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. 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 [1, 2].

Speculative decoding mitigates this bottleneck by separating *proposal* and *verification* [3]. 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 [4].

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 [4]. 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 [5, 6].

However, existing tree-based approaches typically employ *fixed* tree shapes with predetermined depth and branching factor [5, 7]. A fixed structure cannot adapt to varying draft confidence across 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

*Equal contribution.

Preprint.

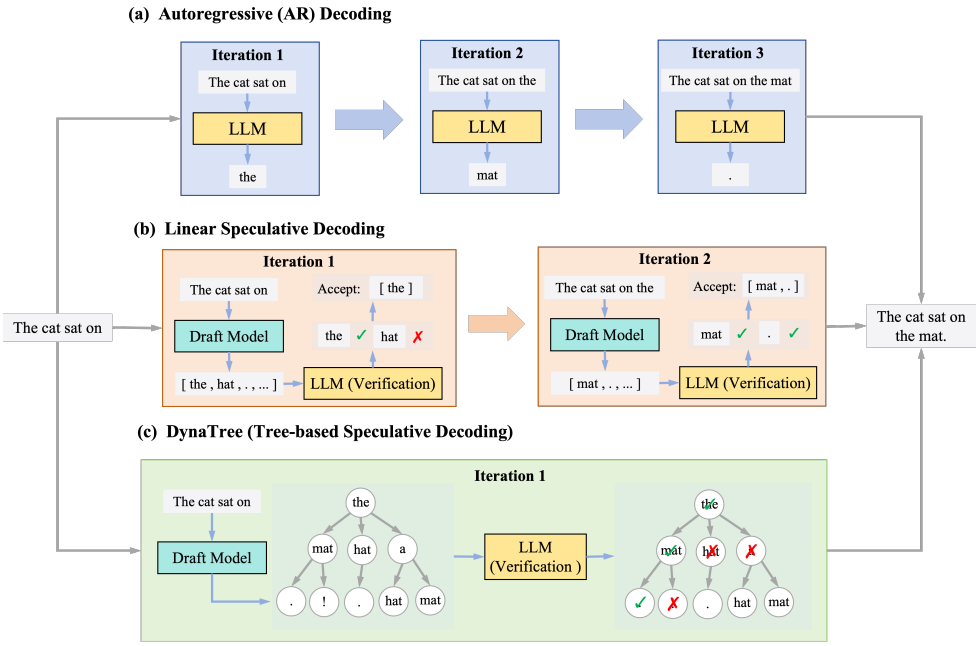

Figure 1: **Comparison of three decoding paradigms. Left:** Autoregressive (AR) decoding generates tokens sequentially, requiring one LLM forward pass per token. **Middle:** Linear speculative decoding drafts a single token chain; early rejection wastes all subsequent drafted tokens. **Right:** Tree-based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

or verification thresholds [8–10], 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 breadth and depth under an explicit node budget. DynaTree (i) adjusts per-node branching based on draft confidence, (ii) dynamically gates depth via early stopping and selective deep expansion, and (iii) applies a lightweight history-based adjustment of a small parameter subset 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 breadth, dynamic 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 $T = 1500$.

2 Related Work

2.1 Speculative Decoding

Speculative decoding improves generation efficiency by verifying draft proposals in parallel while preserving target-model correctness [3, 4, 11]. Systems work highlights that decoding performance is often memory-bound and sensitive to batching, context length, and workload heterogeneity [1, 2, 12, 13]. Recent studies further emphasize serving-oriented considerations for speculative decoding under realistic deployment constraints [14, 15].

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 [5–7, 16]. SpecInfer [5] pioneered practical token-tree verification mechanisms, while OPT-Tree [6] searches for draft tree structures that improve the expected committed prefix length. Medusa [7] explores multi-token prediction via additional decoding heads. 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 [8–10]. In contrast to adaptive *length*-only policies, DynaTree adapts the tree structure itself by jointly controlling breadth and depth and incorporating lightweight history-based adjustment, 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 [17, 18], cost-aware tree construction and pruning [6, 19], and retrieval-augmented pruning heuristics [20]. Other lines of work adapt speculative hyperparameters online [21] or propose alternative parallel decoding schemes beyond a single draft–verify pair [22, 23]. 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. Given a prefix (prompt) $x_{1:t}$, greedy decoding with M_T produces tokens $y_{t+1}, y_{t+2}, \ldots$ where

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

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

3.2 Overview of DynaTree

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

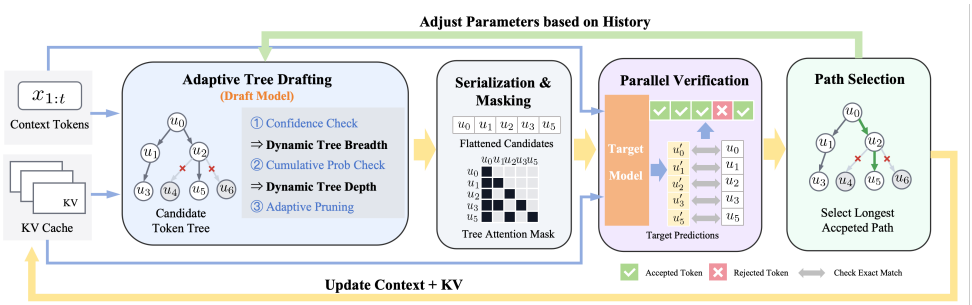

Figure 2: **One iteration of DynaTree decoding.** The process consists of six main stages: (1) *Adaptive Tree Drafting:* The draft model expands a candidate tree with three adaptive mechanisms: confidence-aware branching adjusts the number of child nodes (1–3) per expansion based on draft model confidence;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

3.3 Adaptive Tree Drafting

We maintain a token tree $\mathcal{T} = (\mathcal{N}, \mathcal{E})$ whose nodes $u \in \mathcal{N}$ correspond to drafted tokens. Each node u is associated with: (i) *token* $z_u \in \mathcal{V}$; (ii) *parent* $\pi(u)$; (iii) *depth* $d(u)$ from root; (iv) *draft log-probability* $\ell_u = \log p_D(z_u \mid \text{prefix}(\pi(u)))$; and (v) *cumulative log-probability* $\bar{\ell}_u = \sum_{v \in \text{path}(u)} \ell_v$, where $\text{path}(u)$ denotes all nodes from root to u along the tree.

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

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

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

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

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

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

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

We assume $1 \leq D_0 < D_{\max}$ and $0 < \rho_{\text{stop}} < \rho_{\text{deep}} < 1$. The first condition enforces a hard depth limit; the second implements *early stopping* by terminating branches whose joint draft probability is

too small; and the third allows *deep expansion* beyond D_0 only along sufficiently likely paths. In practice, ρ_{stop} and ρ_{deep} are tuned on a held-out set (Section 4).

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

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

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

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

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

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

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

3.4 Tree Attention for Parallel Verification

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

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

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, 6].

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 [24]. 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 GPU with sufficient memory to host both models. We implement DynaTree in PyTorch [25] on top of HuggingFace Transformers [26]. Across methods, we reuse KV caches where appropriate and synchronize GPU execution for timing to ensure a consistent measurement protocol.

Workloads and data preprocessing. We report results on WikiText-2 [27] and PG-19 [28]. 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 configurations.

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.

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

Method	WikiText-2		PG-19	
	Throughput (t/s)	Speedup	Throughput (t/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×**

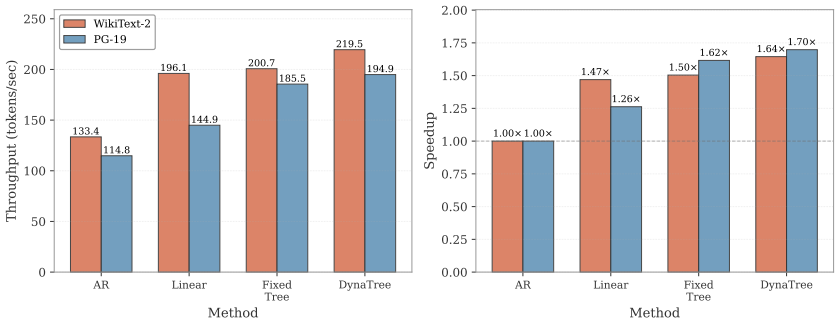

Figure 3: **Throughput and speedup across datasets ($T = 1500$).** Each method is shown with two bars (WikiText-2 vs. PG-19). DynaTree consistently improves over autoregressive and linear speculative decoding on both datasets.

- **Linear speculative decoding:** A linear-chain speculative decoder that drafts K tokens with the draft model and verifies them with the target model in parallel, committing the longest greedy-consistent prefix [3].
- **Fixed Tree:** A static tree speculative decoder with fixed depth D, fixed branching factor B, and node budget $N_{\max}$, representing a non-adaptive tree baseline. We report the configuration used wherever results are shown.
- **DynaTree:** Our full method that augments the fixed-tree backbone with (i) *Dynamic Breadth*, (ii) *Dynamic 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 highest throughput on both datasets, improving over autoregressive decoding, linear speculative decoding, and a fixed-tree baseline. On WikiText-2, DynaTree reaches 219.5 t/s and outperforms a tuned static tree (200.7 t/s), indicating that allocating verification budget according to local model uncertainty can improve the draft–verify trade-off beyond a fixed configuration.

Figure 3 visualizes throughput and speedup. Linear speculative decoding can attain high acceptance when the draft closely matches the target; however, a mismatch truncates the reusable 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. DynaTree further concentrates the node budget on high-confidence regions while limiting expansion in uncertain regions, improving throughput across both datasets.

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

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. Tree-based methods commit

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**

longer prefixes per step and thus require fewer rounds than autoregressive decoding; DynaTree further reduces the number of rounds by adapting the draft structure to recent verification outcomes.

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, isolating the contribution of each adaptive component relative to a fixed-tree backbone. In this regime, adapting breadth alone may introduce control overhead without reliably increasing the committed prefix length, whereas dynamic depth control more directly increases per-iteration progress and yields the dominant throughput gain. Table 4 therefore reports the combined effect of *Dynamic Breadth & Depth*, followed by the additional improvement from *History Adaptation*, which stabilizes the configuration by reacting to recent verification outcomes.

Interpretation. Dynamic depth control primarily improves throughput by increasing per-iteration progress ($\bar{L}$) and reducing the number of verification rounds required to reach T tokens. History Adaptation then refines the draft aggressiveness based on recent outcomes, stabilizing verification efficiency and yielding a further throughput gain without increasing peak memory (Appendix C).

Table 4: **Ablation-style progressive comparison on WikiText-2 ($T = 1500$).** We compare a fixed tree ($D_0 = 5, B = 2$) with variants that add Dynamic Breadth & Depth and History Adaptation. Speedup is computed relative to the autoregressive baseline.

Variant	Throughput (t/s)	Speedup	Δ vs Fixed
Fixed Tree ($D_0 = 5, B = 2$)	188.0±16.5	1.42×	0.0%
+ Dynamic Breadth & Depth	213.2±26.1	1.61×	+13.4%
+ History Adaptation	**218.5±22.2**	**1.65×**	**+16.2%**

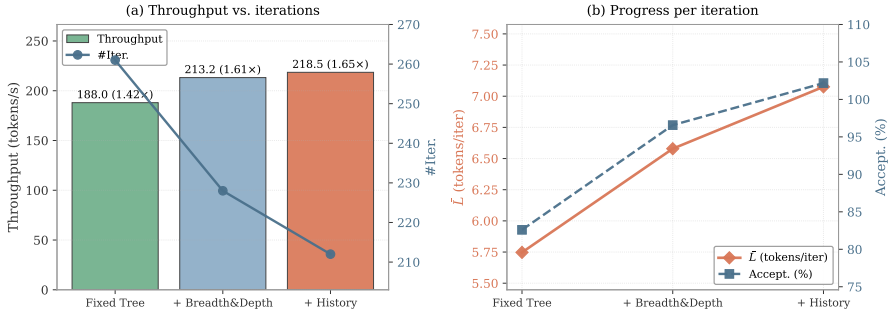

Figure 4: **Ablation progression on WikiText-2 ($T = 1500$).** Each panel combines two complementary metrics. Left: throughput (bars) alongside the number of verification iterations (#Iter.). Right: per-iteration progress ($\bar{L}$, tokens/iter) alongside acceptance rate (separate axis). Dynamic Breadth & Depth increases per-step progress and reduces iterations; History Adaptation further improves per-step progress and iteration count.

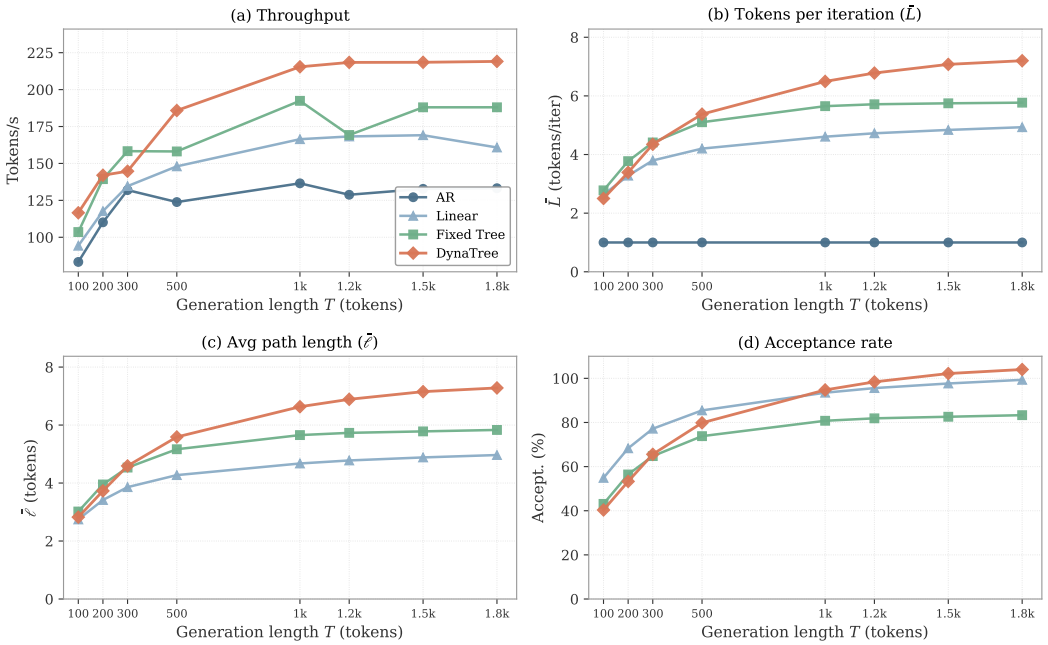

Figure 5: **Sequence length scaling on WikiText-2.** (a) Throughput (tokens/s) as a function of T for AR, linear speculative decoding, a fixed tree baseline, and DynaTree. (b–d) Auxiliary verification statistics: tokens per iteration ($\bar{L}$), average committed path length ($\bar{\ell}$), and acceptance rate.

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 5 summarizes both end-to-end throughput and auxiliary verification statistics across T.

Analysis. Across T, DynaTree maintains strong throughput by sustaining a larger committed prefix per verification round while keeping acceptance rates stable. In contrast, linear speculation is more sensitive to occasional early mismatches: as the generation proceeds, a single rejection can truncate the reusable drafted chain and reduce effective progress per round. The efficiency view (right panel) highlights how changes in $\bar{L}$, $\bar{\ell}$, and acceptance jointly determine throughput, providing a mechanistic explanation of the observed scaling curves.

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

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 nodes in a single target-model pass using a structured attention mask. DynaTree adaptively allocates the verification budget via dynamic breadth and depth control and stabilizes the configuration through history-based adjustment. 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 committing longer greedy-consistent prefixes per verification step. Future work includes evaluating broader serving regimes (e.g., longer contexts, different prompt lengths and batch sizes) and reducing system 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.

[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] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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/2508.08192v1.

[15] 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.

[16] 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.

[17] 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.

[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] 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.

[20] 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.

[21] 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/2505.15141v2.

[22] 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/2402.15678v2.

[23] 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/2410.13344v1.

[24] 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.

[25] 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.

[26] 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.

[27] 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`.

[28] 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`.

A Experimental Configuration Details

Common settings. Unless stated otherwise, we use WikiText-2 prompts, truncate the prompt length to $L_{\max} = 800$, and generate T new tokens with greedy decoding. We evaluate $N = 10$ prompts and discard the first $W = 2$ runs as warmup. 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 B 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 $D_{\max} = 8$, 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.

B Additional Experimental Analyses

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

B.1 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.

B.2 Fixed-tree Hyperparameter Sweep

We perform a fixed-tree hyperparameter sweep under the same evaluation protocol as our main benchmark to ensure the static-tree baseline is reasonably tuned. Figure 7 summarizes the sweep over depth D, branching factor B, and pruning threshold τ on WikiText-2 at $T = 1500$.

B.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 8 reports

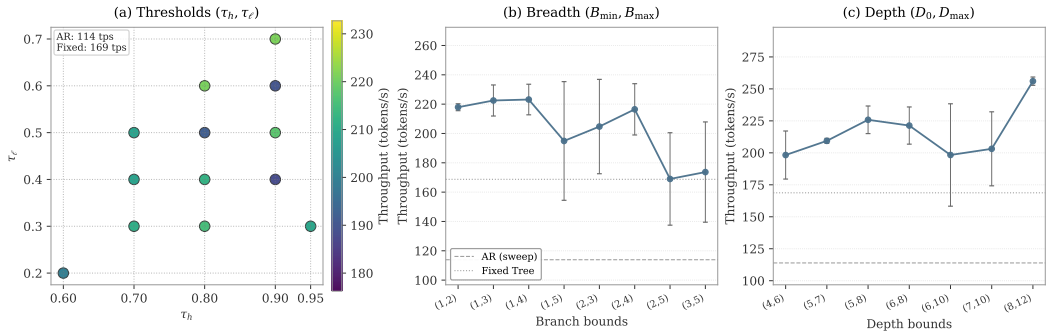

Figure 6: **Sensitivity of DynaTree to drafting hyperparameters (WikiText-2, $T = 1500$). (a)** Threshold sweep shown as a sparse 2D scatter over (τ_h, τ_ℓ), colored by throughput (tokens/s). **(b–c)** Breadth and depth sweeps shown as throughput mean±std across tested pair configurations. Horizontal lines denote the autoregressive and fixed-tree baselines measured in the same sweep run.

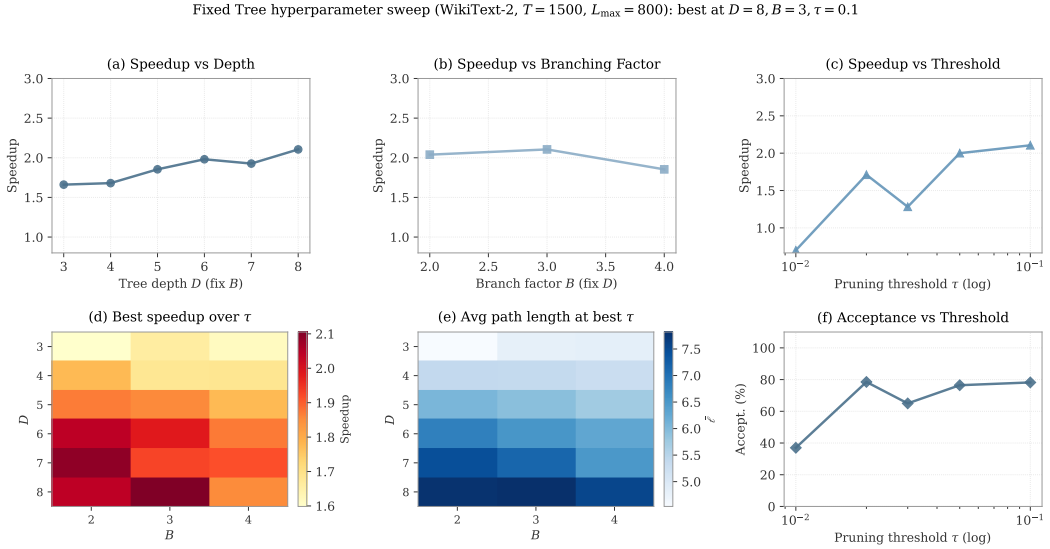

Figure 7: **Fixed-tree hyperparameter sweep under the paper protocol (WikiText-2, $T = 1500$).** Speedup is computed relative to autoregressive decoding under the same setting. Line plots use widened y-axis ranges to emphasize robustness across depth, branching, and threshold settings.

throughput and speedup as functions of $L_{\max}$, illustrating how speculative decoding performance changes as prefill cost increases.

C 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 $T = 1500$ main benchmark. 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.

D DynaTree Iteration Pseudocode

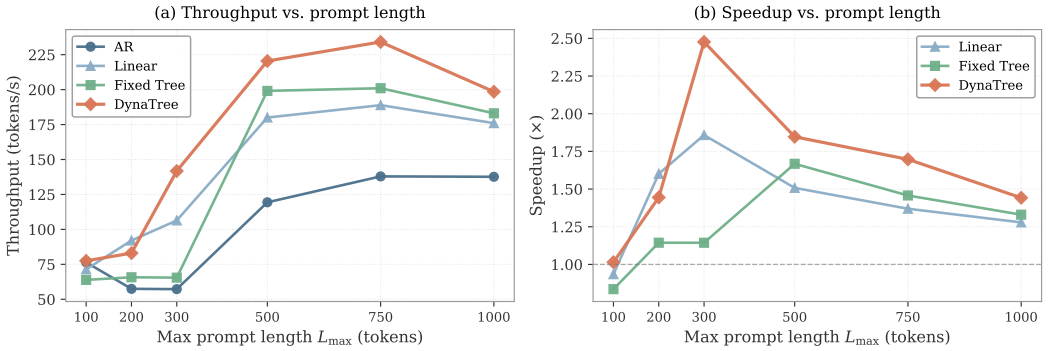

Figure 8: **Prompt length sensitivity on WikiText-2 ($T = 1500$).** Throughput and speedup (relative to AR at the same $L_{\max}$) as functions of the maximum prompt length.

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.

Method	Peak Memory (MB)		Average (MB)	Rel. Change
	PG-19	WikiText-2		
AR (baseline)	6157.7	6110.0	6133.9	0.00%
Linear Spec ($K = 5$)	6360.5	6313.8	6337.2	+3.31%
Fixed Tree ($D = 5, B = 2$)	6358.3	6312.8	6335.5	+3.29%
DynaTree	**6359.4**	**6316.1**	**6337.7**	**+3.32%**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

            continue

        end if ▷ probability-threshold pruning

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

            continue

        end if

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

            continue

        end if ▷ early stopping

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

            continue

        end if ▷ depth gating

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

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

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

    end while

    return $\mathcal{T}$

end function

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

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

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

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

        return $[first]$

    else

        $y \leftarrow$ tokens on P

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

        return y

    end if

end function