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 often employ *fixed* structures that cannot adapt to varying draft model confidence. We propose **DynaTree**, a training-free tree-based speculative decoding framework with *confidence-aware* adaptive drafting that dynamically adjusts tree breadth and depth under an explicit node budget, combined with probability-threshold pruning. In the greedy-decoding setting, DynaTree is exact: it generates the same token sequence as the target model (no distributional bias). Experiments with Pythia-2.8B (target) and Pythia-70M (draft) show that DynaTree improves throughput over autoregressive decoding, linear speculative decoding, and tuned fixed-tree baselines on both WikiText-2 and PG-19. At $T = 1500$, DynaTree reaches 219.5 tokens/s on WikiText-2 (1.64×) and 194.9 tokens/s on PG-19 (1.70×).

1 Introduction

Autoregressive decoding remains the default generation mode for large language models (LLMs), but it is inherently sequential: each token requires a forward pass conditioned on the full prefix. While transformer inference can exploit parallelism during prefill, the decode stage offers limited parallelism and is often bottlenecked by memory traffic and per-token kernel launch overhead [5, 8].

Speculative decoding mitigates this bottleneck by separating *proposal* and *verification* [10]. A lightweight draft model proposes candidate tokens and the target model verifies them in parallel; when verification succeeds, multiple tokens are committed per iteration. With rejection sampling, speculative decoding preserves the exact output distribution of the target model [26].

Most deployed speculative decoders are *linear*: the draft model proposes a single chain of K tokens. This design is brittle under draft–target mismatch: a rejection at an early position discards all downstream draft tokens, wasting both draft computation and target-model verification work [26]. Tree-based speculation offers a natural remedy by exploring multiple continuations in parallel and verifying a token tree with a structured attention mask, increasing the probability that at least one branch matches the target model's greedy continuation [13, 20].

However, existing tree-based approaches typically employ *fixed* tree shapes with predetermined depth and branching factor [3, 13]. A fixed structure cannot adapt to varying draft confidence across

*Equal contribution.

Preprint.

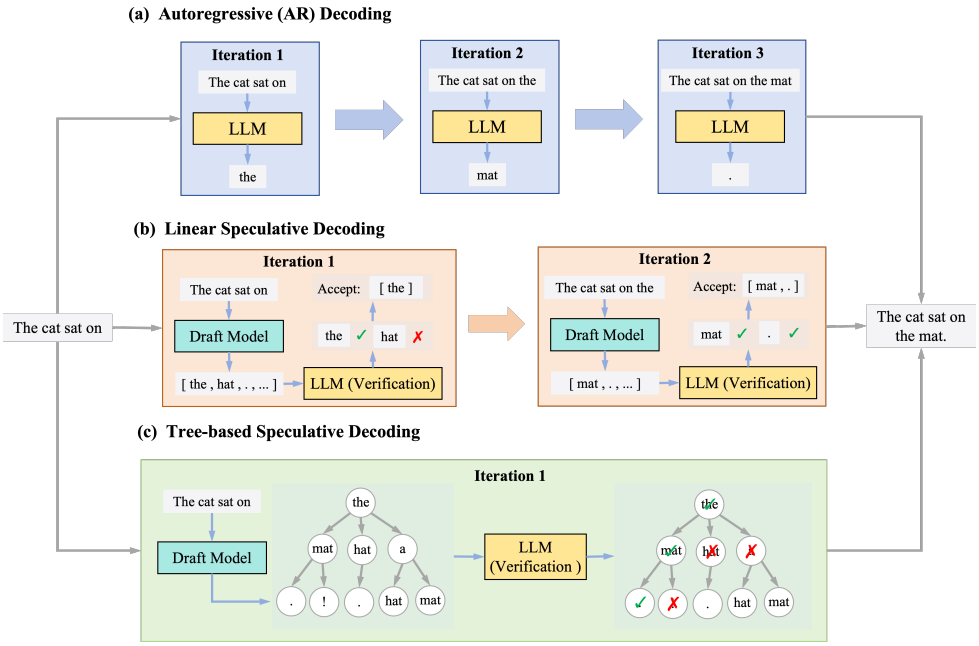

Figure 1: **Comparison of three decoding paradigms. (a) Autoregressive (AR):** the target LLM generates one token per iteration sequentially. **(b) Linear speculative decoding:** a draft model proposes a length-K chain that is verified by the target LLM; once an early token is rejected, all subsequent drafted tokens in the chain are discarded, reducing effective progress per iteration. **(c) Tree-based speculative decoding (e.g., DynaTree):** the draft model proposes a token tree and the target LLM verifies all candidates in parallel (one forward pass with a structured attention mask), then commits the longest greedy-consistent path (plus a bonus token), improving robustness to draft mismatch.

contexts: it may over-explore when the draft distribution is peaked, and under-explore when the next-token distribution is flat, creating an *efficiency gap*. Recent adaptive methods adjust draft length or verification thresholds [14, 18, 28], but most focus on linear speculation rather than restructuring the tree itself.

We propose **DynaTree**, a training-free tree speculative decoder that constructs a draft token tree with *confidence-aware* adaptive drafting under an explicit node budget. DynaTree combines *Dynamic Tree Breadth*, *Dynamic Tree Depth*, and *History Adaptation* to stabilize verification efficiency across iterations. Using tree attention, all drafted nodes are verified in a single target-model forward pass. Empirically, DynaTree improves throughput over autoregressive decoding, linear speculation, and tuned fixed-tree baselines on both WikiText-2 and PG-19 (Table 1).

In summary, our contributions are:

- We propose **DynaTree**, a training-free tree speculative decoding framework that adaptively allocates verification budget via confidence-aware adjustments to tree breadth and depth under an explicit node budget.
- We introduce a lightweight three-component adaptation mechanism (*Dynamic Tree Breadth*, *Dynamic Tree Depth*, and *History Adaptation*) that integrates with tree-attention verification and probability-threshold pruning.
- Experiments on WikiText-2 and PG-19 demonstrate consistent throughput and latency improvements over autoregressive decoding, linear speculative decoding, and tuned fixed-tree baselines, with up to $1.70\times$ speedup at $T = 1500$.

Figure 1 provides a schematic comparison of autoregressive decoding, linear speculative decoding, and tree-based speculative decoding.

2 Related Work

2.1 Speculative Decoding

Speculative decoding improves generation efficiency by verifying draft proposals in parallel while preserving target-model correctness [10, 24, 26]. Systems work highlights that decoding performance is often memory-bound and sensitive to batching, context length, and workload heterogeneity [4, 5, 8, 9]. Recent studies further emphasize serving-oriented considerations for speculative decoding under realistic deployment constraints [1, 19].

2.2 Tree-Based Speculative Decoding

Tree-based speculative decoding generalizes linear drafting by verifying a token tree with a structured attention mask in a single target-model pass [3, 13, 20, 22]. SpecInfer [13] pioneered practical token-tree verification mechanisms, while OPT-Tree [20] searches for draft tree structures that improve the expected committed prefix length. Medusa [3] explores multi-token prediction via additional decoding heads. Recent work further improves the efficiency of tree-structured verification via optimized tree-attention implementations [27]. These approaches typically employ fixed tree structures, which motivates adaptive policies that allocate verification budget based on context-dependent uncertainty.

Adaptive speculative decoding modulates drafting aggressiveness using confidence signals or learned predictors [14, 18, 28]. In contrast to adaptive *length*-only policies, DynaTree adapts the tree structure itself by jointly controlling *Dynamic Tree Breadth* and *Dynamic Tree Depth* and incorporating *History Adaptation*, while remaining training-free.

2.3 Dynamic Pruning Strategies

Exponential candidate growth necessitates pruning to balance exploration against verification cost. Prior work studies early pruning and confidence-guided expansion [25, 29], cost-aware tree construction and pruning [6, 20], and retrieval-augmented pruning heuristics [16]. Other lines of work adapt speculative hyperparameters online [7] or propose alternative parallel decoding schemes beyond a single draft–verify pair [11, 21]. DynaTree adopts probability-threshold pruning under an explicit node budget and focuses on training-free, confidence-aware tree restructuring.

3 Methodology

3.1 Problem Setup and Notation

Let M_T denote a target autoregressive language model and M_D a smaller draft model over vocabulary $\mathcal{V}$. We write $p_T(\cdot \mid \cdot)$ and $p_D(\cdot \mid \cdot)$ for their next-token distributions. Given a prefix (prompt) $x_{1:t}$, greedy decoding with M_T produces tokens $y_{t+1}, y_{t+2}, \ldots$ via

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

Speculative decoding accelerates generation by proposing candidate tokens with M_D and verifying them with M_T. In this paper we focus on greedy decoding; correctness requires that a token is committed only if it matches the target model's greedy argmax under the corresponding conditioning context.

3.2 Overview of DynaTree

DynaTree generalizes linear speculative decoding from a single draft chain to a *draft token tree*. In each iteration, it constructs a candidate tree with M_D and verifies all drafted nodes in a single forward pass of M_T using tree attention. DynaTree's key innovation is *confidence-aware adaptive drafting* that allocates verification budget based on draft uncertainty: **(i) Dynamic Tree Breadth** selects a per-node branching factor from $\{B_{\min}, B_{\text{mid}}, B_{\max}\}$ using confidence thresholds (τ_h, τ_ℓ); **(ii) Dynamic Tree Depth** gates expansion using cumulative path probability with early stopping and selective deep expansion up to $D_{\max}$; and **(iii) History Adaptation** updates a small subset of drafting hyperparameters online using recent acceptance statistics. Throughout, we use **Adaptive Pruning** to denote the combination of probability-threshold pruning with a strict node-budget cap (Figure 3c).

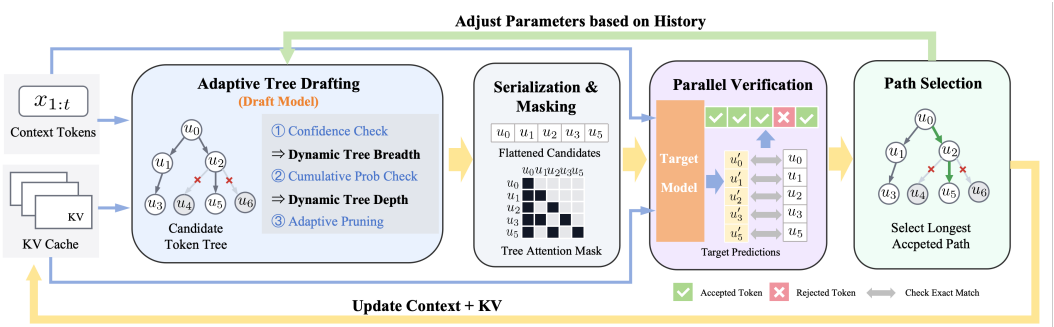

Figure 2: **One iteration of DynaTree decoding (end-to-end pipeline).** (1) **Adaptive Tree Drafting:** the draft model expands a candidate token tree under node budget $N_{\max}$ with confidence-aware breadth/depth and probability-threshold pruning. (2) **Serialization & Masking:** the pruned tree is serialized (BFS order) and converted into a tree-attention mask. (3) **Parallel Verification:** the target model verifies all drafted candidates in a single forward pass. (4) **Path Selection:** select the longest greedy-consistent (accepted) path and commit it (plus a bonus token). (5) **Update Context + KV:** committed tokens update the context and target-model KV cache. (6) **Adjust Parameters based on History:** recent acceptance statistics adjust drafting hyperparameters for the next iteration.

Figure 2 illustrates the end-to-end pipeline of a single DynaTree iteration.

3.3 Adaptive Tree Drafting

Figure 3 provides a schematic view of DynaTree's three adaptive mechanisms: Dynamic Tree Breadth, Dynamic Tree Depth, and Adaptive Pruning.

We maintain a token tree $\mathcal{T} = (\mathcal{N}, \mathcal{E})$ rooted at the current prefix $x_{1:t}$, where each node $u \in \mathcal{N}$ corresponds to a drafted token. Each node u is associated with: (i) token $z_u \in \mathcal{V}$; (ii) parent $\pi(u)$; (iii) depth $d(u)$ from the root; (iv) draft log-probability $\ell_u = \log p_D(z_u \mid \mathrm{ctx}(\pi(u)))$, where $\mathrm{ctx}(\pi(u))$ denotes the unique root-to-$\pi(u)$ token context; and (v) cumulative log-probability $\bar{\ell}_u = \sum_{v \in \mathrm{path}(u)} \ell_v$, where $\mathrm{path}(u)$ denotes the nodes on the root-to-u path. We also use the cumulative path probability $p(u) = \exp(\bar{\ell}_u)$.

Dynamic Tree Breadth. 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

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

where $0 < \tau_\ell < \tau_h < 1$ are confidence thresholds and $1 \leq B_{\min} \leq B_{\mathrm{mid}} \leq B_{\max}$ 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 B.

Dynamic Tree Depth. Let D_0 denote a base depth and $D_{\max}$ a hard maximum depth. We gate expansion using two probability thresholds $\rho_{\mathrm{stop}} < \rho_{\mathrm{deep}}$ on the cumulative path probability $p(u)$. A

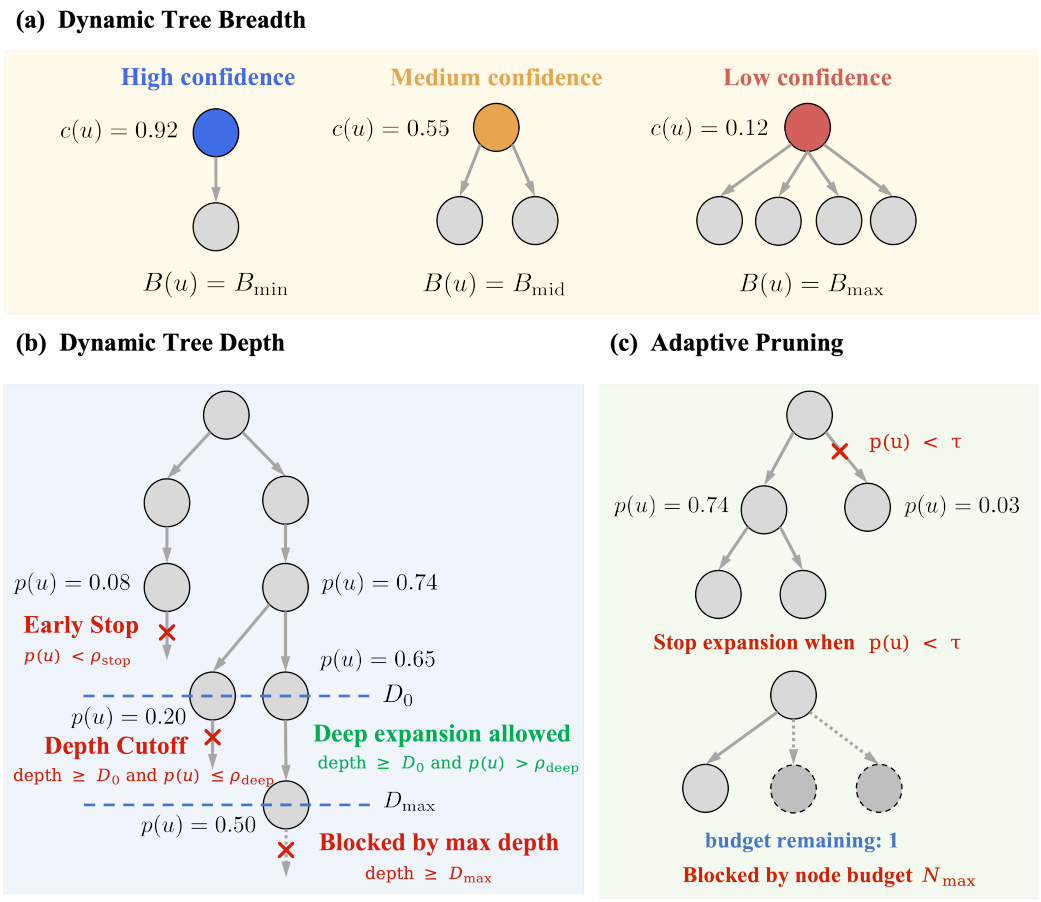

Figure 3: **Illustration of DynaTree's adaptive mechanisms. (a) Dynamic Tree Breadth:** choose per-node branching $B(u) \in \{B_{\min}, B_{\text{mid}}, B_{\max}\}$ based on local confidence $c(u)$. **(b) Dynamic Tree Depth:** control expansion using cumulative path probability $p(u)$ with early stopping ($p(u) < \rho_{\text{stop}}$), a base depth D_0, and selective deep expansion beyond D_0 only when $p(u) > \rho_{\text{deep}}$, up to a hard maximum depth $D_{\max}$. **(c) Adaptive Pruning:** stop expanding nodes whose $p(u) < \tau$ and cap the tree size by a strict node budget $N_{\max}$.

node u at depth $d(u)$ is expandable if and only if

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

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 a small subset of drafting hyperparameters online using recent verification outcomes. Let $a_r \in [0, 1]$ denote the per-iteration acceptance statistic at

iteration r (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. We use a simple proportional-control update around a target acceptance level $a^\star \in (0,1)$. For example, for base depth D_0 and high-confidence threshold τ_h,

$$D_0 \leftarrow \mathrm{clip}(D_0 + \eta_D(\bar{a}_t - a^\star),\, 1,\, D_{\max} - 1)\,, \quad \tau_h \leftarrow \mathrm{clip}(\tau_h - \eta_h(\bar{a}_t - a^\star),\, 0,\, 1)\,,$$

with step sizes $\eta_D, \eta_h > 0$ and $\mathrm{clip}(\cdot)$ enforcing valid ranges. Intuitively, if recent acceptance is high ($\bar{a}_t > a^\star$), we increase drafting aggressiveness; otherwise we become more conservative. The exact parameter subset and schedule follow the same principle and are provided in Appendix B.

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

3.4 Tree Attention for Parallel Verification

To verify all drafted tokens in one target-model forward pass, we *serialize* the tree in breadth-first order (BFS)—equivalently, *flatten* it into an ordered node list $(u_1, \ldots, u_n)$ and token sequence $z_{1:n}$ where $z_i = z_{u_i}$. Let $\mathrm{pos}(u_i) = i$. We then construct a boolean tree-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) among drafted nodes:

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

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 [13, 20].

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 serialized (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. While this rollback-and-rebuild introduces extra target-model work, its cost scales only with the number of committed tokens in that iteration (rather than the full drafted tree) and is amortized by committing multiple tokens per verification round.

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 [2]. Our target model M_T is Pythia-2.8B (2.8B parameters) and our draft model M_D is Pythia-70M (70M parameters). Throughout, we use deterministic greedy decoding so the generated sequence is uniquely determined by the model and prefix.

Hardware and software. All experiments run on a single NVIDIA A100 GPU with batch size 1 (one prompt per run). We implement DynaTree in PyTorch [15] on top of HuggingFace Transformers [23]. Across methods, we reuse KV caches where applicable and synchronize GPU execution for timing to obtain comparable wall-clock measurements.

Workloads and data preprocessing. We report results on WikiText-2 [12] and PG-19 [17]. For each sampled prompt, we generate T new tokens with greedy decoding; unless stated otherwise, $T = 1500$. To control prefill cost, prompts are truncated to a maximum length $L_{\max}$ ($L_{\max} = 800$ for WikiText-2 and $L_{\max} = 1000$ for PG-19). We evaluate $N = 10$ prompts and discard the first $W = 2$ runs as warmup, reporting mean and standard deviation over the remaining runs. Appendix A summarizes the common settings.

4.2 Evaluation Metrics

We use **throughput** (tokens/s) as the primary metric, computed as T divided by the wall-clock decoding time (excluding warmup). We report **speedup** relative to autoregressive decoding, $\text{speedup} = \text{TPS}/\text{TPS}_{\text{AR}}$, under the same dataset and prompt-length cap. To characterize verification efficiency, we report the **acceptance rate** a (fraction of drafted tokens matching the target model's greedy predictions under the corresponding conditioning contexts) and the average **tokens per iteration** $\bar{L}$ (tokens committed per verification round). For tree-based methods, we additionally report the **average committed path length** $\bar{\ell}$ (mean depth of the greedy-consistent committed path before the bonus token). For latency, we include **time-to-first-token** (TTFT) and **time-per-output-token** (TPOT), averaged over prompts.

4.3 Baselines

We compare against the following baselines 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 [10].

Table 1: **Main results ($T = 1500$).** Throughput (tokens/s) and speedup relative to autoregressive decoding on WikiText-2 and PG-19. Values are mean±std over prompts (excluding warmup). For linear speculation, we use $K = 8$ on WikiText-2 and $K = 5$ on PG-19.

Method	WikiText-2		PG-19	
	Throughput (tokens/s)	Speedup	Throughput (tokens/s)	Speedup
AR	133.4±0.5	1.00×	114.8±20.6	1.00×
Linear Spec	196.1±37.8	1.47×	144.9±28.6	1.26×
Fixed Tree ($D = 8, B = 3, \tau = 0.1$)	200.7±41.7	1.50×	185.5±33.2	1.62×
DynaTree	**219.5±22.2**	**1.64×**	**194.9±35.6**	**1.70×**

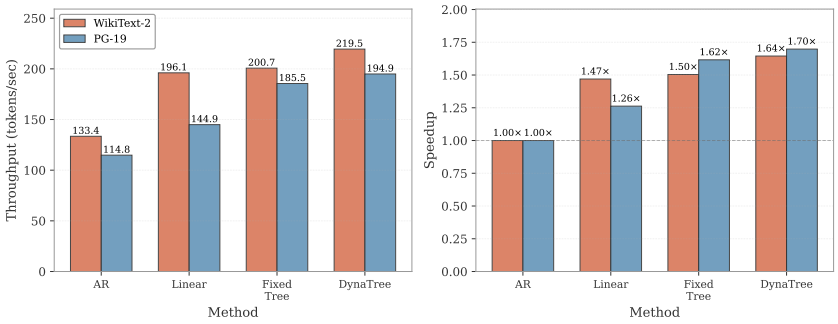

Figure 4: **Throughput and speedup across datasets ($T = 1500$).** Each method is shown with two bars (WikiText-2 vs. PG-19). DynaTree consistently improves over autoregressive and linear speculative decoding on both datasets.

- **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. We report the configuration used wherever results are shown.
- **DynaTree:** Our full method that augments the fixed-tree backbone with (i) *Dynamic Tree Breadth*, (ii) *Dynamic Tree Depth*, and (iii) *History Adaptation*. We ablate these components in Section 4.5.

4.4 Main Results

Table 1 reports end-to-end throughput on WikiText-2 and PG-19 for $T = 1500$. DynaTree achieves the best throughput on both datasets, improving over autoregressive decoding, linear speculative decoding, and a fixed-tree baseline. On WikiText-2, DynaTree reaches 219.5 tokens/s versus 200.7 tokens/s for the static tree, demonstrating that adapting the draft structure can improve the draft–verify trade-off over a fixed configuration under the same evaluation protocol. To ensure a fair comparison, the fixed-tree baseline configuration is selected via a grid search under the same evaluation protocol and reported as the best (or near-best) setting in the sweep (Appendix, Figure 7).

Figure 4 visualizes throughput and speedup. Linear speculative decoding can attain high acceptance when the draft closely matches the target; however, an early mismatch truncates the committed prefix and wastes the remaining drafted tokens in the chain. Tree-based drafting mitigates this failure mode by exploring multiple continuations in parallel and committing the longest greedy-consistent prefix found in the candidate set. DynaTree further reallocates verification budget across the tree based on uncertainty, reducing wasted verification in low-confidence regions while preserving longer committed prefixes in high-confidence regions.

Latency breakdown analysis. Table 2 reports TTFT and TPOT for $T = 1500$. Across datasets, speculative decoding reduces TPOT by amortizing a target-model verification over multiple output tokens; DynaTree achieves the lowest TPOT among compared methods. TTFT reflects both prefill and the first decode iteration and is therefore sensitive to implementation and cache reuse; under our implementation, speculative methods also reduce TTFT.

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). For linear speculation, we use $K = 8$ on WikiText-2 and $K = 5$ on PG-19.

Method	WikiText-2		PG-19	
	TTFT (ms)	TPOT (ms)	TTFT (ms)	TPOT (ms)
AR	18.9±6.1	7.48±0.02	21.8±8.7	9.00±1.69
Linear Spec	14.6±4.0	5.32±1.24	11.1±3.3	7.17±1.42
Fixed Tree ($D = 8, B = 3, \tau = 0.1$)	14.6±4.2	5.30±1.63	9.7±0.4	5.55±0.95
DynaTree	14.4±4.0	**4.59±0.47**	9.6±0.6	**5.30±1.02**

Table 3: **Verification efficiency metrics ($T = 1500$).** We report acceptance rate (**Accept.**), tokens committed per verification iteration ($\bar{L}$), average committed path length before the bonus token ($\bar{\ell}$), and the total number of verification iterations (#Iter.) needed to generate T tokens. Due to the bonus-token convention in tree verification, **Accept.** can slightly exceed 100% for tree-based methods. Values are mean across prompts (excluding warmup).

Method	WikiText-2				PG-19			
	Accept.	$\bar{L}$	$\bar{\ell}$	#Iter.	Accept.	$\bar{L}$	$\bar{\ell}$	#Iter.
AR	–	1.00	–	1500	–	1.00	–	1500
Linear Spec	88.2%	6.82	7.05	220	92.5%	4.56	4.63	329
Fixed Tree	71.0%	6.79	7.10	221	64.3%	6.20	6.43	242
DynaTree	**102.2%**	**7.08**	**7.15**	**212**	**92.2%**	**6.17**	**6.45**	**243**

Verification efficiency. To contextualize throughput gains, Table 3 reports auxiliary verification statistics: tokens committed per verification round ($\bar{L}$), average committed path length ($\bar{\ell}$), and the number of verification rounds required to generate $T = 1500$ tokens. Relative to autoregressive decoding, speculative methods reduce the number of target-model rounds by committing multiple tokens per round; tree-based methods further increase robustness to draft mismatches by verifying multiple continuations. DynaTree improves efficiency by adapting the draft structure online, yielding fewer rounds and larger $\bar{L}$ under the same node budget.

Discussion. Two trends help explain the observed speedups. First, compared with linear drafting, tree-based methods are less sensitive to early mismatches: when the draft model diverges, alternative branches can still contain a greedy-consistent continuation, which increases the expected committed prefix length per verification step. Second, compared with a fixed tree, DynaTree allocates more of the node budget to regions where the draft model is confident while curtailing wasteful expansion in uncertain regions. This adaptivity improves verification efficiency (Table 3) and translates into higher end-to-end throughput (Table 1).

4.5 Ablation Study

We provide a progressive ablation on WikiText-2 under the same $T = 1500$ setting as the main benchmark (Table 4), measuring the contribution of each adaptive component relative to a fixed-tree backbone. Since *Dynamic Tree Breadth* and *Dynamic Tree Depth* are coupled in how they trade verification breadth against the length of the committed path, we report them jointly as *Dynamic Tree Breadth & Depth*. We then add *History Adaptation*, which adjusts drafting hyperparameters based on recent verification outcomes.

Interpretation. The throughput gains correlate with increased per-iteration progress ($\bar{L}$) and fewer verification iterations. History Adaptation provides an additional improvement by adjusting draft aggressiveness online, yielding higher throughput while keeping verification efficiency metrics stable (Figure 5).

Table 4: **Ablation-style progressive comparison on WikiText-2 ($T = 1500$).** We compare a fixed tree ($D = 5, B = 2$) with variants that add Dynamic Tree Breadth & Depth and History Adaptation. Speedup is computed relative to the autoregressive baseline.

Variant	Throughput (tokens/s)	Speedup	Δ vs Fixed
Fixed Tree ($D = 5, B = 2$)	188.0±16.5	1.42×	0.0%
+ Dynamic Tree Breadth & Depth	213.2±26.1	1.61×	+13.4%
+ History Adaptation	**218.5±22.2**	**1.65×**	**+16.2%**

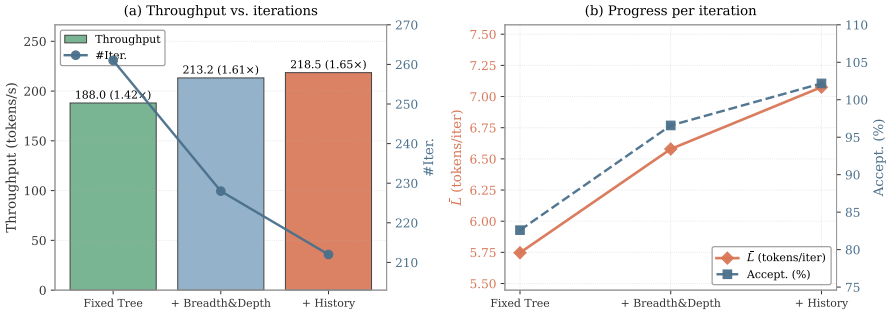

Figure 5: **Ablation progression on WikiText-2 ($T = 1500$).** Each panel combines two complementary metrics. Left: throughput (bars) alongside the number of verification iterations (#Iter.). Right: per-iteration progress ($\bar{L}$, tokens/iter) alongside acceptance rate (separate axis). Dynamic Tree Breadth & Depth increases per-step progress and reduces iterations; History Adaptation further improves per-step progress and iteration count.

4.6 Sequence Length Scaling

We evaluate how performance varies with generation length T on WikiText-2, comparing autoregressive decoding, linear speculative decoding, a fixed tree baseline, and DynaTree. Figure 6 summarizes both end-to-end throughput and auxiliary verification statistics across T.

Analysis. Across T, DynaTree maintains strong throughput by sustaining larger committed progress per verification round while keeping acceptance rates relatively stable. In contrast, linear speculation is more sensitive to occasional early mismatches: a single rejection can truncate the reusable drafted chain and reduce effective progress per round. The auxiliary panels highlight how $\bar{L}$, $\bar{\ell}$, and acceptance jointly determine end-to-end throughput via their effect on the number of verification rounds.

Additional analyses. We place supplementary parameter sensitivity analyses in Appendix C.

5 Conclusion

We presented DynaTree, a greedy-consistent tree-based speculative decoding method that drafts a candidate token tree with a lightweight model and verifies all drafted nodes in a single target-model pass using a structured attention mask. DynaTree adaptively allocates verification budget via *Dynamic Tree Breadth* and *Dynamic Tree Depth* and stabilizes decoding through *History Adaptation*. Across WikiText-2 and PG-19 at $T = 1500$, DynaTree improves throughput and reduces per-token latency relative to autoregressive decoding, linear speculative decoding, and a fixed-tree baseline by increasing the expected number of committed tokens per verification iteration. Our evaluation is limited to deterministic greedy decoding with a single target–draft model pair on a single GPU, and does not cover batch-serving scenarios. Extending adaptive tree verification to batched serving, additional model pairs, and broader workloads, while further reducing system overhead, is a promising direction for future work.

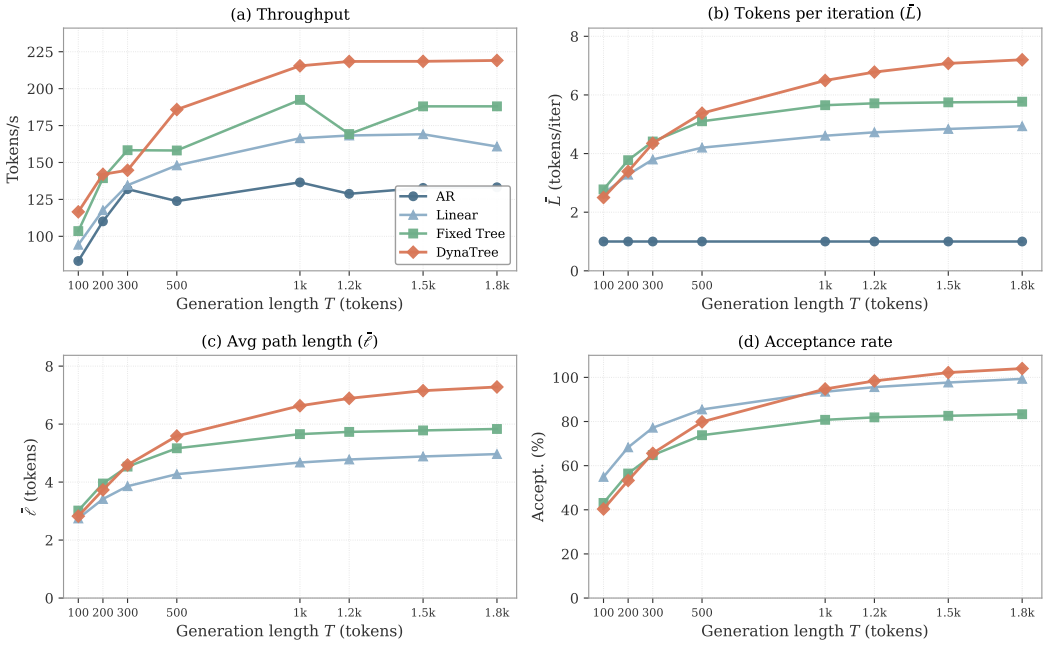

Figure 6: **Sequence length scaling on WikiText-2.** (a) Throughput (tokens/s) as a function of T for AR, linear speculative decoding, a fixed tree baseline, and DynaTree. (b–d) Auxiliary verification statistics: tokens per iteration ($\bar{L}$), average committed path length ($\bar{\ell}$), and acceptance rate.

References

[1] 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.

[2] 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. URL https://arxiv.org/abs/2304.01373.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Yunlong Hou, Fengzhuo Zhang, Cunxiao Du, Xuan Zhang, Jiachun Pan, Tianyu Pang, Chao Du, Vincent Y. F. Tan, and Zhuoran Yang. Banditspec: Adaptive speculative decoding via bandit algorithms, 2025. URL https://arxiv.org/abs/2505.15141.

[8] 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.

[9] 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.

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

[11] Yuxuan Liu, Wenyuan Li, Laizhong Cui, and Hailiang Yang. Cerberus: Efficient inference with adaptive parallel decoding and sequential knowledge enhancement, 2024. URL https://arxiv.org/abs/2410.13344.

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

[13] 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, page 932–949. ACM, April 2024. doi: 10.1145/3620666.3651335. URL http://dx.doi.org/10.1145/3620666.3651335.

[14] Zhiyuan Ning, Jiawei Shao, Ruge Xu, Xinfei Guo, Jun Zhang, Chi Zhang, and Xuelong Li. Casspec: Cascade adaptive self-speculative decoding for on-the-fly lossless inference acceleration of llms, 2025. URL https://arxiv.org/abs/2510.26843.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach 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. URL https://arxiv.org/abs/1912.01703.

[16] Guofeng Quan, Wenfeng Feng, Chuzhan Hao, Guochao Jiang, Yuewei Zhang, and Hao Wang. Rasd: Retrieval-augmented speculative decoding, 2025. URL https://arxiv.org/abs/2503.03434.

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

[18] Jaydip Sen, Subhasis Dasgupta, and Hetvi Waghela. Confidence-modulated speculative decoding for large language models, 2025. URL https://arxiv.org/abs/2508.15371.

[19] Bangsheng Tang, Carl Chengyan Fu, Fei Kou, Grigory Sizov, Haoci Zhang, Jason Park, Jiawen Liu, Jie You, Qirui Yang, Sachin Mehta, Shengyong Cai, Xiaodong Wang, Xingyu Liu, Yunlu Li, Yanjun Zhou, Wei Wei, Zhiwei Zhao, Zixi Qi, Adolfo Victoria, Aya Ibrahim, Bram Wasti, Changkyu Kim, Daniel Haziza, Fei Sun, Giancarlo Delfin, Emily Guo, Jialin Ouyang, Jaewon Lee, Jianyu Huang, Jeremy Reizenstein, Lu Fang, Quinn Zhu, Ria Verma, Vlad Mihailescu, Xingwen Guo, Yan Cui, Ye Hu, and Yejin Lee. Efficient speculative decoding for llama at scale: Challenges and solutions, 2025. URL https://arxiv.org/abs/2508.08192.

[20] 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, 2025. URL https://arxiv.org/abs/2406.17276.

[21] Siqi Wang, Hailong Yang, Xuezhu Wang, Tongxuan Liu, Pengbo Wang, Xuning Liang, Kejie Ma, Tianyu Feng, Xin You, Yongjun Bao, Yi Liu, Zhongzhi Luan, and Depei Qian. Minions: Accelerating large language model inference with aggregated speculative execution, 2024. URL https://arxiv.org/abs/2402.15678.

[22] 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.

[23] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi 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. Huggingface's transformers: State-of-the-art natural language processing, 2020. URL https://arxiv.org/abs/1910.03771.

[24] 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.

[25] 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.

[26] Minghao Yan, Saurabh Agarwal, and Shivaram Venkataraman. Decoding speculative decoding, 2025. URL https://arxiv.org/abs/2402.01528.

[27] Jinwei Yao, Kaiqi Chen, Kexun Zhang, Jiaxuan You, Binhang Yuan, Zeke Wang, and Tao Lin. DeFT: Decoding with flash tree-attention for efficient tree-structured LLM inference. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=2c7pfOqu9k.

[28] Situo Zhang, Hankun Wang, Da Ma, Zichen Zhu, Lu Chen, Kunyao Lan, and Kai Yu. Adaeagle: Optimizing speculative decoding via explicit modeling of adaptive draft structures, 2024. URL https://arxiv.org/abs/2412.18910.

[29] 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.

A Experimental Configuration Details

Data traceability. All appendix figures and tables are generated from the logged JSON result files included in the repository. We do not introduce any unlogged (hand-entered) experimental numbers. When a result file contains an auxiliary `speedup` field, we recompute speedup as $\mathrm{TPS}/\mathrm{TPS}_{\mathrm{AR}}$ under the same setting (Section 4.4).

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. Reported values are mean and standard deviation over the remaining runs.

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. We use a static-tree speculative decoder with a fixed (D, B, τ) configuration and node budget $N_{\max} = 256$. We report the exact configuration used alongside the corresponding results. We additionally report a fixed-tree hyperparameter sweep under the paper protocol in Appendix C to verify that the static-tree baseline used in the main benchmark is reasonably tuned.

DynaTree. Unless stated otherwise, the adaptive configuration uses base depth $D_0 = 5$, maximum depth $D_{\max} = 8$ on WikiText-2 ($D_{\max} = 9$ on PG-19), branch bounds $(B_{\min}, B_{\max}) = (1, 3)$, and confidence thresholds $(\tau_h, \tau_\ell) = (0.9, 0.4)$. History Adaptation updates a small subset of these parameters based on recent verification outcomes.

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
$\mathcal{T} \leftarrow \textsc{DraftTree}(x_{1:t}, B_{\min}, B_{\text{mid}}, B_{\max}, \tau_\ell, \tau_h, D_0, D_{\max}, \rho_{\text{stop}}, \rho_{\text{deep}}, \tau, N_{\max})$
$\mathbf{z}_{1:n} \leftarrow \textsc{BFSSerialize}(\mathcal{T})$; $\mathbf{A} \leftarrow \textsc{TreeAttnMask}(\mathcal{T}, \ell)$ ▷ BFS serialization + tree attention mask (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 Dynamic Tree Breadth, Dynamic Tree Depth, 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** $\exp(\bar{\ell}_u) \le \rho_{\text{deep}}$ **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

B DynaTree Iteration Pseudocode

Reference implementation. Algorithm 1 provides a self-contained pseudocode listing for one greedy-consistent DynaTree iteration, matching the pipeline in Figure 2.

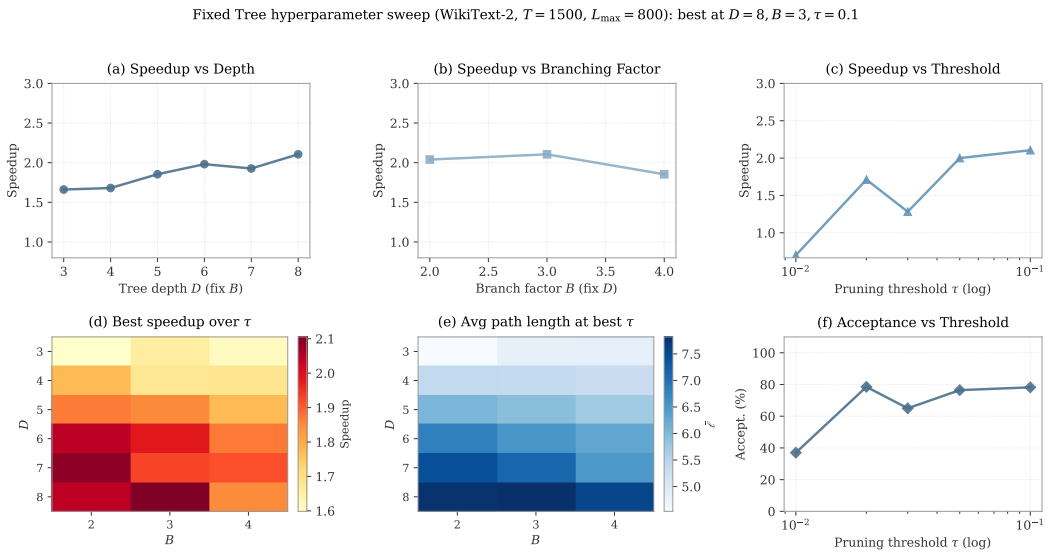

Figure 7: **Fixed-tree hyperparameter sweep under the paper protocol (WikiText-2,** $T = 1500$**).** Speedup is computed relative to autoregressive decoding under the same setting. Line plots use widened y-axis ranges to emphasize robustness across depth, branching, and threshold settings.

C Additional Experimental Analyses

Scope. This appendix reports additional analyses that support the main claims: (i) a fixed-tree sweep that validates the static-tree baseline is reasonably tuned; (ii) sensitivity of DynaTree to drafting hyperparameters; (iii) prompt-length sensitivity; and (iv) memory footprint.

C.1 Fixed-tree Hyperparameter Sweep

We perform a fixed-tree hyperparameter sweep under the paper protocol to ensure the static-tree baseline is reasonably tuned. The sweep varies depth D, branching factor B, and pruning threshold τ while holding the node budget fixed. Figure 7 summarizes how throughput-related metrics change across the grid, and illustrates that the baseline configuration reported in the main tables lies in a stable high-performing region rather than being a brittle outlier.

C.2 Parameter Sensitivity

We study the sensitivity of DynaTree to key drafting hyperparameters on WikiText-2 at $T = 1500$ using a comprehensive sweep that varies confidence thresholds, branch bounds, depth ranges, and selected cross-combinations. Since the sweep does not enumerate a full Cartesian grid for every parameter pair, we visualize thresholds as a sparse 2D scatter and use mean$\pm$std plots for breadth and depth.

C.3 Prompt Length Sensitivity

We evaluate the impact of input context length by varying the maximum prompt length $L_{\max}$ on WikiText-2 under the same $T = 1500$ generation setting as the main benchmark. Figure 9 reports throughput and speedup as functions of $L_{\max}$. This analysis isolates how longer prompts (higher prefill cost and longer KV caches) interact with speculative verification, under otherwise fixed decoding settings.

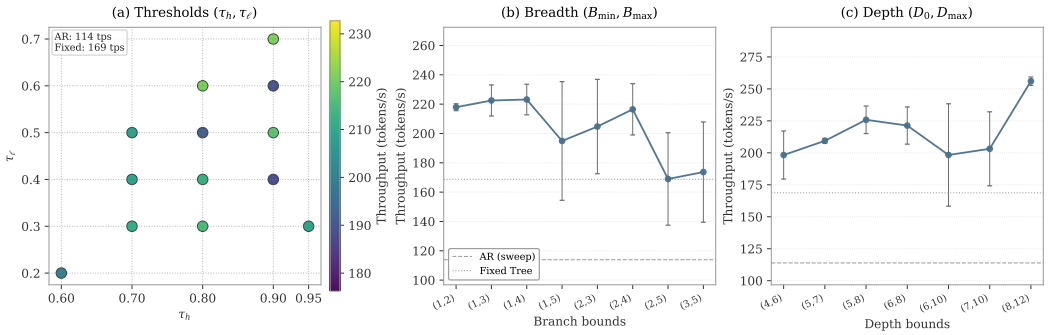

Figure 8: **Sensitivity of DynaTree to drafting hyperparameters (WikiText-2, $T = 1500$). (a)** Threshold sweep shown as a sparse 2D scatter over (τ_h, τ_ℓ), colored by throughput (tokens/s). **(b–c)** Breadth and depth sweeps shown as throughput mean±std across tested pair configurations. Horizontal lines denote the autoregressive and fixed-tree baselines measured in the same sweep run.

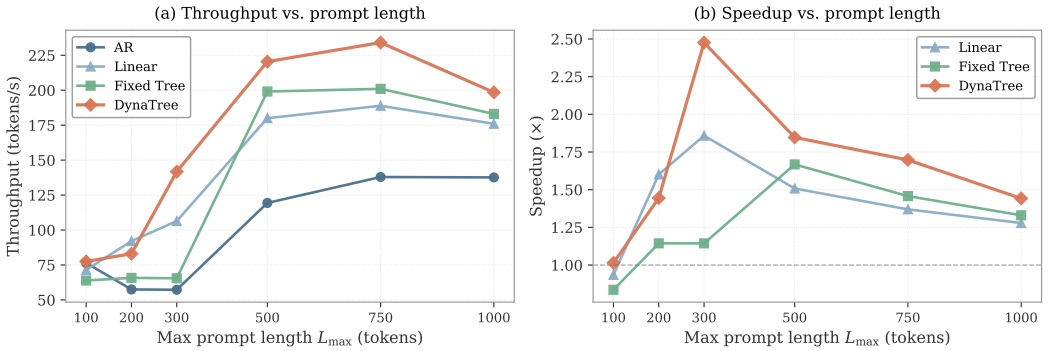

Figure 9: **Prompt length sensitivity on WikiText-2 ($T = 1500$).** Throughput and speedup (relative to AR at the same $L_{\max}$) as functions of the maximum prompt length.

C.4 Memory Footprint Analysis

An important practical consideration for speculative decoding methods is their memory overhead. Table 5 reports peak GPU memory consumption across methods on PG-19 and WikiText-2 during the $T = 1500$ benchmark corresponding to Tables 1–2. Overall, speculative decoding introduces a modest additional peak allocation (about 3% on average in this setup) to maintain the draft-model KV cache and intermediate verification structures.

Table 5: **Peak GPU memory consumption comparison ($T = 1500$).** Peak GPU memory (MB) during generation on PG-19 and WikiText-2 with Pythia-2.8B and Pythia-70M. Relative change is computed against the autoregressive baseline using the average of the two datasets. Linear speculative decoding uses $K = 5$ on PG-19 and $K = 8$ on WikiText-2 (matching Table 1).

Method	Peak Memory (MB)		Average (MB)	Rel. Change
	PG-19	WikiText-2		
AR (baseline)	6156.5	6110.0	6133.3	0.00%
Linear Spec	6363.0	6315.3	6339.2	+3.36%
Fixed Tree (D=8, B=3, τ=0.1)	6354.6	6316.9	6335.7	+3.30%
DynaTree	**6355.0**	**6316.4**	**6335.7**	**+3.30%**