FlowSpec: Distributed Edge LLM Inference with Continuous Pipelined Speculative Decoding

Anonymous ACL submission

Abstract

Distributed inference enables large language model deployment at the edge for real-time response and data privacy. Pipeline Parallelism (PP) suits low-bandwidth edge links but suffers severe under-utilization with sparse decoding requests. Recent works adopt speculative decoding (SD) to provide draft tokens as micro-batches and alleviate pipeline bubbles. However, existing methods suffer from inefficient draft verification on pipeline that delays the acceptance of tokens, and the leaf-only expansion that results in stale low-quality draft in context-aware SD that causes weak acceptance. To address this, we propose FlowSpec, a pipeline-parallel tree-based speculative decoding framework to accelerate context-aware SD on edge devices. FlowSpec integrates three mechanisms to enable efficient continuous SD: (1) score-based step-wise verification enables cross-layer early acceptance of high-confidence drafts; (2) lightweight draft management prunes invalid branches early; and (3) context-aware dynamic expansion leverages the latest context and regenerates fresh draft trees to improve acceptance and pipeline continuity. Experiments on a real-world edge testbed demonstrate 2.44×–3.7× speedups over vanilla autoregressive decoding. Our code is available at https://anonymous.4open.science/r/FlowSpec-BEF9.

1 Introduction

Large language models (LLMs) have been widely adopted across various domains, such as education, finance, and the Internet of Things (IoT) (Minaee et al., 2024). Deploying LLM inference directly at the network edge is increasingly favored for its potential to provide real-time responses and inherently protect data privacy (Gill et al., 2025), particularly as an alternative to cloud-based services that may suffer from peak-period overload (Li et al., 2024a) and raise concerns over sensitive data uploading(Gill et al., 2025). However, the deployment on resource-constrained edge devices faces two primary challenges: (1) limited memory (e.g., several GBs) makes it difficult to run LLMs with billions of parameters, and (2) restricted computational capabilities cause high inference latency. While model compression techniques like pruning (Sanh et al., 2020) and quantization (Lin et al., 2024; Xiao et al., 2023) offer partial remedies, they often incur performance degradation. A promising alternative is distributed inference, which partitions an LLM across multiple devices to enable collaborative parallel inference, thereby overcoming the memory constraints of a single edge device.

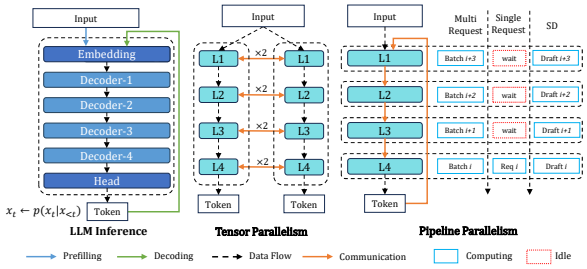


Figure 1: Left: Prefilling and decoding of LLM inference. Middle: Tensor Parallelism (TP). Right: Pipeline Parallelism with multi-/single-request and speculative decoding. L1-L4 here represent the hidden layers.

Among common strategies[1], pipeline parallelism (PP) suits low-bandwidth edge links for its low communication overhead and potential computation-communication overlapping, unlike tensor parallelism (TP) which demands frequent synchronizations (Fig. 1; see Appendix E). However, for LLM autoregressive generation, *the effectiveness of PP diminishes with sparse inference requests* (e.g., single request), which is common at the network edge (Ye et al., 2025). Since each

[1]Detailed discussion of parallel strategies for LLM inference are provided in Appendix A.

decoding step produces only one token, it cannot be further split into micro-batches to occupy the pipeline stages, leading to frequent cold starts and severe under-utilization. Moreover, autoregressive decoding is inherently I/O-bound: generating each token demands heavy memory accesses (i.e., KV cache) with minimal computation.

Speculative decoding (SD) (Leviathan et al., 2023) offers a fundamental resolution to these issues. By partitioning the multi-token draft tree into micro-batches, SD enables effective pipeline parallelism even for single request. Besides, verifying these drafts in a single forward pass generates multiple tokens, significantly reducing the access overhead of KV cache and weights. Thus, the synergy between SD and PP offers a promising solution for efficient edge LLM inference.

To this end, recent works explore asynchronous drafting and verification under pipeline architectures to further mitigate the sequential draft-then-verify dependency (McDanel, 2024; Butler et al., 2024; Yin et al., 2025). By deploying the draft model and base LLM on separate devices, verification of the current draft tokens and generation of new draft tokens can proceed in parallel, which overlaps the latency of draft generation. However, these asynchronous pipeline methods still face critical verification bottleneck: existing layer-wise draft tree verification (Butler et al., 2024; Yin et al., 2025) processes only one layer per pipeline turn, which fundamentally caps the average accepted length per pipeline turn (ALT, see Sec. 4) at 1. In practice, frequent draft rejections and pipeline interruptions further degrade efficiency, causing the actual ALT to fall far below 1. Without using contextual scores to prioritize high-confidence tokens across layers, this delay token acceptance and constrains decoding throughput.

Concurrently, SD has evolved into advanced context-aware frameworks (Li et al., 2024c,b; Du et al., 2024) that improve draft reliability by leveraging contextual information from the base model. Yet, integrating such advanced drafters into asynchronous pipelines exposes a second limitation: existing asynchronous methods expand the draft tree exclusively from leaf nodes. While this strategy suffices for traditional SD where the generated drafts rely solely on the input prefix, it becomes suboptimal for context-aware SD. Since context-aware SD drafting additionally conditions on the base model's latest feedback, the optimal draft distribution shifts as the context updates. Rigidly appending new tokens to leaves ignores such shifts, failing to enhance the stale low-quality draft under the updated context. This misalignment suppresses acceptance rates and frequently stalls the pipeline.

To address these, we propose **FlowSpec**, a pipeline-parallel speculative decoding framework for efficient distributed LLM inference with resource-constrained edge devices. To overcome the verification bottleneck, FlowSpec adopt fast draft initialization and scored-based, step-wise draft verification. It pre-generates a deep draft tree and prioritizes high-confidence candidates in step-wise verification, enabling accepting multiple tokens in a single pipeline turn and early acceptance of deeper draft tokens to accelerate decoding. Meanwhile, to resolve the stale draft issue caused by leaf-only expansion, FlowSpec incorporates a synergistic pruning-and-expansion mechanism for context-aware SD. The context-aware expansion dynamically regenerates fresh draft tree rooted in the latest context rather than rigidly extending from the stale leaf nodes. This mechanism sustains a continuous supply of high-quality drafts, significantly boosting acceptance rates and pipeline continuity. Our key contributions are listed as follows:

- FlowSpec introduces score-based step-wise draft verification that prioritizes high-confidence candidates across layers, enabling early token acceptance to accelerate decoding.
- FlowSpec incorporates synergistic dynamic pruning-and-expansion strategy tailored for context-aware SD to continuously supply high-quality drafts, improving acceptance rates and pipeline continuity.
- We evaluate FlowSpec on a real-world edge testbed. Extensive experiments demonstrate that FlowSpec achieves **2.5**×**–3.7**× speedups over vanilla decoding and **1.2**×**–1.5**× compared to state-of-the-art baselines. An ablation study further demonstrates the effectiveness.

2 Related Works

Distributed edge LLM inference primarily adopts Tensor Parallelism (TP) or Pipeline Parallelism (PP) to break single-device memory limits. TP-based methods (e.g., DeTransformer (Wei et al., 2024), Voltage (Hu and Li, 2024), GALAXY (Ye et al., 2024)) optimize intra-layer communication but remain constrained by frequent synchronization over low-bandwidth edge links. PP-based ap-

proaches ((Agrawal et al., 2023), (Zhang et al., 2024), (Jiang et al., 2026)) improve throughput via chunked scheduling, batching or load balancing. However, these works predominantly target the compute-intensive prefill phase, largely overlooking the I/O-bound decoding stage. Constrained by autoregressive generation, they fail to mitigate severe pipeline under-utilization and high latency in sparse-request edge scenarios, where decoding efficiency remains the critical bottleneck.

Speculative decoding (SD) accelerates autoregressive generation by drafting candidate tokens with a lightweight model and verifying them in parallel on the base LLM (Leviathan et al., 2023). Early tree-based methods (e.g., SpecInfer (Miao et al., 2024), Medusa (Cai et al., 2024)) organize candidates for batch verification to improve acceptance. Recent advanced context-aware frameworks ((Li et al., 2024c; Gloeckle et al., 2024; Du et al., 2024; Li et al., 2025)) significantly improve draft reliability by leveraging base-model hidden states, KV cache, and confidence-driven adaptive tree construction. However, directly deploying these methods in distributed edge scenarios without customized computation/communication scheduling results in suboptimal performance.

Asynchronous speculative decoding aims to break the sequential draft-then-verify dependency by overlapping drafting and verification across pipeline stages. Jupiter (Ye et al., 2025) enables request-level parallel SD with PP, while it applies to only specific outline-based tasks. Early systems (e.g., AMUSD (McDanel, 2024), PEARL (Liu et al., 2025)) enable async execution but limited in sequence-only drafting. PipeInfer (Butler et al., 2024) and PipeDec (Yin et al., 2025) introduces continuous tree-based SD with pruning and expansion. However, their layer-wise expansion and verification significantly constrain decoding throughput. Moreover, with context-aware SD, their rigid leaf-only draft expansion results in stale and low-quality drafts under the updated context, leading to constrained acceptance rates and weak pipeline continuity.

3 Methodology

3.1 FlowSpec Overview

FlowSpec is a pipeline system with $1 + N$ devices. As shown in Fig. 2, the first device (draft stage, D0) employs the draft model, is responsible for draft generation and accepting valid tokens based on the

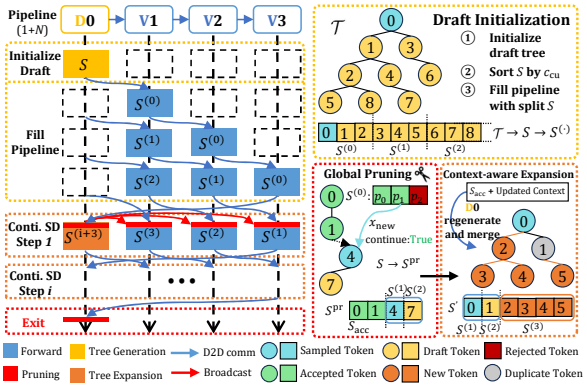


Figure 2: Overview of FlowSpec. Left: the pipeline runtime for a SD round, each round consists of a draft initialization step and multiple continuous SD steps; Right: the key components of FlowSpec.

verification scores from the base LLM. The base LLM is partitioned into N layer blocks distributed to the remaining N devices as verification stages (V1-VN). These stages are only responsible for executing the corresponding forward pass to verify the draft tokens on the base LLM.

To handle an LLM generation request, the input prompt is first processed in the **prefill phase** to compute the initial KV cache and the first sampled token. We adopt chunked prefill (Agrawal et al., 2023) to accelerate prefill with long sequences. After prefill, the system starts the **decoding phase**, which is the main focus of FlowSpec. This phase consists of multiple SD **rounds**. As shown in Fig. 2, an SD round operates as an asynchronous loop consisting of the following key phases:

Draft Initialization (Sec. 3.2). An SD round starts with D0 initializing a draft tree $\mathcal{T}$ with the associated draft sequence S in descending order of the draft scores. S is split into draft segments that sequentially fed into the verification pipeline. Once the first segment completes all verification stages, continuous SD step starts.

Continuous SD Step. The verification of existing drafts and the expansion of new drafts are executed asynchronously in each continuous SD step. Once a draft segment completes verification, D0 receives the verification result and determines the accepted tokens and an additional sampled token. As shown in Fig. 3, a continuous SD step consists of the following key operations:

- **Pruning & Context-aware Expansion**. If the continuous condition is satisfied, D0 performs global pruning to discard invalid draft tokens across all stages (Sec. 3.3). Subsequently, D0

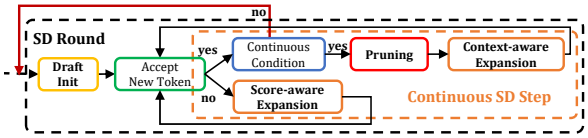


Figure 3: Workflow of the draft stage D0.

regenerates a fresh draft tree rooted from the latest context and merges it with the pruned tree as expansion (Sec. 3.4).

- **Score-aware Expansion**. If no new token are generated (a draft segment may become empty due to the continuous pruning), D0 directly extends the existing draft tree deeper based on the draft scores to maintain a steady stream of draft inputs.

- **Round Termination**. If the continuous condition fails, which means the tree can no longer satisfies the accepted prefix, the current SD round exits and a new round is initiated.

A draft initialization step accounts for $N+1$ pipeline **turns**, i.e., one for drafting on D0 and N for cold start from V1 to VN. A continuous SD step is considered as one pipeline **turn**, since drafting on D0 and verification of draft segments on V1 to VN can be executed in parallel.

In the following, unless specified, a draft sequence or segment includes its tree position IDs and attention mask.

3.2 Draft Initialization: Score-Based Draft Verification by Step

In the draft initialization step, D0 constructs a draft tree $\mathcal{T}$ of depth d_0 with L_0 nodes. Note that each node corresponds to a token, so we use "node" and "token" interchangeably. The root node is the newly sampled x_{new} from the prefill phase or the prior SD round. The construction of $\mathcal{T}$ contains the following procedures.

Initial Tree Generation. D0 sets the root node (layer 0) to be the latest x_{new}. Then, it uses the draft model to generate the initial tree $\mathcal{T}_{\text{base}}$. Inspired by EAGLE-2 (Li et al., 2024b), we use the cumulative confidence scores from the draft model to estimate the acceptance rate of the draft tokens. The tree is initially rooted at x_{new} and extended layer by layer based on the nodes with the highest cumulative scores in each layer. The cumulative score $c_{\text{cu}}(n_i)$ of node n_i in a draft token tree is defined as follows:

$$c_{\text{cu}}(n_i) = \prod_{n_j \in \text{path}(n_i)} c(n_j) = c_{n_i} \times c_{\text{cu}}(\text{parent}(n_i)), \tag{1}$$

where path(n_i) denotes all ancestor nodes along the path from the root node to n_i, and $c(n_j)$ is the confidence score of node n_j from the draft model. To construct the next layer of $\mathcal{T}_{\text{base}}$, the draft model takes the tokens from the previous layer as input and generates the corresponding next-token scores. Based on these scores, the top-k next tokens for each input token form the new layer of $\mathcal{T}_{\text{base}}$. Among these, the top-$k$ nodes with the highest cumulative scores are selected to continue the expansion of the next layer. D0 expands $\mathcal{T}_{\text{base}}$ until it reaches layer d_0. The initial depth d_0 serves as a hyper-parameter in FlowSpec, which will be discussed in Appendix D.

Score-Based Draft Segmentation. Then the top-L_0 nodes in $\mathcal{T}_{\text{base}}$ with the highest cumulative scores are selected to obtain the refined draft token tree $\mathcal{T}$, with the corresponding draft token sequence S sorted in descending order of the draft scores. Since a node always has a higher cumulative score than its children, the selected nodes can form a connected tree $\mathcal{T}$. Then S is split into consecutive, equal-length draft segments $S^{(0)}, ..., S^{(N)}$ with the maximum length of L_{max}. L_{max} is considered the token budget for a draft segment to help balance the forward latency between the verification stages while achieving sufficient compute utilization. These segments are then fed into the verification pipeline one after the other.

Step-wise verification of the draft tree based on cumulative scores offers following advantages. First, the descending-score order prioritizes the verification of higher-scoring draft tokens, increasing the likelihood accepting more tokens in advance, thereby enabling faster and deeper expansion of the draft token tree. Second, each draft segment consists of tokens from multiple tree layers, which allows verifying and accepting multiple tokens in a single pipeline turn (consistent with tree expansion in the following continuous SD steps). This breaks the throughput cap of layer-wise draft verification. An example of score-aware draft segmentation is provided in Appendix B. In addition, since a parent node always has a higher score than its children, this ordering of the draft segments establishes a valid topological order of the tree $\mathcal{T}$, which ensures valid causal relationship between tokens in

the subsequent step-wise verification (i.e., a preceding draft token must be ahead of its successors in verification).

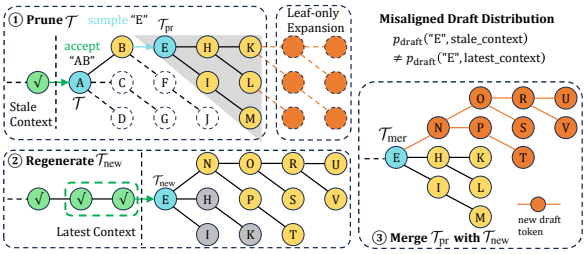


Figure 4: Example of a continuous SD step in *FlowSpec*, including tree pruning (①) and context-aware draft tree expansion (②, ③). Within the same parent, nodes are arranged top-down with decreasing confidence.

3.3 Continuous SD: Tree Pruning

In each continuous SD step, once a draft segment $S^{(i)}$ completes all verification stages, D0 receives its verification output from the base LLM, and computes the output probability of $S^{(i)}$. By comparing the output distributions from the draft model and the base model, tokens that lead to an aligned distribution with the base LLM are accepted, denoted by S_{acc}. Meanwhile, an additional sampled token x_{new} is generated based on the output probability and S_{acc}. After that, D0 determines whether to continue the current SD round according to the **continuous condition**, which is defined as:

$$\texttt{continuous_condition}(\mathcal{T}, S_{\text{acc}}, x_{\text{new}}) := \exists n_{\text{new}} \in \mathcal{T}\ s.t.\ \text{path}_{\mathcal{T}}(n_{\text{new}}) = \{S_{\text{acc}} || x_{\text{new}}\}, \quad (2)$$

where "||" denotes sequence concatenation, and $\text{path}_{\mathcal{T}}(n_{\text{new}})$ represents the sequence of tokens from the root to n_{new}. If the continuous condition is satisfied, which means x_{new} is already in the verification pipeline corresponding to $n_{\text{new}} \in \mathcal{T}$, thus the current SD round continues. Otherwise, violation of the continuous condition means that all remaining draft tokens become invalid. In this case, FlowSpec discards these invalid ones and their associated cache to avoid redundant overhead, exits the current SD round and initiates a new one.

The tree pruning process is illustrated in part ① of Fig. 4. To prune $\mathcal{T}$, D0 identifies all invalid branches according to S_{acc} and x_{new}. The nodes to be retained include n_{new} and all its descendant nodes in $\mathcal{T}$, and these nodes form the pruned tree $\mathcal{T}_{\text{pr}}$ with the new root node n_{new}. The corresponding pruned sequence S_{pr} preserves the original order of the retained tokens.

To maintain consistency across the pipeline, all verification stages (V1-VN) should synchronously prune their local draft segments and KV-cache based on D0's decisions. Instead of transmitting full tree structures, D0 only broadcasts a compact set of retained token indices $\mathcal{I}_{\text{retain}} = \mathcal{I}_{\text{acc}} \cup \mathcal{I}_{\text{pr}}$ to the downstream stages. Each stage then maps these global indices to its local context, efficiently discarding invalid draft tokens, hidden states and redundant KV cache entries while preserving the correct causal relationships and attention masks.

This lightweight, index-driven pruning mechanism eliminates redundant computation and memory overhead across the pipeline system with minimal communication cost (detailed in Appendix E). The efficient and correct draft management is an essential mechanism to support continuous speculative decoding on the pipeline system. Detailed implementations regarding global-to-local index mapping, draft segment truncation, and KV cache management are provided in Appendix C.

3.4 Continuous SD: Tree Expansion

Unless the current SD round terminates, D0 performs tree expansion in each continuous SD step. This process expands the current draft token tree to continuously supply high-quality draft tokens for verification and hence reduces the chance of cold starts in pipeline parallelism. There are two possible situations for tree expansion as follows:

Context-aware Expansion. When new tokens S_{acc} are accepted, D0 performs context-aware expansion immediately after pruning. Tokens retained in $\mathcal{T}_{\text{pr}}$ were generated under stale context and may misalign with $\mathcal{T}_{\text{new}}$ generated under the latest context. As shown in part ①-② of Fig. 4, $\mathcal{T}_{\text{new}}$ generated under the latest may differ from $\mathcal{T}_{\text{pr}}$, e.g., the top children node of "E" is "N" in $\mathcal{T}_{\text{new}}$ while "H" in $\mathcal{T}_{\text{pr}}$. Traditional leaves-only expansion directly extends from these outdated branches and ignores this misalignment, making it difficult to satisfy the *continuous condition* (Eq. 2) in the subsequent steps. To address this, FlowSpec regenerates a fresh tree $\mathcal{T}_{\text{new}}$ from the latest context and merges it with $\mathcal{T}_{\text{pr}}$. By explicitly constructing draft paths that originate from the verified prefix, this strategy significantly increases the probability of satisfying the continuous condition. Consequently, the current SD round is better sustained, ensuring a continuous supply of context-aligned drafts and minimizing disruptive pipeline restarts.

Specifically, D0 generates $\mathcal{T}_{\text{new}}$ of size L_{exp} and

depth d_{exp} based on the $\{S_{\text{acc}}||x_{\text{new}}\}$. $\mathcal{T}_{\text{new}}$ is constructed similarly to the initial draft tree and shares the updated $\mathcal{T}_{\text{base}}$ as its source. Then $\mathcal{T}_{\text{pr}}$ and $\mathcal{T}_{\text{new}}$ are merged as an updated $\mathcal{T}$ by appending only those nodes in $\mathcal{T}_{\text{new}}$ that are new to $\mathcal{T}_{\text{pr}}$. To identify new draft tokens in $\mathcal{T}_{\text{new}}$, D0 checks whether their paths exist in $\mathcal{T}_{\text{pr}}$, using a preloaded hash table of paths $\mathcal{P}_{\text{pr}} = \{\text{path}_{\mathcal{T}_{\text{pr}}}(n)|n \in \mathcal{T}_{\text{pr}}\}$ for efficient retrieval. Then the set of new draft tokens from $\mathcal{T}_{\text{new}}$ can be defined as $\mathcal{N}_{\text{new}} = \{n \in \mathcal{T}_{\text{new}}|\text{path}_{\mathcal{T}_{\text{new}}}(n) \notin \mathcal{P}_{\text{pr}}\}$. These tokens form a new draft segment S_{app}, which serves as the input for the next continuous SD step. S_{app} is then appended S_{pr}, resulting in the updated draft sequence $S = \{S_{\text{pr}}||S_{\text{app}}\}$. As shown in part ③ of Fig. 4, our tree expansion strategy has the potential to expand the draft tree from root to bottom with new draft tokens based on the updated context, significantly improving the continuity of pipelined SD.

Score-aware Expansion. Since pruning is applied to the remaining draft segments, some segments received by D0 may already be reduced to empty sequences. In this case, the context is not updated. To ensure a continuous supply of new draft tokens, D0 also expands $\mathcal{T}$ at this point.

Specifically, D0 expands the current $\mathcal{T}_{\text{base}}$ with additional d_{se} layers (as mentioned in Sec. 3.2) to a deeper draft tree $\mathcal{T}'_{\text{base}}$ from which we excluded all nodes that already exist in the current $\mathcal{T}$, and select the top-L_{se} nodes with the highest cumulative scores from the remaining ones as the expansion of $\mathcal{T}$. The new corresponding draft segment S_{se} is also appended to the draft sequence by $S \leftarrow \{S||S_{\text{se}}\}$, serving as the new draft input of the verification pipeline. As a result, we efficiently verify a larger draft tree, leading to a higher acceptance rate of draft tokens while ensuring a steady stream of new drafts for pipelined verification.

4 Experiment

4.1 Setups

Hardware. The testbed consists of 5 `NVIDIA Jetson Orin Nano` (NVIDIA Corporation, 2025) units, each equipped with 8GB CPU-GPU unified memory, interconnected via a local area network with a maximum bandwidth of 1000Mbps.

Models and Datasets. We evaluated FlowSpec on 4 LLMs with associated EAGLE draft models, i.e., LLaMA2-Chat 7B, 13B (Touvron et al., 2023) and Vicuna-v1.3 7B, 13B (Chiang et al., 2023). W4A16 quantization is applied to 13B models. The model weights are sourced from Hugging Face (Hugging Face, 2025). Performance is validated across 6 different downstream tasks, i.e., multi-turn dialogue, code generation, mathematical reasoning, instruction understanding, text summarization, and question answering. The associated datasets are MT-bench (Zheng et al., 2023), HumanEval (Chen et al., 2021), GSM8K (Cobbe et al., 2021), Alpaca (Taori et al., 2023), CNN/Daily Mail (Nallapati et al., 2016), and Natural Questions (Kwiatkowski et al., 2019), respectively.

Comparison[2]**.** We compare FlowSpec against 4 representative methods in distributed edge LLM inference, and vanilla autoregressive decoding on pipeline is served as the baseline. Megatron-LM (Mega-LM) (Shoeybi et al., 2019) enables tensor parallelism (TP) for forward pass on the base model. Galaxy (Ye et al., 2024) enhances TP by overlapping ring-based All-Reduce with split linear layers. Chunk-PP (Agrawal et al., 2023; Ye et al., 2025) accelerates long-draft tree verification with PP by splitting the draft tree into chunks for parallel pipeline verification. PipeDec (Yin et al., 2025) enables asynchronous SD with pruning and expansion in a layer-wise tree verification manner and leaf-only draft tree expansion. All methods adopt the context-aware EAGLE SD draft model (Li et al., 2024c,b). Implementation details for above methods are provided in Appendix F.

Metrics[3]**.** We consider the following metrics:

- **Tokens Per Second (TPS)**: The number of generated tokens per second in the **decoding phase**, which reflects the overall efficiency.
- **Speedup Ratio (SR)**: The TPS acceleration achieved compared to **vanilla autoregressive decoding** on the same pipeline testbed.
- **Average Accepted Length per Pipeline Turn (ALT)**: Total new tokens divides the total pipeline turns, demonstrating the decoding performance for pipeline methods.
- **Average Accepted Length per SD Round (ALR)**: Total new total tokens divides the total SD rounds, reflects the continuity of SD with pipeline parallelism.

[2]Due to prohibitively low throughput (i.e., TPS<1) with 13B-int4 models on the testbed, we evaluate Mega-LM and Galaxy only with 7B-fp16 models.

[3]Since FlowSpec adopts standard accept-reject sampling of SD that ensures identical output distribution to the base LLM, we omit the evaluation of output quality.

Table 1: Main results: FlowSpec vs. baselines on 6 datasets, under greedy (T=0) and stochastic (T=1) sampling (20 samples each). L2 and V denote LLaMA2-Chat and Vicuna-v1.3; 7B/13B indicate model sizes.

		MT-bench		HumanEval		GSM8K		Alpaca		CNN/DM		Natural Ques.		Mean	
Model	Method	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑
Temperature = 0															
L2 7B	Chunk-PP	6.72	1.90×	7.16	2.02×	6.25	1.77×	6.29	1.78×	5.06	1.48×	5.42	1.53×	6.15	1.75×
	Mega-LM	2.25	0.63×	2.41	0.68×	2.05	0.58×	2.06	0.58×	1.61	0.47×	1.75	0.49×	2.02	0.57×
	Galaxy	2.34	0.66×	2.50	0.70×	2.10	0.59×	2.12	0.60×	1.73	0.51×	1.78	0.50×	2.10	0.59×
	PipeDec	7.43	2.01×	7.56	2.14×	7.31	2.07×	7.25	2.05×	6.24	1.83×	6.83	1.93×	7.10	2.02×
	FlowSpec	**9.94**	**2.82×**	**10.44**	**2.96×**	**9.12**	**2.58×**	**9.01**	**2.55×**	**7.17**	**2.11×**	**8.03**	**2.27×**	**8.95**	**2.55×**
V 7B	Chunk-PP	7.42	2.09×	7.34	2.06×	6.83	1.92×	6.24	1.75×	4.95	1.45×	5.19	1.46×	6.32	1.79×
	Mega-LM	2.48	0.69×	2.46	0.69×	2.22	0.62×	2.00	0.56×	1.61	0.47×	1.66	0.46×	2.07	0.59×
	Galaxy	2.56	0.72×	2.54	0.71×	2.26	0.63×	2.05	0.57×	1.72	0.50×	1.69	0.47×	2.14	0.61×
	PipeDec	7.71	2.17×	7.66	2.15×	7.52	2.11×	7.35	2.07×	6.39	1.87×	6.88	1.93×	7.25	2.05×
	FlowSpec	**10.88**	**3.06×**	**10.63**	**2.99×**	**9.78**	**2.75×**	**9.19**	**2.59×**	**7.32**	**2.15×**	**7.62**	**2.15×**	**9.23**	**2.62×**
L2 13B	Chunk-PP	1.45	2.41×	2.63	1.00×	1.37	2.28×	1.25	2.08×	1.12	1.89×	1.13	1.88×	1.31	2.18×
	PipeDec	1.64	2.73×	1.69	2.81×	1.63	2.71×	1.60	2.66×	1.47	2.49×	1.55	2.58×	1.59	2.65×
	FlowSpec	**2.41**	**4.02×**	**2.63**	**4.38×**	**2.33**	**3.88×**	**2.11**	**3.52×**	**1.86**	**3.15×**	**1.94**	**3.23×**	**2.21**	**3.68×**
V 13B	Chunk-PP	1.58	2.63×	1.56	2.60×	1.42	2.36×	1.25	2.08×	1.15	1.95×	1.03	1.72×	1.33	2.21×
	PipeDec	1.68	2.80×	1.70	2.83×	1.67	2.78×	1.62	2.70×	1.45	2.45×	1.51	2.51×	1.60	2.66×
	FlowSpec	**2.62**	**4.37×**	**2.63**	**4.38×**	**2.38**	**3.97×**	**2.10**	**3.50×**	**1.89**	**3.20×**	**1.77**	**2.95×**	**2.23**	**3.72×**
Temperature = 1															
L2 7B	Chunk-PP	6.56	1.85×	6.63	1.87×	6.20	1.75×	5.87	1.66×	4.69	1.38×	5.30	1.50×	5.88	1.67×
	Mega-LM	2.13	0.60×	2.17	0.61×	2.07	0.58×	1.93	0.54×	1.54	0.45×	1.74	0.49×	1.93	0.55×
	Galaxy	2.15	0.61×	2.31	0.65×	2.06	0.59×	1.99	0.56×	1.66	0.49×	1.83	0.52×	2.00	0.57×
	PipeDec	7.37	2.08×	7.39	2.09×	7.42	2.10×	7.28	2.06×	6.23	1.83×	6.98	1.97×	7.11	2.08×
	FlowSpec	**9.51**	**2.70×**	**9.85**	**2.79×**	**9.01**	**2.55×**	**8.83**	**2.50×**	**6.87**	**2.02×**	**7.93**	**2.25×**	**8.67**	**2.47×**
V 7B	Chunk-PP	6.26	1.77×	6.32	1.79×	6.08	1.72×	5.39	1.52×	4.41	1.30×	4.58	1.30×	5.51	1.57×
	Mega-LM	2.01	0.57×	2.11	0.60×	1.93	0.55×	1.82	0.52×	1.46	0.43×	1.40	0.42×	1.80	0.51×
	Galaxy	2.11	0.60×	2.19	0.62×	2.03	0.57×	1.86	0.53×	1.57	0.46×	1.52	0.45×	1.89	0.54×
	PipeDec	7.45	2.11×	7.53	2.13×	7.30	2.06×	7.00	1.98×	6.04	1.82×	6.57	1.89×	7.02	2.00×
	FlowSpec	**9.48**	**2.69×**	**9.66**	**2.74×**	**8.71**	**2.47×**	**8.42**	**2.39×**	**6.70**	**1.98×**	**7.19**	**2.04×**	**8.37**	**2.39×**
L2 13B	Chunk-PP	1.35	2.25×	1.52	2.53×	1.30	2.16×	1.20	2.00×	1.07	1.81×	1.15	1.91×	1.26	2.10×
	PipeDec	1.65	2.75×	1.70	2.83×	1.63	2.71×	1.59	2.65×	1.47	2.49×	1.58	2.63×	1.60	2.66×
	FlowSpec	**2.31**	**3.85×**	**2.59**	**4.32×**	**2.20**	**3.67×**	**2.04**	**3.40×**	**1.72**	**2.92×**	**1.97**	**3.28×**	**2.14**	**3.57×**
V 13B	Chunk-PP	1.31	2.18×	1.41	2.35×	1.17	1.95×	1.06	1.76×	1.01	1.71×	0.97	1.61×	1.16	1.93×
	PipeDec	1.64	2.73×	1.64	2.73×	1.59	2.65×	1.55	2.58×	1.41	2.39×	1.49	2.48×	1.55	2.58×
	FlowSpec	**2.24**	**3.73×**	**2.33**	**3.88×**	**2.02**	**3.37×**	**1.81**	**3.02×**	**1.70**	**2.88×**	**1.64**	**2.73×**	**1.96**	**3.27×**

Table 2: ALT & ALR. Average accepted length per pipeline turn/SD round for methods under temperature=0 (for all six datasets).

		7B-version		13B-version	
Model	Method	ALT	ALR	ALT	ALR
L2	Chunk-PP	0.54	4.89	0.55	5.02
	PipeDec	0.59	5.68	0.59	5.76
	FlowSpec	**1.01**	**8.67**	**1.01**	**8.93**
V	Chunk-PP	0.55	5.00	0.55	5.02
	PipeDec	0.60	5.94	0.60	5.81
	FlowSpec	**1.02**	**9.96**	**1.04**	**9.28**

To ensure accurate time measurements, each method undergoes the same warm-up procedure prior to testing in order to equalize cache hit rates.

4.2 Main Results

Decoding Performance. Table 1 presents the results of FlowSpec and the baselines across multiple tasks on 4 LLMs. From the table, we observe that FlowSpec significantly outperforms both Chunk-PP and PipeDec on a variety of tasks. Across multiple tasks, FlowSpec achieves an average speedup of up to **1.73**× compared to Chunk-PP. Notably, TP-based methods (Mega-LM and Galaxy) exhibit significantly lower TPS, as about 75% of their execution time is consumed by communications.

Under the greedy sampling (i.e., Temperature=0), the decoding performance of FlowSpec far surpasses that of PipeDec and Chunk-PP, with speedup rates up to 3.06× for 7B models and 4.38× for 13B models. As for stochastic sampling (i.e., Temperature=1), experimental results highlights FlowSpec's remarkable advantage. FlowSpec achieves higher TPS than baselines across all six tasks, with speedup rates up to 2.79× for 7B models and 4.32× for 13B models. In general, FlowSpec obtains approximately average speedup rates of 2.5× for 7B models and 3.6× for 13B models compared to Chunk-PP, and about 1.2-1.4× speedups compared to PipeDec. The speedup of FlowSpec is also consistent for both greedy sampling and stochastic sampling. Owing to the uncertainty inherent in stochastic sampling, the acceleration achieved across different datasets and models exhibits appreciable variability.

Table 3: Ablation Results using the LLaMA2-Chat 7B model, showing the TPS across all six datasets under two sampling settings.

Method	MT-bench		HumanEval		GSM8K		Alpaca		CNN/DM		Natural Ques.		Mean	
	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑	TPS↑	SR↑
Temperature = 0														
Chunk-PP	6.72	1.90×	7.16	2.02×	6.25	1.77×	6.29	1.78×	5.06	1.48×	5.42	1.53×	6.15	1.75×
Pruned PP	8.25	2.33×	8.53	2.41×	7.92	2.24×	7.97	2.25×	6.61	1.94×	7.44	2.10×	7.78	2.21×
FlowSpec w/o SBD	9.38	2.65×	9.95	2.81×	8.83	2.50×	8.72	2.47×	6.98	2.05×	7.87	2.22×	8.62	2.45×
FlowSpec	**9.94**	**2.82×**	**10.14**	**2.96×**	**9.12**	**2.58×**	**9.01**	**2.55×**	**7.17**	**2.11×**	**8.03**	**2.27×**	**8.95**	**2.55×**
Temperature = 1														
Chunk-PP	6.56	1.85×	6.63	1.87×	6.20	1.75×	5.87	1.66×	4.69	1.38×	5.30	1.50×	5.88	1.67×
Pruned PP	7.97	2.25×	8.23	2.33×	7.83	2.21×	7.48	2.11×	6.31	1.85×	7.09	2.00×	7.48	2.13×
FlowSpec w/o SBD	8.84	2.50×	9.09	2.57×	8.82	2.49×	8.44	2.39×	6.57	1.93×	7.76	2.19×	8.25	2.35×
FlowSpec	**9.51**	**2.70×**	**9.85**	**2.79×**	**9.01**	**2.55×**	**8.83**	**2.50×**	**6.92**	**2.04×**	**7.93**	**2.25×**	**8.67**	**2.47×**

ALT & ALR. Table 2 shows the mean value of average accept length of each pipeline **turn** and SD **round** (explained in Sec. 3.1) across all six datasets. We can approximately consider that all pipeline turns have equal latency across different methods, thus ALT reflects the overall decoding performance. Results show that FlowSpec has a larger average accepted length for each pipeline turn compared to the baselines, demonstrating its effectiveness in accelerating speculative decoding. Moreover, the ALR of FlowSpec exhibits significantly improved continuity on SD with pipeline parallelism, enabling generating more tokens in a single SD round and reduced pipeline cold starts, leading to increased decoding efficiency.

While the layer-wise verification design of PipeDec limits the throughput of SD, FlowSpec unlocks the performance of advanced context-aware approaches by accepting more tokens in a single pipeline step. The context-aware expansion strategy also improves the continuity of context-aware SD on pipeline, which further boosts efficiency.

4.3 Ablation Study

Table 3 presents the results for different module configurations of FlowSpec on multiple tasks using LLaMA2-Chat 7B under two sampling conditions (Temperature = 0 or 1). *Chunk-PP* follows the same setting as above. *Pruned Pipeline Parallelism (Pruned PP)* extends Chunk-PP by leveraging Tree Pruning. *FlowSpec w/o Score-Based Draft (FlowSpec w/o SBD)* omits Score-Based Draft Segmentation while applying Tree Expansion on top of Pruned PP. The different combinations of components confirm the effectiveness of each module.

Score-Based Draft Segmentation. Comparing FlowSpec to FlowSpec w/o SBD in Table 3 directly highlights the benefit of the Score-Based component. By sorting draft tokens according to their scores, Score-Based Draft Segmentation accepts more draft tokens early to accelerate decoding in the draft initialization phase. As a result, FlowSpec achieves higher throughput FlowSpec w/o SBD.

Tree Pruning. Comparing Chunk-PP and Pruned PP demonstrates the effectiveness of tree pruning. By discarding invalid branches early, pruning eliminates redundant verification steps, leading to improved SD efficiency and performance.

Tree Expansion. Table 3 demonstrates substantial speedups of Tree Expansion. Notably, on the multi-turn dialogue task, FlowSpec w/o SBD achieves significantly higher throughput than Pruned PP for both sampling cases. Similar performance gains are observed across other datasets. This indicates that Tree Expansion improves continuity and reduces PP cold starts, leading to significant performance enhancement.

5 Conclusion

This paper introduces FlowSpec, a pipeline-parallel distributed LLM inference framework tailored to accelerate advanced context-aware speculative decoding on resource-constrained edge devices. To address inefficient draft verification and the stale draft issues inherent in existing methods, FlowSpec integrates three synergistic mechanisms: score-based step-wise verification for cross-layer early token acceptance; lightweight draft pruning to eliminate redundant overhead; and context-aware dynamic tree expansion that regenerates fresh drafts using the latest context for improved acceptance and pipeline continuity. Real-word testbed evaluations demonstrate that FlowSpec significantly outperforms state-of-the-art baselines, delivering 2.4x-3.7× speedups over autoregressive decoding while maintaining robust pipeline continuity.

Limitations

Despite the effectiveness of FlowSpec, some limitations still exist:

Concurrent serving: FlowSpec targets latency-sensitive, sparse-request edge scenarios (e.g., IoT gateways), where pipeline under-utilization is the core bottleneck. Extending our single-request asynchronous method to concurrently continuous batching introduces orthogonal challenges in multi-tree scheduling and KV-cache multiplexing. Adapting FlowSpec to balance per-request latency with system throughput in concurrent settings on resource-constrained edge devices remains a valuable direction for future work.

Pipeline scalability: FlowSpec is optimized for small-scale edge clusters where the number of available devices naturally bounds the pipeline depth. Scaling to massive pipeline stages inevitably incurs linear growth in inter-stage communication latency, a fundamental constraint of pipeline parallelism over bandwidth-constrained networks. For larger-scale deployments, integrating hybrid parallelism strategies helps control the pipeline depth while scaling efficiently.

References

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*.

Branden Butler, Sixing Yu, Arya Mazaheri, and Ali Jannesari. 2024. Pipeinfer: Accelerating llm inference using asynchronous pipelined speculation. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–19.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 34 others. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

Karl Cobbe, Vineet Kosaraju, Mo Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168.

Cunxiao Du, Jing Jiang, Xu Yuanchen, Jiawei Wu, Sicheng Yu, Yongqi Li, Shenggui Li, Kai Xu, Liqiang Nie, Zhaopeng Tu, and Yang You. 2024. Glide with a cape: a low-hassle method to accelerate speculative decoding. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Sukhpal Singh Gill, Muhammed Golec, Jianmin Hu, Minxian Xu, Junhui Du, Huaming Wu, Guneet Kaur Walia, Subramaniam Subramanian Murugesan, Babar Ali, Mohit Kumar, and 1 others. 2025. Edge ai: A taxonomy, systematic review and future directions. *Cluster Computing*, 28(1):1–53.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. 2024. Better & faster large language models via multi-token prediction. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Chenghao Hu and Baochun Li. 2024. When the edge meets transformers: Distributed inference with transformer models. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 82–92.

Hugging Face. 2025. Hugging face models. `https://huggingface.co/models`. Accessed: 2025-12-07.

Jie Jiang, Zhixiong Chen, Hyundong Shin, and Arumugam Nallanathan. 2026. Edge inference for large language models with pipeline parallelism and batching. *IEEE Transactions on Communications*, 74:8390–8406.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Matthew Kelcey, Jacob Devlin, Kenton Lee, Kristina N. Toutanova, Llion Jones, Ming-Wei Chang, Andrew Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural questions: a benchmark for question answering research. *Transactions of the Association of Computational Linguistics*.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR.

Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. 2024a. Llm inference serving: Survey of recent advances and opportunities. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024b. EAGLE-2: Faster inference of language models with dynamic draft trees. In *Empirical Methods in Natural Language Processing*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024c. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: scaling up inference acceleration of large language models via training-time test. *CoRR*, abs/2503.01840.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for llm compression and acceleration. In *MLSys*.

Tianyu Liu, Yun Li, Qitan Lv, Kai Liu, Jianchen Zhu, Winston Hu, and Xiao Sun. 2025. PEARL: parallel speculative decoding with adaptive draft length. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Bradley McDanel. 2024. Amusd: Asynchronous multi-device speculative decoding for llm acceleration. *arXiv preprint arXiv:2410.17375*.

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. 2024. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 932–949, New York, NY, USA. Association for Computing Machinery.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. *arXiv preprint arXiv:2402.06196*.

Ramesh Nallapati, Bowen Zhou, Cícero Nogueira dos Santos, Çaglar Gülçehre, and Bing Xiang. 2016. Abstractive text summarization using sequence-to-sequence rnns and beyond. In *Conference on Computational Natural Language Learning*.

NVIDIA Corporation. 2025. Nvidia jetson orin nano super developer kit. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/. Accessed: 2025-12-07.

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, and 2 others. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Victor Sanh, Thomas Wolf, and Alexander M. Rush. 2020. Movement pruning: Adaptive sparsity by fine-tuning. *ArXiv*, abs/2005.07683.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, and 1 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Yuanxin Wei, Shengyuan Ye, Jiazhi Jiang, Xu Chen, Dan Huang, Jiangsu Du, and Yutong Lu. 2024. Communication-efficient model parallelism for distributed in-situ transformer inference. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*.

Shengyuan Ye, Jiangsu Du, Liekang Zeng, Wenzhong Ou, Xiaowen Chu, Yutong Lu, and Xu Chen. 2024. Galaxy: A resource-efficient collaborative edge ai system for in-situ transformer inference. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, pages 1001–1010.

Shengyuan Ye, Bei Ouyang, Liekang Zeng, Tianyi Qian, Xiaowen Chu, Jian Tang, and Xu Chen. 2025. Jupiter: Fast and resource-efficient collaborative inference of generative llms on edge devices. *arXiv preprint arXiv:2504.08242*.

Haofei Yin, Mengbai Xiao, Rouzhou Lu, Xiao Zhang, Dongxiao Yu, and Guanghui Zhang. 2025. Pipedec: Low-latency pipeline-based inference with dynamic

speculative decoding towards large-scale models. *arXiv preprint arXiv:2504.04104*.

Mingjin Zhang, Xiaoming Shen, Jiannong Cao, Zeyang Cui, and Shan Jiang. 2024. Edgeshard: Efficient llm inference via collaborative edge computing. *IEEE Internet of Things Journal*, pages 1–1.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Haotong Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *ArXiv*, abs/2306.05685.

A Generation and Parallelism of LLMs

A.1 LLM Inference and Generation

Decoder-only LLM Architecture: Modern LLMs adopt a decoder-only architecture, composed of a token embedding layer followed by multiple stacked transformer decoder layers. Each decoder layer integrates self-attention mechanisms, feed-forward networks (FFNs) and residual connections, enabling LLMs to capture complex contextual dependencies. The final output of the last decoder layer is passed through a classification head to produce probability distributions for the next token. An example is illustrated in the left part of Fig 1.

Generation Phases of LLMs: The sequence generation of LLMs relies on the key-value caching (KV cache) mechanism, which divides the generation process into two phases: **prefilling** and **decoding**. In the **prefilling phase**, the LLM processes the entire input prompt in a single forward pass to generate the probability of the first new token and save the KV cache for all initial input tokens. This phase is characterized by high computational intensity and is typically a computational bottleneck. Then in the **decoding phase**, the LLM autoregressively generates the next new token based on the previous one. Each step involves a low-FLOPs forward propagation and requires frequent access to the KV cache and model parameters, making this phase inherently I/O-bound.

A.2 Parallelism Strategies

The growth rate of the model parameter scales far outpaces that of hardware performance improvement, resulting in an increasing demand for distributed LLM deployment. To scale LLM inference efficiently, various parallelism strategies have been proposed. Besides model-partitioning methods TP and PP mentioned previously, other strategies are listed as follows:

Data parallelism (DP) distributes distinct inference requests across multiple complete model replicas to enable parallel computation, typically employed to enhance the throughput of training and inference on high-performance computing (HPC) systems. However, DP becomes infeasible in edge scenarios with constrained memory and sparse requests.

Sequence parallelism (SP) partitions the input along the sequence dimension to enable parallel layer execution. Similar to TP, SP also requires synchronization (e.g., All-Gather) within each layer, but critically, it necessitates a complete model replica on each device. This conflicts with the memory constraints and low-bandwidth nature of typical edge environments, limiting its practicality.

In summary, **only the strategies with model partitioning** such as TP and PP can address the memory constraints of edge devices. However, they face key challenges in LLM decoding under edge scenarios. TP suffers from high communication overhead due to frequent synchronization under limited bandwidth, while PP typically struggles with low concurrency and resource utilization when dealing with sparse inference requests.

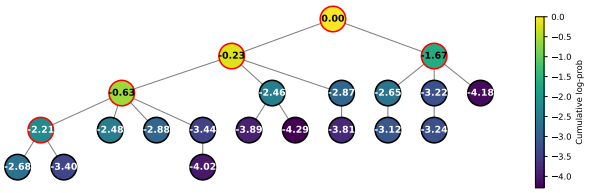


Figure 5: Visualization of a real generated draft tree. The number and color in each draft node demonstrate the cumulative log-probability.

B Visualized Draft Tree with Scores

Figure 5 shows an example of a generated draft tree with 20 draft tokens. When constructing a draft token tree, the tokens with higher confidence scores are more preferable to be selected, thus the draft tree visually appears "left-heavy and right-light". In case of $L_{\text{max}} = 5$, the first draft segment (with outline in red) includes draft tokens from layers 1-4, meaning that at most 4 tokens can be accepted in this single segment. Compare to layer-wise draft verification, the score-aware segmented verification can bring cross-layer verification in a single segment and early token acceptance.

C Pruning Details

Besides D0, V1 to VN also require consistent pruning to update their local draft segments. Let $\mathcal{I}_{\text{acc}}$ and $\mathcal{I}_{\text{pr}}$ denote the ordered indices of the accepted tokens and retaining tokens of $\mathcal{T}_{\text{pr}}$ in the sequence S before pruning. D0 sends the full retaining set $\mathcal{I}_{\text{retain}} = \mathcal{I}_{\text{acc}} \cup \mathcal{I}_{\text{pr}}$ as the pruning information (a few hundreds of bytes) to V1-VN. Let l_{glo} denote the global context offset (i.e., the total length of the input prompt and all accepted tokens), which will be updated by $l_{\text{glo}} = l_{\text{glo}} + |S_{\text{acc}}|$ after each pruning procedure. Then each element in $\mathcal{I}_{\text{retain}}$ is

incremented by l_{glo} such that the element indicates the position index of the associated token in the global sequence. For each stage, the pruning operation is applied to both the local draft segment and KV cache based on the following two index sets:

- Draft segment pruning: The local positions in the current draft segment corresponding to the tokens that need to be retained in the pruned tree $\mathcal{T}_{\text{pr}}$, defined as $\mathcal{I}_{\text{local}} = \{i - l_{\text{cache}} | i \in \mathcal{I}_{\text{retain}}, l_{\text{cache}} \leq i < l_{\text{cache}} + l_{\text{s}}\}$. Here, l_{cache} and l_{s} denote the length of the KV cache and the length of the draft segment at the current stage, respectively. The indices are mapped from the global positions to local draft segment positions by subtracting l_{cache}.

- KV cache pruning: The indices of nodes (tokens) to be retained in the KV cache are those associated with indices in set $\mathcal{I}_{\text{incache}} = \{i \in \mathcal{I}_{\text{retain}} | i < l_{\text{cache}}\}$. Since KV cache includes the complete context, the indices used for cache pruning correspond to global positions.

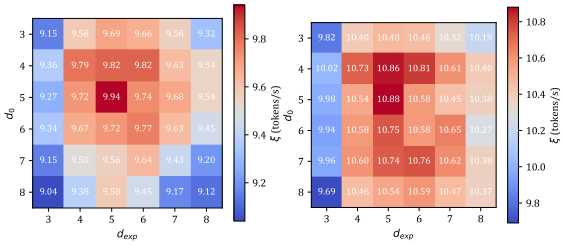
To prune a local draft segment (including its hidden states, tree position IDs and attention mask), we retain only the entries corresponding to $\mathcal{I}_{\text{local}}$, and the full draft sequence S is accordingly pruned to be S_{pr} consisting of the pruned segments from V1 to VN. To update the KV cache, we first retain the KV cache entries from to the previous context (i.e., positions before l_{glo}). For the KV cache associated with the draft tokens (i.e., positions $\geq l_{\text{glo}}$), we only keep those indexed by $\mathcal{I}_{\text{incache}}$. This removes redundant KV cache entries and intermediate results, ensuring both consistency and memory efficiency in subsequent decoding steps.

D Tree Parameters

D.1 Tree Parameters Analysis

As introduced in Section 3, FlowSpec controls draft generation via three key structural parameters: tree depth (d_0, d_{exp}), tree size (L_0, L_{exp}), and tree width (k).

For tree size, the initial draft length L_0 can be set relatively large (e.g. $L > NL_{\max}$), since T_0 is segmented and verified incrementally across V1–VN. In contrast, the expansion size L_{exp} must account for potential overlap with the existing tree and post-expansion pruning; empirically, setting L_{exp} such that its *effective* contribution approaches $L_{\max}$ helps balance verification workloads and mitigate



(a) L2-7B (b) V-7B

Figure 6: FlowSpec's TPS with varying tree depth d_0 and d_{exp} on MT-bench.

pipeline bubbles. While tree width (k) may affect the inference latency of the draft model, it empirically takes a small value ($k < L_{\max}$), thus it makes less impacts on the system performance. We follow the empirically optimal setting of k in EAGLE-2(Li et al., 2024b).

As for the tree depth, it critically affects both the drafting overhead and the expected number of accepted tokens per verification step. First, the draft generation latency t_{draft} scales linearly with depth d, i.e., $t_{\text{draft}} \approx dt_D$, where t_D is the latency to execute a single forward pass of the draft model (i.e., generating one tree layer). Besides, a deeper draft tree increases the chance of accepting more tokens per verification, enhancing the expected throughput.

In the draft initialization step, a draft tree of depth d_0 is generated. Since this step provides the initial input for a new SD round before pipeline overlap can be established, its latency is fully exposed. Therefore, d_0 should balance token acceptance gains against the drafting latency, accounting for draft model accuracy, inference cost and pipeline architecture. In each continuous SD step, while the existing draft tokens undergo verification, D0 expands the draft tree by d_{exp} additional layers to produce new candidates. To maximize D0's pipeline utilization, D0 should fully exploit the available verification window without becoming the system bottleneck. Let t_V denote the execution latency of each verification stage, setting $d_{\text{exp}} \approx t_V/t_D$ ensures balanced latency in each continuous SD step between the draft stage D0 and verification stages V1-VN, while also provides sufficiently deep exploration for subsequent speculation.

D.2 Tree Depth Evaluation

As analyzed in Section D, the draft tree depth may serve as a key trade-off parameter between computational cost of the draft stage and overall generation efficiency. For deploying the EAGLE (Li

et al., 2024c) draft model and the 7B base model on our testbed, the inference latency $t_d \approx 14$ms and $t_v \approx 80$ms. Based on this latency profile, we evaluate FlowSpec across varying tree depths d_0 and d_{exp} using the LLaMA-2-Chat model on the MT-Bench benchmark. Results are shown in Figure 6. FlowSpec achieves optimal performance when the initial tree depth d_0 and extension depth d_{exp} are both set to 5 (around t_d/t_v). Under this configuration, the draft stage D0 effectively saturates the time budget of each pipeline iteration closely, producing sufficiently deep draft trees and fully exploiting the draft model's capacity. Crucially, this depth aligns the computation load of D0 with that of the subsequent verification stages (V1–VN), ensuring balanced stage latencies and thereby minimizing pipeline bubbles and improving overall throughput.

E Communication

Communication in FlowSpec consists of two parts: (i) pruning metadata (see Sec. C) and (ii) draft segments or intermediate pipeline results. Although (i) is introduced as additional communication for cross-device draft pruning, its excessively low transmission volume together with low-latency transmission in edge scenarios, typically ensure minimal overhead. (ii) is the inevitable transmission in pipeline systems and constitutes the main communication component in FlowSpec.

Given that (ii) dominates the communication footprint, its overhead in bandwidth-constrained edge scenarios may affect FlowSpec's practical deployment performance. PP-based approaches, owing to their inherently lower inter-device transmission volume and opportunity for comm-comp overlap, are better suited to low-bandwidth deployment scenarios compared to TP-based methods. Nevertheless, when bandwidth becomes excessively constrained, the resulting increase in inter-device transmission latency can no longer be effectively masked, thereby degrading the overall throughput of the pipelined inference. First, transmission latency should be kept below the execution latency of each pipeline stage to ensure that communication overhead can be fully overlapped with computation during continuous SD steps. While this can be satisfied by excessively low bandwidth in most cases. Further more, higher transmission latency delays the arrival of inputs at the final pipeline stage during cold-start, thereby increasing the draft initialization overhead. However, the improved pipeline continuity mitigates this issue.

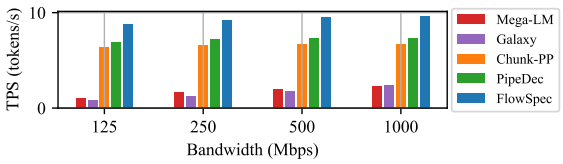


Figure 7: FlowSpec's TPS under different bandwidths for LLaMA2-chat on MT-bench.

We have also evaluated FlowSpec with other methods under varying bandwidth. Figure 7 illustrates the TPS of the aforementioned methods under varying inter-device bandwidth conditions. As observed, the performance of the PP-based approaches are less sensitive to bandwidth variations compared to the TP-based ones. Specifically, when the bandwidth decreases from 1000 Mbps to 125 Mbps, the PP-based methods retain over 90% of its original performance, whereas TP-based methods suffers a substantial degradation, dropping to approximately 40% of its original throughput. Moreover, across all bandwidth constraints, FlowSpec consistently outperforms other baselines, achieving a speedup over 1.37× compared to Chunk-PP.

F Implementation Details

All operations within the draft stage associated with the draft tree are performed on CPU for efficiency, given their low parallelism, whereas the model forward computations of the draft model and the base model are delegated to GPU.

We build the testbed using `NVIDIA JetPack 5.1.2 (L4T R35.4.1)` with `CUDA 11.4` support. We employ the ARM-aarch64-specific `PyTorch 1.11.0+cu114` (Paszke et al., 2019), installed via NVIDIA's official Python 3.8 wheel. The speculative decoding component in our core codebase extends the original EAGLE-2 implementation.

For each `NVIDIA Jetson Orin Nano` unit, its actual available GPU memory is $\sim$ 5GB. To deploy the 13B parameter model on the rest 4 devices, we employed W4A16 quantization (4-bit storage while preserving 16-bit computation precision) via the `bitsandbytes` library integrated in HuggingFace `transformers`. The primary cause of the significantly lower decoding performance on the 13B models is the additional dequantization overhead in the runtime.

All evaluated methods leverage the advanced EAGLE speculative decoding framework. The

draft model is deployed on a single separate device, with the base LLM partitioned across the rest deices (via TP or PP). For vanilla auto-regressive decoding on the testbed, only the last 4 devices are utilized for deploying the base LLM split in layer-wise. Besides, continuous pipeline pruning of draft segments can cause workload imbalance across verification stages. To mitigate this, the initial maximal segment length L_{max} should be properly set to ensure its forward latency on the base LLM remains within a low ratio (e.g., 10%) of that for a single-token sequence. For fair comparison, we set $L_{\text{max}} = 16$ and $L_0 = 80$ correspondingly. We also set $d_0 = 5$, taking into account the forward latency ratio between the draft model and the base model stages (t_V/t_D), alongside its empirically optimal performance in EAGLE-2 and our tree depth evaluations above. $k = 10$ is set for all methods except for PipeDec.

Detailed implementations are list as following:

- **Chunk-PP**: the draft tree are generated with size L_0 and depth d_0 in a EAGLE-2 style, then the tree are verified incrementally in segments with length L_{max} by the verification pipeline.
- **Mega-LM**: similar to Chunk-PP, but the draft teee verification in executed in TP style.
- **Galaxy**: similar to Chunk-PP, but the draft tree verification is executed in Galaxy style (optimized TP fine-grained asynchronous All-Reduce).
- **PipeDec**: it enables asynchronous draft generation and verification in a layer-wise manner and leaf-only expansion. Therefore, the tree width k and the maximal draft segment length L_{max} is inherently the same. Thus we set $L_{\text{max}} = k = 16$, which also aligns with its origin settings.
- **FlowSpec**: we further configure $L_{\text{exp}} = 64$ and $d_{\text{exp}} = 5$. As derived in Appendix D, d_{exp} is set to align the draft stage latency with the pipeline stage bubble or verification stage budget ($d_{\text{exp}} \approx t_V/t_D$), preventing D0 from becoming a pipeline bottleneck while maintaining sufficient tree depth. L_{exp} is chosen to compensate for tokens discarded during pruning, ensuring the post-pruning segment length approximates L_{max} for balanced stage workloads. The tree width k is constrained to keep single-layer draft generation within the stage latency budget, rather than relying on external empirical defaults.

To prevent out-of-memory (OOM) issues during evaluation on long inputs, we set the maximum output length to 256 or all methods. This constraint is applied uniformly across baselines and FlowSpec to ensure fair comparison. All other experimental settings mirror those of the main experiments.

G Towards Concurrent Requests

While FlowSpec currently targets single-request scenarios to eliminate under-utilization of pipeline, extending it to concurrent continuous batching offers a direct pathway to maximize edge device utilization. This extension introduces following interconnected design directions:

(1) Speculation-Aware Kernel Co-design. Existing serving primitives (e.g., PagedAttention for KV-cache multiplexing, FlashAttention for batched verification) provide a foundational stack. However, efficiently verifying irregular, interleaved draft trees from concurrent requests remains a bottleneck. Developing *dynamic-shape kernels* with JIT compilation support to fuse sparse attention operations, minimize kernel launch overhead, and adapt to variable tree depths is a critical systems challenge.

(2) Multi-Level Priority Scheduling. Resource-constrained edge concurrency requires a global token budget to maintain pipeline stage balance. Within this budget, a hierarchical scheduler could dynamically allocate verification slots by jointly considering per-request SLOs, draft confidence scores, and overall draft acceptance rates. This enables priority-aware token verification that balances individual latency guarantees with global throughput.

(3) Concurrency-Adaptive Draft Control. Sustaining high throughput demands dynamically scaling draft generation parameters (e.g., d_0, $d_{\text{exp}}L_{\text{exp}}$) based on real-time request concurrency and observed acceptance feedback. Low concurrency may justify deeper or wider draft trees to fully saturate compute resources, whereas high concurrency should trigger conservative drafting to avoid wasted computation and maintain high system throughput. Designing such concurrency-aware adaptive policies remains a promising direction for future work.