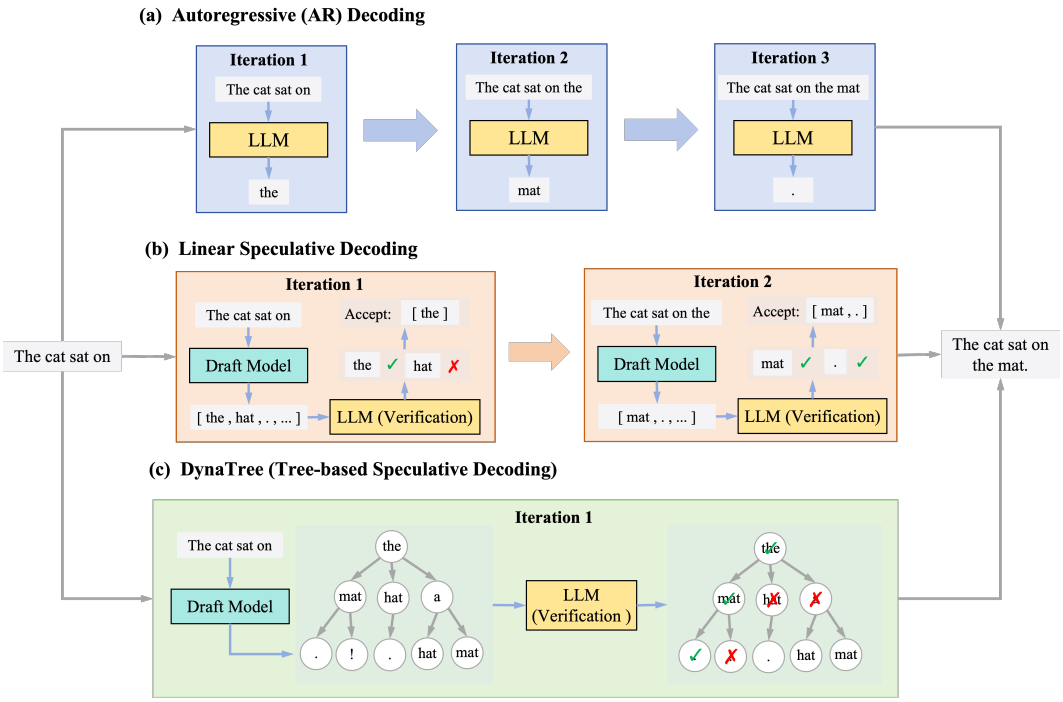

Figure 1: **Comparison of three decoding paradigms. Left:** Autoregressive (AR) decoding generates tokens sequentially, requiring one LLM forward pass per token. **Middle:** Linear speculative decoding drafts a single token chain; early rejection wastes all subsequent drafted tokens. **Right:** Tree-based speculative decoding (DynaTree) explores multiple paths in parallel, enabling recovery from draft errors and committing longer sequences per iteration. The multi-path exploration fundamentally addresses the brittleness of linear drafting while maintaining output correctness through structured tree attention verification.

rate heuristics, improving throughput by 47%. While promising, these methods focus on linear speculation or require learned predictors. In contrast, DynaTree directly restructures trees via confidence-aware branching, dynamic depth control, and historical adjustment—achieving 16.3% gains over fixed configurations while remaining training-free.

2.3 Dynamic Pruning Strategies

Exponential candidate growth necessitates pruning to balance exploration and verification cost. ProPD [16] employs top-k early prediction heads and weighted regression to remove low-utility branches, reducing computation by 2×. CAST [17] formalizes cost-aware breadth and depth pruning, explicitly modeling verification overhead per layer. DySpec [18] uses greedy confidence-guided expansion, while RASD [19] prunes retrieval candidates outside the draft model's top-k predictions. AdaSD [20] introduces hyperparameter-free thresholds based on entropy and Jensen-Shannon distance, achieving 49% speedups. These methods differ in adaptation mechanisms—from offline heuristics to online predictors. DynaTree adopts probability-threshold pruning with explicit node budgets, prioritizing simplicity and training-free integration while maintaining strong empirical performance.

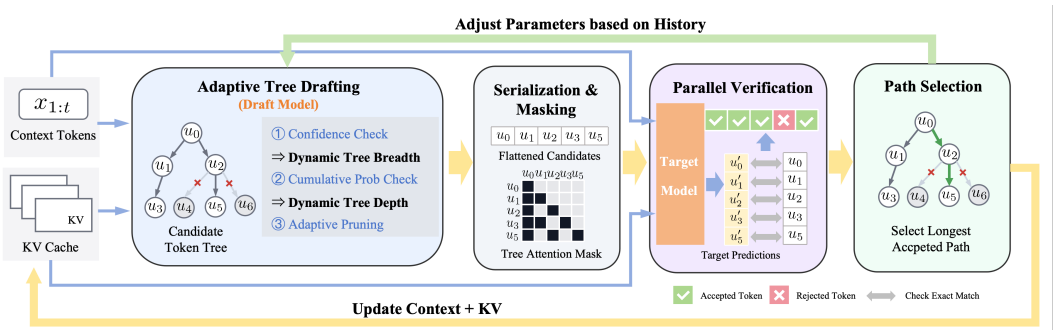

Figure 2: **One iteration of DynaTree decoding.** The process consists of six main stages: (1) *Adaptive Tree Drafting:* The draft model expands a candidate tree with three adaptive mechanisms: confidence-aware branching adjusts the number of child nodes (1–3) per expansion based on draft model confidence; dynamic depth control implements early stopping for low cumulative probability branches and deep expansion for high-probability paths; adaptive pruning removes branches below probability threshold τ or exceeding node budget $N_{\max}$. (2) *Flattening & Masking:* The pruned tree is serialized in breadth-first order, and a causal attention mask is constructed to ensure each node attends only to its ancestors. (3) *Parallel Verification:* The target model verifies all candidates in a single forward pass. (4) *Path Selection:* The longest path where drafted tokens match the target model's greedy predictions is identified. (5) *Cache Update:* The committed tokens are used to update the context and key-value cache for the next iteration. (6) *Historical Adjustment:* Acceptance rate from recent rounds feeds back to adjust confidence thresholds and base depth for the next iteration. This three-phase adaptive mechanism enables efficient multi-path exploration while maintaining correctness guarantees for greedy decoding.

3 Methodology

3.1 Problem Setup and Notation

Let M_T denote a target autoregressive language model and M_D a smaller draft model. Given a prefix (prompt) $x_{1:t}$, greedy decoding with M_T produces tokens $y_{t+1}, y_{t+2}, \ldots$ where

$$y_i \;=\; \arg\max_{v \in \mathcal{V}} p_T(v \mid x_{1:i-1}).$$

Speculative decoding accelerates generation by proposing candidate tokens with M_D and verifying them with M_T, while preserving the greedy output when the verification rule only commits tokens that match the target greedy predictions.

3.2 Overview of DynaTree

DynaTree generalizes linear speculative decoding from a single draft chain to a *draft token tree*. In each iteration, it first performs *adaptive tree drafting* with M_D, where the effective tree breadth and depth are determined on-the-fly from the draft distribution and the cumulative path probability. Concretely, for a draft node u with draft logits $\mathbf{h}(u)$, we define the draft confidence $c(u) = \max_{v \in \mathcal{V}} \mathrm{softmax}(\mathbf{h}(u))_v$ and use it to select a per-node branching factor $B(u) \in \{B_{\min}, B_{\mathrm{mid}}, B_{\max}\}$. We further control expansion depth via the cumulative probability $p(u) = \exp(\bar{\ell}_u)$, combining early stopping for low-probability branches with deeper expansion along high-probability paths. The resulting candidate tree is then **verified in parallel** in a single forward pass of M_T using tree attention, followed by **greedy path selection** and **KV-cache update** for committed tokens. Finally, DynaTree maintains a short window of recent acceptance statistics to adjust drafting thresholds for subsequent iterations.

3.3 Adaptive Tree Drafting

We maintain a token tree $\mathcal{T} = (\mathcal{N}, \mathcal{E})$ whose nodes $u \in \mathcal{N}$ correspond to drafted tokens. Each node u is associated with: (i) *token* $z_u \in \mathcal{V}$; (ii) *parent* $\pi(u)$; (iii) *depth* $d(u)$ from root; (iv) *draft log-*

probability $\ell_u = \log p_D(z_u \mid \text{prefix}(\pi(u)))$; and (v) *cumulative log-probability* $\bar{\ell}_u = \sum_{v \in \text{path}(u)} \ell_v$, where path$(u)$ denotes all nodes from root to u along the tree.

Tree expansion. Starting from the current prefix $x_{1:t}$, we construct $\mathcal{T}$ in a breadth-first manner under a strict node budget $N_{\max}$. For any expandable node u, let $q_D(\cdot \mid u)$ denote the draft distribution conditioned on the unique root-to-u prefix, and define the local confidence

$$c(u) \;=\; \max_{v \in \mathcal{V}} q_D(v \mid u).$$

We select a *per-node* branching factor via a confidence rule

$$B(u) = \begin{cases} B_{\min}, & c(u) \geq \tau_h, \\ B_{\text{mid}}, & \tau_\ell \leq c(u) < \tau_h, \\ B_{\max}, & c(u) < \tau_\ell, \end{cases}$$

where $0 < \tau_\ell < \tau_h < 1$ are confidence thresholds and $1 \leq B_{\min} \leq B_{\text{mid}} \leq B_{\max}$ are integer branch bounds. and expand u by adding the $B(u)$ highest-probability children under $q_D(\cdot \mid u)$. To adapt depth, we use the cumulative path probability $p(u) = \exp(\bar{\ell}_u)$: low-probability branches are terminated early, while high-probability paths may be expanded beyond a base depth. Concretely, a node at depth $d(u)$ is eligible for expansion only if it satisfies the depth-gating rule (defined in the next paragraph) and $|\mathcal{N}| < N_{\max}$. Implementation details for cache reuse during expansion are deferred to Appendix D.

Dynamic depth control. Let D_0 denote a base depth and $D_{\max}$ a hard maximum depth. We gate expansion using two probability thresholds $\rho_{\text{stop}} < \rho_{\text{deep}}$ on the cumulative path probability $p(u)$. A node u at depth $d(u)$ is expandable if and only if

$$d(u) < D_{\max} \quad \wedge \quad p(u) \geq \rho_{\text{stop}} \quad \wedge \quad \Big(d(u) < D_0 \;\; \vee \;\; p(u) \geq \rho_{\text{deep}} \Big).$$

We assume $1 \leq D_0 < D_{\max}$ and $0 < \rho_{\text{stop}} < \rho_{\text{deep}} < 1$. The first condition enforces a hard depth limit; the second implements *early stopping* by terminating branches whose joint draft probability is too small; and the third allows *deep expansion* beyond D_0 only along sufficiently likely paths. In practice, ρ_{stop} and ρ_{deep} are tuned on a held-out set (Section 4).

Adaptive pruning under a node budget. To reduce wasted verification on unlikely branches, we further prune any leaf u whose cumulative probability falls below a global threshold $\tau \in (0, 1)$:

$$p(u) \;<\; \tau \quad \Longrightarrow \quad \text{prune } u.$$

This rule focuses the target-model verification budget on paths that are jointly plausible under the draft model. Additionally, we enforce a strict node budget $N_{\max}$ during construction; when $|\mathcal{N}| = N_{\max}$, expansion stops and remaining frontier nodes are treated as leaves.

Historical adjustment. Finally, DynaTree adapts drafting thresholds online using recent verification outcomes. Let $a_r \in [0, 1]$ denote the per-iteration acceptance statistic at iteration r (i.e., the fraction of drafted tokens committed in that iteration), and let $\bar{a}_t$ be the sliding-window mean over the last W iterations:

$$\bar{a}_t \;=\; \frac{1}{W} \sum_{i=0}^{W-1} a_{t-i}.$$

Here W is a fixed window size. When $\bar{a}_t$ is high, we make drafting more aggressive (e.g., increasing D_0 or lowering τ_h); when $\bar{a}_t$ is low, we become more conservative to avoid verification waste. We defer the exact update schedule to Appendix D.

We provide full pseudocode for one DynaTree iteration in Appendix D.

3.4 Tree Attention for Parallel Verification

To verify all drafted tokens in one target-model forward pass, we *flatten* the tree in breadth-first order (BFS), producing a sequence $z_{1:n}$ where each token corresponds to one node and all ancestors appear earlier than descendants. We then construct a boolean attention mask $\mathbf{A} \in \{0, 1\}^{n \times (t+n)}$ such that

each drafted token attends to: (i) all prefix tokens $x_{1:t}$, and (ii) only its ancestors (including itself) in the flattened tree:

$$\mathbf{A}_{i,j} = \begin{cases} 1, & 1 \le j \le t, \\ 1, & j = t + \mathrm{pos}(v) \text{ for some ancestor } v \in \mathrm{Anc}(u_i) \cup \{u_i\}, \\ 0, & \text{otherwise.} \end{cases}$$

This mask ensures the conditional distribution computed at each node matches the distribution of sequential decoding along its unique root-to-node path, while enabling parallel verification across different branches [5, 14].

3.5 Greedy Path Selection and Cache Update

Verification signals. Let $\hat{y}_{t+1} = \arg\max p_T(\cdot \mid x_{1:t})$ be the target model's greedy next token from the prefix (available from the prefix logits). For each tree node u with flattened position i, the target forward pass outputs logits $\mathbf{s}_i$, whose argmax $\hat{y}(u) = \arg\max \mathbf{s}_i$ corresponds to the greedy *next-token* prediction after consuming the path to u.

Longest valid path. DynaTree commits the longest path $u_0 \to u_1 \to \cdots \to u_m$ such that the drafted token at each node matches the target greedy prediction from its parent context:

$$z_{u_0} = \hat{y}_{t+1}, \quad z_{u_k} = \hat{y}(u_{k-1}) \ \text{ for } k = 1, \ldots, m.$$

If no drafted token matches the first greedy prediction, we fall back to committing $\hat{y}_{t+1}$ (one token progress). After committing the matched draft tokens, we append one *bonus* token $\hat{y}(u_m)$ from the target model, mirroring the greedy speculative decoding convention and ensuring steady progress.

KV-cache management. Tree verification may populate key-value states for branches that are ultimately not committed. To maintain consistency with sequential decoding, we must restore the cache to the state corresponding to the committed prefix. Concretely, after identifying the committed path, we: (i) discard all cached key-value pairs beyond the original prefix length t; and (ii) perform a forward pass of the committed tokens through the target model to populate the cache correctly for the next iteration. This ensures that subsequent iterations start from an identical cache state as sequential greedy decoding would produce.

3.6 Correctness for Greedy Decoding

We sketch the correctness argument for greedy decoding (the setting used throughout our experiments). The tree attention mask guarantees that for any node u, the target logits at u are computed from exactly the same conditioning context as in sequential decoding along the root-to-u path. DynaTree commits a drafted token *only if* it equals the target greedy argmax under that context. Therefore, every committed token matches the token that greedy decoding with M_T would produce at that position. The cache rollback-and-rebuild step ensures the subsequent iteration starts from an identical KV state. Consequently, DynaTree generates exactly the same token sequence as greedy decoding with the target model, while reducing the number of expensive target-model forward passes by verifying many candidate tokens in parallel.

3.7 Complexity Discussion

Let $n = |\mathcal{N}| \le N_{\max}$ be the number of drafted nodes. Drafting requires $O(n)$ one-token forward passes of the draft model (with cache reuse across expansions). Verification requires a single target-model forward pass over n tokens with a structured attention mask. Dynamic pruning reduces n in uncertain regions by discarding low-probability branches, improving the trade-off between draft overhead and verification parallelism.

4 Experiments

4.1 Experimental Setup

Models. We evaluate DynaTree using models from the Pythia family [21]. Our target model M_T is Pythia-2.8B (2.8 billion parameters), and our draft model M_D is Pythia-70M (70 million parameters).

All experiments use deterministic greedy decoding, ensuring that the output sequence is uniquely determined by the model and prefix.

Hardware and software. All experiments are conducted on a single NVIDIA GPU with sufficient memory to accommodate both models simultaneously. We implement DynaTree in PyTorch [22] using the HuggingFace Transformers library [23] for model loading and inference, leveraging dynamic key-value cache structures to minimize memory overhead during tree verification.

Workloads and data preprocessing. Our primary benchmarks use WikiText-2 [26] and PG-19 [25]. For each sampled prompt, we generate T new tokens using greedy decoding, where $T = 1500$ for the main results (Section 4.4). We truncate prompts to a maximum length $L_{\max}$ to control prefill cost, using $L_{\max} = 800$ for WikiText-2 and $L_{\max} = 1000$ for PG-19. We evaluate $N = 10$ prompts and discard the first $W = 2$ runs as warmup; we report the mean and standard deviation over the remaining runs. To ensure fair comparison, we synchronize GPU execution and reset cached states between methods. Full experimental configurations are summarized in Appendix A.

4.2 Evaluation Metrics

We report **throughput** (tokens per second) as the primary metric, computed as T divided by the wall-clock decoding time (excluding warmup). We additionally report **speedup** relative to autoregressive decoding, i.e., $\mathrm{speedup} = \mathrm{TPS}/\mathrm{TPS}_{\mathrm{AR}}$. To characterize verification efficiency, we measure the **acceptance rate** a, defined as the fraction of drafted tokens that match the target model's greedy predictions under the corresponding conditioning contexts, and the average **tokens per iteration** $\bar{L}$, i.e., the number of committed tokens per verification round. We also report **average path length** $\bar{\ell}$ (the mean depth of the greedy-consistent committed path before the bonus token) to reflect how far each verification step progresses. For latency, we include **time-to-first-token** (TTFT) and **time-per-output-token** (TPOT), averaged over prompts.

4.3 Baselines

We compare the proposed confidence-aware adaptive decoding against the following baselines and variants under identical greedy decoding settings:

- **Autoregressive (AR):** Standard greedy decoding with the target model, serving as the performance baseline.
- **Linear speculative decoding:** A linear-chain speculative decoder that drafts K tokens with the draft model and verifies them with the target model in parallel, committing the longest greedy-consistent prefix [3].
- **Fixed Tree:** A static tree speculative decoder with fixed depth D, fixed branching factor B, and node budget $N_{\max}$, representing a non-adaptive tree baseline.
- **DynaTree:** Our final method, which adds three adaptive components on top of the fixed-tree backbone: Phase 1 (Dynamic Breadth), Phase 2 (Dynamic Depth), and Phase 3 (History Adaptation).

4.4 Main Results

Table 1 reports end-to-end throughput on WikiText-2 and PG-19 for $T = 1500$ token generation. Across both datasets, DynaTree yields the highest throughput and improves over both autoregressive decoding and linear speculative decoding. Compared with a fixed tree, DynaTree provides an additional gain on both datasets (e.g., 218.1 t/s vs. 188.6 t/s on WikiText-2), indicating that adapting the draft structure to local model uncertainty improves the draft–verify trade-off beyond a static configuration.

Figure 3 visualizes the throughput and speedup comparison. Linear speculative decoding can achieve high acceptance but is fragile to early mismatches: once a divergence occurs, the remaining drafted tokens in the chain cannot be reused. Tree-based drafting mitigates this failure mode by exploring multiple continuations in parallel and committing the longest greedy-consistent prefix. DynaTree further concentrates the verification budget on high-confidence regions while limiting wasted expansion in uncertain regions, improving throughput across both datasets.

Table 1: **Main results ($T = 1500$).** Throughput (t/s) and speedup relative to autoregressive decoding on WikiText-2 and PG-19. Values are mean±std over prompts (excluding warmup).

Method	WikiText-2		PG-19	
	Throughput (t/s)	Speedup	Throughput (t/s)	Speedup
AR	135.2±0.6	1.00×	133.7±0.7	1.00×
Linear Spec ($K = 5$)	163.2±26.0	1.21×	154.1±25.4	1.15×
Fixed Tree ($D = 5, B = 2$)	188.6±16.7	1.39×	183.7±24.8	1.37×
DynaTree	**218.1±21.6**	**1.61×**	**192.1±34.2**	**1.44×**

Table 2: **Latency metrics ($T = 1500$).** We report TTFT (latency to first output token) and TPOT (average per-token latency) on WikiText-2 and PG-19. Values are mean±std over prompts (excluding warmup).

Method	WikiText-2		PG-19	
	TTFT (ms)	TPOT (ms)	TTFT (ms)	TPOT (ms)
AR	23.2±14.8	7.38±0.03	20.0±4.5	7.46±0.04
Linear Spec ($K = 5$)	15.7±4.8	6.28±1.06	10.5±1.9	6.71±1.42
Fixed Tree ($D = 5, B = 2$)	14.9±3.9	5.34±0.51	10.4±0.4	5.54±0.75
DynaTree	19.4±16.2	**4.62±0.46**	9.9±0.6	**5.38±1.03**

Latency breakdown analysis. Table 2 reports TTFT and TPOT on both datasets for $T = 1500$. Across datasets, speculative decoding reduces TTFT relative to autoregressive decoding, and DynaTree achieves the lowest TPOT by committing longer prefixes per verification step.

4.5 Ablation Study

We report an ablation-style progressive comparison on WikiText-2 under the same $T = 1500$ setting as the main benchmark, isolating how each adaptive component contributes on top of a fixed-tree backbone. In this regime, enabling Dynamic Breadth alone can slightly reduce throughput due to additional control overhead, while Dynamic Depth provides the dominant gain. For clarity, Table 3 reports the combined effect of *Dynamic Breadth & Depth*, and the additional benefit of *History Adaptation*.

Additional analyses. We place supplementary sensitivity analyses (sequence-length scaling and parameter sensitivity) in Appendix B.

5 Conclusion

We introduced DynaTree, a tree-based speculative decoding framework that drafts multiple candidate continuations and verifies them in parallel using tree attention, while controlling verification cost via probability-threshold pruning and an explicit node budget. Across Pythia models, DynaTree improves decoding throughput over autoregressive decoding and consistently outperforms strong speculative decoding baselines. Our results suggest that multi-branch exploration, coupled with lightweight pruning, is an effective way to better utilize target-model verification compute under strict budget constraints. A key direction for future work is improving robustness across diverse prompts and long-context settings, and reducing overhead via kernel-level optimizations and hardware-aware tree construction.

References

[1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with IO-awareness, 2022. URL `https://arxiv.org/abs/2205.14135`.

[2] 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

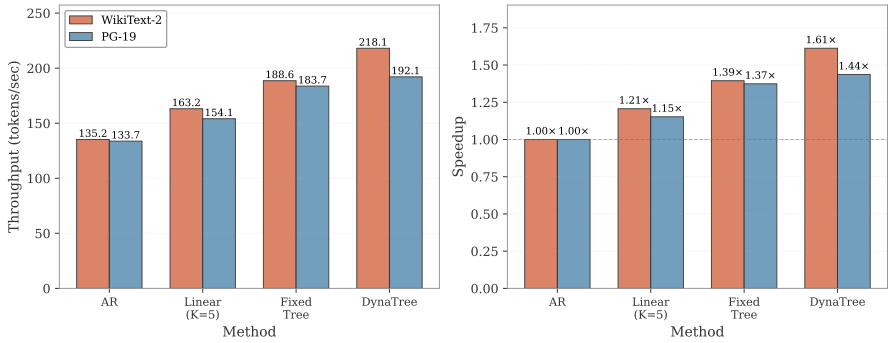

Figure 3: **Throughput and speedup across datasets ($T = 1500$).** Each method is shown with two bars (WikiText-2 vs. PG-19). DynaTree consistently improves over autoregressive and linear speculative decoding on both datasets.

Table 3: **Ablation-style progressive comparison on WikiText-2 ($T = 1500$).** We compare a fixed tree ($D_0 = 5, B = 2$) with variants that add Dynamic Breadth & Depth and History Adaptation. Speedup is computed relative to the autoregressive baseline.

Variant	Throughput (t/s)	Speedup	Δ vs Fixed
Fixed Tree ($D_0 = 5, B = 2$)	188.6±16.7	1.39×	0.0%
+ Dynamic Breadth & Depth	213.7±26.0	1.58×	+13.3%
+ History Adaptation	**218.1±21.6**	**1.61×**	**+15.6%**

language model serving with PagedAttention, 2023. URL https://arxiv.org/abs/2309.06180.

[3] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023. URL https://arxiv.org/abs/2211.17192.

[4] Minghao Yan, Saurabh Agarwal, and Shivaram Venkataraman. Decoding speculative decoding, 2024. URL https://arxiv.org/abs/2402.01528.

[5] 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 large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pages 932–949. ACM, April 2024. doi: 10.1145/3620666.3651335. URL https://doi.org/10.1145/3620666.3651335.

[6] 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, 2024. URL https://arxiv.org/abs/2401.10774.

[7] Jaydip Sen and Saurabh Dasgupta. Confidence-modulated speculative decoding for large language models, 2024. URL https://arxiv.org/abs/2508.15371.

[8] Songlin Zhao, Yue Zhang, Hao Li, Qianhui Zhong, Haotian Wang, and Qiao Xu. Adaeagle: Optimizing speculative decoding via explicit modeling of adaptive draft structures, 2024. URL https://arxiv.org/abs/2412.18910.

[9] Kaifeng Zhang, Xuefan Hu, Kun Huang, Aoran Li, Yue Wu, and Yong Zhou. Cas-spec: Cascade adaptive self-speculative decoding for on-the-fly lossless inference acceleration of llms, 2025. URL https://arxiv.org/abs/2510.26843.

[10] Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding, 2024. URL https://arxiv.org/abs/2401.07851.

[11] Fahao Chen, Peng Li, Tom H. Luan, Zhou Su, and Jing Deng. Spin: Accelerating large language model inference with heterogeneous speculative models, 2025. URL https://arxiv.org/abs/2503.15921.

[12] Jaeseong Lee, Seung-Won Hwang, Aurick Qiao, Gabriele Oliaro, Ye Wang, and Samyam Rajbhandari. Owl: Overcoming window length-dependence in speculative decoding for long-context inputs, 2025. URL https://arxiv.org/abs/2510.07535.

[13] Gregor Bachmann, Sotiris Anagnostidis, Albert Pumarola, Markos Georgopoulos, Artsiom Sanakoyeu, Yuming Du, Edgar Schönfeld, Ali Thabet, and Jonas Kohler. Judge decoding: Faster speculative sampling requires going beyond model alignment, 2025. URL https://arxiv.org/abs/2501.19309.

[14] Jikai Wang, Yi Su, Juntao Li, Qingrong Xia, Zi Ye, Xinyu Duan, Zhefeng Wang, and Min Zhang. Opt-tree: Speculative decoding with adaptive draft tree structure. *Transactions of the Association for Computational Linguistics*, 13:188–199, 2025. doi: 10.1162/tacl_a_00735. URL https://doi.org/10.1162/tacl_a_00735.

[15] Yepeng Weng, Qiao Hu, Xujie Chen, Li Liu, Dianwen Mei, Huishi Qiu, Jiang Tian, and Zhongchao Shi. Traversal verification for speculative tree decoding, 2025. URL https://arxiv.org/abs/2505.12398.

[16] Shuzhang Zhong, Zebin Yang, Meng Li, Ruihao Gong, Runsheng Wang, and Ru Huang. Propd: Dynamic token tree pruning and generation for llm parallel decoding, 2024. URL https://arxiv.org/abs/2402.13485.

[17] Yinrong Hong, Zhiquan Tan, and Kai Hu. Inference-cost-aware dynamic tree construction for efficient inference in large language models, 2025. URL https://arxiv.org/abs/2510.26577.

[18] Yunfan Xiong, Ruoyu Zhang, Yanzeng Li, Tianhao Wu, and Lei Zou. Dyspec: Faster speculative decoding with dynamic token tree structure, 2024. URL https://arxiv.org/abs/2410.11744.

[19] Yiming Chen, Yikai Wang, Yize Li, Jing Sun, Xingjian Tang, Ning Zhu, and Lei Li. Rasd: Retrieval-augmented speculative decoding, 2025. URL https://arxiv.org/abs/2503.03434.

[20] Yichen Zhang, Tianyu Guan, Zijie Wang, Yan Yan, Yuhao Lv, Heng Zhou, Xupeng Miao, Lingxiao Cao, Hongyi Yin, Chen Mei, Ming Xue, Donghua Yuan, Yifan Mei, Bo Xing, Bowen Zhou, Jiankun Sun, Libin Lan, Kaimeng Wang, and Xuelong Cheng. Adasd: Adaptive speculative decoding for efficient language model inference, 2024. URL https://arxiv.org/abs/2512.11280.

[21] Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling, 2023.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[23] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing, 2020.

[24] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024. URL https://arxiv.org/abs/2309.17453.

Table 4: **Sequence length scaling on WikiText-2.** Results are computed from the experiment logs released with this work. Throughput is in tokens/s (mean±std across prompts), and speedup is relative to AR at the same T.

T	AR (t/s)	Fixed Tree (t/s)	Speedup	DynaTree (t/s)	Speedup	Accept.
100	81.6±17.2	109.6±27.4	1.34×	110.0±36.8	1.35×	39.2%
200	121.3±21.7	140.9±27.0	1.16×	135.6±35.9	1.12×	49.0%
300	132.1±2.1	157.6±26.0	1.19×	159.7±32.4	1.21×	62.6%
500	133.4±0.9	165.3±32.1	1.24×	178.1±42.1	1.33×	77.3%
750	127.2±14.4	183.7±12.9	1.44×	190.7±36.6	1.50×	86.1%
1000	135.7±0.7	192.2±13.6	1.42×	210.0±27.0	1.55×	92.0%

[25] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019. URL `https://arxiv.org/abs/1911.05507`.

[26] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016. URL `https://arxiv.org/abs/1609.07843`.

A Experimental Configuration Details

Common settings. Unless stated otherwise, we use WikiText-2 prompts, truncate the prompt length to $L_{\max} = 800$, and generate T new tokens with greedy decoding. We evaluate $N = 10$ prompts and discard the first $W = 2$ runs as warmup.

PG-19 setting. For PG-19, we use the same model pair and greedy decoding, with a maximum prompt length $L_{\max} = 1000$.

Fixed-tree baseline. For the static-tree baseline, we use depth $D = 5$, branching factor $B = 2$, and a node budget $N_{\max} = 256$.

DynaTree. Unless stated otherwise, the adaptive configuration uses base depth $D_0 = 5$, maximum depth $D_{\max} = 8$, branch bounds $(B_{\min}, B_{\max}) = (1, 3)$, and confidence thresholds $(\tau_h, \tau_\ell) = (0.9, 0.4)$.

B Additional Experimental Analyses

Speedup computation. Some auxiliary result files include a placeholder `speedup` field; throughout this appendix we report speedup as the ratio between the method throughput and the corresponding autoregressive throughput under the same setting.

B.1 Sequence Length Scaling

We evaluate how throughput varies with generation length T on WikiText-2, comparing autoregressive decoding, a fixed tree configuration, and DynaTree. Table 4 reports throughput and speedup for each T.

B.2 Parameter Sensitivity

We study the sensitivity of DynaTree to confidence thresholds (τ_h, τ_ℓ) and branch bounds $(B_{\min}, B_{\max})$ on WikiText-2 at $T = 500$. Table 5 reports representative configurations with the highest throughput in `results/adaptive/sensitivity/paper_benchmark_sensitivity.json`.

Table 5: **Parameter sensitivity on WikiText-2 ($T = 500$).** Results are computed from the experiment logs released with this work. Speedup is relative to AR under the same setting.

Setting	Throughput (t/s)	Speedup	Accept.
AR	99.2±22.3	1.00×	0.0%
$(\tau_h, \tau_\ell) = (0.9, 0.4),\ (B_{\min}, B_{\max}) = (1, 4)$	180.5±29.6	1.82×	81.1%
$(\tau_h, \tau_\ell) = (0.8, 0.3),\ (B_{\min}, B_{\max}) = (1, 3)$	179.0±27.3	1.80×	79.7%
$(\tau_h, \tau_\ell) = (0.8, 0.3),\ (B_{\min}, B_{\max}) = (1, 2)$	178.3±29.2	1.80×	78.6%
$(\tau_h, \tau_\ell) = (0.8, 0.3),\ (B_{\min}, B_{\max}) = (1, 4)$	174.8±31.3	1.76×	77.3%

Table 6: **Peak GPU memory consumption comparison.** We measure peak memory usage during 500-token generation on PG-19 (long-form fiction) and WikiText-2 (structured articles) with Pythia-2.8B and Pythia-70M models. All speculative methods incur minimal memory overhead (<1% relative to baseline), with DynaTree adding only 0.45% on average to accommodate tree structures and draft KV cache. Values show mean peak memory across 10 runs; relative change is computed against the autoregressive baseline.

Method	Peak Memory (MB)		Average (MB)	Rel. Change
	PG-19	WikiText-2		
AR (baseline)	5855.1	5798.6	5826.9	0.00%
Linear K=6	5817.3	5786.3	5801.8	−0.43%
Linear K=7	5817.7	5786.2	5801.9	−0.43%
DynaTree (D=6, B=2)	**5883.7**	**5822.9**	**5853.3**	**+0.45%**
DynaTree (D=7, B=2)	5883.7	5822.9	5853.3	+0.45%

C Memory Footprint Analysis

An important practical consideration for speculative decoding methods is their memory overhead. Table 6 reports peak GPU memory consumption across methods on PG-19 and WikiText-2 datasets during 500-token generation with Pythia models. All measurements are taken using PyTorch's memory profiler during steady-state generation (excluding initial model loading).

Key observations: (i) DynaTree incurs minimal memory overhead (+0.45% on average) compared to the autoregressive baseline, adding only 26 MB to accommodate the draft model's KV cache and intermediate tree structures; (ii) Linear speculative methods show similarly negligible overhead (<1%), as they maintain only a small fixed-size buffer of candidate tokens. Across all methods, memory overhead remains well within 1% of the baseline, confirming that speculative decoding's primary cost is computational rather than memory-related. This makes DynaTree suitable for memory-constrained deployment scenarios where the target and draft models can already fit in GPU memory.

D DynaTree Iteration Pseudocode

Algorithm 1 DynaTree: one iteration (greedy-consistent).

Require: Prefix tokens $x_{1:t}$; target KV cache $\mathcal{K}_T$; prefix next-token logits $\mathbf{s}_{\text{last}}$; branch bounds $B_{\min} \le B_{\text{mid}} \le B_{\max}$; confidence thresholds $0 < \tau_\ell < \tau_h < 1$; base depth D_0; max depth $D_{\max}$; depth thresholds $0 < \rho_{\text{stop}} < \rho_{\text{deep}} < 1$; pruning threshold τ; node budget $N_{\max}$; history window W.

Ensure: Committed tokens $y_{t+1:t+L}$ and updated $\mathcal{K}_T$.

$\ell \leftarrow \textsc{SeqLen}(\mathcal{K}_T)$ ▷ record prefix cache length

$\mathcal{T} \leftarrow \textsc{DraftTree}(x_{1:t}, B_{\min}, B_{\text{mid}}, B_{\max}, \tau_\ell, \tau_h, D_0, D_{\max}, \rho_{\text{stop}}, \rho_{\text{deep}}, \tau, N_{\max})$

$\mathbf{z}_{1:n} \leftarrow \textsc{BFSFlatten}(\mathcal{T})$; $\mathbf{A} \leftarrow \textsc{TreeMask}(\mathcal{T}, \ell)$ ▷ prefix + ancestors only

$\mathbf{s}_{1:n} \leftarrow M_T(\mathbf{z}_{1:n}; \mathbf{A}, \mathcal{K}_T)$; $\hat{\mathbf{y}} \leftarrow \arg\max \mathbf{s}_{1:n}$

$y_{t+1:t+L} \leftarrow \textsc{SelectCommit}(\mathcal{T}, \hat{\mathbf{y}}, \mathbf{s}_{\text{last}})$

$\mathcal{K}_T \leftarrow \textsc{Crop}(\mathcal{K}_T, \ell)$; $\mathcal{K}_T \leftarrow M_T(y_{t+1:t+L}; \mathcal{K}_T)$ ▷ rollback + rebuild

$\textsc{UpdateHistory}(y_{t+1:t+L}, \mathcal{T}, W)$; $\textsc{Adjust}(\tau_\ell, \tau_h, D_0)$ ▷ historical adjustment

return $y_{t+1:t+L}$

function $\textsc{DraftTree}(x_{1:t}, B_{\min}, B_{\text{mid}}, B_{\max}, \tau_\ell, \tau_h, D_0, D_{\max}, \rho_{\text{stop}}, \rho_{\text{deep}}, \tau, N_{\max})$ ▷ Draft a candidate token tree with adaptive branching, dynamic depth control, and pruning under a node budget.

  Run M_D on $x_{1:t}$; let u_0 be the $\top 1$ token; initialize $\mathcal{T}$ with root u_0

  $\mathcal{A} \leftarrow \{u_0\}$ ▷ active frontier

  while $|\mathcal{A}| > 0$ **and** $|\mathcal{T}| < N_{\max}$ **do**

    Pop an element u from $\mathcal{A}$

    if $\exp(\bar{\ell}_u) < \tau$ **then**

      continue

    end if ▷ probability-threshold pruning

    if $d(u) \ge D_{\max}$ **then**

      continue

    end if

    if $\exp(\bar{\ell}_u) < \rho_{\text{stop}}$ **then**

      continue

    end if ▷ early stopping

    if $d(u) \ge D_0$ **and** $\exp(\bar{\ell}_u) < \rho_{\text{deep}}$ **then**

      continue

    end if ▷ depth gating

    Do one cached step of M_D from u; compute confidence $c(u) = \max \text{softmax}(\mathbf{h}(u))$

    Set $B(u) \leftarrow B_{\min}$ if $c(u) \ge \tau_h$, $B_{\max}$ if $c(u) < \tau_\ell$, else B_{mid}

    Take $\top B(u)$ next-token candidates; add each child v to $\mathcal{T}$ and push v into $\mathcal{A}$ until $N_{\max}$

  end while

  return $\mathcal{T}$

end function

function $\textsc{SelectCommit}(\mathcal{T}, \hat{\mathbf{y}}, \mathbf{s}_{\text{last}})$ ▷ Select the longest greedy-consistent path (plus one bonus token).

  $first \leftarrow \arg\max \mathbf{s}_{\text{last}}$

  Find the longest path P where root token $= first$, and for each edge $(u \to v)$, token$(v) = \hat{\mathbf{y}}[\text{pos}(u)]$

  if $P = \emptyset$ **then**

    return $[first]$

  else

    $y \leftarrow$ tokens on P

    Append one bonus token $\hat{\mathbf{y}}[\text{pos}(\text{last}(P))]$

    return y

  end if

end function