# STARSD: ONE-FOR-MANY SPECULATIVE DECODING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Speculative decoding accelerates autoregressive generation by introducing a draft-then-verify procedure, but most existing approaches are designed for single-node execution and do not scale well to multi-accelerator clusters used for serving modern Large Language Models (LLMs). We present StarSD, a one-for-many speculative decoding framework that uses a single draft model to serve multiple target models across distributed nodes via a star topology. StarSD decouples drafting and verification, enabling effective sharing of draft computation, and preventing distributed accelerators from remaining idle under bursty workloads. We provide a system-level analysis that characterizes when and why a single draft model can remain fully utilized by multiple verifiers, yielding predictable latency and utilization gains. Extensive experiments in real-world distributed inference settings demonstrate that StarSD simplifies deployment and supports flexible resource allocation across heterogeneous accelerators, while maintaining output quality. These results indicate that StarSD is a practical and scalable framework for bringing speculative decoding to memory-constrained multi-session serving.

## 1 INTRODUCTION

Large Language Models (LLMs) have become a central interface for a wide range of applications, and their deployment increasingly involves serving large volumes of concurrent inference requests on the service provider's shared accelerator clusters. Under such workloads, inference systems face sustained pressure on throughput, Service Level Objective (SLO) related latency, and hardware efficiency, often operating near the limits of memory and compute. A key source of this pressure is *autoregressive* decoding, where each output token requires a serial forward pass, directly constraining end-to-end latency and limiting achievable throughput. This bottleneck has motivated many decoding acceleration methods that restructure generation to reduce the cost of sequential decoding. Among them, speculative decoding provides lossless acceleration by using a lightweight *draft* model to propose multiple tokens and a large *target* model to verify them in parallel, preserving the target model's output distribution while reducing decoding time (Leviathan et al., 2023; Chen et al., 2023). This property distinguishes speculative decoding from approximate acceleration methods such as early-exit decoding (Elbayad et al., 2019; Schuster et al., 2022), because it maintains exact generation semantics while introducing parallel verification for several tokens, making it well-suited for latency-sensitive serving. Recent methods build on this advantage by improving draft quality and reducing verification overhead, which increases acceptance rates and yields further decoding speedups (Li et al., 2024a; Miao et al., 2024; Li et al., 2024b).

However, most speculative decoding designs implicitly adopt a centralized execution mode, in which the draft and target models are co-located on a single accelerator, e.g., a Graphics Processing Unit (GPU). This pattern becomes increasingly limiting as modern LLMs continue to grow in parameter scale and inference workloads that impose sustained memory pressure through large Key-Value (KV) caches, causing the combined footprint to approach or even exceed the capacity of a single accelerator. For instance, co-locating a Vicuna-7B (Zheng et al., 2023) target model with the lightweight EAGLE draft model can leave little memory headroom (Li et al., 2024a) for an RTX 4090 (24GB), restricting support for long-context serving (see Appendix C for detailed analysis). Moreover, as LLMs expand into vertical domains and are increasingly deployed on resource-constrained edge devices, e.g., PCs, mobile phones, or NVIDIA Jetson, co-location becomes even less practical due to tighter memory and compute budgets. Although substantial work has explored memory-efficient inference techniques (Kwon et al., 2023; Sheng et al., 2023), these methods do not remove the need



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

to host the target-draft model pair on every accelerator, which constrains batching efficiency and scheduling flexibility.

Consequently, speculative decoding-aided LLM inference systems require a spatial degree of freedom, enabling distributed placement that allows drafting and verification to be placed independently while cooperating tightly to share accelerator resources. In principle, this flexibility supports higher utilization by decoupling compute-intensive drafting from request-driven verification, and applies both within a single server across multiple accelerators and distributed clusters across machines. However, simply separating the draft and target models reveals an accelerator-level inefficiency due to a mismatch between the model inference pattern and the operating characteristics of modern accelerators. Specifically, the draft-then-verify workflow alternates between drafting, verification, and communication, while the target model verifies proposed tokens, the draft model remains idle. This alternating execution introduces *idle gaps* that limit end-to-end throughput. In addition, disaggregation makes the draft execution *bursty*, with short compute phases separated by millisecond-scale gaps. Such bursty execution can prevent the draft model from staying in a warm compute state, increasing per-token latency and its variance compared to continuous execution (Schall et al., 2023).

Addressing these challenges requires a speculative decoding framework that coordinates drafting and verification across machines while keeping modern accelerators steadily occupied. We propose StarSD, a one-for-many distributed speculative decoding framework in which a single draft model assists multiple target instances. StarSD multiplexes asynchronous verification returns through a request buffer, isolates the state of each target instance, and executes requests in a work-conserving loop so that the draft accelerator can process available work without waiting for all targets to finish a round. StarSD yields two complementary benefits under multi-instance workloads. First, it amortizes verification and communication return latency across multiple target instances, which reduces draft idle gaps and increases aggregate throughput. Second, the continuous request stream keeps the draft side closer to a steady state execution regime, improving per-token compute efficiency and reducing execution variability. To analyze StarSD under concurrent serving, we develop an analytical system model that separates progress, measured by the expected number of accepted tokens per round, from time, measured by round duration. This separation characterizes underloaded and fully loaded regimes and identifies the load transition at which idle gaps disappear. Our contributions are summarized below.

- We propose StarSD as a one-for-many distributed speculative decoding framework, where a single draft model serves multiple target instances. StarSD isolates per-target state, multiplexes asynchronous verification returns through a request buffer, and enables work-conserving draft execution whenever pending requests are available.
- We develop an analytical system model that separates per-round progress from per-round time. The model explains the load regimes and scaling behavior of StarSD, identifies the transition point where draft idle gaps vanish, and characterizes the tradeoff between per-target throughput and system-wide throughput.
- We evaluate StarSD across various deployments, including intra-machine and inter-machine settings, showing consistent benefits in different speculative decoding backends.

## 2 Related Work

**Speculative Decoding.** Speculative decoding accelerates autoregressive generation by separating proposal and verification. A lightweight draft model proposes candidate continuations, and a stronger target model verifies them while preserving the target distribution (Leviathan et al., 2023; Chen et al., 2023). Follow-up work improves efficiency by verifying multiple candidates per iteration. Specifically, tree-based methods verify token trees in parallel to exploit GPU parallelism (Miao et al., 2024; Chen et al., 2024; Wang et al., 2025). Systems such as EAGLE (Li et al., 2024a) use tree-style expansion to find longer verifiable prefixes, while other variants use multi-branch or multi-head drafting under exact acceptance rules (Cai et al., 2024). Since acceptance length depends on draft-target agreement, prior work improves draft quality via distillation (Zhou et al., 2023; Hu et al., 2025) and online updates (Liu et al., 2023).

**LLM Serving Under Memory Pressure.** LLM serving is often bottlenecked by the KV Cache footprint, which limits batch size and constrains serving throughput under long contexts and concur-

rent requests. vLLM reduces KV Cache fragmentation through paged memory management (Kwon et al., 2023). Beyond single-GPU serving, DistServe disaggregates prefill and decode to reduce interference and improve goodput (Zhong et al., 2024), while Splitwise separates prompt and token phases across machines to improve resource utilization (Patel et al., 2024). These systems show that accelerator memory is a scarce resource in practical LLM serving. Under realistic batch sizes and KV Cache pressure, dedicating target-side memory to a fixed colocated draft model can reduce the flexibility of serving and weaken the resource efficiency that these systems aim to improve.

**Distributed and Networked Speculative Decoding.** Disaggregating draft and target models turns speculative decoding into a distributed pipeline, where communication latency, asynchronous verification returns, and scheduling directly affect accelerator utilization. DSI accelerates a single request by orchestrating multiple target and drafter instances through speculation parallelism (Timor et al., 2024), while edge-oriented systems such as SpecEdge, place lightweight drafts on edge device and keep verification on the server to reduce serving cost or communication volume (Li et al., 2025; Park et al., 2025; Venkatesha et al., 2025; Gao et al., 2025). A detailed qualitative comparison is provided in Appendix J. Overall, once drafting and verification are disaggregated, performance depends not only on acceptance behavior but also on communication efficiency and scheduling policy.

# 3 System Model

We model StarSD as a shared draft service where one draft instance serves multiple target instances under asynchronous verification returns. The model separates per-round progress from per-round time to characterize draft idle gaps, the transition to fully-loaded execution, and the tradeoff between per-target throughput and system-wide throughput as the number of target instances increases.

## 3.1 Iteration Definitions

Let $\mathcal{M}_p$ and $\mathcal{M}_q$ denote the target and draft model types, and let $M_p^{(i)}$ and $M_q$ denote their deployed instances. StarSD uses one draft instance $M_q$ to serve $N$ target instances $\{M_p^{(i)}\}_{i=1}^{N}$. System iterations are indexed by $\gamma \in \{1, \ldots, \gamma_{\text{end}}\}$, where each iteration corresponds to one draft-side scheduling round and $\gamma_{\text{end}}$ is the total number of scheduling rounds in a decoding session. In the $\gamma$-th iteration, $M_q$ processes one speculative request from each active target instance in a work-conserving order. For each request, $M_q$ generates a speculative draft of depth $d$, and the corresponding target instance verifies the draft and returns the accepted prefix. The draft depth $d$ upper-bounds the number of accepted tokens per request in one round and determines both the draft workload and the verification workload.

## 3.2 Per-iteration Acceptance Length

We quantify per-iteration progress by the number of tokens accepted across all target instances.

**Acceptance probability.** We start from a token-level acceptance notion that captures the agreement between the draft model type $\mathcal{M}_q$ and the target model type $\mathcal{M}_p$. We consider the common multi-session serving case where all target instances are instantiated from the same target model type $\mathcal{M}_p$, while different indices $i$ correspond to different decoding sessions with different prefixes. Conditioned on a verified prefix $y$, let $p_i(\cdot \mid y)$ and $q(\cdot \mid y)$ denote the next-token distributions of target instance $M_p^{(i)}$ and draft instance $M_q$, respectively. We define the token-level acceptance probability at prefix $y$ as

$$\beta_i(y) \triangleq \sum_x \min\{p_i(x \mid y),\, q(x \mid y)\} = 1 - D_{\text{LK}}(p_i(\cdot \mid y),\, q(\cdot \mid y))\,. \tag{1}$$

where $D_{\text{LK}}(p, q) = \frac{1}{2}\sum_x |p(x) - q(x)|$ denotes the total variation distance between $p$ and $q$ (Leviathan et al., 2023). This quantity measures how likely a token proposed by $M_q$ is accepted by $M_p^{(i)}$ under the prefix $y$.

**Expected accept length.** We next lift token-level acceptance to iteration-level progress. Let $l_i(d) \in \{0, 1, \ldots, d\}$ denote the accept length of $M_p^{(i)}$ in one iteration, defined as the number of consecutive proposed tokens accepted by $M_p^{(i)}$. This random variable measures the realized progress of target instance $M_p^{(i)}$ in a single round. Directly modeling the full distribution of $l_i(d)$ is unnecessary for throughput analysis. It is enough to express its expectation through prefix acceptance events, since accepting at least $j$ tokens means that the first $j$ proposed tokens all pass verification. This gives the following tail-sum form as

**Lemma 3.1** (Tail-sum identity for bounded accept length)**.** *For any $l_i(d) \in \{0, 1, \ldots, d\}$, we have*

$$\mathbb{E}[l_i(d)] = \sum_{j=1}^{d} \Pr(l_i(d) \geq j)\,. \tag{2}$$

*Proof.* Since $l_i(d)$ is non-negative and bounded by $d$, it can be written as $l_i(d) = \sum_{j=1}^{d} \mathbf{1}\{l_i(d) \geq j\}$, where $\mathbf{1}\{\cdot\}$ is the indicator function. Taking the expectation on both sides and using the linearity of expectation gives $\mathbb{E}[l_i(d)] = \sum_{j=1}^{d} \mathbb{E}[\mathbf{1}\{l_i(d) \geq j\}] = \sum_{j=1}^{d} \Pr(l_i(d) \geq j)$. □

**Closed-form Expression.** To connect the tail-sum form to token-level acceptance, we follow the standard speculative decoding analytical simplification (Chen et al., 2023) and treat token acceptances within one iteration as i.i.d. Bernoulli trials with acceptance rate $\beta_i$ (Leviathan et al., 2023). Under this approximation, we have

$$\mathbb{E}[l_i(d)] \approx \sum_{j=1}^{d} \beta_i^j = \frac{\beta_i \left(1 - \beta_i^d\right)}{1 - \beta_i}. \tag{3}$$

This expression makes the depth tradeoff explicit. When $\beta_i < 1$, increasing the draft depth from $d$ to $d + 1$ adds only $\beta_i^{d+1}$ expected tokens, so the marginal progress decreases with depth. Aggregating across all active target instances gives the expected system progress in iteration $\gamma$ as

$$\mathbb{E}[l]_\gamma \triangleq \sum_{i=1}^{N} \mathbb{E}[l_i(d)] \approx \sum_{i=1}^{N} \frac{\beta_i \left(1 - \beta_i^d\right)}{1 - \beta_i}. \tag{4}$$

This term captures how much useful decoding progress StarSD obtains in one scheduling round, serving as the system progress term when interpreting throughput.

### 3.3 Per-iteration Time Decomposition

We now characterize the *time* component of throughput by characterizing the duration of a single iteration under concurrent service. In each iteration, $M_q$ serves $N$ depth-$d$ requests, one from each active target instance $M_p^{(i)}$. The service time for a single depth-$d$ request is

$$S(d) \triangleq d\, t_s, \tag{5}$$

where $t_s$ is the time for $M_q$ to generate one layer of draft tokens, which remains nearly constant due to parallel execution within a modern accelerator.

After $M_q$ finishes serving a request from target instance $M_p^{(i)}$, the next request from the same target can arrive only after communication and verification finish. We define this return time as

$$Z(d) \triangleq t_c + t_v, \tag{6}$$

where $t_c$ is the communication time and $t_v$ is the verification time of $M_p^{(i)}$ for the received draft.

We consider $M_q$ is work-conserving. When the request buffer is non-empty, $M_q$ continuously serves pending requests, e.g., in first-in-first-out order. Each target instance $M_p^{(i)}$ issues its next request only after completing verification of the previous draft. Under this execution model, the main source of draft-side idleness is the uncovered return time from target instances. The following lemma gives a sufficient condition under which this return time is fully hidden by serving other targets as

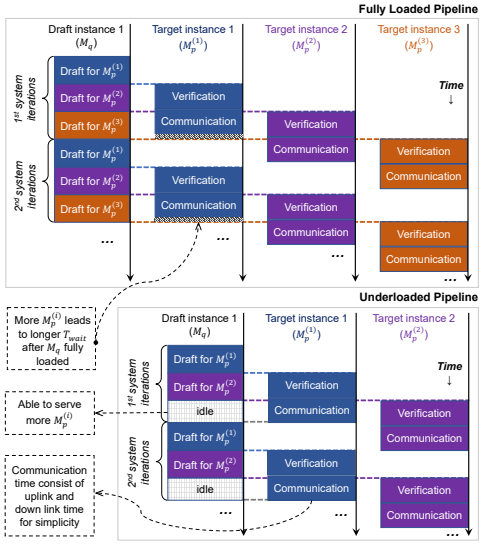



Figure 1: Execution cases of StarSD under different load conditions.

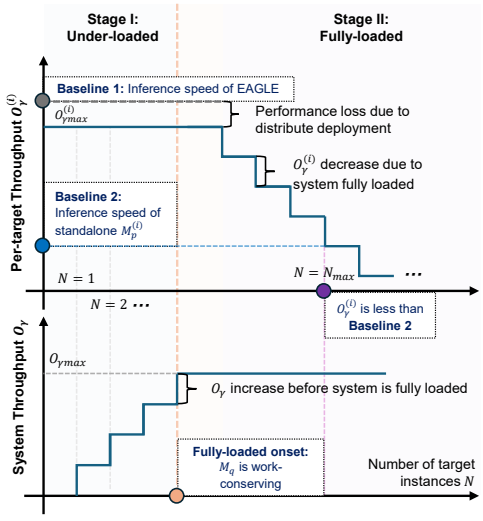



Figure 2: StarSD behavior when scaling the number of target instances.

**Lemma 3.2** (A sufficient non-idling condition for $M_q$)**.** *If*

$$Z(d) \leq (N-1)\, S(d), \tag{7}$$

*then a work-conserving $M_q$ does not idle while waiting for verification results from target instances.*

*Proof.* After $M_q$ serves a request from target instance $M_p^{(i)}$, the earliest time at which $M_p^{(i)}$ can issue its next request is $Z(d)$. During this interval, $M_q$ can serve one pending request from each of the remaining $N-1$ target instances, which takes $(N-1)S(d)$ time in total. If Eq. (7) is satisfied, the next request from $M_p^{(i)}$ is available no later than the time $M_q$ finishes serving the other targets. Therefore, the request buffer remains non-empty whenever $M_q$ is ready to continue service, and a work-conserving $M_q$ does not idle due to verification returns. □

### 3.4 Per-iteration Throughput

We now combine the progress term in Eq. (4) with the time decomposition above. Based on the non-idling condition in Lemma 3.2, the iteration time has two regimes, as illustrated in Fig. 1.

**Fully-loaded case.** When Eq. (7) holds, the return time $Z(d)$ of any active $M_p^{(i)}$ can be covered by the $M_q$-side service time spent on the other $N-1$ active $M_p^{(i)}$. Once the request buffer is populated, $M_q$ can continuously serve pending requests without draft-side idle gaps. The per-iteration time is

$$T_{\gamma_{\text{full}}} = NS(d), \tag{8}$$

since one iteration consists of $N$ depth-$d$ requests, each taking $S(d)$ time on $M_q$.

**Under-loaded case.** When Eq. (7) does not hold, the service time spent on the other $N-1$ target instances is not enough to hide the return time of the original target instance. The draft instance may then wait for the next request to arrive. We define the resulting idle gap as

$$T_{\text{idle}} \triangleq \max\{0,\ Z(d) - (N-1)S(d)\}\,. \tag{9}$$

Thus, the general per-iteration time can be expressed as

$$T_\gamma = NS(d) + T_{\text{idle}}. \tag{10}$$

Finally, the throughput of the $\gamma$-th iteration is

$$O_\gamma \triangleq \frac{\mathbb{E}[l]_\gamma}{T_\gamma}. \tag{11}$$

This formulation separates useful decoding progress from round duration. It shows whether throughput is mainly limited by acceptance length, captured by $\mathbb{E}[l]_\gamma$, or by time overheads, captured by $T_{\text{idle}}$ and $NS(d)$. This decomposition also implies the scaling pattern shown in Fig. 2. As $N$ increases, StarSD first hides draft idle gaps and then reaches a fully loaded draft regime, after which per-target throughput decreases. Section 4.2 analyzes this behavior in detail.

# 4 STARSD IMPLICATION

This section translates the analytical conditions into StarSD implication principles and examines how throughput scales as the number of target instances increases.

## 4.1 STARSD DESIGN

StarSD follows the execution structure of the system model through three design properties.

**Work-conserving draft execution.** To match Eqs. (8)-(10) under asynchronous returns, $M_q$ maintains a global request buffer and an outgoing buffer. A non-blocking service loop continuously selects pending requests when the request buffer is non-empty. Thus, any uncovered return time from target instances is exposed as the idle gap $T_{\text{idle}}$ in Eq. (9).

**Per-target state isolation.** To preserve the acceptance model in Eqs. (1) and (2) under concurrency, StarSD isolates all per-target state by `tag`. This state includes KV cache, verified prefixes, draft metadata, and routing information, preventing interference across target instances.

**Independent return paths.** To match the return-time decomposition in Eq. (6), StarSD assigns each target a unique `tag` and a dedicated port during a one-time handshake. These ports are reused during steady-state execution, allowing verification returns from different targets to arrive independently.

Figure 3 summarizes the StarSD execution pipeline, and Algorithm 2 in Appendix K gives its pseudocode implementation. Verified prefixes from multiple target instances enter a shared request buffer and are served by the draft instance in a work-conserving order. The drafted tokens are routed back to their corresponding targets for parallel verification, and the accepted prefixes re-enter the buffer for subsequent iterations. By decoupling request arrival, draft execution, and verification return, StarSD avoids cross-target head-of-line blocking and keeps the draft side active whenever pending work is available.

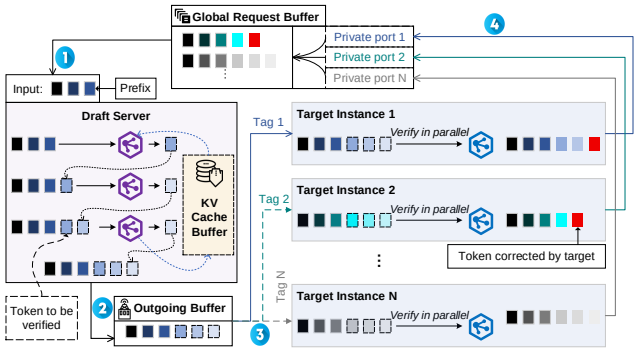



Figure 3: The Pipeline of StarSD.

## 4.2 INSIGHTS

While StarSD enables one-for-many serving, its throughput exhibits a trade-off as shown in Fig. 2.

*Stage I: Under-loaded.* When $N$ is small, $M_q$ is underloaded and the global request buffer is frequently empty. Adding more active target instances $M_p^{(i)}$ improves utilization by reducing draft-side idle gaps. In this regime, the per-$M_p^{(i)}$ throughput $O_\gamma^{(i)}$ stays approximately constant, while the system throughput $O_\gamma$ increases with $N$ because the uncovered return time is gradually amortized across more concurrent requests. This case corresponds to the violation of Eq. (7), where the uncovered return time appears as $T_{\text{idle}}$ in Eq. (9), and $T_\gamma$ follows Eq. (10).

*Stage II: Fully-loaded.* As $N$ increases, the idle gap vanishes, the global request buffer becomes persistently non-empty, and $M_q$ turns work-conserving. Beyond this point, increasing $N$ no longer increases the service capacity of $M_q$ but introduces queueing in the global request buffer. Consequently, the waiting time before a request is served increases $T_\gamma$, which reduces the per-$M_p^{(i)}$

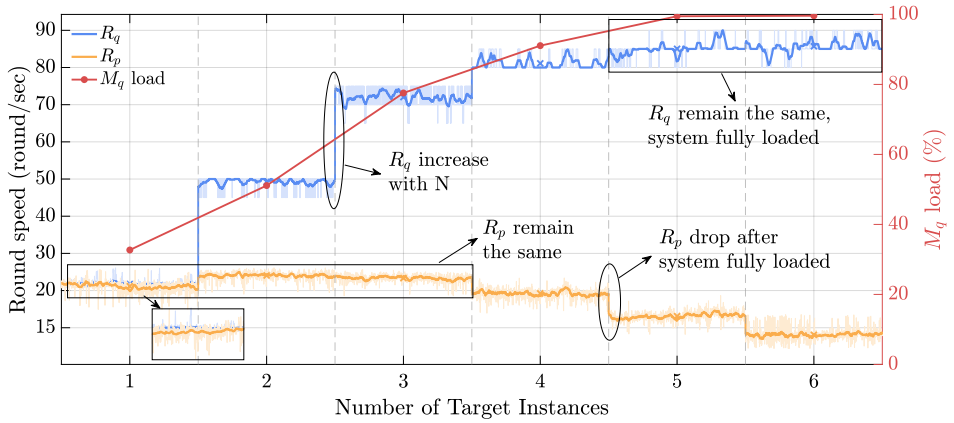



Figure 4: Round speed $R_q$ and $R_p$, and load of $M_q$ under varying numbers of target instances $N$.

throughput $O_\gamma^{(i)}$, while $O_\gamma$ saturates near its maximum. Formally, the full-load onset is the smallest $N$ such that Eq. (7) holds. Equivalently, for

$$N \geq N_{\text{full}} \triangleq \left\lceil \frac{Z(d)}{S(d)} \right\rceil + 1, \tag{12}$$

we have $T_{\text{idle}} = 0$ and $T_\gamma$ follows Eq. (8).

**Admission control under full load.** Full load does not mean that StarSD should stop admitting target instances. Additional targets can still improve system-level throughput as long as the resulting per-target throughput $O_\gamma^{(i)}$ remains above the throughput of standalone $M_p^{(i)}$. Let $N_{\max}$ denote the largest active set size that satisfies this condition. Beyond $N_{\max}$, additional concurrency makes each target slower than serving without the shared draft instance $M_q$. This observation motivates admission control at $M_q$, where new target requests are rejected when $N > N_{\max}$ to preserve per-target throughput above standalone inference.

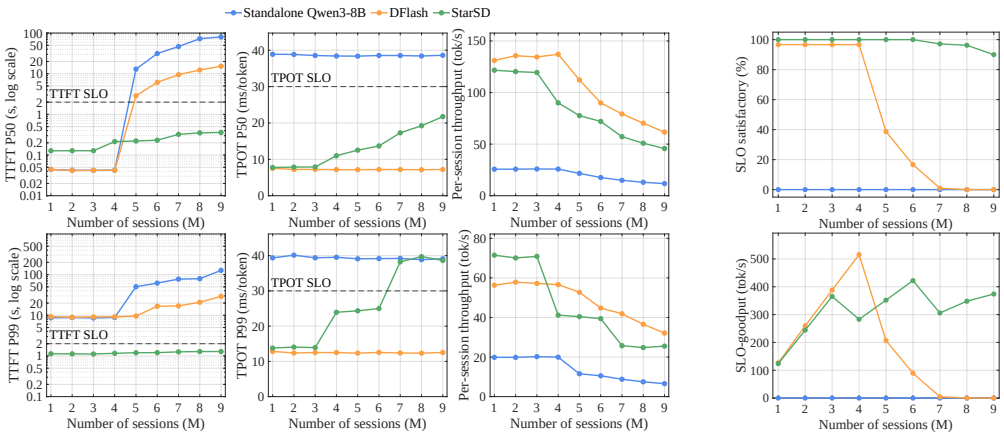
**Packing advantage in practice.** Beyond throughput, StarSD improves practical memory packing on accelerators. Colocating one target instance with its draft model forms fixed pairs that occupy large, indivisible memory blocks. Under a fixed accelerator memory budget, such coarse-grained pairing limits how many target instances can be placed concurrently and leaves fragmented memory regions. By decoupling drafting from verification and providing the draft model as a shared service, StarSD's finer-grained placement allows accelerator memory to be primarily filled with target instances, increasing target density and enabling either more concurrent replicas or longer context lengths under the same hardware budget.

## 5 Experiments

We evaluate StarSD along three question-driven dimensions. First, we test whether increasing the number of target instances produces the two-stage scaling behavior predicted by the system model and identifies an effective operating window for the shared-draft pipeline. Second, we test whether StarSD improves user-facing service quality under multi-session overload compared with standalone LLM serving and co-located speculative decoding under the same accelerator budget. Third, we test whether StarSD can serve as a backend-agnostic shared-draft runtime whose one-for-many scaling behavior persists across speculative decoding backends and workloads.

### 5.1 Scaling Effect and Operating Window

We first evaluate the scaling behavior of StarSD using EAGLE-2 (Li et al., 2024b) as the speculative decoding backend, Vicuna-7B as the target model, and MT-Bench (Zheng et al., 2023) as the workload. We vary the number of target instances $N$ served by a single draft instance $M_q$ and measure round speed, throughput, draft-side load, idle time, and queueing time. Fig. 4 reports the




(a) Request metrics. top: P50; bottom: P99. (b) SLO metrics.

Figure 5: Multi-session serving comparison with increasing number of concurrent session M.

round speeds $R_q$ and $R_p$, which are proportional to $O_\gamma$ and $O_\gamma^{(i)}$ as defined in Appendix F. Table 1 summarizes the corresponding system metrics.

When $N$ increases from 1 to 3, StarSD moves through the underloaded regime discussed in Section 4.2. The measured idle gap $T_{\text{idle}}$ decreases from 30.6 ms to 3.4 ms, and the load of $M_q$ increases from 33.3% to 75.4%. As verification and communication returns are increasingly hidden by draft-side service for other targets, $O_\gamma$ increases from 79.9 to 246.3 tokens/s. We also observe a draft-side burstiness effect. As $T_{\text{idle}}$ decreases, the measured per-layer draft time $t_s$ changes from 2.24 ms at $N = 1$ to 1.36 ms at $N = 2$ and $N = 3$, suggesting that idle gaps affect the execution state of $M_q$. A controlled analysis of this effect is provided in Appendix L. The main benefit window appears from $N = 3$ to $N = 5$. In this range, idle time is nearly eliminated while queueing remains moderate. At $N = 4$, $T_{\text{idle}}$ is only 0.6 ms and the load of $M_q$ reaches 91.4%. At $N = 5$, $T_{\text{idle}}$ becomes zero and the load of $M_q$ reaches 99.5%, matching the fully loaded condition in Eq. (12). Beyond this window, adding targets mainly increases $T_{\text{wait}}$ rather than $O_\gamma$. From $N = 5$ to $N = 6$, $O_\gamma$ increases only from 285.5 to 288.7 tokens/s, while $T_{\text{wait}}$ grows from 13.0 ms to 22.3 ms and $O_\gamma^{(i)}$ drops from 69.49 to 57.78 tokens/s. This confirms the operating-window prediction that StarSD should admit more targets only while the gain in draft utilization outweighs the queueing cost.

Table 1: System performance metrics as the number of target instances $N$ increases.

| Metric | EAGLE | StarSD with different numbers of target instances $N$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 |
| ***Throughput (tokens/s)*** | | | | | | | |
| $O_\gamma$ | 94.3 | 79.9 | 167.6 | 246.3 | 266.6 | 285.5 | **288.7** |
| $O_\gamma^{(i)}$ | 118.5 | 86.6 | **100.9** | 98.3 | 80.94 | 69.49 | 57.78 |
| ***Time breakdown (ms)*** | | | | | | | |
| $t_s$ | 1.78 | 2.24 | 1.36 | 1.36 | 1.42 | 1.46 | 1.46 |
| $T_{\text{wait}}$ | — | 0.01 | 0.50 | 0.60 | 3.90 | 13.0 | 22.3 |
| $T_{\text{idle}}$ | — | 30.6 | 9.8 | 3.4 | 0.6 | 0.0 | 0.0 |
| ***Round speed (round/s)*** | | | | | | | |
| $R_q$ | 26.8 | 20.90 | 49.25 | 72.26 | 81.54 | 85.52 | **85.84** |
| $R_p$ | 26.8 | 20.90 | **24.07** | 23.38 | 19.73 | 16.64 | 14.10 |
| ***Utilization (%)*** | | | | | | | |
| $M_q$ load | — | 33.3 | 50.3 | 75.4 | 91.4 | **99.5** | **99.6** |

## 5.2 Multi-session Serving Comparison

To answer the second evaluation question, we compare StarSD with standalone LLM serving and co-located DFlash (Chen et al., 2026), a recent state-of-the-art speculative decoding system, under the same 4-GPU budget on a single 4×RTX 4090 server. The standalone baseline deploys four Qwen3-8B (Team, 2025) instances without speculative decoding. Co-located DFlash deploys four independent speculative decoding instances, each hosting both the draft and target models. StarSD uses DFlash as its backend and deploys one shared draft instance with three Qwen3-8B target instances. Thus, all methods use the same number of GPUs, but allocate them differently. We vary the number of concurrent sessions from $M = 1$ to $M = 9$. Each session runs the full AIME25 (Zhang & Math-AI, 2025) dataset and submits prompts sequentially, where the next prompt is issued only after the previous inference finishes. All methods use a maximum generation length of 2048 tokens and the same First-in-First-out (FIFO) request buffer. We report request-level latency and service

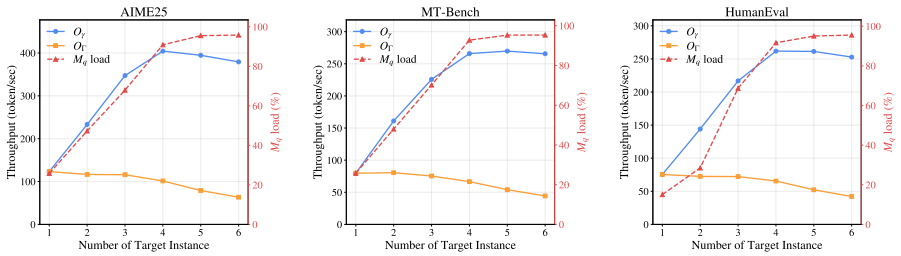



Figure 6: Throughput and $M_q$ load as the number of target instances varies, using DFlash as the backend across AIME25, MT-Bench, and HumanEval.

quality metrics, including Time to First Token (TTFT), Time per Output Token (TPOT), queueing delay, SLO satisfaction rate, and SLO-goodput, with formal definitions in Appendix D. Appendix E provides additional measurements that isolate the performance difference between co-located deployment and StarSD.

Fig. 5 shows that StarSD trades low-load peak throughput for better overload behavior. Since one GPU is reserved for the shared draft instance, StarSD has fewer target instances than co-located DFlash, which increases TPOT when many sessions share the draft service. However, when the number of sessions exceeds the number of available instances, StarSD avoids the severe queueing-induced TTFT growth observed in co-located DFlash and maintains higher SLO satisfaction. As a result, StarSD achieves higher SLO-goodput in the overloaded regime, indicating that it delivers more tokens while still satisfying user-facing latency constraints. This result supports StarSD for multi-session serving, where stable service quality is more important than low-load peak throughput.

### 5.3 Generality Across Backends and Datasets

To answer the third evaluation question, we examine whether the shared-draft runtime of StarSD depends on a specific speculative decoding backend or workload. We instantiate the same one-for-many runtime with DFlash and vary the number of target instances from $N = 1$ to $N = 6$. We evaluate three workloads, including MT-Bench for conversation, AIME25 for reasoning, and HumanEval (Chen et al., 2021) for code generation. Detailed per-dataset measurements and trace-level delay breakdowns are provided in Appendix H.

Fig. 6 shows that StarSD increases system throughput $O_\gamma$ and the load of $M_q$ as the number of target instances grows across all three workloads. This trend confirms that the shared-draft pipeline can improve draft-side utilization beyond the EAGLE-2 setting as in Section 5.1. The peak-throughput point, however, differs across datasets because workload-specific prefixes change acceptance behavior and therefore change how quickly requests return to the draft side. Comparing Fig. 4 with Fig. 6 further shows backend-level differences. EAGLE-2 and DFlash reach the fully loaded regime at different values of $N$, since their draft costs and verification patterns differ. These results show that StarSD is a backend-agnostic shared-draft runtime, while its best operating point should be selected according to the backend and workload.

## 6 Conclusion

StarSD is a one-for-many speculative decoding framework that allows a single draft model to efficiently serve multiple target instances. By keeping the draft side work-conserving under asynchronous verification, StarSD eliminates idle gaps that limit the efficiency in distributed deployments. Experiments across heterogeneous settings show that this design improves utilization and throughput without modifying model weights or generation semantics, making speculative decoding more practical under realistic memory and deployment constraints. Beyond raw throughput, the multi-session serving evaluation shows that StarSD provides stronger robustness under overloaded workloads by maintaining higher SLO satisfaction and SLO-goodput, which indicates its advantage for user-facing serving scenarios where stable latency is critical. Additional experiments across datasets and speculative decoding backends further show that the one-for-many scaling behavior is a general system-level property of shared-draft speculative decoding.

## AI USE STATEMENT

Generative AI tools were used to assist with language editing and to improve the clarity and readability of the manuscript. They were not used to develop the research methodology, formulate mathematical claims, design experiments, implement the proposed method, process experimental data, or interpret the experimental results. All AI-assisted edits were reviewed and revised by the authors, who take full responsibility for the final content of the paper.

## REPRODUCIBILITY STATEMENT

We provide the implementation and experimental details needed to reproduce the results of StarSD. The shared-draft execution model, analytical assumptions, and operating regimes are described in Section 3, while the main experimental methodology, workloads, hardware configurations, baselines, and evaluation settings are reported in Section 5. The appendix further specifies the request-level evaluation metrics in Appendix D, the StarSD runtime and state-management implementation in Appendix K, and additional cross-machine and large-model experimental configurations in Appendix I. Reproducibility details for the draft-side timing and GPU-frequency experiments are provided in Appendices L and M, and the models, speculative decoding backends, evaluation codebases, benchmarks, and their licenses are summarized in Appendix P. We also provide anonymized source code and experimental scripts as supplementary material to facilitate reproduction of the reported results.

## REFERENCES

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. *arXiv preprint arXiv:2401.10774*, 2024.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Jian Chen, Yesheng Liang, and Zhijian Liu. DFlash: Block Diffusion for Flash Speculative Decoding. *arXiv preprint arXiv:2602.06036*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable and robust speculative decoding. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 129531–129563. Curran Associates, Inc., 2024.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive transformer. *arXiv preprint arXiv:1910.10073*, 2019.

Luyao Gao, Jianchun Liu, Hongli Xu, Xichong Zhang, Yunming Liao, and Liusheng Huang. Collaborative speculative inference for efficient LLM inference serving. *arXiv e-prints*, art. arXiv:2503.10325, March 2025.

Yuezhou Hu, Jiaxin Guo, Xinyu Feng, and Tuo Zhao. Adaspec: Selective knowledge distillation for efficient speculative decoders. *arXiv preprint arXiv:2510.19779*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286, 23–29 Jul 2023.

Jiaxi Li, Yue Zhu, Eun Kyung Lee, and Klara Nahrstedt. Revisiting disaggregated large language model serving for performance and energy implications. *arXiv e-prints*, art. arXiv:2601.08833, November 2025.

Xiangchen Li, Dimitrios Spatharakis, Saeid Ghafouri, Jiakun Fan, Hans Vandierendonck, Deepu John, Bo Ji, and Dimitrios S Nikolopoulos. Sled: A speculative LLM decoding framework for efficient edge serving. In *Proceedings of the Tenth ACM/IEEE Symposium on Edge Computing*, pp. 1–8, 2025.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-2: Faster inference of language models with dynamic draft trees. *arXiv preprint arXiv:2406.16858*, 2024b.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Online speculative decoding. *arXiv preprint arXiv:2310.07177*, 2023.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 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*, pp. 932–949, 2024.

Jinwoo Park, Seunggeun Cho, and Dongsu Han. Specedge: Scalable edge-assisted serving framework for interactive LLMs. *arXiv preprint arXiv:2505.17052*, 2025.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture*, pp. 118–132. IEEE, 2024.

David Schall, Andreas Sandberg, and Boris Grot. Warming up a cold front-end with ignite. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pp. 254267, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614258. URL https://doi.org/10.1145/3613424.3614258.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 35:17456–17472, 2022.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single GPU. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Nadav Timor, Jonathan Mamou, Daniel Korat, Moshe Berchansky, Oren Pereg, Moshe Wasserblat, Tomer Galanti, Michal Gordon, and David Harel. Distributed speculative inference (DSI): Speculation parallelism for provably faster lossless language model inference. *arXiv preprint arXiv:2405.14105*, 2024.

Yeshwanth Venkatesha, Souvik Kundu, and Priyadarshini Panda. Fast and cost-effective speculative edge-cloud decoding with early exits. *arXiv e-prints*, art. arXiv:2505.21594, May 2025.

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, 02 2025. ISSN 2307-387X.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging LLM-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation*, pp. 193–210, 2024.

Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. Distillspec: Improving speculative decoding via knowledge distillation. *arXiv preprint arXiv:2310.08461*, 2023.

# A Overview of Supplementary Material

This appendix provides additional motivation, analysis, implementation details, deployment studies, and experimental results that complement the main paper. The supplementary material is organized into five parts.

1. **Motivation and resource implications.** We further motivate the one-for-many design from practical deployment constraints and quantify the target-side memory capacity recovered by draft-target decoupling in Appendices B and C.
2. **Metrics, analytical justification, and speculative decoding details.** We define the request-level serving metrics used in our evaluation, quantify the overhead of distributed execution in the $N = 1$ setting, provide additional mathematical justification for the draft- and target-side round-speed metrics, and summarize the EAGLE-based speculative decoding procedure in Appendices D to G.
3. **Extended evaluation and system comparisons.** We provide complete cross-dataset generality results, additional cross-machine and large-model scaling measurements, an analysis of how communication latency affects the predicted saturation point, and a comparison with representative distributed speculative decoding systems, including an end-to-end comparison with SpecEdge under identical open-loop arrivals and equal accelerator budgets, in Appendices H to J.
4. **Implementation and draft-side execution behavior.** We describe the runtime implementation of StarSD, including connection establishment, asynchronous request handling, communication-compute decoupling, and tag-indexed state management. We also provide controlled measurements of draft-side execution dynamics, GPU-frequency effects, warm-up behavior, and inference-time variation in Appendices K to M.
5. **Deployment, limitations, and broader considerations.** We illustrate representative intra-node, inter-node, and edge-server deployment topologies, discuss the limitations of the current design, summarize the licenses and terms of the existing assets used in our evaluation, and discuss the broader impact and security implications of StarSD in Appendices N to Q.

# B Motivation

Recent advances in speculative decoding have largely focused on improving the draft model by better aligning its token distribution with the target model. However, most existing frameworks implicitly assume that the draft instance $M_q$ and the target instance $M_p$ can be co-located on a single accelerator, e.g., a GPU. In practical deployments, this assumption often does not hold. Many real systems operate under heterogeneous and resource-constrained settings, including edge clusters and mid-scale cloud servers, where accelerator memory is limited and shared by multiple services. The weights may barely fit, but the remaining memory must still accommodate KV cache and runtime buffers. In practice, this sharply reduces the KV cache budget, hence affect the context length, which is especially problematic in multi-tenant deployments where one GPU also serves other inference tasks. Even with compression or quantization, co-locating $M_q$ and $M_p$ can remain impractical or introduce nontrivial accuracy, engineering, and operational costs.

A natural response is to disaggregate the deployed instances $M_q$ and $M_p$ across different accelerators, e.g., GPU. Distributed speculative decoding has been explored in prior work, but most existing designs do not address a core inefficiency inherent to the draft-then-verify workflow. When the target instance $M_p$ is verifying draft tokens, the accelerator hosting the draft instance $M_q$ is idle; when $M_q$ is drafting, the $M_p$ accelerator is underutilized. This alternating execution pattern reduces effective hardware utilization and makes $M_q$ execution bursty, with short compute phases separated by idle gaps induced by verification latency and inter-accelerator communication. These idle gaps have system-level consequences that are also largely overlooked in prior work. Even when the decoding depth is unchanged, bursty execution prevents $M_q$ from sustaining a stable high-throughput execution state and may trigger repeated per-iteration transients (e.g., kernel reactivation and cache coldness), leading to increased draft-step latency and higher variance across decoding iterations.

More fundamentally, the generated output is ultimately dictated by the target instance $M_p$, while $M_q$ serves as auxiliary compute overhead. From a system perspective, this overhead should be amortized across multiple target instances rather than dedicated to a single $M_p$. These observations motivate a one-for-many design in which a single $M_q$ multiplexes requests from multiple $M_p^{(i)}$. Such a design allows verification responses to arrive more continuously, reduces idle gaps on the draft accelerator, and improves system-level utilization.

## C Memory Saving from Draft-Target Decoupling

Target-side memory pressure by removing the draft model's resident weights and draft-side KV cache from the target GPU. Let $M$ be the available GPU memory, $W_B$ and $W_D$ be the target and draft model weights, and $k_B$ and $k_D$ be the per-token KV Cache footprints of the target and draft models. If only weights and KV cache are considered, the maximum supported context length without and with draft co-location can be estimated as

$$L_{\text{base}} \approx \frac{M - W_B}{k_B}, \qquad L_{\text{full}} \approx \frac{M - W_B - W_D}{k_B + k_D}. \tag{C.1}$$

Here $L_{\text{base}}$ denotes the target-only setting, while $L_{\text{full}}$ denotes the co-located target-plus-draft setting.

For illustration, we instantiate this estimate with a Vicuna-7B target and an EAGLE-Vicuna draft on an RTX 4090 (24GB). We use $W_B \approx 16$GB and $W_D \approx 1.48$GB. The measured KV Cache footprints are $k_B \approx 147$KB per token for the target and $k_D \approx 4$KB per token for the draft. This gives

$$L_{\text{base}} \approx \frac{24\text{GB} - 16\text{GB}}{147\text{KB}} \approx 54\text{K}, \tag{C.2}$$

and

$$L_{\text{full}} \approx \frac{24\text{GB} - 16\text{GB} - 1.48\text{GB}}{147\text{KB} + 4\text{KB}} \approx 43\text{K}. \tag{C.3}$$

Thus, separating the draft from the target side recovers approximately

$$\Delta L = L_{\text{base}} - L_{\text{full}} \approx 11\text{K} \tag{C.4}$$

tokens of theoretical KV Cache capacity, corresponding to roughly a $26\%$ increase over the co-located setting.

This recovered capacity can be decomposed into the contribution from removing draft weights and the contribution from removing draft KV cache:

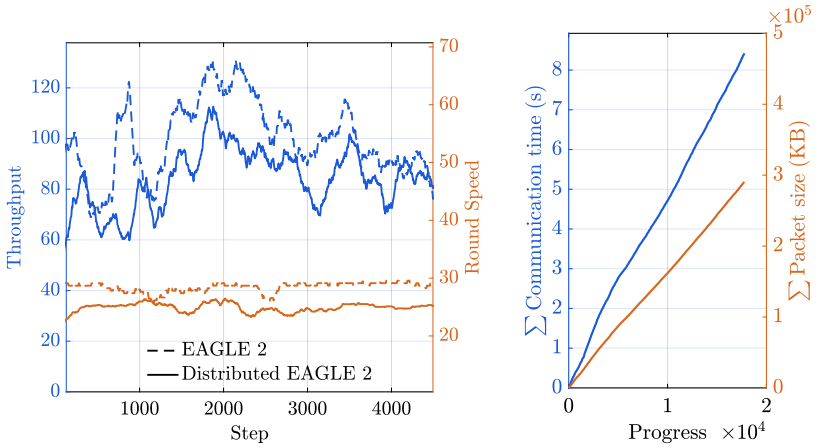
$$\Delta L \approx \frac{W_D}{k_B} + \frac{k_D}{k_B} L_{\text{full}} \approx \frac{1.48\text{GB}}{147\text{KB}} + \frac{4\text{KB}}{147\text{KB}} \times 43\text{K} \approx 11\text{K}. \tag{C.5}$$

This decomposition shows that most of the recovered context capacity comes from removing the draft model weights from the target GPU, while the draft KV cache contributes additional savings.

This estimate is idealized and does not include CUDA runtime memory, allocator fragmentation, temporary activation buffers, attention workspaces, or framework-specific reserved memory. In practice, these overheads reduce the absolute context length, but the same principle remains: draft-target decoupling frees target-side memory that can be used for longer contexts, larger KV cache, or higher serving concurrency. The numerical values above are specific to this target-draft pair and backend implementation. The exact memory saving depends on the speculative decoding backend, the target and draft model sizes, their KV Cache footprints, precision formats, and runtime memory overheads.

## D Evaluation Metrics

For each request $r$, let $t_{\text{sub}}(r)$ be the time when the prompt enters the serving system, $t_{\text{start}}(r)$ the time when an instance starts processing it, $t_{\text{first}}(r)$ the time when the first output token becomes




(a) Performance Comparison. (b) Communication Cost.

Figure 7: Throughput loss due to distributed deployment of EAGLE-2 and communication overhead.

available, $t_{\text{fin}}(r)$ the finish time, and $L_{\text{out}}(r)$ the number of generated output tokens. We define

$$\text{TTFT}(r) = t_{\text{first}}(r) - t_{\text{sub}}(r), \tag{D.1}$$

$$\text{TPOT}(r) = \frac{t_{\text{fin}}(r) - t_{\text{first}}(r)}{\max(1, L_{\text{out}}(r) - 1)}, \tag{D.2}$$

$$T_{\text{queue}}(r) = t_{\text{start}}(r) - t_{\text{sub}}(r). \tag{D.3}$$

Here, TTFT includes queueing delay and the time to produce the first token, while TPOT measures the average generation time per output token after the first token.

We use an interactive serving SLO where a request is considered SLO-satisfied if

$$\text{TTFT}(r) \leq 2.0\,\text{s} \quad \wedge \quad \text{TPOT}(r) \leq 30\,\text{ms/token}. \tag{D.4}$$

Given a request set $\mathcal{R}$, the SLO satisfaction rate is

$$\text{SLO satisfaction rate} = \frac{|\{r \in \mathcal{R} : \text{SLO-satisfied}(r)\}|}{|\mathcal{R}|}. \tag{D.5}$$

We further define SLO-goodput as the number of output tokens from SLO-satisfied requests per experiment wall-clock time:

$$\text{SLO-goodput} = \frac{\sum_{r \in \mathcal{R}:\text{SLO-satisfied}(r)} L_{\text{out}}(r)}{T_{\text{exp}}}, \tag{D.6}$$

where $T_{\text{exp}}$ is the total wall-clock time of the experiment.

## E Distributed Behavior of StarSD

Since StarSD does not modify model weights or the speculative decoding algorithm, performance differences arise solely from system effects, namely communication overhead introduced by distribution and potential gains from parallel execution. Thus, we evaluate StarSD on the full MT-Bench benchmark (Zheng et al., 2023) using EAGLE-2 (Li et al., 2024b) as a representative backend. To isolate the cost of distribution, we consider the $N = 1$ case, where the draft instance $M_q$ and the target instance $M_p$ are placed on different accelerators. In this configuration, StarSD reduces to a distributed deployment of EAGLE-2, enabling a direct measurement of cross-device communication overhead.

As shown in Fig. 7(*a*), we report throughput from two perspectives, (1) system throughput $O_\gamma$, which fluctuates noticeably across MT-Bench due to prompt-dependent acceptance lengths, and (2) the round speed $R_q$ of $M_q$ which is proportional to $O_\gamma$ and prove in F.1, it provides a smoother

perspective to observe performance. To reflect the cost of distributed deployment, we measure end-to-end serving performance in the deployed setting, so the experiment results naturally include cross-device overheads such as communication latency shown in Fig. 7(*b*), message serialization and deserialization, and GPU-CPU memory transfers. Our implementation is deployment-agnostic and supports multi-accelerator execution either within a single machine or across machines. Accordingly, we use socket-based communication instead of hardware-specific backends, e.g., NVLink, trading some performance for portability. This observation aligns with recent findings in distributed LLM serving that cross-accelerator data movement can incur non-trivial overhead and lead to a measurable performance drop under equivalent accelerator resources (Li et al., 2025).

# F Mathematical Justification

## F.1 Metric Justification of Draft and Verify Round Speed

We consider a fixed dataset $\mathcal{D}$, a fixed decoding configuration, and a fixed model pair $(M_p, M_q)$ with depth $d$. Across deployments, we keep the model weights and the decoding algorithm unchanged. Hence, for any prefix $y$ encountered during decoding on $\mathcal{D}$, the next-token distributions $p(\cdot \mid y)$ and $q(\cdot \mid y)$ are invariant up to numerical nondeterminism, and the induced acceptance behavior is invariant in expectation. Under the standard approximation, the expected accepted length per draft proposal

$$\bar{\ell} \triangleq \mathbb{E}[l(d)], \tag{F.1}$$

is deployment-invariant.

**Round speeds.** Let $R_q$ denote the rate per unit time at which the draft model executes the token-generation routine that produces a depth-$d$ proposal, each such execution yields an accepted length $l(d)$ in the subsequent verification step. Let $R_p$ denote the rate per unit time at which a target instance executes a verification routine. Both $R_q$ and $R_p$ are measured in the steady state.

**Throughput as tokens per unit time.** Recall that the system throughput (11) from the draft-side view is

$$O_\gamma \triangleq \frac{\mathbb{E}[l]_\gamma}{T_\gamma}, \tag{F.2}$$

where $\mathbb{E}[l]_\gamma$ is the expected number of accepted tokens contributed in an interval of duration $T_\gamma$. From the target-side view, we similarly define the per-target throughput for instance $i$ as

$$O_\gamma^{(i)} \triangleq \frac{\mathbb{E}[l]_\gamma^{(i)}}{T_\gamma}. \tag{F.3}$$

**Positive correlation with $R_q$.** Over any time window of length $T_\gamma$, the number of draft-generation executions is approximately $R_q T_\gamma$ in expectation. Each execution contributes an accepted length with expectation $\bar{\ell}$. Therefore,

$$\mathbb{E}[l]_\gamma \;\approx\; (R_q T_\gamma)\,\bar{\ell}, \qquad O_\gamma \;=\; \frac{\mathbb{E}[l]_\gamma}{T_\gamma} \;\approx\; \bar{\ell}\, R_q. \tag{F.4}$$

Since $\bar{\ell} > 0$ under fixed $\mathcal{D}$ and decoding setup, $O_\gamma$ is monotone increasing in $R_q$, i.e., $R_q$ is positively correlated with $O_\gamma$.

**Positive correlation with $R_p$.** Fix a target instance $M_p^{(i)}$. Over a window of length $T_\gamma$, the expected number of verification executions by this instance is approximately $R_p T_\gamma$. Each verification round finalizes one proposal and advances the decoded sequence by an accepted length with expectation $\bar{\ell}$. Thus,

$$\mathbb{E}[l]_\gamma^{(i)} \;\approx\; (R_p T_\gamma)\,\bar{\ell}, \qquad O_\gamma^{(i)} \;=\; \frac{\mathbb{E}[l]_\gamma^{(i)}}{T_\gamma} \;\approx\; \bar{\ell}\, R_p, \tag{F.5}$$

which implies that $O_\gamma^{(i)}$ is monotone increasing in $R_p$ and hence $R_p$ is positively correlated with the throughput of the $i$-th target instance.

**Bottleneck regime.** In practice, draft and verification must be matched, so the realized throughput is governed by the slower side. This is consistent with the upper-bound intuition

$$O_\gamma \lesssim \bar{\ell} \min\{R_q, R_p\}, \tag{F.6}$$

and increasing either $R_q$ or $R_p$ cannot decrease throughput.

**Special case: 1-for-1 in StarSD.** When there is only one target instance, i.e., $N = 1$, the system has a single verification stream, so the per-target throughput coincides with the system throughput

$$O_\gamma^{(i)} = O_\gamma. \tag{F.7}$$

Moreover, the execution forms a tightly coupled loop, where each draft-generation round on $M_q$ produces exactly one proposal, and each proposal triggers exactly one verification round on $M_p$. Therefore, in steady state, the number of draft rounds and verification rounds are equal over any long horizon, implying

$$R_q = R_p. \tag{F.8}$$

Combining (F.5) and (F.7), both metrics induce the same monotone proxy for throughput in the 1-for-1 setting:

$$O_\gamma = \bar{\ell}\, R_q = \bar{\ell}\, R_p. \tag{F.9}$$

## G EAGLE ALGORITHM DETAILS

**Setting and notation.** Let $y$ denote the *verified prefix* at the beginning of one speculative iteration. EAGLE uses a draft model $M_q$ to construct a $k$-ary proposal tree of depth $d$, and a target model $M_p$ to verify candidate paths. A candidate path is represented as $\texttt{tree_path} = [x_1, \ldots, x_d]$. During tree expansion, the draft model $M_q$ produces next-token distributions at each depth. We denote the distribution at depth $t$ by $q_t(\cdot)$. The correction step uses the corresponding draft probability $q_{l+1}(x)$ in $\texttt{norm}(\max(0, p_{l+1}(x) - q_{l+1}(x)))$. During verification, running $M_p$ along the prefixes induced by a path produces target-side distributions $p_1(\cdot), \ldots, p_{d+1}(\cdot)$.

**Initialization.** EAGLE starts each iteration from a verified prefix $y$. In the first iteration, $M_q$ samples a root token $x$ and an accompanying `hidden_state` from the distribution conditioned on $y$ provided by $M_p$, in later iterations, it anchors the new proposal tree at the last verified token $y[-1]$.

**Tree expansion with caching.** The proposal tree is expanded layer-by-layer to depth $d$ with top-$k$ branching. Before expanding each layer, the algorithm initializes the draft KV cache from $y$ or incrementally updates it following the currently selected `target_path`. The expansion routine EXPANDLAYER is executed in parallel over the current tree frontier, and the draft probabilities $q_t(\cdot)$ needed by the correction step are stored in the tree metadata.

**Verification and accept-length computation.** For each candidate path $\pi = [x_1, \ldots, x_d]$, the target model $M_p$ is evaluated on the sequence of induced prefixes $y$, $y{+}x_1, \ldots, y{+}[x_1, \ldots, x_d]$ to produce $p_1(\cdot), \ldots, p_{d+1}(\cdot)$. The algorithm samples $r_1, \ldots, r_d$ and gets the accepted length $l$ using the test in Alg. 1.

**Correction and best-path selection.** If $l < d$, the algorithm forms a corrected distribution $p'(x) \leftarrow \texttt{norm}(\max(0,\, p_{l+1}(x) - q_{l+1}(x)))$ and appends one corrected token after the accepted prefix. Otherwise, it keeps the accepted prefix of length $l$ (which equals $d$ in this case). Finally, it selects the path with the largest accepted length and returns $y + \texttt{target_path}$ along with the updated `hidden_state` of the selected path.

**Algorithm 1** EAGLE Speculative Decoding

**Input:** $M_q$, $M_p$, verified prefix $y$, depth $d$, top-$k$ $k$
**Output:** $y +$ `target_path` and updated `hidden_state`
`accept_length` $\leftarrow 0$ *// best accepted length among all verified paths*
`target_path` $\leftarrow [\,]$ *// path selected for this iteration*

**if** `First Iteration` **then**
  *// Initialize root token and hidden state to start* $M_q$
  $x$, `hidden_state` $\sim p(y)$
  `tree` $\leftarrow x$ *// root of the proposal tree*
**else**
  *// Use the last verified token as the tree root*
  `tree` $\leftarrow y[-1]$
**end if**

*// Build a depth-*$d$ *proposal tree with branching factor* $k$
**for** $t = 1$ to $d$ **do**
  *// KV Cache maintenance for draft expansion (prefix init or incremental update)*
  **if** `kv_cache` is empty **then**
    `kv_cache` $\leftarrow$ INITKV($y$, `hidden_state`) *// cache corresponds to the verified prefix*
  **else**
    `kv_cache` $\leftarrow$ UPDATEKV(`target_path`, `kv_cache`, `hidden_state`) *// follow the currently selected path*
  **end if**
  *// Expand one tree layer in parallel using* $M_q$ *(top-*$k$ *per node)*
  `tree` $\leftarrow$ EXPANDLAYER(`tree`, `hidden_state`, $M_q$, $k$)
**end for**

*// Verify candidate paths in parallel*
**for** $n = 1$ to $k^d$ **do**
  `tree_path` $\leftarrow$ `tree`[$n$]
  *//* `tree_path` *corresponds to* $\pi = [x_1, \ldots, x_d]$
  *// Run* $M_p$ *along prefixes induced by the path, can be executed in parallel*
  $p_1(x), \ldots, p_{d+1}(x) \leftarrow M_p(y), \ldots, M_p(y + [x_1, \ldots, x_d])$
  *// Determine accept length on this path via stochastic test*
  $r_1 \sim U(0,1), \ldots, r_d \sim U(0,1)$
  $l \leftarrow \min\left(\{i - 1 \mid 1 \le i \le d,\ r_i > p_i(x)\} \cup \{d\}\right)$
  *// Prepare continuation (either accept full prefix, or apply one-step correction)*
  $p'(x) \leftarrow p_{l+1}(x)$
  **if** $l < d$ **then**
    $p'(x) \leftarrow$ `norm`$(\max(0,\ p_{l+1}(x) - q_{l+1}(x)))$
    `tree_path` $\leftarrow$ `tree_path`[:$l$].`append`($t \sim p'(x)$) *// accept first* $l$ *tokens, then sample one corrected token*
  **else**
    `tree_path` $\leftarrow$ `tree_path`[:$l$] *// accept the whole depth-*$d$ *path*
  **end if**
  *// Select the path with the largest accept length*
  **if** $l >$ `accept_length` **then**
    `accept_length` $\leftarrow l$
    `target_path` $\leftarrow$ `tree_path`
  **end if**
**end for**

**return** $y +$ `target_path`, `hidden_state`

## H SUPPLEMENTARY MATERIAL OF STARSD GENERALITY

**Supplementary analysis.** Table 2 reports the complete numerical results for the DFlash-based generality study. The additional measurements show that all three workloads reach a near work-conserving draft regime around $M = 4$, where $M_q$ load exceeds 90%. Beyond this point, the dominant delay shifts from draft-side idle time to queueing wait time: $t_{\text{idle}}$ drops to below 1 ms at large $M$, while $t_{\text{wait}}$ increases rapidly as requests accumulate in the global buffer. This explains why $O_\gamma$ saturates while $O_\Gamma$ decreases after the operating point.

**Trace-level evidence.** Figure 8 further provides per-step traces of $t_c$, $t_{\text{wait}}$, and $t_{\text{idle}}$ on AIME25. The traces show that $t_c$ remains relatively small and stable, whereas the main varying component is the exchange between idle time and queueing wait time. Thus, the supplementary results attribute the observed scaling behavior primarily to pipeline occupancy and buffer contention.

Table 2: Scaling behavior of StarSD using DFlash as backend.

| Dataset | Metric | Number of concurrent sessions $N$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 |
| **AIME25** | $O_\gamma$ (tok/s) | 123.3 | 233.2 | 347.2 | 404.4 | 394.6 | 379.2 |
| | $O_\Gamma$ (tok/s) | 123.3 | 116.6 | 115.9 | 101.4 | 79.1 | 63.4 |
| | $dt_s$ (ms) | 8.78 | 9.26 | 9.02 | 10.42 | 11.33 | 11.91 |
| | $t_{\text{wait}}$ (ms) | 0.11 | 0.83 | 1.29 | 4.09 | 15.10 | 24.84 |
| | $t_{\text{idle}}$ (ms) | 30.31 | 11.42 | 4.82 | 1.52 | 0.96 | 0.86 |
| | $M_q$ load (%) | 25.8 | 47.3 | 67.9 | 90.9 | 95.5 | 95.8 |
| **MT-Bench** | $O_\gamma$ (tok/s) | 79.7 | 161.0 | 225.7 | 265.9 | 269.8 | 265.6 |
| | $O_\Gamma$ (tok/s) | 79.7 | 80.5 | 75.4 | 66.6 | 54.0 | 44.3 |
| | $dt_s$ (ms) | 8.81 | 8.65 | 9.26 | 10.43 | 10.58 | 10.76 |
| | $t_{\text{wait}}$ (ms) | 0.10 | 0.32 | 1.32 | 4.27 | 14.23 | 23.61 |
| | $t_{\text{idle}}$ (ms) | 29.89 | 10.48 | 4.36 | 1.11 | 0.80 | 0.81 |
| | $M_q$ load (%) | 25.8 | 48.0 | 70.2 | 92.8 | 95.3 | 95.3 |
| **HumanEval** | $O_\gamma$ (tok/s) | 75.5 | 144.1 | 216.8 | 261.8 | 261.3 | 252.8 |
| | $O_\Gamma$ (tok/s) | 75.5 | 72.5 | 72.3 | 65.5 | 52.3 | 42.2 |
| | $dt_s$ (ms) | 5.50 | 5.74 | 9.29 | 10.33 | 10.75 | 11.17 |
| | $t_{\text{wait}}$ (ms) | 0.11 | 0.42 | 1.43 | 4.02 | 14.37 | 24.55 |
| | $t_{\text{idle}}$ (ms) | 34.84 | 15.27 | 4.71 | 1.25 | 0.85 | 0.83 |
| | $M_q$ load (%) | 15.0 | 28.5 | 68.7 | 91.6 | 95.1 | 95.5 |

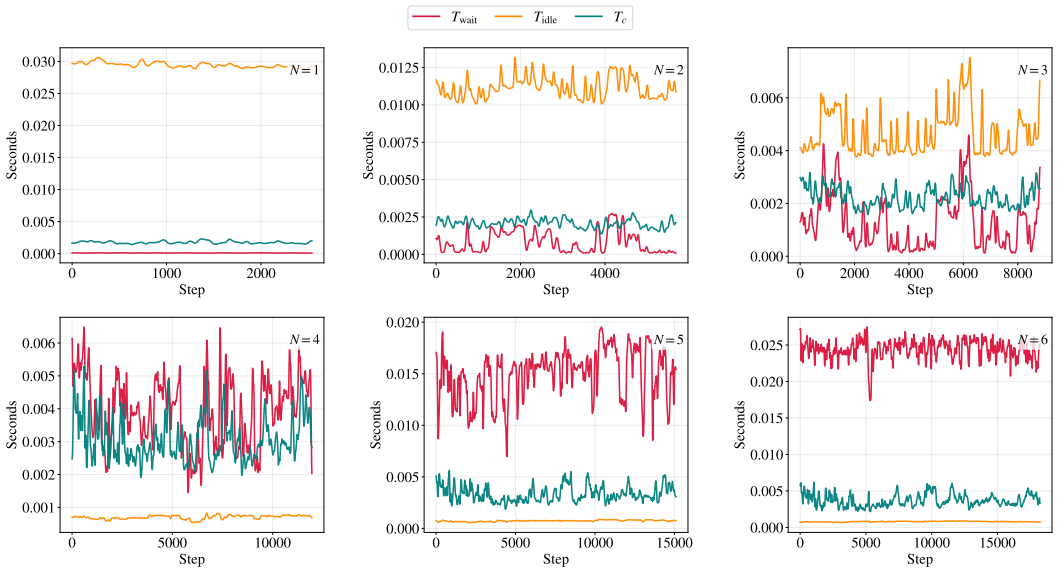



Figure 8: Comparison between $t_c$, $t_{\text{wait}}$ and $t_{\text{idle}}$

Table 3: Communication overhead under intra-node and inter-node deployment. The proportion is measured relative to the end-to-end time of one speculative-decoding iteration.

| Deployment | Communication latency | Iteration proportion |
|---|---|---|
| Intra-node | 1.61 ms | 2.60% |
| Inter-node | 2.43 ms | 3.80% |

# I Additional Deployment and Scaling Analysis

This section provides additional measurements that complement the main evaluation of StarSD. We first clarify the cross-machine configuration used in the target-count scaling experiment and quantify the communication overhead introduced by inter-node execution. We then relate the measured communication latency to the saturation condition derived in Section 3, and finally evaluate StarSD with a substantially larger target model to examine whether the same one-for-many behavior persists beyond the 7-8B model scale.

## I.1 Cross-Machine Deployment and Communication Overhead

The target-count scaling experiment in the main text uses a two-server deployment containing seven GPUs. The first server contains four RTX 4090 GPUs and hosts the shared draft instance together with three target instances. The second server contains three A100 GPUs and hosts the additional target instances when the number of targets exceeds three. Therefore, the configurations with $N = 4$, 5, and 6 involve actual inter-machine execution. Communication between the two servers uses the same socket-based messaging path as the rest of StarSD, and the reported end-to-end measurements include serialization, network transfer, and the associated runtime overheads.

To quantify the cost introduced by crossing the machine boundary, we measure the communication latency $t_c$ and its fraction of one speculative-decoding iteration under intra-node and inter-node execution. Table 3 summarizes the results.

Moving from intra-node to inter-node execution increases the measured communication latency from 1.61 ms to 2.43 ms, corresponding to only 3.80% of the end-to-end iteration time in the evaluated local-area network. Consequently, communication remains a relatively small component of the speculative-decoding cycle in this deployment.

The behavior of the shared draft as $N$ increases further illustrates where the system bottleneck appears. From $N = 4$ to $N = 6$, draft utilization increases from 91.4% to 99.6%, while the observed draft idle interval decreases from approximately 0.6 ms to zero. Over the same range, aggregate throughput increases from 266.6 to 288.7 tokens/s, whereas queueing time increases from 3.9 ms to 22.3 ms. At $N = 5$, draft utilization already reaches approximately 99.5% and the idle gap disappears. Increasing the number of targets further therefore primarily increases contention for the shared draft rather than exposing additional communication-induced idle time. These measurements indicate that, within the evaluated two-server LAN environment, the shared draft reaches compute saturation before communication fan-in becomes a dominant bottleneck.

## I.2 Communication Latency and the Saturation Point

The cross-machine measurements can also be interpreted through the saturation analysis in the main text. Recall that the minimum number of active target instances required to continuously occupy the draft is

$$N_{\text{full}} = \left\lceil \frac{Z(d)}{S(d)} \right\rceil + 1, \tag{I.1}$$

where $S(d)$ is the effective draft-side service time for speculation depth $d$, and $Z(d)$ denotes the time between completion of a draft operation and the corresponding target result becoming available for subsequent drafting. In the system model, $Z(d)$ includes target verification and the communication path between the draft and target workers.

An increase in communication latency enlarges $Z(d)$ and can therefore increase the number of targets required to hide the return latency. Importantly, this dependence is stepwise rather than con-

Table 4: StarSD scaling with LLaMA2-Chat 70B on eight A800 GPUs.

| Metric | $N = 1$ | $N = 2$ | $N = 3$ |
|---|---|---|---|
| Throughput (tokens/s) | 31.8 | 47.6 | 62.2 |
| Queueing delay (s) | 0.0002 | 0.0008 | 0.0010 |
| Draft idle time (s) | 0.1750 | 0.0794 | 0.0485 |

tinuous because of the ceiling operation in Equation I.1. The predicted saturation point changes only when the additional return latency is sufficiently large for $Z(d)/S(d)$ to cross the next integer boundary.

In our measured deployment, moving from intra-node to inter-node communication increases the communication component by only $2.43 - 1.61 = 0.82$ ms. This increase is small relative to the effective draft service interval and does not cross the next saturation boundary for the evaluated configuration. As a result, introducing the second server does not change the predicted full-load transition. This observation is consistent with the measured target-count sweep: the draft becomes effectively continuously occupied around $N = 5$, after which additional targets primarily increase queueing rather than further reducing draft idle time. Thus, the analytical condition also provides a direct interpretation of how larger network delays would affect StarSD: sufficiently large increases in return latency require additional concurrent targets to maintain continuous draft execution, whereas smaller increases that remain within the same service-time interval leave $N_{\text{full}}$ unchanged.

### I.3 Scaling to a Larger Target Model

The main evaluation primarily uses 7-8B target models. To examine whether the one-for-many behavior persists when target verification is substantially more expensive, we additionally evaluate StarSD with LLaMA2-Chat 70B on eight NVIDIA A800 GPUs. We vary the number of target instances from $N = 1$ to $N = 3$ while retaining a single shared draft instance. Table 4 reports the resulting throughput, queueing delay, and draft idle time.

Increasing the number of target instances from one to three raises aggregate throughput from 31.8 to 62.2 tokens/s while reducing the draft idle interval from 0.175 s to 0.0485 s. Queueing remains negligible throughout this range. The same qualitative behavior observed with the smaller target models therefore remains visible with the 70B target: additional independently progressing targets provide more opportunities for the shared draft to perform useful work while other targets are executing verification. This result provides additional evidence that the one-for-many execution pattern is not specific to the 7-8B model configurations used in the primary evaluation.

## J Comparison with Distributed Speculative Decoding Systems

Distributed speculative decoding systems differ not only in their communication mechanisms, but also in their target serving regimes, resource assumptions, and optimization objectives. To clarify these design tradeoffs, Table 5 compares StarSD with DSI and SpecEdge along five dimensions: primary objective, target regime, communication model, scheduling focus, and resource organization. StarSD targets stable per-session service under many concurrent sessions by sharing one draft instance across multiple target sessions. In contrast, DSI primarily reduces single-request latency by assigning additional speculative resources to the same request, while SpecEdge targets edge-cloud cost efficiency and interactive latency through edge-side drafting and server-side verification. These systems are designed for different operating regimes and optimization objectives; the comparison is therefore intended to clarify their distinct design choices rather than to establish that one system is uniformly preferable to another.

### J.1 End-to-End Comparison with SpecEdge

Among the distributed speculative decoding systems in Table 5, SpecEdge provides a public implementation and supports a direct end-to-end comparison. We emphasize that the purpose of this experiment is not to establish a general ranking between StarSD and SpecEdge. The two systems

Table 5: Comparison between StarSD and representative distributed speculative decoding systems.

| Method | Primary objective | Target regime | Communication model | Scheduling focus | Resource organization |
|---|---|---|---|---|---|
| **StarSD** | Per-session stability | One shared draft serving multiple concurrent target sessions | Asynchronous draft-verifier interaction through a global request buffer | Work-conserving draft-side scheduling to eliminate idle gaps | Decouples the draft from the target side, leaving more accelerator memory for target KV cache |
| **DSI** | Single-request latency reduction | Multiple target and draft instances jointly accelerate the same request | Multi-instance speculation parallelism to hide verification latency | Lookahead-controlled orchestration of target servers so verification does not block drafting | Primarily spends additional compute resources on each query and is not designed for improving many-session resource packing |
| **SpecEdge** | Edge–cloud cost efficiency and interactive latency | Edge drafting with server-side verification in WAN settings | Exchanges token outputs rather than intermediate activations or full model states | Server-side pipeline-aware scheduling across multiple edge requests | Moves drafting to the edge and concentrates server resources on verification, focusing on WAN communication volume and cost efficiency |

are designed around different deployment assumptions: SpecEdge focuses on edge–cloud speculative inference, whereas StarSD focuses on sharing draft computation across multiple concurrent target-side sessions. We therefore use this comparison to examine how these different resource organizations behave under the same accelerator budget and request workload.

The experiment is conducted on a server with six RTX 4090 GPUs, while the allocated GPU budget $B$ is varied from two to six. For each budget, SpecEdge deploys one target worker and $B-1$ independent draft clients, whereas StarSD deploys one shared draft worker and $B-1$ independent target workers. Thus, increasing the GPU budget expands different sides of the speculative-decoding pipeline: SpecEdge increases draft-side parallelism around one target worker, while StarSD increases target-side parallelism around one shared draft worker.

Both systems replay the same pre-generated Poisson arrival trace with $\lambda = 0.4$ requests/s. The workload consists of 20 requests randomly sampled from AIME25, with each request generating up to 512 tokens. Requests are submitted according to their scheduled arrival times without waiting for preceding requests to complete, yielding an open-loop workload. We use TTFT and TPOT thresholds of $2.0$ s and $0.12$ s, respectively, for the SLO evaluation.

Table 6: End-to-end throughput and SLO performance of StarSD and SpecEdge under the same Poisson open-loop workload. SLO GP denotes SLO goodput.

| GPUs | System | E2E Thr. (tok/s) | SLO Sat. (%) | SLO GP (tok/s) |
|---|---|---|---|---|
| 2 | SpecEdge | 51.24 | 5 | 2.3 |
| 2 | StarSD | 34.12 | 0 | 0.0 |
| 3 | SpecEdge | 90.61 | 10 | 8.6 |
| 3 | StarSD | 90.59 | 15 | 13.6 |
| 4 | SpecEdge | 105.63 | 15 | 15.8 |
| 4 | StarSD | 156.75 | 95 | 148.8 |
| 5 | SpecEdge | 131.05 | 20 | 27.7 |
| 5 | StarSD | 157.06 | 95 | 149.1 |
| 6 | SpecEdge | 141.78 | 50 | 72.4 |
| 6 | StarSD | 156.14 | 95 | 148.2 |

The results show a resource-configuration crossover under the evaluated workload. With two GPUs, StarSD has only one target worker and therefore cannot yet exploit sharing one draft across multiple independently progressing targets. With three GPUs, the two systems achieve nearly identical end-to-end throughput, although their latency profiles differ. As the GPU budget increases further, the two resource organizations lead to increasingly different serving behavior.

At four GPUs, StarSD achieves 156.75 tokens/s, compared with 105.63 tokens/s for SpecEdge, corresponding to a $1.48\times$ throughput difference under this workload. Under the TTFT and TPOT thresholds defined above, StarSD achieves 95% SLO satisfaction and 148.8 tokens/s SLO goodput, compared with 15% and 15.8 tokens/s for SpecEdge. At six GPUs, StarSD maintains 95% SLO

Table 7: Latency comparison under the same open-loop workload. TTFT and TPOT report P50 values.

| GPUs | System | P50 TTFT (s) | P50 TPOT (s) |
|---|---|---|---|
| 2 | SpecEdge | 80.73 | 0.021 |
| 2 | StarSD | 2.97 | 0.484 |
| 3 | SpecEdge | 29.46 | 0.022 |
| 3 | StarSD | 1.27 | 0.124 |
| 4 | SpecEdge | 18.37 | 0.026 |
| 4 | StarSD | 0.85 | 0.020 |
| 5 | SpecEdge | 7.20 | 0.024 |
| 5 | StarSD | 0.93 | 0.018 |
| 6 | SpecEdge | 2.41 | 0.024 |
| 6 | StarSD | 0.74 | 0.017 |

satisfaction and 148.2 tokens/s SLO goodput, while SpecEdge reaches 50% and 72.4 tokens/s, respectively.

These results should be interpreted in the context of the different design goals of the two systems rather than as a universal advantage of either approach. SpecEdge allocates additional accelerators primarily to draft-side parallelism around a single target worker, consistent with its edge-cloud deployment focus, whereas StarSD allocates additional accelerators to independently progressing target workers around one shared draft, consistent with its focus on multi-session serving. Under the particular open-loop workload evaluated here, the latter organization provides higher aggregate target-side serving capacity once multiple target workers can be deployed. This experiment is also an end-to-end comparison of the native implementations rather than a controlled ablation of resource organization alone; the measured differences therefore include system-specific factors such as speculative decoding backends, acceptance behavior, and batching mechanisms.

# K Implementation Details of StarSD

Algorithm 2 gives the pseudocode for StarSD. This appendix provides implementation-level details and covers the following topics.

- One-time connection establishment protocol.
- Steady-state request execution loop and payload handling.
- Decoupling communication from draft inference on $M_q$ using separate logical components or threads.
- Tag-indexed state management, including KV cache lifecycle and other per-session metadata.
- How these details realize the design requirements in Section 5.
- Work-conserving execution to support Eq. (8)-(10).
- Per-session isolation to keep $\beta_i(y)$ in Eq. (1) and $l_i(d)$ in Eq. (2) well-defined.
- Clean time decomposition to preserve $Z(d) = t_c + t_v$ in Eq. (6) and $S(d) = dt_s$ in Eq. (5).

## K.1 Connection Establishment

**One-time handshake per $M_p^{(i)}$.** Each $M_p^{(i)}$ performs an initial handshake with $M_q$ through a public port. Upon the first request from this $M_p^{(i)}$, $M_q$ allocates (i) a private communication port and (ii) a unique tag. All subsequent messages between this $M_p^{(i)}$ and $M_q$ are sent through the private port, and the tag is attached to every request to identify the corresponding per-session states at $M_q$. By isolating the control plane, i.e., handshake from the data plane, i.e., per-iteration messages, we ensure $t_c$ in Eq. (6) corresponds to the minimal, repeated payload transfer.

**Algorithm 2** StarSD: One-for-Many Serving Runtime

**Input:** $M_q$, $\{M_p^{(i)}\}_{i=1}^{N}$, Prefix $\hat{y}$.
**Preparation:**
· Handshake: assign each target a unique
· `tag` and establish a dedicated port.
· $\mathcal{C}[\texttt{tag}]$: per-session KV cache for `tag`.
· $\mathcal{Q}_{in}$: global request buffer $(\texttt{tag}, y)$.
· $\mathcal{Q}_{out}$: outgoing queue of replies $(\texttt{tag}, \hat{y})$.

*## In machine hosting $M_p^{(i)}$*
**if** `recv` $\hat{y}$ from $M_q$ **then**
  $y \leftarrow M_p^{(i)}(\hat{y})$ *// refer to Algorithm 1 for verification details*
  `send` $y$ to $M_q$ on the port identified by `tag`
**end if**

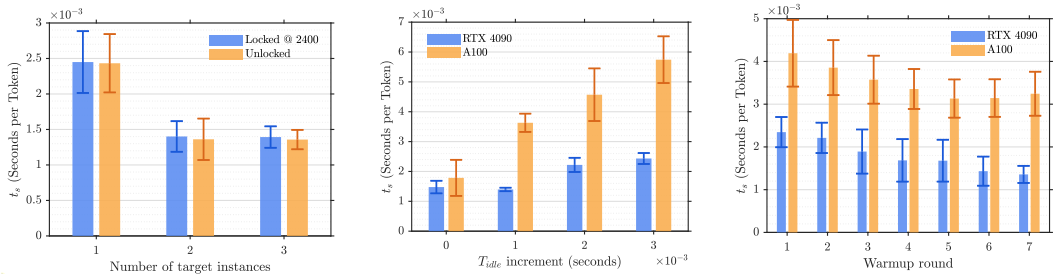
*## In machine hosting $M_q$*
*// Receiver*
**while** true **do**
  $(\texttt{tag}, y) \leftarrow$ `recv` from any $M_p^{(i)}$
  `push` $(\texttt{tag}, y)$ to $\mathcal{Q}_{in}$
**end while**
*// Draft inference*
**if** $\mathcal{Q}_{in}$ is non-empty **then**
  $(\texttt{tag}, y) \leftarrow$ `pop` $\mathcal{Q}_{in}$
  $(\hat{y}, \mathcal{C}[\texttt{tag}]) \leftarrow M_q(y;\ \mathcal{C}[\texttt{tag}])$ *// refer to Algorithm 1 for inference details*
  `push` $(\texttt{tag}, \hat{y})$ to $\mathcal{Q}_{out}$
**end if**
*// Sender*
**if** $\mathcal{Q}_{out}$ **not empty then**
  $(\texttt{tag}, \hat{y}) \leftarrow$ `pop` $\mathcal{Q}_{out}$
  `send` $\hat{y}$ back to $M_p^{(i)}$ on the port identified by `tag`
**end if**

### K.2 Steady-State Request Execution

**Request execution loop.** Given a user prompt, $M_p^{(i)}$ begins decoding and produces an initial token. Then, in each iteration, $M_p^{(i)}$ sends the verified prefix or the newly verified token, along with its tag, to $M_q$ via the private port. The payload follows the minimal-transfer design of the distributed EAGLE setting to reduce the per-iteration communication cost. Upon receiving a tagged request, $M_q$ pushes it into the global request buffer. When the $M_q$ inference instance becomes available, it selects a buffered request using a work-conserving scheduling policy (e.g., first-in-first-out). Before inference, $M_q$ retrieves the corresponding KV cache using the tag, executes draft inference, and updates the KV cache upon completion. The draft output is written to the per-$M_p^{(i)}$ communication channel and sent back to $M_p^{(i)}$ via the corresponding private port, where it is verified and used to form the next request.

### K.3 Decoupling Communication and Draft Inference on $M_q$

As shown in Fig. 3, $M_q$ consists of two logical components: (i) a communication component that receives requests and appends them to the global request buffer, and (ii) an inference component that repeatedly selects and executes buffered requests as long as the buffer is non-empty. Each active $M_p^{(i)}$ is associated with its own private port and a dedicated communication channel at $M_q$, preventing cross-session interference while keeping $M_q$ work-conserving. This decoupling is required for the service-time model $S(d) = dt_s$ in Eq. (5) in which I/O stalls do not backpressure the compute loop, so the measured draft-side service remains dominated by pure compute. In addition, it helps




(a) Effect of locking GPU frequency.

(b) Effect of $T_{\text{idle}}$.

(c) Effect of warm-up rounds.

Figure 9: Per-token generation time analysis of $M_q$ under warm-up at $T_{\text{idle}} = 0.03$ s, GPU frequency control, and $T_{\text{idle}}$ variation.

keep the per-iteration transfer cost accounted for in $Z(d) = t_c + t_v$ in Eq. (6), rather than being mixed into the compute-side service time.

### K.4 Tag-Indexed State Management

**KV cache isolation and lifecycle.** $M_q$ maintains a mapping from tag to KV cache. When an $M_p^{(i)}$ starts a new decoding session, $M_q$ initializes an empty KV cache entry for its tag; during each iteration, the inference component loads and updates this KV cache to avoid recomputation of historical activations. When the decoding session finishes, the KV cache entry associated with the tag is released to reclaim memory. This tag-indexed management prevents cross-session contamination and enables efficient lookup and update with many active $M_p^{(i)}$. Importantly, this isolation is necessary for session-conditioned quantities such as $\beta_i(y)$ in Eq. (1) and $l_i(d)$ in Eq. (2) to remain well-posed, where each return must advance the *same* session prefix and KV state associated with that tag.

**Extending to other per-$M_p^{(i)}$ metadata.** The same tag-indexed design applies to other per-$M_p^{(i)}$ states maintained at $M_q$, such as speculative metadata and bookkeeping information required by the execution loop. By enforcing strict per-tag isolation, StarSD preserves correctness under concurrency while keeping $M_q$ scalable in both memory usage and scheduling.

## L Details of Burstness Bebavior of $M_q$

### L.1 Execution Regimes and Performance Dynamics

As suggested by the results in Table 1, a major source of StarSD's system-level performance gain comes from eliminating cold-start and idle gaps on the draft side. To further analyze this effect and verify that the observed variation in the inference speed of $M_q$ is not specific to a particular accelerator, we conduct controlled experiments by forming a single draft–target pair and evaluating it independently on an RTX 4090 GPU and on an A100 GPU.

We first rule out GPU frequency scaling as a confounding factor by instrumenting the graphics clock before each inference. As shown in Fig. 9($a$), the same slowdown trend appears under both the default Dynamic Voltage and Frequency Scaling (DVFS) policy and a fixed clock of 2400 MHz, indicating that frequency changes are not the cause. To precisely control idle behavior, we then inject artificial delays on the draft side by pausing $M_q$ for a prescribed duration after receiving each request, thereby inducing a target $T_{\text{idle}}$. Figure 9($b$) shows that per-token inference time remains stable when $T_{\text{idle}}$ is either small or large, but degrades in an intermediate regime where increasing $T_{\text{idle}}$ leads to a monotonic slowdown and higher run-to-run variance. Together, these results indicate that sustained, continuous execution keeps the draft model in a favorable operating state, while intermittent idle gaps introduce transient inefficiencies that increase both latency and variability.

We further examine the warm-up effect shown in Fig. 9($c$) by inserting additional draft-side executions after an idle gap and before normal generation begins. In this setup, $M_q$ generates a small num-

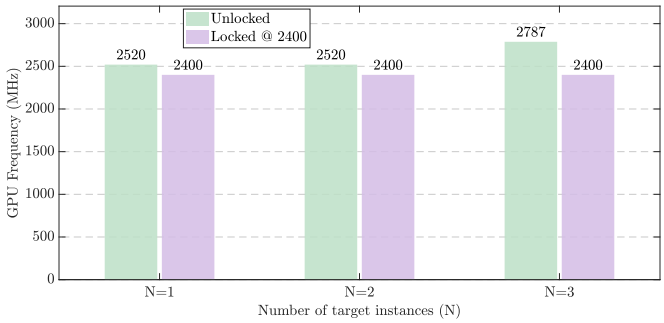



Figure 10: Measured GPU graphics clock frequency before each draft-side inference under the default DVFS setting versus a fixed 2400 MHz lock.

ber of warm-up tokens to transition back toward a steady execution state. As the number of warm-up rounds increases, the per-token inference speed improves and then stabilizes, with diminishing gains beyond a modest threshold. At low warm-up counts, the execution behavior is comparable to the $N = 1$ setting in Table 1, which exhibits an idle gap of approximately $T_{\text{idle}} \approx 0.03$ second and degraded performance. As warm-up rounds increase, the effective idle impact is reduced to about $0.014$ second, leading to faster and more stable inference.

This section supplements the frequency-sensitivity study in Fig. 9(*b*) by verifying that GPU frequency control is indeed taking effect during our experiments. To verify whether GPU frequency control is effective, we log the measured frequency just before each draft-side inference begins in both the default setting and when the GPU graphics clock is locked.

**GPU frequency lock command.** We enable persistence mode and lock the graphics clock to 2400 MHz using the following commands:

**GPU Frequency Lock Commands**

```
sudo nvidia-smi -pm 1
sudo nvidia-smi -i 0 -lgc 2400,2400
```

Unless otherwise specified, the "Default (unlocked)" setting means leaving the GPU in its default DVFS behavior, without applying a fixed graphics clock.

**Frequency probing via NVML.** After applying the frequency lock or leaving it unlocked for the default setting, we probe the instantaneous graphics clock *immediately before* the draft model starts inference in each iteration. Concretely, we query NVML through `pynvml` by using the following command:

**GPU Frequency Measurement**

```
pynvml.nvmlDeviceGetClockInfo(\_gpu\_handle, pynvml.NVML\_CLOCK\
    _GRAPHICS)
```

We record these measurements for each experimental configuration, which means different numbers of target instances $N$. In post-processing, we discard spurious outliers where the returned frequency is very low, which can occur during handle initialization or end of string task, and then aggregate the remaining samples to produce the summary shown in Fig. 10.

**Observed frequency stability.** Figure. 10 confirms that the locked configuration stays at 2400 MHz across all tested $N$, whereas the default configuration exhibits higher and varying graphics clocks. This verification ensures that the trends reported in Fig. 9(*b*) are valid.

## M Draft Inference Time Under Different Experimental Settings

**Draft inference speed** We report the draft-side inference time measured during our experiments using EAGLE-2's Vicuna-7B setting as backend. To obtain accurate GPU kernel timing under CUDA's asynchronous execution, we use `torch.cuda.Event(enable_timing=True)` and compute elapsed time via `event.elapsed_time()` with an explicit `torch.cuda.synchronize()` to ensure all kernels have completed.

**CUDA Event Timing Snippet**

```
starter = torch.cuda.Event(enable_timing=True)
ender = torch.cuda.Event(enable_timing=True)

starter.record()
# draft forward pass
ender.record()

torch.cuda.synchronize()
infer_ms = starter.elapsed_time(ender) # milliseconds
```

**Draft inference time trend.** Figure. 11 plots the measured draft-side inference time under different numbers of target instances $N$ on RTX 4090 server. As $N$ increases from 1 to 2 or above, the inference time drops markedly, where $N$=1 trace concentrates around $\sim$16 ms with pronounced jitter and frequent spikes, whereas the $N$=2 and $N$=3 traces form much tighter bands around $\sim$9 ms with only occasional outliers.

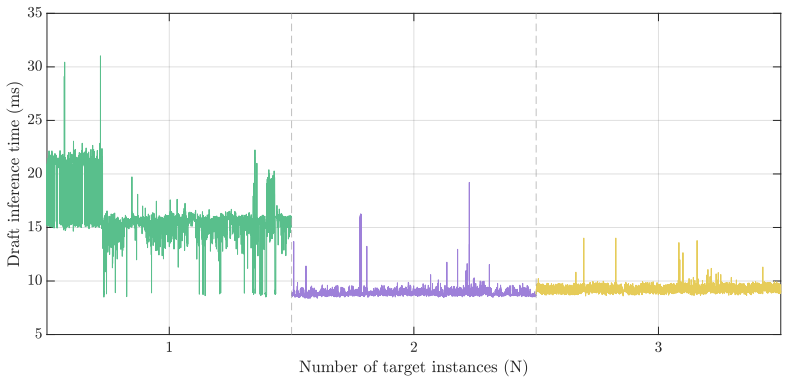



Figure 11: Draft-side inference time under different numbers of target instances $N$.

**Summary statistics.** Table 8 summarizes the draft-side inference time. As $N$ increases from 1 to 2-3, the mean inference time decreases substantially and the variance drops sharply, indicating a faster and much more stable draft-side execution.

Table 8: Draft-side inference time statistics.

| $N$ | Mean (ms) | Variance ($\mathrm{ms}^2$) |
|---|---|---|
| 1 | 16.06 | 6.34 |
| 2 | 8.86 | 0.30 |
| 3 | 9.17 | 0.12 |

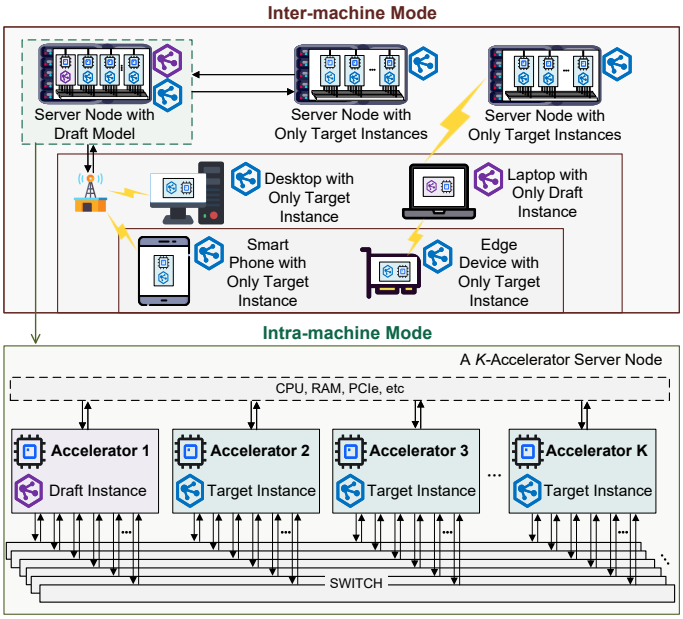



Figure 12: **Deployment illustrations for StarSD. (A) Intra-MG:** draft instance runs on one GPU and serves multiple target instances on other GPUs within the same machine, where communication can leverage intra-machine high-speed GPU interconnects, e.g., NVLink. **(B) Inter-CL:** draft instance runs as a draft server on one machine while target instances are placed on other machines, communication crosses the network boundary and is implemented via socket-based messaging. **(C) Inter-ES:** draft instance runs on one edge device, e.g., a laptop, whereas the target instances run on other edge devices, e.g., NVIDIA Jetson, and communication follows the same Inter-CL setup.

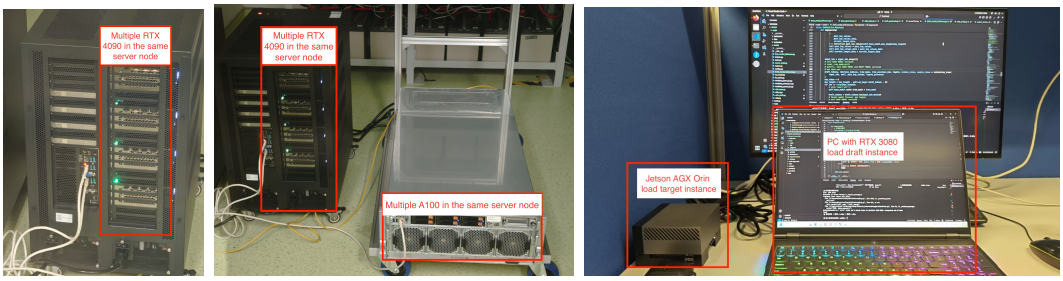
## N Deployment Illustrations

StarSD is designed to support disaggregated speculative decoding across a range of deployment topologies as shown in Fig. 12, from intra-node multi-GPU setups to networked multi-node clusters and heterogeneous edge-server environments. To make these supported scenarios explicit, we summarize three representative deployment modes:

- **Intra-Node Multi-GPU (Intra-MG)**: *machine-internal cross-GPU* within a single machine.
- **Inter-Node Cluster (Inter-CL)**: *cross-machine* deployment over a network.
- **Edge-Server Hybrid (Inter-ES)**: *cross-device* deployment (e.g., Jetson-PC) emphasizing heterogeneity and portability.

**Demo A: Intra-Node Multi-GPU (Intra-MG).** In Intra-MG, the draft server hosting $M_q$ is placed on one GPU, while the target instances $\{M_p^{(i)}\}$ are placed on other GPUs within the same node. A representative configuration is to place $M_q$ on a single GPU and run three target instances on the remaining GPUs of a 4-GPU server. This topology matches the *machine-internal cross-GPU* scenario and is a natural fit when co-locating $M_q$ and $M_p^{(i)}$ on the same device is infeasible due to memory pressure. Since all communication stays within one machine, StarSD can exploit high-bandwidth intra-host GPU transport, e.g., NVLink, to reduce transfer overhead. The real-world setup is shown in Fig. 13(*a*).

**Demo B: Inter-Node Cluster (Inter-CL).** In Inter-CL, the draft server and target instances are placed on different machines, corresponding to the *cross-machine* deployment over a network. A representative configuration is to host the draft instance alongside a subset of target instances on a single multi-GPU server, e.g., one GPU for draft and three GPUs for targets, while placing additional




(a) Intra-MG (b) Inter-CL (c) Inter-ES

Figure 13: Photograph of the inter-server node deployment, the intra-server node deployment, and the Jetson-PC hybrid deployment.

target instances on another server equipped with a different GPU type to form a heterogeneous cluster. Because the dataflow must traverse the network, we use a socket-based messaging layer for portability across commodity Ethernet and heterogeneous cluster environments, while keeping the StarSD execution logic unchanged. More specialized transports, e.g., RDMA or cluster-specific collectives, can be integrated as an optimization, but are not required for StarSD to function. The real-world setup is shown in Fig. 13(*b*).

**Demo C: Edge-Server Hybrid (Inter-ES).** To demonstrate *cross-device* portability, we also include an edge-server deployment in which one side runs on an embedded edge device, e.g., a Jetson, and the other side runs on a desktop/server GPU machine. This scenario stresses heterogeneous compute capability and network constraints, and therefore uses the same socket-based messaging layer as Inter-CL. The real-world setup is shown in Fig. 13(*c*).

## O LIMITATIONS

StarSD improves shared-draft utilization and provides stronger robustness under multi-session serving, the current design has several limitations.

- **Operating-window dependence and shared-draft bottleneck.** StarSD is most beneficial when the number of active target instances is large enough to hide draft-side idle gaps but not so large that the shared draft instance becomes the dominant bottleneck. Under light load, reserving one accelerator for the shared draft model can reduce peak raw throughput compared with co-located speculative decoding, because fewer accelerators are available for target-side serving. Under heavy load, once $M_q$ becomes fully loaded, additional sessions mainly increase queueing delay in the shared request buffer, which can increase TPOT and reduce per-target throughput. This indicates that practical deployments need admission control or a future many-for-many extension with multiple draft instances.
- **SLO-aware scheduling is evaluated but not yet optimized.** The current evaluation reports user-facing metrics such as TTFT, TPOT, SLO satisfaction rate, and SLO-goodput, and shows that StarSD is more robust than co-located baselines under overloaded workloads. However, the runtime still follows a simple work-conserving shared-draft design instead of an explicit SLO-aware or fairness-aware scheduler. Therefore, StarSD does not provide formal guarantees for per-session fairness, deadline satisfaction, or tail-latency isolation when sessions have heterogeneous priorities, output lengths, or latency objectives.
- **Communication portability trades off against efficiency.** StarSD currently uses a socket-based messaging layer to support intra-machine, inter-machine, and edge-server deployments with the same execution logic. This design improves portability across heterogeneous environments, but it is not the most optimized transport for high-performance clusters. Serialization, deserialization, GPU-CPU data movement, and small-message overheads may become more significant in cross-machine or edge settings. More specialized transports, such as RDMA, CUDA-aware communication, or cluster-specific collectives, could further reduce communication overhead.

Table 9: Existing assets used in StarSD.

| Asset | Type | Usage in StarSD | License / Terms |
|---|---|---|---|
| `lmsys/vicuna-7b-v1.3` | Model | Target model in the EAGLE-2-based scaling and distributed-behavior experiments. | Non-commercial license. |
| `yuhuili/EAGLE-Vicuna-7B-v1.3` | Draft model | EAGLE draft model paired with Vicuna-7B v1.3 in the EAGLE-2 backend experiments. | Apache-2.0; used together with the corresponding target-model license. |
| `SafeAILab/EAGLE` | Codebase / backend | Baseline speculative decoding implementation and EAGLE-2 backend used for the scaling experiments. | Apache-2.0. |
| `Qwen/Qwen3-8B` | Model | Target model in standalone serving, co-located DFlash, and StarSD+DFlash multi-session experiments. | Apache-2.0. |
| `z-lab/Qwen3-8B-DFlash-b16` | Draft model | DFlash draft model used with `Qwen/Qwen3-8B` in DFlash-based StarSD experiments. | MIT. |
| `z-lab/dflash` | Codebase / backend | DFlash speculative decoding backend used for multi-session serving and backend-generality experiments. | MIT. |
| `lm-sys/FastChat/MT-Bench` | Evaluation code / benchmark | Source of MT-Bench questions and evaluation utilities used for conversational workload evaluation. | FastChat codebase: Apache-2.0. |
| `math-ai/aime25` | Dataset / benchmark | AIME25 reasoning benchmark used in multi-session serving and generality experiments. | Apache-2.0. |
| `openai/openai_humaneval` | Dataset / benchmark | Code-generation benchmark used in backend and dataset generality experiments. | MIT. |

## P Existing Assets and Licenses

We use existing models, speculative decoding backends, and benchmark datasets only for research evaluation. Table 9 summarizes the assets used in our experiments, their roles in StarSD, and their licenses or terms of use. All assets are cited in the main paper or appendix, and we follow the corresponding licenses and usage restrictions.

## Q Impact Statement

StarSD enables one-for-many speculative decoding by decoupling drafting and verification across distributed accelerators, allowing a single draft model to serve multiple target instances in a work-conserving manner. This can improve utilization and reduce deployment overheads under memory and resource constraints, supporting more flexible allocation across heterogeneous devices in both server and edge serving environments. StarSD may also introduce risks. By lowering the cost of LLM inference, it can make large-scale misuse easier. More importantly, however, the shared draft model can become a concentrated trust point, as it may transiently hold multiple sessions prompts, intermediate prefixes, and cached decoding state across served targets. If the draft service is compromised, an attacker could potentially access sensitive user inputs or session traces, leading to severe privacy leakage. To mitigate these risks, deployments should treat the draft service as a high-sensitivity component. Enforce strong isolation between sessions, minimize and protect in-memory and on-disk artifacts including KV caches and logs, use authenticated and encrypted channels for inter-node communication, and apply access control, rate limiting, and auditing to detect abuse and intrusion.