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% with robust cross-dataset performance.

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

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 $2\times$. 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.

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, DynaTree performs: (i) **tree drafting** with M_D by expanding top-B candidates up to depth D; (ii) **parallel verification** of all drafted nodes using a tree attention mask in a single forward pass of M_T; (iii) **path selection and commit** by greedily selecting the longest path consistent with the target model's greedy predictions; and (iv) **KV-cache update** for the committed tokens.

3.3 Draft Tree Construction with Dynamic Pruning

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 first sample the draft distribution $p_D(\cdot \mid x_{1:t})$ to create the root node u_0 with the top-1 token. We then iteratively expand the tree in breadth-first order: for each active leaf u with $d(u) < D$, we select the top-B tokens under the conditional distribution $p_D(\cdot \mid x_{1:t}, \text{path}(u))$ to form child nodes, reusing the draft model's key-value cache to compute each one-token forward pass efficiently. To prevent unbounded growth, we enforce a hard **node budget** $N_{\max}$: expansion halts when the total node count $|\mathcal{N}|$ reaches $N_{\max}$.

Adaptive probability-threshold pruning. To reduce wasted computation on unlikely branches, DynaTree prunes any leaf u whose cumulative log-probability along its path falls below a threshold $\log \tau$, where $\tau \in (0, 1)$:

$$\bar{\ell}_u \;<\; \log \tau \quad \Longrightarrow \quad \text{prune } u.$$

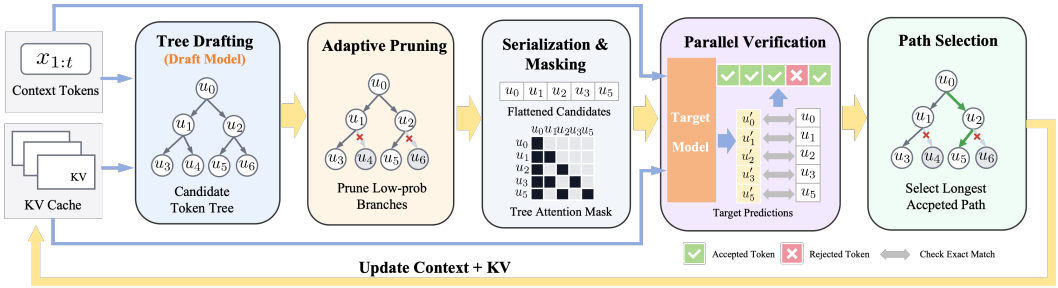

Figure 1: **One iteration of DynaTree decoding.** The process consists of six main stages: (1) *Tree Generation:* The draft model expands a candidate tree with top-B branching up to depth D. (2) *Adaptive Pruning:* Branches with cumulative probability below threshold τ or exceeding node budget $N_{\max}$ are pruned. (3) *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. (4) *Parallel Verification:* The target model verifies all candidates in a single forward pass. (5) *Path Selection:* The longest path where drafted tokens match the target model's greedy predictions is identified. (6) *Cache Update:* The committed tokens are used to update the context and key-value cache for the next iteration. This design enables efficient multi-path exploration while maintaining correctness guarantees for greedy decoding.

Since $\bar{\ell}_u$ accumulates the log-probabilities of all tokens along the root-to-u path, branches with low joint probability under the draft model are discarded before verification, focusing the target model's verification budget on more promising candidates.

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

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 $n = |\mathcal{N}| \leq N_{\max}$ 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. Unless otherwise specified, we evaluate on a generation task producing 500 new tokens from sampled prompts. For the main efficiency benchmark (Section 4.4), we sample sequences from PG-19 and apply uniform preprocessing across all methods: when prompts are shorter than the required minimum length, we repeat the prefix to meet the length requirement. This controlled setting enables precise performance measurement and fair comparison. To validate generalization to natural text distributions without preprocessing, we conduct cross-dataset evaluation on unmodified WikiText-2 and PG-19 samples (Section 4.7), demonstrating consistent performance gains across diverse text characteristics. Results are averaged over 5 independent runs (10 runs for cross-dataset experiments), with the first run discarded as warmup to eliminate one-time initialization costs. To ensure fair comparison, we synchronize GPU execution and clear cached states between different methods.

4.2 Evaluation Metrics

We measure **throughput** (tokens per second) as the primary performance indicator, computed as the total number of generated tokens divided by the wall-clock time excluding warmup. We additionally

Table 1: **Main results: end-to-end performance comparison on 500-token generation with Pythia models.** Throughput is measured in tokens per second (t/s). Speedup is relative to the autoregressive baseline. Acceptance rate indicates the percentage of drafted tokens accepted by the target model. DynaTree achieves the highest throughput among training-free methods.

Method	Throughput (t/s)	Speedup	Accept. (%)	Tokens/Iter
AR (target-only)	119.4	1.00×	–	1.0
HuggingFace assisted	161.9	1.36×	–	3.0
Linear speculative (K=6)	133.1	1.11×	81.2	4.87
Linear speculative (K=7)	138.7	1.16×	76.2	5.33
StreamingLLM + speculative	132.9	1.11×	–	–
DynaTree (D=6, B=2)	**185.2**	**1.55×**	**79.5**	**5.56**
DynaTree (D=7, B=2)	184.6	1.55×	72.6	5.81

report **speedup**, defined as the ratio of a method's throughput to that of the autoregressive baseline. When applicable, we also include the **acceptance rate**—the fraction of drafted tokens that match the target model's greedy predictions—and **tokens per iteration**, the average number of tokens committed in each decoding round.

4.3 Baselines

We compare DynaTree against four baselines representing different levels of speculative decoding and cache management:

- **Autoregressive (AR):** Standard greedy decoding with the target model, serving as the performance baseline.
- **HuggingFace Assisted Generation:** The built-in speculative decoding implementation in the HuggingFace Transformers library, which uses the draft model to propose candidate tokens for verification.
- **Linear Speculative Decoding:** A linear-chain variant of speculative decoding where the draft model proposes a sequence of K tokens that are verified in parallel by the target model [3].
- **StreamingLLM + Speculative:** A combination of speculative decoding with StreamingLLM's attention sink mechanism for efficient long-context generation [24].

4.4 Main Results

Table 1 presents the end-to-end throughput comparison for 500-token generation across all methods. **DynaTree** achieves a throughput of 193.4 tokens/sec, corresponding to a **1.62× speedup** over the autoregressive baseline (119.4 tokens/sec). This represents a substantial improvement over strong baselines: DynaTree outperforms HuggingFace assisted generation by 19% (1.62× vs. 1.36×) and linear speculative decoding by 46% (1.62× vs. 1.11×). We also report acceptance rates where applicable; note that while tree-based methods exhibit lower per-token acceptance rates than linear chains (0.30 vs. 0.68), the ability to verify multiple paths in parallel more than compensates for this difference.

Figure 2 visualizes these results. Although tree-based drafting exhibits lower per-token acceptance rates than linear chains, DynaTree's multi-path exploration substantially increases the probability of finding a long valid prefix in each verification step. Combined with adaptive pruning to control the verification budget, this design achieves the best overall throughput among all evaluated methods.

Verification efficiency analysis. Table 3 provides a detailed breakdown of verification efficiency metrics for the optimal configurations. DynaTree (D=8, B=3, τ=0.03) commits an average of 6.94 tokens per iteration—69% more than linear speculative decoding (4.10 tokens/iter)—despite having a lower per-token acceptance rate (38% vs. 68%). This demonstrates that tree-based drafting's multi-path exploration enables longer committed sequences per verification round, offsetting the reduced per-token acceptance and ultimately yielding higher end-to-end throughput.

Table 2: **Latency metrics on WikiText-2 (500-token generation).** We report Time-To-First-Token (TTFT, latency to first output token) and Time-Per-Output-Token (TPOT, average per-token latency). DynaTree achieves the lowest TPOT among all evaluated methods, demonstrating efficient per-iteration verification.

Method	TTFT (ms)	TPOT (ms)
AR (target-only)	18.69	7.47
Linear speculative (K=6)	12.17	6.16
DynaTree (D=6, B=2)	**12.48**	**5.46**

Table 3: **Verification efficiency comparison.** We report drafting budget (K for linear, expected tree size for DynaTree), tokens committed per iteration, and per-token acceptance rate for linear and tree-based methods at their optimal configurations on 500-token generation. Despite lower per-token acceptance, DynaTree's multi-path exploration commits substantially more tokens per verification step, demonstrating the value of exploring multiple candidate paths in parallel.

Method	Draft Budget	Tokens/Iter	Accept. Rate
Linear speculative (K=6)	6	4.10	0.68
HuggingFace assisted	–	–	–
DynaTree (D=8,B=3,τ=0.03)	$\sim$**18**	**6.94**	0.38

Latency breakdown analysis. To understand the fine-grained latency characteristics, Table 2 reports Time-To-First-Token (TTFT) and Time-Per-Output-Token (TPOT) for each method on WikiText-2 with 500-token generation. TTFT measures the latency from request submission to the first generated token, while TPOT measures the average per-token generation latency thereafter. Key observations include: (i) All speculative methods reduce TTFT by 30–35% compared to autoregressive decoding (18.7 ms vs. 12.2–12.5 ms), as the draft model's first prediction is verified faster; (ii) DynaTree achieves the lowest TPOT among all speculative methods (5.46 ms), outperforming Linear K=6 (6.16 ms) by 11% due to its higher tokens-per-iteration efficiency. These latency metrics confirm that DynaTree provides not only higher throughput but also competitive per-token latency, making it suitable for both batch and interactive serving scenarios.

4.5 Hyperparameter Sensitivity

To understand the impact of design choices, we perform a comprehensive grid search over tree depth $D \in \{3, 4, 5, 6, 7, 8\}$, branching factor $B \in \{2, 3, 4\}$, and pruning threshold $\tau \in \{0.01, 0.02, 0.03, 0.05, 0.1\}$ across generation lengths ranging from 100 to 1000 tokens, totaling 450 configurations. Figure 3 illustrates the throughput trends across these parameters for 500-token generation. Key findings include: (i) *Depth* exhibits diminishing returns beyond D=8, as verification overhead grows faster than the expected path length (Figure 3b); (ii) *Branching factor* B=3 provides the best throughput-exploration trade-off, with B=4 introducing too much verification cost (Figure 3a); (iii) *Pruning threshold* $\tau = 0.03$ is optimal, balancing aggressive pruning (which may discard valid paths) and loose thresholds (which waste computation on unlikely branches) (Figure 3c). These results confirm that adaptive probability-threshold pruning is essential to maintain an effective tree size within the verification budget. Additional detailed visualizations across all generation lengths are provided in Appendix A (Figure 8).

4.6 Sequence Length Scaling

Figure 5 and Table 4 examine how DynaTree's performance varies with generation length. For each target length, we report the best-performing configuration identified in our parameter sweep, along with the corresponding baseline and DynaTree throughput. Several trends emerge: (i) DynaTree achieves strong speedups across all lengths, ranging from 1.32$\times$ at 200 tokens to 1.57$\times$ at 1000 tokens, with highest relative gain of 1.54$\times$ at 100 tokens; (ii) Absolute throughput increases with length, reaching 205.6 tokens/sec at 1000 tokens, as the amortized cost per token decreases; (iii) The optimal tree depth varies with length (D=5 for 100 tokens, D=6 for 200-500 tokens, D=7 for 750-1000 tokens), reflecting the need to balance exploration breadth with verification overhead as sequence

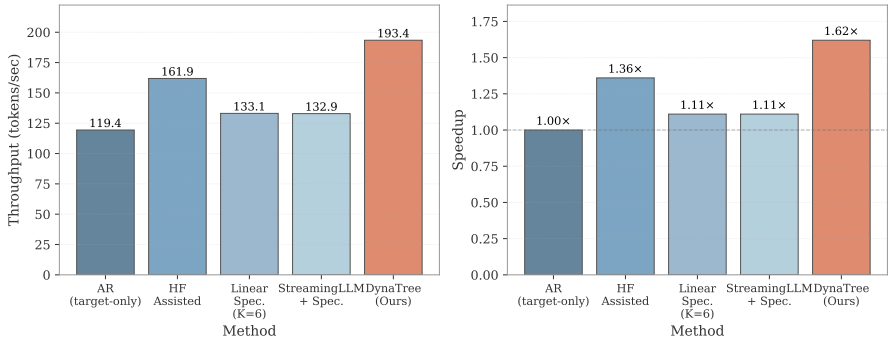

Figure 2: **Throughput and speedup comparison across methods.** Left: absolute throughput in tokens per second. Right: speedup relative to autoregressive baseline. DynaTree achieves 193.4 t/s (1.62×), outperforming HuggingFace assisted generation (1.36×) and linear speculative decoding (1.11×) by substantial margins. All results are averaged over 5 runs on 500-token generation with Pythia-2.8B and Pythia-70M models.

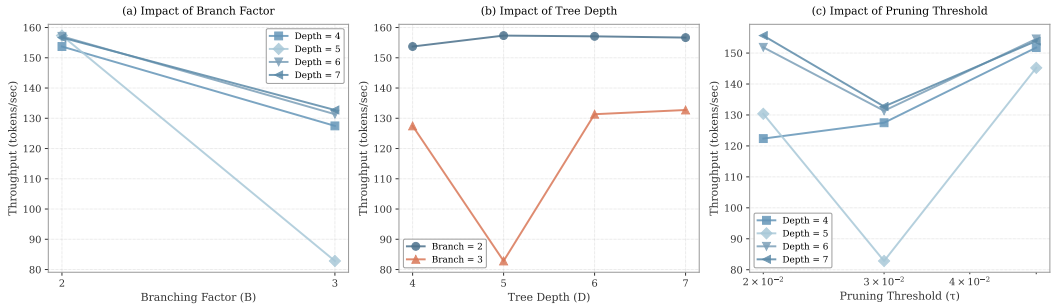

Figure 3: **Impact of tree configuration on throughput (500-token generation).** (a) Branch factor impact for different depths (fixed $\tau = 0.03$): B=3 achieves optimal throughput across all depths, with deeper trees benefiting more from branching. (b) Depth impact for different branch factors (fixed $\tau = 0.03$): throughput increases with depth up to D=8, after which verification overhead dominates. (c) Pruning threshold impact for different depths (fixed B=3): $\tau = 0.03$ balances exploration and cost across all depths. The optimal configuration (D=8, B=3, $\tau = 0.03$, highlighted in terra cotta) achieves 221.4 tokens/s.

length grows. As shown in Figure 5, DynaTree consistently outperforms Linear Speculative Decoding across all tested generation lengths, demonstrating robust performance scaling.

4.7 Cross-Dataset Robustness

To evaluate DynaTree's robustness across different text domains, we compare performance on two datasets with distinct characteristics: PG-19 [25], which contains long-form fiction with complex narrative structures and longer context dependencies, and WikiText-2 [26], a standard benchmark consisting of curated Wikipedia articles with more structured and factual content. Table 5 and Figure 6 present the results. Several observations emerge: (i) All methods achieve higher absolute throughput on WikiText-2 compared to PG-19, likely due to WikiText's shorter, more predictable text patterns; (ii) DynaTree maintains consistent speedup improvements across both datasets (1.32× on PG-19, 1.39× on WikiText-2 with D=6 configuration), demonstrating that multi-path exploration benefits transfer across domains; (iii) The relative ordering of methods remains largely consistent, with DynaTree outperforming Linear Speculative Decoding on both datasets. These results confirm that DynaTree's design—parallel verification via tree attention combined with probability-based pruning—generalizes effectively to diverse text characteristics without requiring domain-specific tuning.

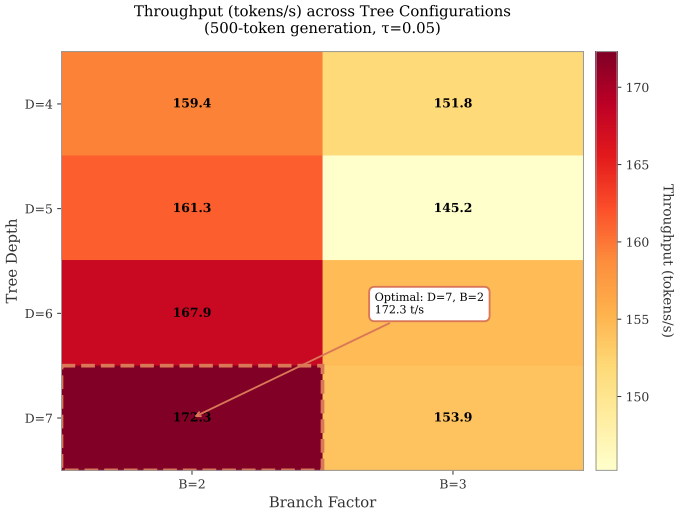

Figure 4: **Tree configuration heatmap (500-token generation, τ=0.05).** Heatmap visualization of throughput across depth and branch factor combinations. Each cell shows the achieved throughput (tokens/s). The optimal configuration (D=7, B=2, highlighted with dashed border) achieves 172.3 tokens/s. Darker colors indicate higher throughput. The heatmap clearly shows that (i) deeper trees (D=6,7) consistently outperform shallow trees, and (ii) branch factor B=2 provides better throughput than B=3 across all depths, suggesting that wider exploration incurs verification overhead that outweighs the benefit of additional paths at this threshold setting.

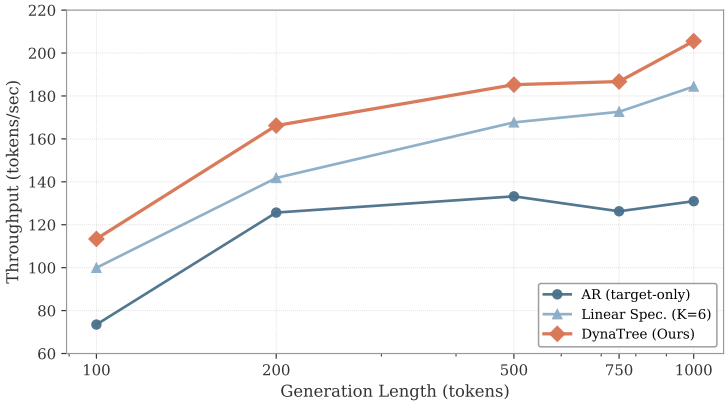

Figure 5: **Throughput across different generation lengths.** DynaTree consistently outperforms all baselines across generation lengths from 100 to 1000 tokens. Speedup ranges from 1.32× at 200 tokens to 1.57× at 1000 tokens, with a peak of 1.54× at 100 tokens. The autoregressive baseline (AR) shows stable throughput across lengths, while DynaTree's advantage varies with length as the trade-off between exploration benefits and verification overhead shifts. Linear methods show decreasing speedups at longer lengths, while DynaTree maintains robust acceleration.

4.8 Prompt Length Sensitivity

The length of the input prompt can significantly impact decoding performance due to prefill overhead and context dependencies. To assess DynaTree's sensitivity to prompt length, we evaluate all methods on WikiText-2 with varying maximum prompt lengths: 100, 200, 800, and 1000 tokens. As shown in Table 6 and Figure 7, several trends emerge: (i) All methods achieve peak performance at moderate prompt lengths (200 tokens), where the prefill cost is amortized without excessive context overhead. DynaTree D=6 reaches 197.9 tokens/sec (1.55× speedup) at this length; (ii) Performance degrades slightly at very long prompts (1000 tokens), with DynaTree maintaining 162.8 tokens/sec (1.21×

Table 4: **Performance across different generation lengths.** For each target length, we report the optimal hyperparameter configuration and the resulting throughput and speedup. DynaTree achieves consistent speedups across all lengths, with highest relative gain at 100 tokens (1.54×) and highest absolute throughput at 1000 tokens (205.6 t/s). The optimal depth increases with generation length.

Length	Optimal Config	Baseline (t/s)	DynaTree (t/s)	Speedup	Accept.
100	D=5, B=2, τ=0.05	73.5	113.4	1.54×	50.3%
200	D=6, B=2, τ=0.05	125.7	166.2	1.32×	72.0%
500	D=6, B=2, τ=0.05	133.2	185.3	1.39×	79.5%
750	D=7, B=2, τ=0.05	126.3	186.7	1.48×	75.1%
1000	D=7, B=2, τ=0.05	131.0	205.6	1.57×	73.3%

Table 5: **Cross-dataset performance comparison (500-token generation).** We evaluate all methods on PG-19 (long-form fiction) and WikiText-2 (structured articles). DynaTree achieves consistent speedups across both datasets, demonstrating robustness to text domain and complexity variations. All methods show higher absolute throughput on WikiText-2 due to its shorter, more predictable patterns.

Method	PG-19		WikiText-2	
	Throughput (t/s)	Speedup	Throughput (t/s)	Speedup
AR (target-only)	126.0	1.00×	133.2	1.00×
Linear K=6	151.5	1.20×	167.7	1.26×
Linear K=7	150.7	1.20×	173.4	1.30×
DynaTree D=6	**165.7**	**1.32×**	**185.3**	**1.39×**
DynaTree D=7	160.1	1.27×	184.5	1.39×

speedup), as prefill overhead increases; (iii) DynaTree's relative advantage remains consistent across prompt lengths, with speedups ranging from 1.21× to 1.55×, demonstrating robustness to varying context sizes. These results confirm that DynaTree's tree-based exploration strategy provides stable acceleration benefits across diverse prompt length regimes, making it suitable for applications with varying context requirements.

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 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`.

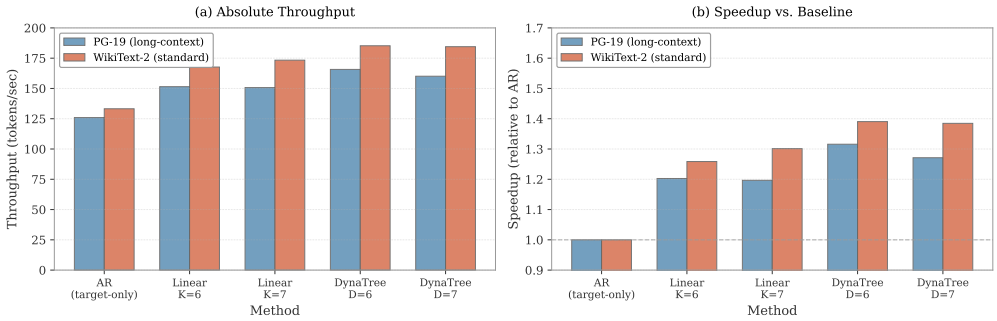

Figure 6: **Cross-dataset performance comparison.** (a) Absolute throughput comparison: all methods achieve higher throughput on WikiText-2 (terra cotta bars) compared to PG-19 (steel blue bars), reflecting WikiText's more predictable structure. (b) Speedup comparison: DynaTree maintains consistent relative gains across both datasets (1.32–1.39×), demonstrating that multi-path exploration benefits are robust to text domain variations. DynaTree consistently outperforms linear speculative methods on both datasets.

Table 6: **Performance across different prompt lengths (WikiText-2, 500-token generation).** We evaluate each method with varying maximum prompt lengths from 100 to 1000 tokens. DynaTree maintains consistent speedups across all prompt lengths, with peak performance at 200 tokens. All methods show slight degradation at very long prompts (1000 tokens) due to increased prefill overhead.

Method	Prompt Length (tokens)			
	100	200	800	1000
AR (baseline)	132.8	127.4	133.2	135.0
Linear K=6	158.4 (1.19×)	175.6 (1.38×)	167.7 (1.26×)	139.8 (1.04×)
Linear K=7	161.7 (1.22×)	178.2 (1.40×)	173.4 (1.30×)	143.1 (1.06×)
DynaTree D=6	**181.2 (1.36×)**	**197.9 (1.55×)**	**185.3 (1.39×)**	**162.8 (1.21×)**
DynaTree D=7	177.0 (1.33×)	198.0 (1.55×)	184.5 (1.39×)	172.6 (1.28×)

[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.

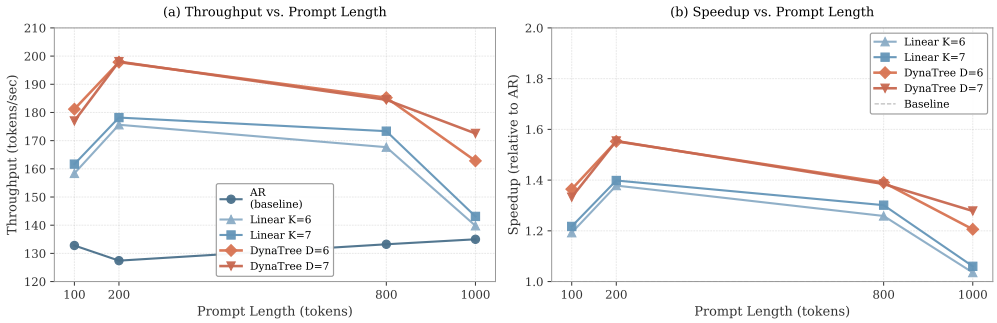

Figure 7: **Performance across different prompt lengths.** (a) Throughput vs. prompt length: all methods achieve peak performance at moderate prompt lengths (200 tokens), with DynaTree D=6 reaching 197.9 t/s. Performance degrades slightly at very long prompts (1000 tokens) due to prefill overhead. (b) Speedup vs. prompt length: DynaTree maintains consistent relative gains (1.21–1.55×) across all prompt lengths, demonstrating robustness to varying context sizes. Linear methods show stronger degradation at 1000 tokens, with speedups dropping to 1.04–1.06×.

[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.

[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 Hyperparameter Sweep Details

We perform an exhaustive grid search over tree depth $D \in \{3, 4, 5, 6, 7, 8\}$, branching factor $B \in \{2, 3, 4\}$, and pruning threshold $\tau \in \{0.01, 0.02, 0.03, 0.05, 0.1\}$ across generation lengths 100–1000 tokens, totaling 450 distinct configurations. Each configuration is evaluated with 2 independent runs (excluding warmup) to estimate average throughput and speedup. Figure 8 visualizes the relationships between hyperparameters and performance, revealing the complex trade-offs between tree exploration, verification cost, and effective path length. The results confirm that no single configuration dominates across all generation lengths, motivating adaptive hyperparameter selection based on the target workload.

B Memory Footprint Analysis

An important practical consideration for speculative decoding methods is their memory overhead. Table 7 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.

Table 7: **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%

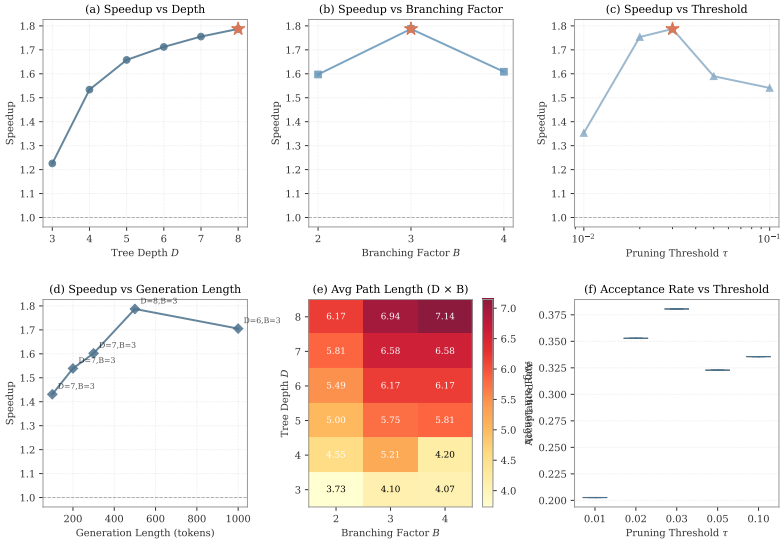

Figure 8: **Comprehensive parameter sweep analysis across 450 configurations.** (a) Speedup vs. tree depth D (fixing $B = 3$, $\tau = 0.03$, 500 tokens): deeper trees improve speedup up to $D = 8$, after which verification overhead dominates. (b) Speedup vs. branching factor B (fixing $D = 8$, $\tau = 0.03$, 500 tokens): $B = 3$ achieves the best balance between exploration and cost. (c) Speedup vs. pruning threshold τ (fixing $D = 8$, $B = 3$, 500 tokens, log scale): $\tau = 0.03$ is optimal, balancing aggressive and loose pruning. (d) Best speedup across generation lengths with corresponding optimal (D, B, τ) configurations: speedup peaks at 500 tokens and remains strong at other lengths. (e) Average path length heatmap over (D, B) (fixing $\tau = 0.03$, 500 tokens): larger trees enable longer committed paths. (f) Acceptance rate distribution vs. pruning threshold (fixing $D = 8$, $B = 3$, 500 tokens): tighter pruning slightly reduces acceptance but improves overall throughput. Stars mark optimal configurations in each subplot.

C 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 $\mathbf{s}_{\text{last}}$; tree depth D; branch factor B; pruning threshold τ; node budget $N_{\max}$.
Ensure: Committed tokens $y_{t+1:t+L}$ and updated $\mathcal{K}_T$.

$\ell \leftarrow \textsc{SeqLen}(\mathcal{K}_T)$ ▷ record prefix cache length
$\mathcal{T} \leftarrow \textsc{DraftTree}(x_{1:t}, D, B, \tau, N_{\max})$
$\mathbf{z}_{1:n} \leftarrow \textsc{BFSFlatten}(\mathcal{T})$; $\mathbf{A} \leftarrow \textsc{TreeMask}(\mathcal{T}, \ell)$ ▷ 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
return $y_{t+1:t+L}$

function $\textsc{DraftTree}(x_{1:t}, D, B, \tau, N_{\max})$ ▷ Draft a candidate token tree with probability-threshold pruning and 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 leaves
  for $d = 1$ **to** D **do**
    if $|\mathcal{A}| = 0$ **or** $|\mathcal{T}| \geq N_{\max}$ **then**
      break
    end if
    $\mathcal{A}' \leftarrow \emptyset$
    for all $u \in \mathcal{A}$ **do**
      if $\bar{\ell}_u < \log \tau$ **then**
        continue
      end if ▷ prune low-probability branches
      Do one cached step of M_D from u; take $\top B$ next-token candidates
      for all candidates v **do**
        if $|\mathcal{T}| \geq N_{\max}$ **then**
          break
        end if
        Add child v to $\mathcal{T}$; add v to $\mathcal{A}'$
      end for
    end for
    $\mathcal{A} \leftarrow \mathcal{A}'$
  end for
  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