DynaTree: Dynamic Tree-based Speculative Decoding with Adaptive Pruning 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, common linear variants explore only a single draft chain per step and can waste substantial computation when early tokens are rejected. We propose **DynaTree**, a tree-based speculative decoding framework that drafts multiple candidate continuations via top-k branching and verifies the resulting token tree in one forward pass using tree attention. To control the exponential growth of the draft tree, DynaTree applies adaptive pruning that removes low-probability branches under an explicit node budget. Experiments on Pythia models show that DynaTree improves decoding throughput by up to $1.62\times$ over standard autoregressive generation and consistently outperforms strong speculative decoding baselines across generation lengths.

1 Introduction

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

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

In practice, most speculative decoding systems employ *linear* drafting: the draft model proposes a single chain of K tokens. This design is brittle under draft–target mismatch: a rejection at an early position forces all subsequent drafted tokens to be discarded, wasting both draft computation and target-model verification work, and constraining achievable speedups [4].

We argue that the single-path constraint is unnecessary. When multiple plausible next tokens compete, exploring several continuations in parallel increases the chance that at least one path aligns with the target model, thereby improving the expected number of accepted tokens per verification step. This motivates *tree-based* speculation, where the draft expands multiple candidates via top-k branching and the target verifies the resulting token tree with a structured, causality-preserving attention mask.

[*]Equal contribution.

The central challenge is controlling verification cost: naive tree expansion grows exponentially with depth and branching. We present **DynaTree**, a tree-based speculative decoding framework with a lightweight adaptive pruning mechanism that removes low-probability branches while enforcing an explicit node budget. Empirically, DynaTree achieves up to $1.62\times$ throughput improvement over standard autoregressive decoding on Pythia models and consistently outperforms strong speculative decoding baselines. In summary, our contributions are: (i) a practical tree-based speculative decoding algorithm with efficient tree attention verification; (ii) an adaptive pruning strategy that stabilizes the depth–breadth trade-off under a fixed verification budget; and (iii) an extensive empirical study characterizing these trade-offs across generation lengths.

2 Related Work

2.1 Speculative Decoding

Speculative decoding accelerates autoregressive generation by decoupling *proposal* and *verification*: a lightweight draft model proposes multiple tokens, and the target model verifies these candidates in parallel while preserving the exact output distribution through rejection sampling [3, 4]. Empirical and theoretical analyses highlight that achievable speedups depend critically on the acceptance behavior induced by the draft–target mismatch, and on the additional overhead introduced by drafting and verification [4, 5]. Recent systems-level studies further emphasize robustness challenges across heterogeneous request distributions and long-context inputs [6–8]. Despite strong progress, the dominant implementation remains *linear* drafting, where a single speculative chain is proposed per iteration; when early tokens are rejected, downstream drafted tokens are discarded, causing substantial wasted computation and limiting utilization of parallel verification.

2.2 Tree-Based and Parallel Decoding

To overcome the single-path limitation, recent work explores *tree-based* speculative decoding, where multiple candidate continuations are drafted and verified in a single target-model forward pass using structured attention masks. SpecInfer [9] instantiates this idea in an LLM serving setting by building a token tree and verifying it efficiently. OPT-Tree [10] further studies *adaptive* tree construction, selecting tree shapes to maximize expected acceptance length under a fixed verification budget. Alternative verification strategies such as traversal-style verification have also been explored for speculative trees [11]. In parallel, Medusa [12] pursues multi-token generation by augmenting a base model with multiple decoding heads and verifying the induced candidate tree; unlike draft–target speculative decoding, it requires model-specific fine-tuning. Our work follows the draft–target paradigm but focuses on practical tree construction and verification under strict budget constraints, emphasizing dynamic pruning as a lightweight mechanism to stabilize performance.

2.3 Dynamic Pruning Strategies

Tree-based methods must contend with the exponential growth of candidates with depth and branching. ProPD [13] proposes dynamic token-tree pruning and generation, leveraging early signals to remove low-utility branches before full verification. Cost-aware formulations further model verification overhead and explicitly optimize the trade-off between exploration and target-model computation [14]. DySpec [15] employs greedy, confidence-guided expansion to adapt tree structures online. DynaTree is closely related to these approaches: we adopt a probability-threshold-based pruning rule coupled with an explicit node budget, aiming for a simple, training-free mechanism that is easy to integrate while maintaining strong speedups in practice.

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

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

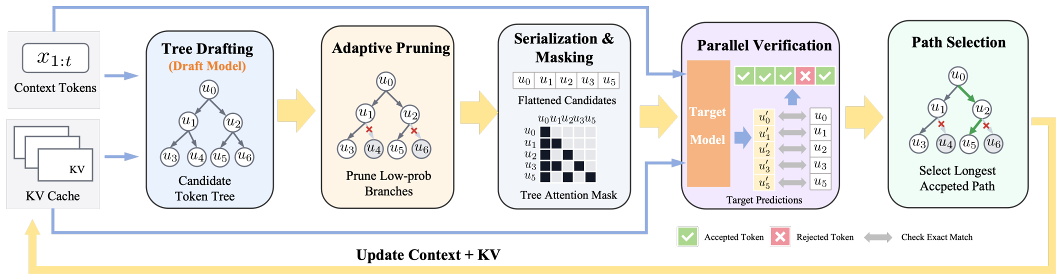

Figure 1: **One iteration of DynaTree decoding.** The process consists of six main stages: (1) *Tree Generation:* The draft model expands a candidate tree with top-B branching up to depth D. (2) *Adaptive Pruning:* Branches with cumulative probability below threshold τ or exceeding node budget $N_{\max}$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

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

3.2 Overview of DynaTree

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

3.3 Draft Tree Construction with Dynamic Pruning

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

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

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

$$\bar{\ell}_u < \log \tau \implies \text{prune } u.$$

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

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

3.4 Tree Attention for Parallel Verification

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

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

This mask ensures the conditional distribution computed at each node matches the distribution of sequential decoding along its unique root-to-node path, while enabling parallel verification across different branches [9, 10].

3.5 Greedy Path Selection and Cache Update

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

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

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

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

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

3.6 Correctness for Greedy Decoding

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

3.7 Complexity Discussion

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

4 Experiments

4.1 Experimental Setup

Models. We evaluate DynaTree using models from the Pythia family [16]. Our target model M_T is Pythia-2.8B (2.8 billion parameters), and our draft model M_D is Pythia-70M (70 million parameters). All experiments use deterministic greedy decoding, ensuring that the output sequence is uniquely determined by the model and prefix.

Hardware and software. All experiments are conducted on a single NVIDIA GPU with sufficient memory to accommodate both models simultaneously. We implement DynaTree in PyTorch [17] using the HuggingFace Transformers library [18] for model loading and inference, leveraging dynamic key-value cache structures to minimize memory overhead during tree verification.

Workloads. Unless otherwise specified, we evaluate on a generation task producing 500 new tokens from a fixed technical prompt. Results are averaged over 5 independent runs, with the first run discarded as warmup to eliminate one-time initialization costs. To ensure fair comparison, we synchronize GPU execution and clear cached states between different methods.

4.2 Evaluation Metrics

We measure **throughput** (tokens per second) as the primary performance indicator, computed as the total number of generated tokens divided by the wall-clock time excluding warmup. We additionally report **speedup**, defined as the ratio of a method's throughput to that of the autoregressive baseline. When applicable, we also include the **acceptance rate**—the fraction of drafted tokens that match the target model's greedy predictions—and **tokens per iteration**, the average number of tokens committed in each decoding round.

4.3 Baselines

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

- **Autoregressive (AR):** Standard greedy decoding with the target model, serving as the performance baseline.
- **HuggingFace Assisted Generation:** The built-in speculative decoding implementation in the HuggingFace Transformers library, which uses the draft model to propose candidate tokens for verification.
- **Linear Speculative Decoding:** A linear-chain variant of speculative decoding where the draft model proposes a sequence of K tokens that are verified in parallel by the target model [3].
- **StreamingLLM + Speculative:** A combination of speculative decoding with StreamingLLM's attention sink mechanism for efficient long-context generation [19].

4.4 Main Results

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

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

Table 1: **Main results: end-to-end performance comparison on 500-token generation with Pythia models.** Throughput is measured in tokens per second (t/s). Speedup is relative to the autoregressive baseline. Acceptance rate is reported only for methods where this metric is explicitly tracked during drafting and verification. DynaTree achieves the highest throughput and speedup across all baselines.

Method	Throughput (t/s)	Speedup	Accept.
AR (target-only)	119.4	1.00	–
HuggingFace assisted	161.9	1.36	–
Linear speculative (K=6)	133.1	1.11	0.68
StreamingLLM + speculative (cache=1024)	132.9	1.11	–
DynaTree (ours)	**193.4**	**1.62**	0.30

Table 2: **Verification efficiency comparison.** We report drafting budget (K for linear, expected tree size for DynaTree), tokens committed per iteration, and per-token acceptance rate for linear and tree-based methods at their optimal configurations on 500-token generation. Despite lower per-token acceptance, DynaTree's multi-path exploration commits substantially more tokens per verification step, demonstrating the value of exploring multiple candidate paths in parallel.

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

Verification efficiency analysis. Table 2 provides a detailed breakdown of verification efficiency metrics for the optimal configurations. DynaTree (D=8, B=3, τ=0.03) commits an average of 6.94 tokens per iteration—69% more than linear speculative decoding (4.10 tokens/iter)—despite having a lower per-token acceptance rate (38% vs. 68%). This demonstrates that tree-based drafting's multi-path exploration enables longer committed sequences per verification round, offsetting the reduced per-token acceptance and ultimately yielding higher end-to-end throughput.

4.5 Ablation Study: Progressive Component Addition

To isolate the contribution of each algorithmic component, we conduct an ablation study by progressively adding features to the baseline linear speculative decoder. Table 3 summarizes the results. Starting from linear speculative decoding (K=6, 1.11$\times$ speedup), adding tree structure with shallow depth and loose pruning (D=4, B=3, τ=0.01) improves speedup to 1.43$\times$, demonstrating the value of multi-path exploration. Further optimizing the tree depth and tightening the pruning threshold (D=8, B=3, τ=0.03) yields the best speedup of 1.79$\times$, highlighting the importance of balancing tree size and verification cost. Note that absolute throughput values here are measured under a consistent parameter-sweep harness and may differ slightly from the end-to-end benchmark in Table 1 due to different measurement protocols, but the relative improvements clearly demonstrate the algorithmic gains from (i) parallel path verification and (ii) adaptive depth with calibrated pruning.

4.6 Hyperparameter Sensitivity

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

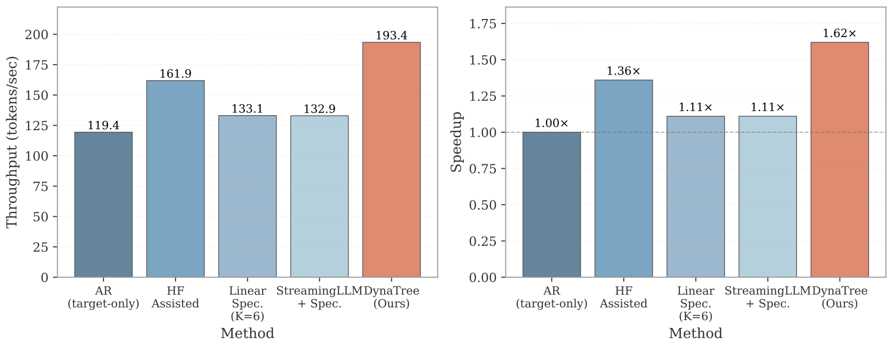

Figure 2: **Throughput and speedup comparison across methods.** Left: absolute throughput in tokens per second. Right: speedup relative to autoregressive baseline. DynaTree achieves 193.4 t/s ($1.62\times$), outperforming HuggingFace assisted generation ($1.36\times$) and linear speculative decoding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

Table 3: **Ablation study: progressive addition of algorithmic components.** Starting from linear speculative decoding, we incrementally add tree structure and optimize hyperparameters. Each row shows the configuration and resulting performance. The progressive improvements demonstrate the value of multi-path exploration and adaptive pruning.

Method	Configuration	Throughput (t/s)	Speedup
Linear speculative	K=6	133.1	1.11
+ Tree structure	$D{=}4, B{=}3, \tau{=}0.01$	176.6	1.43
+ Depth & pruning optimization	$D{=}8, B{=}3, \tau{=}0.03$	**221.4**	**1.79**

4.7 Sequence Length Scaling

Figure 3 and Table 4 examine how DynaTree's performance varies with generation length. For each target length, we report the best-performing configuration identified in our parameter sweep, along with the corresponding baseline and DynaTree throughput. Several trends emerge: (i) Speedup increases monotonically from $1.43\times$ at 100 tokens to a peak of $1.79\times$ at 500 tokens, as the one-time setup and verification overhead is amortized over more generated tokens; (ii) Beyond 500 tokens, speedup plateaus or slightly decreases ($1.71\times$ at 1000 tokens), likely due to changing token statistics and the accumulation of small KV-cache management costs; (iii) The optimal tree depth varies with length (D=7 for shorter sequences, D=8 for 500 tokens, D=6 for 1000 tokens), reflecting the need to adapt exploration depth to the generation horizon. As shown in Figure 3, DynaTree consistently outperforms other speculative decoding baselines across all tested generation lengths, demonstrating robust performance scaling.

5 Conclusion

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

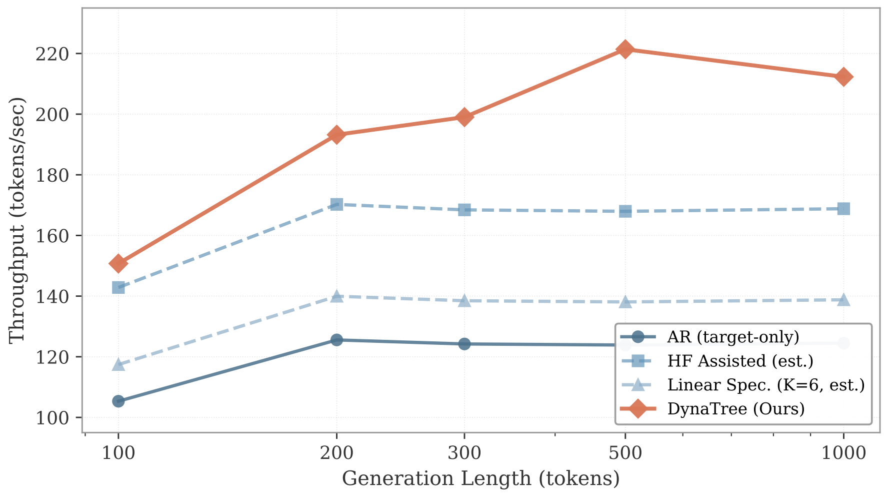

Figure 3: **Throughput across different generation lengths.** DynaTree consistently outperforms all baselines across generation lengths from 100 to 1000 tokens. Performance improves from 1.43× speedup at 100 tokens to a peak of 1.79× at 500 tokens, then plateaus at 1.71× for 1000 tokens. The autoregressive baseline (AR) shows stable throughput across lengths, while DynaTree's advantage grows with length until verification overhead balances exploration gains. HuggingFace Assisted and Linear Speculative Decoding throughputs are estimated from their observed speedup ratios at 500 tokens (1.36× and 1.11× respectively), applied to baseline performance at each length.

Table 4: **Performance across different generation lengths.** For each target length, we report the optimal hyperparameter configuration and the resulting throughput and speedup. DynaTree achieves consistent speedups across all lengths, peaking at 1.79× for 500-token generation. The optimal depth varies with length, indicating the importance of adaptive configuration.

Length	Optimal (D, B, τ)	Baseline (t/s)	DynaTree (t/s)	Speedup	Accept.
100	(7,3,0.03)	105.3	150.7	1.43	0.31
200	(7,3,0.03)	125.5	193.2	1.54	0.36
300	(7,3,0.03)	124.2	199.0	1.60	0.38
500	(8,3,0.03)	123.9	221.4	1.79	0.38
1000	(6,3,0.05)	124.5	212.3	1.71	0.37

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] 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`.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

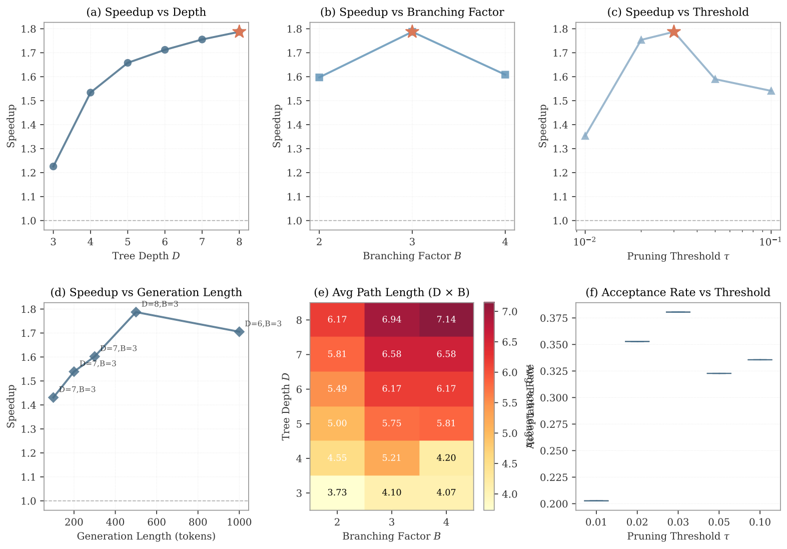

Figure 4: **Comprehensive parameter sweep analysis across 450 configurations.** (a) Speedup vs. tree depth D (fixing $B = 3$, $\tau = 0.03$, 500 tokens): deeper trees improve speedup up to $D = 8$, after which verification overhead dominates. (b) Speedup vs. branching factor B (fixing $D = 8$, $\tau = 0.03$, 500 tokens): $B = 3$ achieves the best balance between exploration and cost. (c) Speedup vs. pruning threshold τ (fixing $D = 8$, $B = 3$, 500 tokens, log scale): $\tau = 0.03$ is optimal, balancing aggressive and loose pruning. (d) Best speedup across generation lengths with corresponding optimal (D, B, τ) configurations: speedup peaks at 500 tokens and remains strong at other lengths. (e) Average path length heatmap over (D, B) (fixing $\tau = 0.03$, 500 tokens): larger trees enable longer committed paths. (f) Acceptance rate distribution vs. pruning threshold (fixing $D = 8$, $B = 3$, 500 tokens): tighter pruning slightly reduces acceptance but improves overall throughput. Stars mark optimal configurations in each subplot.

A Hyperparameter Sweep Details

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

B DynaTree Iteration Pseudocode

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

```
Require: Prefix tokens $x_{1:t}$; target KV cache $\mathcal{K}_T$; prefix next-token logits $\mathbf{s}_{\text{last}}$; tree depth $D$; branch factor $B$;
    pruning threshold $\tau$; node budget $N_{\max}$.
Ensure: Committed tokens $y_{t+1:t+L}$ and updated $\mathcal{K}_T$.
 1: $\ell \leftarrow$ SEQLEN($\mathcal{K}_T$)                                          ▷ record prefix cache length
 2: $\mathcal{T} \leftarrow$ DRAFTTREE($x_{1:t}, D, B, \tau, N_{\max}$)
 3: $\mathbf{z}_{1:n} \leftarrow$ BFSFLATTEN($\mathcal{T}$); $\mathbf{A} \leftarrow$ TREEMASK($\mathcal{T}, \ell$)              ▷ prefix + ancestors only
 4: $\mathbf{s}_{1:n} \leftarrow M_T(\mathbf{z}_{1:n}; \mathbf{A}, \mathcal{K}_T)$; $\hat{\mathbf{y}} \leftarrow \arg\max \mathbf{s}_{1:n}$
 5: $y_{t+1:t+L} \leftarrow$ SELECTCOMMIT($\mathcal{T}, \hat{\mathbf{y}}, \mathbf{s}_{\text{last}}$)
 6: $\mathcal{K}_T \leftarrow$ CROP($\mathcal{K}_T, \ell$); $\mathcal{K}_T \leftarrow M_T(y_{t+1:t+L}; \mathcal{K}_T)$                     ▷ rollback + rebuild
 7: return $y_{t+1:t+L}$

 8: function DRAFTTREE($x_{1:t}, D, B, \tau, N_{\max}$)     ▷ Draft a candidate token tree with probability-threshold
    pruning and a node budget.
 9:     Run $M_D$ on $x_{1:t}$; let $u_0$ be the $\top 1$ token; initialize $\mathcal{T}$ with root $u_0$
10:     $\mathcal{A} \leftarrow \{u_0\}$                                                      ▷ active leaves
11:     for $d = 1$ to $D$ do
12:         if $|\mathcal{A}| = 0$ or $|\mathcal{T}| \geq N_{\max}$ then
13:             break
14:         end if
15:         $\mathcal{A}' \leftarrow \emptyset$
16:         for all $u \in \mathcal{A}$ do
17:             if $\bar{\ell}_u < \log \tau$ then
18:                 continue
19:             end if                                            ▷ prune low-probability branches
20:             Do one cached step of $M_D$ from $u$; take $\top B$ next-token candidates
21:             for all candidates $v$ do
22:                 if $|\mathcal{T}| \geq N_{\max}$ then
23:                     break
24:                 end if
25:                 Add child $v$ to $\mathcal{T}$; add $v$ to $\mathcal{A}'$
26:             end for
27:         end for
28:         $\mathcal{A} \leftarrow \mathcal{A}'$
29:     end for
30:     return $\mathcal{T}$
31: end function

32: function SELECTCOMMIT($\mathcal{T}, \hat{\mathbf{y}}, \mathbf{s}_{\text{last}}$)  ▷ Select the longest greedy-consistent path (plus one bonus token).
33:     $first \leftarrow \arg\max \mathbf{s}_{\text{last}}$
34:     Find the longest path $P$ where root token $= first$, and for each edge $(u \to v)$, token$(v) = \hat{\mathbf{y}}[\text{pos}(u)]$
35:     if $P = \emptyset$ then
36:         return $[first]$
37:     else
38:         $y \leftarrow$ tokens on $P$
39:         Append one bonus token $\hat{\mathbf{y}}[\text{pos}(\text{last}(P))]$
40:         return $y$
41:     end if
42: end function
```