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

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

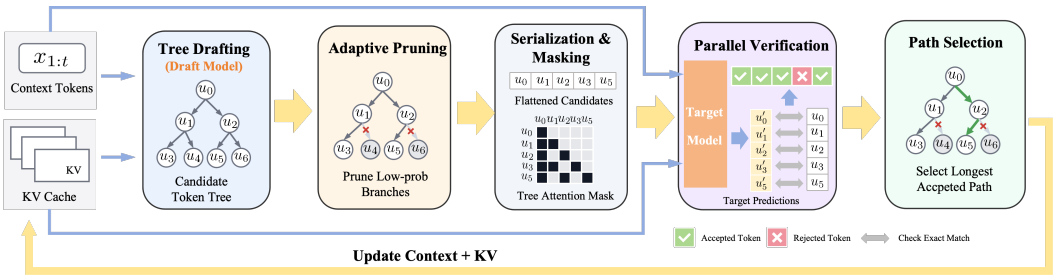

Figure 1: **Architecture overview of DynaTree.** Each iteration drafts a candidate token tree with a small *draft model*, prunes unlikely branches under a probability threshold and node budget, serializes the remaining nodes and constructs a *tree causal attention mask*, then verifies all candidates in parallel with a single forward pass of the *target model*. A longest valid path (tokens matching the target model's greedy top-1 predictions) is committed, and the context and KV cache are updated for the next iteration.

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 stores: token $z_u \in \mathcal{V}$, parent $\pi(u)$, depth $d(u)$, draft log-probability $\ell_u = \log p_D(z_u \mid \text{prefix}(\pi(u)))$, and cumulative log-probability $\bar{\ell}_u = \sum_{v \in \text{path}(u)} \ell_v$.

Expansion. Starting from the current prefix $x_{1:t}$, we first obtain the draft distribution $p_D(\cdot \mid x_{1:t})$ and create the root node u_0 using the top-1 token. Then, for each active leaf u with $d(u) < D$, we expand children by selecting the top-B tokens under $p_D(\cdot \mid x_{1:t+\text{pos}(u)})$ (implemented via cached one-token forward passes). To bound computation, we enforce a hard **node budget** $N_{\max}$ and stop expansion when $|\mathcal{N}| \geq N_{\max}$.

Probability-threshold pruning (V2). To avoid wasting draft computation on unlikely branches, DynaTree prunes any leaf u whose cumulative probability falls below a threshold $\tau \in (0, 1)$:

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

This rule corresponds exactly to the implementation in 'TreeSpeculativeGeneratorV2', where $\bar{\ell}_u$ is accumulated along the path and compared against $\log \tau$.

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 KV states for non-committed branches. To maintain correctness, we crop the target cache back to the pre-iteration prefix length t and then forward only the committed tokens to rebuild the cache (as implemented in '$_u pdate_t ree_c ache$').

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 on the Pythia family with a target model M_T = Pythia-2.8B and a draft model M_D = Pythia-70M, using greedy decoding throughout (`do_sample=False`).

Hardware and software. All experiments are run on a single NVIDIA GPU. We use PyTorch 2.x and HuggingFace Transformers 4.x with `DynamicCache` for KV management.

Table 1: Main decoding throughput on Pythia (500-token generation). Higher is better.

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

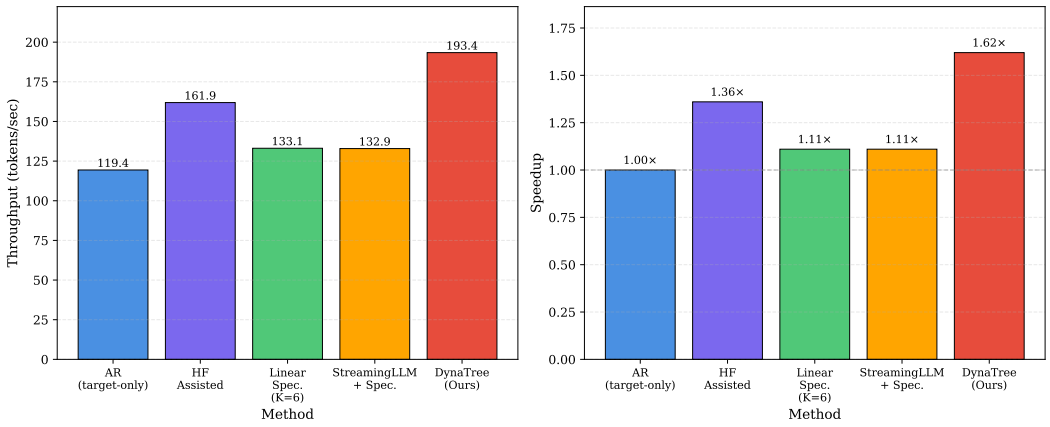

Figure 2: **Main results comparison.** DynaTree achieves the highest throughput (193.4 tokens/sec) and speedup (1.62×) compared to all baselines, outperforming HuggingFace assisted generation and linear speculative decoding.

Workloads. Unless otherwise stated, we generate 500 new tokens from a fixed technical prompt and report averages over 5 runs, skipping the first run as warmup. For each run, we synchronize the GPU before timing and clear caches between methods to reduce cross-run interference.

4.2 Metrics

We report **throughput** (tokens/sec) as the primary metric. We also report **speedup** relative to the autoregressive baseline, and (when available) **acceptance rate** and **tokens per iteration** (average committed tokens per verification step).

4.3 Baselines

We compare against: (i) **AR** greedy decoding with the target model; (ii) **HuggingFace assisted generation** (built-in speculative decoding using `assistant_model`); (iii) **linear speculative decoding** implemented with a draft chain of length K; and (iv) **StreamingLLM + speculative decoding** for long-context cache compression [16].

4.4 Main Results

Table 1 summarizes end-to-end throughput on 500-token generation. **DynaTree (ours)** achieves the best speedup (1.62×) and outperforms both HuggingFace assisted generation (1.36×) and linear speculative decoding (1.11×). We additionally report the draft-token acceptance rate when available.

Although the acceptance rate of tree-based drafting is lower than that of linear drafting, DynaTree improves throughput by increasing the probability of finding a long prefix consistent with the target greedy trajectory within a single verification step, while pruning low-probability branches to keep the verification set compact.

Table 2: Ablation study (progressive component addition) extracted from the sweep results.

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

Table 3: Best DynaTree performance across different generation lengths.

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

4.5 Ablation Study: Progressive Component Addition

To isolate the contribution of tree structure and pruning/optimization, we use the exhaustive sweep to form a progressive ablation sequence (Linear $\rightarrow$ Tree $\rightarrow$ Optimized Tree). Table 2 reports throughput and speedup under the sweep harness. Although absolute throughput can differ from the end-to-end benchmark due to different measurement harnesses and warmup effects, the progressive improvements highlight the algorithmic value of (i) parallel path verification and (ii) deeper trees with calibrated pruning.

4.6 Hyperparameter Sensitivity (Summary)

We conduct an extensive sweep over tree depth D, branching factor B, and pruning threshold τ across multiple generation lengths (100–1000 tokens), totaling 450 configurations. For clarity and space, we place the detailed sweep plots in Appendix A and summarize the key trend: increasing depth and branching improves exploration but can reduce throughput if verification cost grows faster than the expected committed length; probability-threshold pruning is essential to keep the effective tree size within budget.

4.7 Sequence Length Scaling

Table 3 reports the best-performing DynaTree configuration selected per generation length, together with baseline throughput and speedup. We observe that speedup increases from short to medium lengths as verification overhead is amortized, peaking around 500 tokens in our setting.

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.

A Hyperparameter Sweep Details

We perform a 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, totaling

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

Require: Prefix tokens $x_{1:t}$; target KV cache $\mathcal{K}_T$; prefix next-token logits $\mathbf{s}_{\text{last}}$; tree depth D; branch factor B; pruning threshold τ; node budget $N_{\max}$.
Ensure: Committed tokens $y_{t+1:t+L}$ and updated $\mathcal{K}_T$.

$\ell \leftarrow \textsc{SeqLen}(\mathcal{K}_T)$ ▷ record prefix cache length
$\mathcal{T} \leftarrow \textsc{DraftTree}(x_{1:t}, D, B, \tau, N_{\max})$
$\mathbf{z}_{1:n} \leftarrow \textsc{BFSFlatten}(\mathcal{T})$; $\mathbf{A} \leftarrow \textsc{TreeMask}(\mathcal{T}, \ell)$ ▷ prefix + ancestors only
$\mathbf{s}_{1:n} \leftarrow M_T(\mathbf{z}_{1:n}; \mathbf{A}, \mathcal{K}_T)$; $\hat{\mathbf{y}} \leftarrow \arg\max \mathbf{s}_{1:n}$
$y_{t+1:t+L} \leftarrow \textsc{SelectCommit}(\mathcal{T}, \hat{\mathbf{y}}, \mathbf{s}_{\text{last}})$
$\mathcal{K}_T \leftarrow \textsc{Crop}(\mathcal{K}_T, \ell)$; $\mathcal{K}_T \leftarrow M_T(y_{t+1:t+L}; \mathcal{K}_T)$ ▷ rollback + rebuild
return $y_{t+1:t+L}$

function $\textsc{DraftTree}(x_{1:t}, D, B, \tau, N_{\max})$ ▷ Draft a candidate token tree with probability-threshold pruning and a node budget.
  Run M_D on $x_{1:t}$; let u_0 be the $\top 1$ token; initialize $\mathcal{T}$ with root u_0
  $\mathcal{A} \leftarrow \{u_0\}$ ▷ active leaves
  for $d = 1$ **to** D **do**
    if $|\mathcal{A}| = 0$ **or** $|\mathcal{T}| \geq N_{\max}$ **then**
      break
    end if
    $\mathcal{A}' \leftarrow \emptyset$
    for all $u \in \mathcal{A}$ **do**
      if $\bar{\ell}_u < \log \tau$ **then**
        continue
      end if ▷ prune low-probability branches
      Do one cached step of M_D from u; take $\top B$ next-token candidates
      for all candidates v **do**
        if $|\mathcal{T}| \geq N_{\max}$ **then**
          break
        end if
        Add child v to $\mathcal{T}$; add v to $\mathcal{A}'$
      end for
    end for
    $\mathcal{A} \leftarrow \mathcal{A}'$
  end for
  return $\mathcal{T}$
end function

function $\textsc{SelectCommit}(\mathcal{T}, \hat{\mathbf{y}}, \mathbf{s}_{\text{last}})$ ▷ Select the longest greedy-consistent path (plus one bonus token).
  $first \leftarrow \arg\max \mathbf{s}_{\text{last}}$
  Find the longest path P where root token $= first$, and for each edge $(u \to v)$, $\text{token}(v) = \hat{\mathbf{y}}[\text{pos}(u)]$
  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

450 configurations. Figure 3 visualizes the depth–breadth–threshold trade-off, together with the relationship between effective tree size and speedup.

B DynaTree Iteration Pseudocode

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.

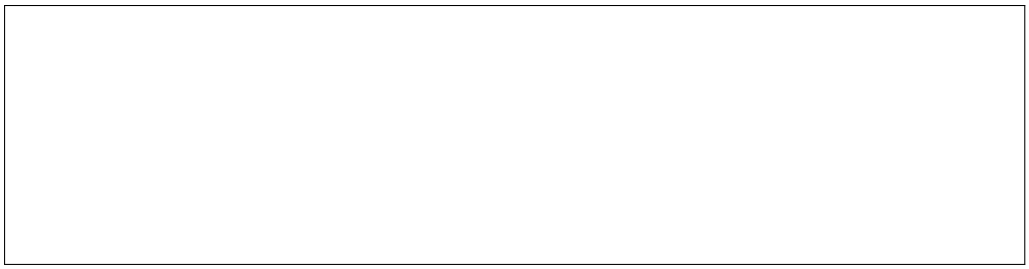

Figure 3: **Parameter sweep (placeholder).** Suggested 6-panel plot: speedup vs depth, vs branching factor, vs threshold; speedup vs length; average tree size heatmap; acceptance/tokens-per-round distribution.

[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] 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.