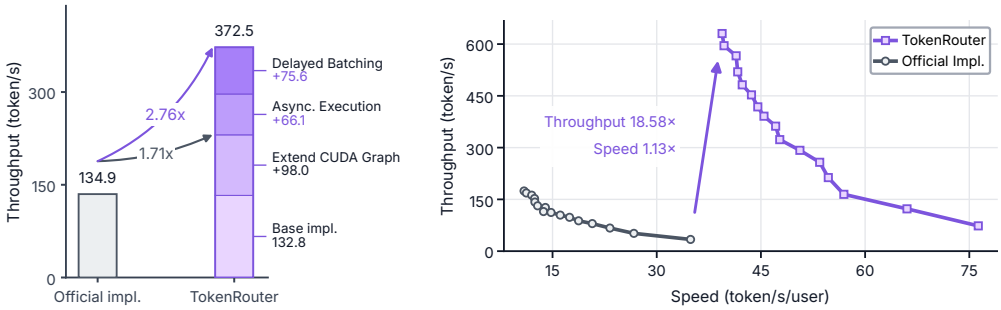
# TokenRouter: Efficient Serving System for Token-Level LLM Routing

**Tianyu Fu**[*1], **Tengxuan Liu**[*1], **Ruoxi Wang**[1], **Yixin Dong**[2], **Yi Ge**[1], **Yichen You**[1], **Yu Wang**[†1]

[1]Tsinghua University [2]Carnegie Mellon University

## Abstract

Large language model (LLM) routing distributes inference work across different models, advancing the cost-quality Pareto frontier of LLM serving. While coarse-grained routing at the *session* or *query* level has been widely adopted in production systems, recent algorithmic work shows that fine-grained *token*-level routing can yield substantial efficiency and quality gains. However, efficiently serving token-level routed inference poses significant challenges to existing systems. Built on single-LLM assumptions, current systems suffer from severe step desynchronization and frequent batch admission delays under token-level routing, and they also impose high implementation complexity on developers. To address these challenges, we design TokenRouter, an efficient and developer-friendly serving system for token-level routed LLM inference. TokenRouter follows the principle of *request-centric programming, model-centric execution*: developers describe routing logic from the perspective of a single request, while the runtime launches a subserver for each LLM and dispatches requests asynchronously. Each subserver employs a delayed-batching scheduler, whose optimal hyperparameters are derived from a mathematical throughput model of the system. Across diverse routing algorithms, workloads, and model pairs, TokenRouter achieves 2.01–64.15× higher decoding throughput than existing systems, substantially advancing the serving efficiency of token-level LLM routing. Our code is available at `https://github.com/thu-nics/TokenRouter`.




(a) Throughput gain breakdown. (b) Throughput–speed trade-off.

Figure 1: TokenRouter improves serving efficiency for token-level routing. (a) At concurrency 8, engineering optimizations yield a 1.71× throughput gain, and TokenRouter achieves 2.76× overall, relative to the official R2R implementation. (b) Throughput–speed trade-off. TokenRouter achieves 18.58× higher throughput than R2R under a stricter SLO.

[*]Equal contribution.

[†]Corresponding author: Yu Wang (yu-wang@tsinghua.edu.cn).

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

# 1 Introduction

Large language models (LLMs) have demonstrated strong capabilities across a wide range of applications. To meet diverse demands, modern LLMs vary in size and domain expertise, showing different latencies and capabilities. To exploit this diversity and improve the cost-quality Pareto frontier of LLM inference, model routing has emerged as a common serving paradigm.

Coarse-grained routing at *session* or *query* level is already widely used in production LLM serving systems. For example, commercial systems, such as ChatGPT [1] and Cursor [2], direct each user query to different backend models based on predicted difficulty or topic. Academic systems such as RouteLLM [3] also show that query-level routing can achieve strong intelligence at a lower cost with preference data. However, in these systems, once a request is routed, the response is generated entirely by a single model until the next round.

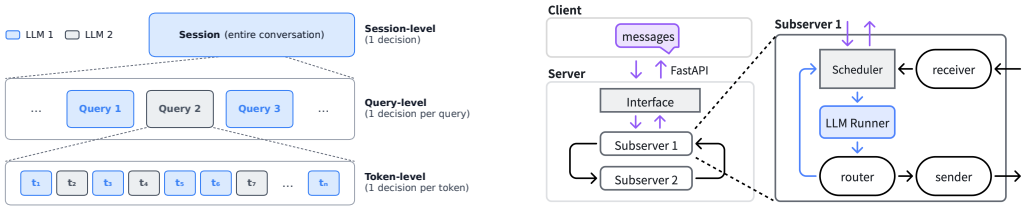
Recent algorithmic work shows that routing at the finer *token* level can unlock benefits that query-level routing cannot reach [4–12]. On the efficiency side, token-level routing exploits the sharp variation in generation difficulty within a single query. For example, R2R [5] can match a 32B model's quality while routing only 5% of *tokens* to the 32B model and decoding the rest with a 1.5B model, versus routing around 40% of *queries* to the 32B model. On the quality side, token-level routing allows models with complementary expertise to collaborate in a single response, surpassing the generation quality of any single LLM [9, 11, 13]. It also provides smooth control over the entire cost-quality trade-off, rather than the few discrete operating points imposed by coarser routing granularities.

However, existing LLM serving systems lack efficient support for token-level routing. Current systems such as SGLang [14] and vLLM [15] are designed for single-LLM serving, where all active requests are synchronous at each decoding step. Token-level routing breaks this assumption by allowing the target LLM to change at every decoding step, creating three main challenges. First, **step desynchronization**. Different LLMs differ widely in per-step latency. Keeping requests in a single batch synchronized forces every step to wait for the slowest model, leaving the faster models idle. Second, **batch admission delay**. Because token-level routing causes frequent model switches, a routed request often arrives while its target model is still processing a previous batch. It must then wait to be admitted into the next batch, creating bubbles and fragmenting batches. Third, **implementation complexity**. Current serving systems expose no programming interface for per-step routing decisions. Implementing token-level routing inside these systems therefore requires extensive modifications to a large codebase. The routing logic must also be carefully coordinated with tightly coupled features such as continuous batching and prefix caching.

To address these challenges, we propose TokenRouter, an efficient and developer-friendly serving system for token-level LLM routing. The key design principle is **request-centric programming, model-centric execution**: developers describe a routing algorithm by tracing how a single request flows across cooperating LLMs, while TokenRouter handles efficient serving through asynchronous model-centric subservers. This separation hides the system complexity of routing serving from developers and gives the runtime full control to optimize cross-model execution and batch scheduling. For the user client, TokenRouter exposes a single external interface, allowing it to serve as a drop-in replacement for standard single-LLM servers. Our main contributions are summarized as follows.

- **Request-centric programming interface.** We provide a route-send-receive interface that lets developers express diverse token-level routing algorithms by simply tracing how a single request moves across cooperating LLMs.
- **Asynchronous execution for serving optimization.** We propose decoupled tri-loop execution that addresses step desynchronization across cooperating LLMs, with a handoff-resume mechanism to simplify inter-model switches.
- **Delayed batching for scheduling optimization.** We propose a delayed-batching scheduler that reduces average batch admission latency, with optimal hyperparameters derived from a throughput-based mathematical model.

TokenRouter brings a high-efficiency LLM serving framework to the token-routing community, turning theoretical benefits into real-world speedups over mature single-LLM serving systems. Experiments across different algorithms, workloads, and model pairs show a 2.01–64.15$\times$ throughput boost over existing implementations.




(a) Routing levels in LLM serving.

(b) TokenRouter system architecture.

Figure 2: (a) Routing granularities in LLM serving. Token-level routing allows decisions at each token, whereas session- or query-level routing binds an entire request to one model. (b) TokenRouter's model-centric runtime. Each cooperating LLM is hosted by an autonomous subserver that implements three user-defined routing functions (route, send, receive) and runs three decoupled scheduler loops, shown in purple (client-server), blue (decoding), and black (inter-model). The Server Interface exposes a single external endpoint to the client.

## 2 Related Work

**Model routing**. Model routing distributes requests across candidate LLMs to improve efficiency or quality. Most methods operate at the session or query level, choosing models for entire requests or conversation turns. Efficiency-oriented routers typically send easier requests to cheaper models under a quality constraint [3, 16–21], while quality-oriented routers select the expert best suited to each request and may combine model outputs to surpass any single LLM [22–25]. This coarse granularity lets production serving stacks [1] reuse standard single-LLM batching and scheduling. TokenRouter instead targets token-level routing, which requires coordinating model switches within a response.

**Token-level routing**. Token-level routing selects a model for each token or short span, enabling collaboration in a single response. Efficiency-oriented methods invoke larger models for difficult tokens or segments, using trained lightweight routers [5, 6, 8, 12, 26] or heuristic signals such as confidence and entropy [27, 28]. Quality-oriented methods combine model expertise by accepting expert tokens [7, 10] or fusing output distributions [9, 11]. Despite different policies, these methods share an execution lifecycle: select next model, transfer the request state, and receive its tokens or distributions to continue generation. This lifecycle motivates TokenRouter's route-send-receive programming interface.

**LLM serving systems**. Modern LLM serving systems accelerate inference through continuous batching [29], paged KV-cache management [15], and efficient structured generation runtimes [14]. These techniques work well for single-model decoding, where each request stays with one model and advances at that model's pace. Speculative decoding also uses multiple models, but couples draft generation and target verification in a predictable pattern [30–32]. Token-level routing instead allows data-dependent model switches at each token, leading to irregular model-local arrivals and fragmented batches. TokenRouter addresses these dynamics with a model-centric runtime that hosts each model as an autonomous subserver, preserving continuous batching and prefix caching locally.

## 3 Programming Interface

To provide a simple and general programming interface, TokenRouter abstracts diverse token-level routing algorithms with three main components: **route**, **send**, and **receive**. These components allow developers to describe when a request should switch models, what state should be transferred, and how returned tokens should be accepted. Section 3.1 explains the design principle behind this interface, and Section 3.2 presents the concrete API, followed by an example in Section 3.3.

### 3.1 Principle: Sequential Request-centric Programming

TokenRouter is designed to support diverse token-level routing algorithms while keeping their implementation simple and intuitive. Although TokenRouter executes requests through an asynchronous, model-centric runtime, developers should not need to reason about low-level batching, scheduling, or

asynchronous execution. Instead, they should only describe a routing algorithm by tracing how a single request moves among cooperating LLMs.

Therefore, we revisit the common lifecycle of a request in token-level routing algorithms. A request first enters an entry model and performs a decoding step. After the forward pass, an algorithm-specific router decides whether the request should continue locally or be delegated to another model. If the request is delegated, the current model gathers the necessary context, such as the latest generated tokens, and sends it to the target model. The target model receives the request, continues decoding, and repeats the same route-send-receive process. This lifecycle can be written as:

$$\textbf{receive} \rightarrow \text{decode} \rightarrow \textbf{route} \rightarrow \textbf{send} \rightarrow \text{peer } \textbf{receive} \rightarrow \text{peer decode} \rightarrow \text{peer } \textbf{route} \rightarrow \cdots .$$

Based on this observation, TokenRouter lets developers implement token-level routing algorithms by filling in only three components: **route**, **send**, and **receive**. Developers program the algorithm from a simple request-centric view, while the TokenRouter runtime executes it with model-centric subservers. This *request-centric programming, model-centric execution* abstraction is the key principle behind TokenRouter's programming interface.

### 3.2 Core Interface

We now detail the three components that developers implement in TokenRouter.

**route** (`result`) is called after each decoding step of an LLM. It determines a destination index for each request in the batch based on the forward-pass result such as hidden states, next-token logits, and sampled tokens. A destination of 0 means that the request continues decoding locally, while a nonzero destination points to the corresponding peer model.

**send** (`req`) is called when **route** delegates a request. It returns a customized `PeerReq` message that carries the request ID, the suffix of tokens that the peer model has not yet seen, a status field indicating whether the request is still active or has terminated, and any algorithm-specific fields needed by the receiver. A position anchor `req.loc` is also attached to indicate the decoding position before the `PeerReq` is sent. The receiver can use this anchor to determine the minimal suffix that should be sent back. The exact token suffix is algorithm-dependent. For example, a scheduler may exclude or include the just-sampled token depending on whether the peer should regenerate it.

**receive** (`peer_req`) is called when a `PeerReq` arrives from a peer model. It converts the `PeerReq` to the local request format so that the receiver model can continue decoding. By default, **receive** commits all transferred tokens from the `PeerReq`, which is sufficient for most token-level routing algorithms. Developers only need to override **receive** when the handoff semantics are algorithm-specific. For example, an algorithm may attach additional state before resuming decoding.

### 3.3 Running Examples

Figure 3 shows an example of the interface for the small-model side of *R-Stitch* [27]. The SLM delegates high-entropy tokens to the LLM, while the LLM sends low-entropy tokens back.

Table 1 summarizes the routing behavior of the five token-level routing algorithms evaluated in this work. Despite using different routing signals, each is expressible with **route** function. For CITER, R2R, and Co-LLM, Model $i$ invokes Model $j$ upon detecting an uncertain token, while Model $j$ returns control after generating one token. R-Stitch instead adopts a symmetric design, where the two models switch control based on token entropy. ME further generalizes token-level routing to more than two models, selecting the next model after every token according to the ensemble weights.

## 4 System Design

This section presents TokenRouter's model-centric runtime, which aims to efficiently serve token-routed inference. Section 4.1 first gives an overview of the system architecture. Section 4.2 addresses *step desynchronization* through asynchronous tri-loop execution and a handoff-resume mechanism. Section 4.3 addresses *batch admission delay* with delayed batching, which trades a small waiting time for larger same-model batches. Our implementation builds on SGLang [14]; Appendix B and Appendix C describe more routing-specific engineering optimizations.

```python
class RStitchSLM(BaseRouter):
    # self.tau (threshold) and self.peer_idx (LLM subserver index) come from the config.

    def route(self, result):
        # Delegate high-entropy tokens to the LLM peer; keep the rest local.
        probs = result.logits_output.next_token_logits.softmax(dim=-1)
        entropy = -(probs * probs.log()).sum(-1) / math.log(probs.shape[-1])
        destination = torch.zeros_like(entropy, dtype=torch.long)
        destination[entropy > self.tau] = self.peer_idx
        return destination.cpu()

    def send(self, req) -> PeerReq:
        # Drop the just-decoded uncertain token; ask the LLM to regenerate.
        token_ids = req.origin_input_ids + req.output_ids[:-1]
        peer_anchor = req.peer_locs[self.peer_idx] or 0
        return PeerReq(rid=req.rid, token_ids=token_ids[peer_anchor:],
                       loc=len(token_ids), finished=req.finished(), ...)

    # receive falls back to the default and accepts all returned tokens.
```

Figure 3: Small-model side of the R-Stitch [27] in TokenRouter. **route** sends high-entropy tokens to the large model and keeps low-entropy tokens local. **send** excludes the uncertain token and transfers only the peer-unseen suffix, letting the large model regenerate from that position. **receive** uses the default accept-all conversion for returned large-model tokens. The large-model side is identical, except that > is replaced by <= in **route** to dispatch low-entropy tokens to the SLM.

Table 1: Routing behavior of the evaluated token-level routing algorithms. Each entry under Model $i$ or Model $j$ specifies the condition under which that model routes generation to another model.

| **Algorithm** | **#Models** | **Model $i$** | **Model $j$** |
|---|---|---|---|
| CITER [12] | 2 | low-confidence token | after one token |
| R2R [5] | 2 | predicted-divergent token | after one token |
| R-Stitch [27] | 2 | high-entropy token | low-entropy token |
| Co-LLM [10] | 2 | high-deferral token | after one token |
| ME [33] | $\geq$2 | after every token, according to ensemble weights | |

### 4.1 System Overview

TokenRouter's runtime is organized as multiple subservers behind a single external interface, as shown in Figure 2. Internally, each subserver hosts one candidate LLM. It owns a scheduler, an LLM runner with a private KV-cache pool, and three user-defined functions to construct the router, sender, and receiver. Subservers progress independently and communicate through peer requests. Their identical internal layout makes it easy to scale to more LLMs by launching additional subservers.

Externally, TokenRouter exposes a single server interface, which receives client requests and forwards them to a designated entry subserver. The client-side protocol is therefore identical to that of standard single-LLM servers such as vLLM [15] and SGLang [14], regardless of how many LLMs participate in routing behind the interface. This design allows TokenRouter to serve as a drop-in replacement for existing single-LLM servers.

### 4.2 Efficient Serving with Asynchronous Execution

TokenRouter addresses the step desynchronization challenge by adding a decoupled inter-model loop and a handoff-resume mechanism on top of the model-centric architecture. Figure 4(a) illustrates the desynchronization problem that arises when token-level routing directly follows the synchronous design of single-LLM serving systems. If requests at the same logical decoding step are synchronized, the whole batch must wait for the slowest model.

Modern serving systems also support speculative decoding, but its two models run in a tightly coupled, periodic pattern, whereas token-level routing invokes random model access. Forcing routing

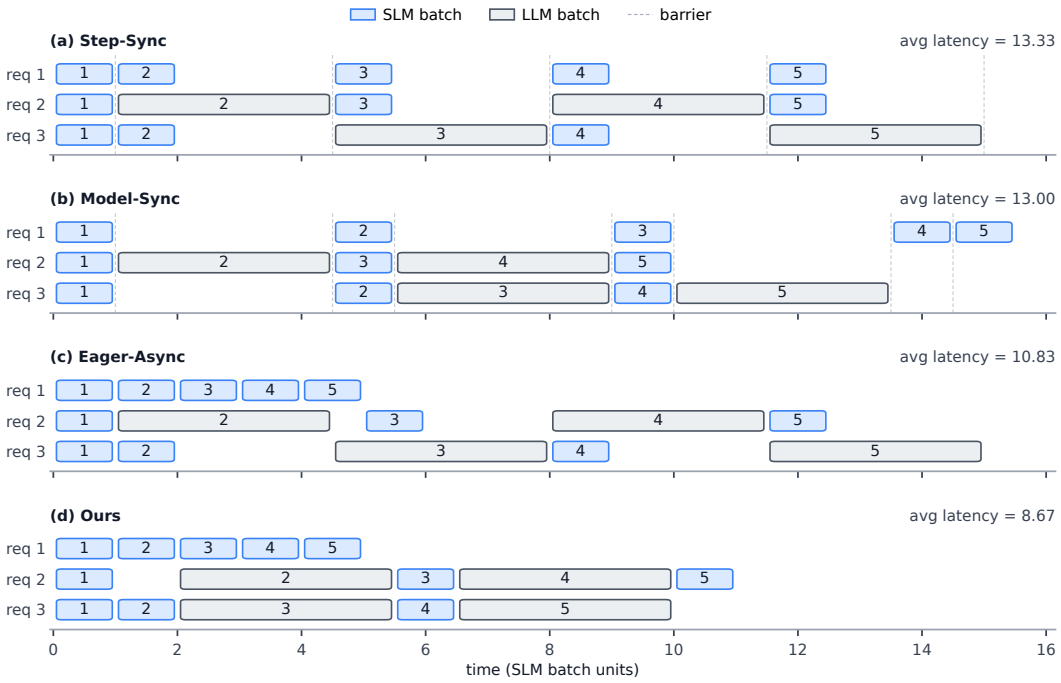



Figure 4: Scheduling strategies of token-level routing. (a) Step-Sync sets step barriers; the next step starts only after all models finish the current one. (b) Model-Sync runs one model at a time; the inactive model's requests stall. (c) Eager-Async processes peer requests immediately, causing large batch admission delay. (d) TokenRouter preserves asynchronous scheduling but delays LLM requests until enough same-model requests accumulate, amortizing the cost across a wider batch.

algorithms to mimic speculative decoding serving would also incur severe idle time (Figure 4(b)). To avoid global synchronization, TokenRouter adopts an asynchronous design (Figure 4(c, d)).

**Decoupled tri-loop execution**. To support inter-model handoff, TokenRouter extends the standard execution structure of single-LLM serving with an additional decoupled loop. As shown in Figure 2, a standard server has two asynchronous loops: a *client-server loop* (purple) for request admission and response streaming, and a *decoding loop* (blue) for request scheduling and model execution. TokenRouter adds a third decoupled loop, the *inter-model loop* (black), to send and receive peer requests between subservers. This third loop allows a routed request to leave the local batch and resume when the peer request returns. While the routed request is being processed by the peer model, the remaining local requests continue executing at the pace of the local decoding loop. When the peer request returns, the scheduler admits the resumed request back into the local batch. Thus, each subserver progresses at its own pace, while routed requests move asynchronously across models.

**Handoff and resume across models**. TokenRouter designs a handoff and resume mechanism for the decoupled inter-model loop. The naive handoff treats a returning request as a fresh call, which redoes prefix matching and KV-cache allocation, incurring unnecessary costs. To deal with this, TokenRouter introduces a *pending* state, an intermediate status between the two standard states *running* and *finished*. A *pending* request is skipped by the scheduler, but its serving state is preserved. At handoff, the routed request is marked as *pending*. Upon resumption, *pending* is toggled to *running* or *finished* according to the peer's output. If not finished, TokenRouter appends new tokens to the request and admits it into the local batch as if it had never left. By skipping prefix matching and KV reallocation, handoff-resume reduces inter-model transitions to little more than a token append.

### 4.3 Efficient Scheduling with Delayed Batching

While asynchronous execution removes synchronization across subservers, each subserver still needs a scheduling policy that handles sparse and irregular routed-token arrivals. We define *batch admission delay* as the waiting time between a request's arrival at a subserver and the start of its execution.

Table 2: Model pairs and benchmarks used by different token-level routing algorithms.

| System | Model Pair | Benchmark |
|---|---|---|
| R2R | DeepSeek-R1-Distill-Qwen-1.5B / 32B | AIME [35] |
| CITER | Qwen2-1.5B / 72B | CommonsenseQA [36] |
| Co-LLM | LLaMA2-7B (tuned) / 70B | GSM8K [37] |
| R-Stitch | L1-1.5B-Short / QwQ-32B | AIME [35] |

Figure 4(c) illustrates the issue under eager asynchronous scheduling. If a subserver starts a new decoding step as soon as any peer request arrives, requests that arrive while the model is already processing a batch cannot be admitted until that step finishes. For example, request 3's token 3 cannot start until the LLM finishes decoding request 2's token 2, incurring the admission delay from time 2 to time 4.5. Because token-level routing causes frequent model switches and high per-step latency variance, such admission stalls occur repeatedly and lead to fragmented batching.

**Delayed-batching scheduler**. To reduce admission delay, we propose a delayed-batching scheduler. The key idea is to wait for a short, controlled interval so that more requests targeting the same model can be executed together. Specifically, the scheduler of model $i$ sbuffers received requests and launches a batch only when the buffer size reaches a threshold $B_i$. Figure 4(d) shows an example where the SLM subserver uses $B_0 = 1$, while the LLM subserver uses $B_1 = 2$. Although an early-arriving request waits longer than under eager-async in Figure 4(c), later-arriving requests can be admitted with much less waiting, so the workload's average batch admission delay drops.

**Throughput-optimal threshold**. TokenRouter analytically determines the throughput-optimal delayed-batching threshold $B_i^*$. This requires balancing a strict trade-off: too small $B_i$ causes large batch admission delay, while too large $B_i$ traps requests in the queue. We model the routing process as a Discrete-Time Markov Chain (DTMC). Given a token-level routing algorithm with $M$ models and token-commit matrix $\mathbf{C}$, concurrency $N$, routing probability $\mathbf{P}$, and per-step decoding latency $L_i$ of each LLM, our model derives the throughput as a function of $B$ and searches the threshold $B^*$ that maximizes the throughput. Full details are provided in Appendix D.

# 5 Experiment

## 5.1 Setup

**Token-level routing algorithms**. We evaluate five token-level routing algorithms in TokenRouter, namely CITER [12], R2R [5], R-Stitch [27], Co-LLM [10], and ME [33]. CITER, R2R, R-Stitch, and Co-LLM route between Qwen3-0.6B and Qwen3-32B [34], while ME routes among Qwen3-0.6B, Qwen3-8B, and Qwen3-32B. For CITER, we train the router and set $\tau = 0.98$. For R2R, we use the official router and threshold. For R-Stitch, we set $\tau = 0.03$. For Co-LLM, we fine-tune Qwen3-0.6B and set $\eta = 0.15$. For ME, we set the ensemble weight to 0.5, 0.3, and 0.2. We also evaluate each algorithm using the model pair and the dataset from its original paper, as summarized in Table 2.

**Hardware**. All experiments run on an 8×A100-80G GPU server. For two-model algorithms, the small model is deployed on one GPU and the large model on two GPUs with tensor parallelism, with two models sharing two GPUs via CUDA MPS. For ME, each model is deployed on one GPU.

**Baselines**. We compare TokenRouter against two baselines. *Official Code* refers to the algorithm's released implementation, available for all but R-Stitch (whose code is not released). Among these, Co-LLM uses a per-model vLLM [15] server with continuous batching, while CITER and R2R are inference engines without continuous batching. *Std. Serving* is a standard SGLang-based baseline that we build for a fair comparison. It uses one SGLang server per LLM; each server returns one token at a time, and an external dispatcher routes each token between servers.

**Workloads**. We evaluate three workloads spanning different input/output lengths. (1) For *Low-effort reasoning* (short input, short output), we use AIME2024 [35] prompts (∼100 tokens) with a maximum output length of 2,048. (2) For *High-effort reasoning* (short input, long output), we use AIME2024 prompts with a maximum output length to 8,192, keeping only problems whose Qwen3-32B solutions exceed 8,192 tokens. (3) For *Agentic* tasks (multi-turn), we use SWE-Smith [38] trajectories of around 8,192 input tokens with a maximum output length of 1,024.

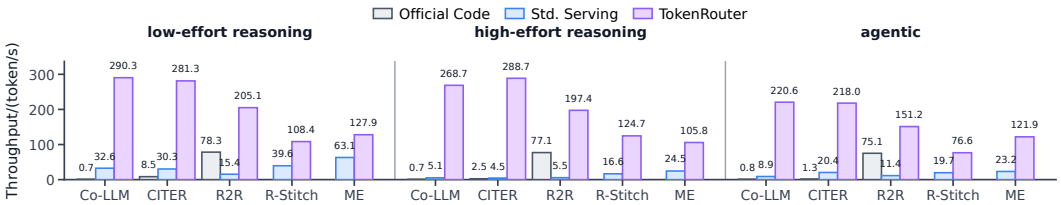



Figure 5: Throughput (token/s) comparison across serving systems at concurrency 4.

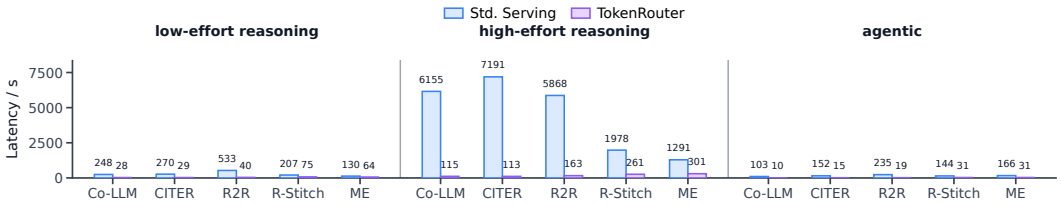



Figure 6: Latency (s) comparison across serving systems at concurrency 4.

**Metrics**. We report the throughput and the end-to-end request completion latency. For inference engines, we form a batch of $N$ requests and process the batch. For serving systems, we launch $N$ workers, each of which issues a new request as soon as the previous one completes. We measure throughput as the total number of generated output tokens divided by elapsed time. For agentic workloads, end-to-end latency is measured separately for each turn.

### 5.2 Serving Efficiency

As shown in Figure 5 and 6, across all 15 algorithm–workload combinations, TokenRouter consistently outperforms the baselines, improving throughput by 2.01–64.15× compared with the stronger baseline. TokenRouter also reduces the end-to-end latency of Std. Serving by 2.03–63.64×. Although Std. Serving substantially improves upon the official Co-LLM and CITER implementations through asynchronous execution, it lacks TokenRouter's handoff-and-resume mechanism. Consequently, the serving state of a routed request cannot be preserved across model switches, and each returning token must be treated as a new request. This incurs substantial overhead for operations such as prefix matching and KV-cache updates. We further break down the per-step latency of Std. Serving to quantify this overhead in Table 10.

TokenRouter also scales gracefully with output length. When we compare low-effort and high-effort reasoning workloads (max output 2,048 vs. 8,192 tokens), TokenRouter's throughput holds steady while Std. Serving loses 58.1–85.2% of its throughput.

When evaluated on the original settings of the token-level routing algorithms, as shown in Table 3, TokenRouter reduces latency by 2.78–21.61× over the official implementations while substantially improving throughput by 2.73–21.97×. These results show that TokenRouter efficiently supports diverse token-level routing algorithms under their original algorithmic settings.

**Throughput–Speed Trade-off.** Figure 1b evaluates the trade-off between throughput and per-user speed as the concurrency increases from 1 to 16. When the concurrency increases from 1 to 16, TokenRouter's throughput increases by 8.61×, while retaining 51.7% of its single-user speed. In contrast, official R2R reaches only 5.14× and retains 31.3% of its single-user speed. Notably, at concurrency 16, TokenRouter achieves 18.58× higher throughput than official R2R at concurrency 1, while still providing 1.13× higher per-user speed. This means that, TokenRouter delivers substantially higher throughput even under a stricter per-user speed Service Level Objective (SLO).

### 5.3 Ablation Study

**Gain breakdown**. Figure 1a and Table 5 breaks down TokenRouter's throughput. Engineering optimizations with extended CUDA graphs raise throughput to 1.71–2.02× the official R2R implementation. On top of these gains, asynchronous execution and delayed batching further increase

Table 3: TTFT, throughput, and end-to-end latency at concurrency 4, comparing across serving systems under original settings.

| Algorithm | Implementation | Throughput/(token/s) | TTFT/s | Latency/s |
|---|---|---|---|---|
| R2R | LLM-only | 145.30 | 0.13 | 411.98 |
| | Official Code | 89.62 | 0.11 | 751.15 |
| | TokenRouter | 244.56 | 0.11 | **270.19** |
| CITER | LLM-only | 123.72 | 0.083 | 2.68 |
| | Official Code | 17.16 | 0.036 | 7.64 |
| | TokenRouter | 149.31 | 0.036 | **0.48** |
| Co-LLM | LLM-only | 134.79 | 0.069 | 13.39 |
| | Official Code | 3.46 | 1.14 | 247.61 |
| | TokenRouter | 76.02 | 0.067 | **11.46** |
| R-Stitch | LLM-only | 150.41 | 0.15 | 360.76 |
| | Official Code | N/A | N/A | N/A |
| | TokenRouter | 140.58 | 0.067 | **151.30** |

| SLM | LLM | R2R | TokenRouter | Gain |
|---|---|---|---|---|
| 0.6B | 8B | 105.1 | 336.98 | 3.21× |
| 0.6B | 32B | 77.08 | 197.40 | 2.56× |
| 1.7B | 8B | 103.37 | 285.41 | 2.76× |
| 4B | 8B | 106.67 | 211.81 | 1.99× |

Table 4: R2R vs. TokenRouter across SLM-LLM pairs. Throughput is reported in token/s.

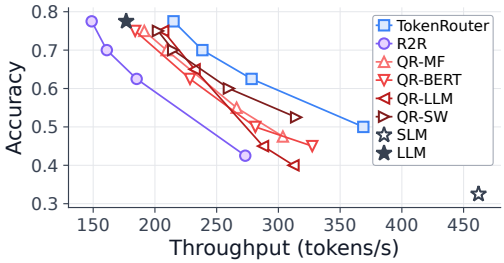



Figure 7: Throughput–accuracy trade-off. QR denotes query-level routing from RouteLLM.

throughput, reaching 2.76–2.81× overall at $N = 4$ and $N = 8$. These results demonstrate that TokenRouter's overall performance gain stems from the combined benefits of these optimizations.

**Effect of model pairs**. Table 4 evaluates TokenRouter against the most competitive R2R system baseline across diverse model pairs. Compared with the official implementation of R2R, TokenRouter achieves 1.99–3.21× throughput improvement across different model pairs. These results suggest that TokenRouter's optimizations are effective across different SLM and LLM scales.

**Parallel strategy**. Table 6 studies how tensor-parallelism (TP) configurations for the SLM and LLM affect throughput. Taking the two-GPU setting as an example, increasing the LLM TP size from 1 to 2 improves throughput by 40.6% at $N = 1$ and 22.8% at $N = 4$. In contrast, once the LLM uses TP size 2, further increasing the SLM TP size from 1 to 2 brings only 0.8% and 0.4% throughput improvements, respectively. This indicates that additional TP resources are substantially more beneficial to the LLM than to the SLM. Therefore, we prioritize tensor parallelism for the LLM and serve the SLM with TP size 1 by default throughout the paper.

### 5.4 Advancing the Pareto Frontier

We evaluate the throughput-accuracy trade-off of TokenRouter, the official R2R implementation, and query-level routing baselines on AMC23, using the Qwen3-0.6B and Qwen3-32B pair with greedy decoding and a maximum output length of 8,192 tokens. All requests are sent concurrently; throughput is computed per request as output tokens divided by elapsed time; and SLM/LLM memory fractions are held equal across systems for a fair comparison. Figure 7 shows that TokenRouter shifts token-level routing to a new Pareto frontier, making fine-grained routing more competitive than coarse-grained routing on state-of-the-art frameworks such as SGLang.

Table 5: Ablation study of different optimization components. Throughput is reported in token/s.

| **Method** | $N = 8$ | $N = 4$ | $N = 1$ |
|---|---|---|---|
| TokenRouter | **372.48** | **210.61** | **73.02** |
| - Delayed batching | 296.86 | 161.26 | 73.02 |
| - Async execution | 230.79 | 143.88 | 73.02 |
| - Router CUDA graph | 228.10 | 142.33 | 72.62 |
| - LLM CUDA graph | 164.08 | 95.55 | 51.22 |
| - SLM CUDA graph ($\approx$ R2R) | 132.78 | 75.54 | 34.20 |
| R2R official code | 134.89 | 73.27 | 36.17 |

Table 6: Throughput across parallel strategies. Throughput is reported in token/s.

| **#GPU** | **TP$_{\text{SLM}}$** | **TP$_{\text{LLM}}$** | $N$=1 | $N$=4 |
|---|---|---|---|---|
| 2 | 1 | 1 | 50.37 | 160.74 |
| | 1 | 2 | 70.83 | 197.40 |
| | 2 | 2 | 71.41 | 198.13 |
| 8 | 4 | 4 | 93.23 | 284.62 |
| | 1 | 8 | 109.67 | 296.34 |
| | 2 | 8 | 106.02 | 289.08 |
| | 4 | 8 | 110.02 | 297.63 |
| | 8 | 8 | 105.59 | 288.35 |

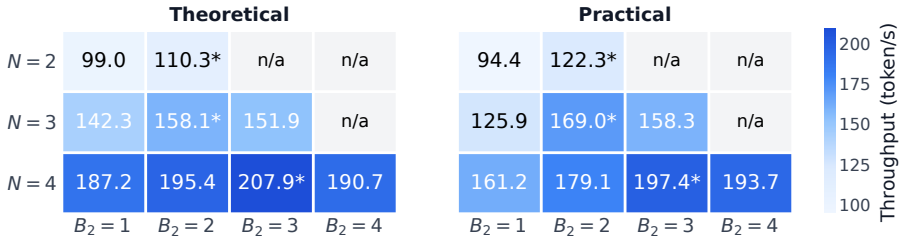



Figure 8: Predicted and measured R2R throughput under different concurrency levels and delayed-batching thresholds. We fix the SLM threshold at $B_0 = 1$ and vary the LLM threshold $B_1$. An asterisk (*) marks the highest throughput for each concurrency level.

### 5.5 Empirical Validation of the Throughput Model

We empirically validate the throughput model using R2R, where model 0 denotes the SLM and model 1 the LLM. For R2R, the routing-probability matrix $\mathbf{P}$ and token-commit matrix $\mathbf{C}$ are given by

$$\mathbf{P} = \begin{bmatrix} 1-p & p \\ 1 & 0 \end{bmatrix}, \qquad \mathbf{C} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}. \tag{1}$$

Under the setup described in Section 5.1, $p = 0.35$. The measured decoding-step latencies of two models are $L_0 = 6.0\,\text{ms}$ for the SLM and $L_1 = 27.9\,\text{ms}$ for the LLM.

Figure 8 fixes $B_0 = 1$ and compares the predicted throughput against measurements obtained on hardware while varying $B_1$. The model closely captures the throughput trends across both concurrency levels and batching thresholds. More importantly, at every evaluated concurrency level, the threshold predicted to maximize throughput coincides with the empirically optimal threshold. These results show that the proposed model accurately captures the effect of delayed batching on system throughput and can guide the selection of throughput-optimal batching thresholds.

## 6 Conclusion

We presented TokenRouter, an efficient serving system for token-level LLM routing that decouples routing algorithms from serving-system optimization. TokenRouter lets developers express routing semantics through a request-centric route-send-receive interface, while the runtime handles asynchronous model-centric execution. We overcome the step desynchronization and batch admission delay challenges for efficient serving with asynchronous execution and the delayed-batching scheduler. Across routing algorithms, workloads, and model pairs, TokenRouter consistently achieves 2.01–64.15$\times$ higher decoding throughput than existing implementations, providing a strong foundation for future token-routing research.

**Limitation and future work**. Our mathematical model assumes that the number of tokens between two consecutive sends follows an geometric distribution, which holds for most token-level routing algorithms. Handling corner cases that violate this assumption is left for future work.

## Acknowledgments and Disclosure of Funding

This work was supported by National Natural Science Foundation of China (No. 62325405, 62104128, U19B2019, U21B2031, 61832007, 62204164, 92364201), Tsinghua EE Xilinx AI Research Fund, and Beijing National Research Center for Information Science and Technology (BNRist). We thank for all the support from Infinigence-AI.

## References

[1] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

[2] Cursor Team. Introducing cursor router. `https://cursor.com/blog/router`, July 2026. Accessed: 2026-07-25.

[3] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[4] Tianyu Fu, Yichen You, Zekai Chen, Guohao Dai, Huazhong Yang, and Yu Wang. Think-at-hard: Selective latent iterations to improve reasoning language models. *arXiv preprint arXiv:2511.08577*, 2025.

[5] Tianyu Fu, Yi Ge, Yichen You, Enshu Liu, Zhihang Yuan, Guohao Dai, Shengen Yan, Huazhong Yang, and Yu Wang. R2r: Efficiently navigating divergent reasoning paths with small-large model token routing. *arXiv preprint arXiv:2505.21600*, 2025.

[6] Chengsong Huang, Tong Zheng, Langlin Huang, Jinyuan Li, Haolin Liu, and Jiaxin Huang. Relayllm: Efficient reasoning via collaborative decoding. *arXiv preprint arXiv:2601.05167*, 2026.

[7] Lifeng Jin, Baolin Peng, Linfeng Song, Haitao Mi, Ye Tian, and Dong Yu. Collaborative decoding of critical tokens for boosting factuality of large language models. *arXiv preprint arXiv:2402.17982*, 2024.

[8] Siteng Liao, Yuzhu Liang, Hengzhong Rao, Xizhao Luo, and Tian Wang. Alpharouter: Token-level routing between slm and llm with reinforcement learning and tree search. In *Forty-third International Conference on Machine Learning*, 2026.

[9] Arushi Rai, Qiang Zhang, Hanqing Zeng, Yunkai Zhang, Dipesh Tamboli, Xiangjun Fan, and Zhuokai Zhao. Taro: Token-level adaptive routing for llm test-time alignment. *arXiv preprint arXiv:2603.18411*, 2026.

[10] Zejiang Shen, Hunter Lang, Bailin Wang, Yoon Kim, and David Sontag. Learning to decode collaboratively with multiple language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12974–12990, 2024.

[11] Nuoya Xiong, Yuhang Zhou, Hanqing Zeng, Zhaorun Chen, Furong Huang, Shuchao Bi, Lizhu Zhang, and Zhuokai Zhao. Token-level llm collaboration via fusionroute. *arXiv preprint arXiv:2601.05106*, 2026.

[12] Wenhao Zheng, Yixiao Chen, Weitong Zhang, Souvik Kundu, Yun Li, Zhengzhong Liu, Eric P Xing, Hongyi Wang, and Huaxiu Yao. Citer: Collaborative inference for efficient large language model decoding with token-level routing. *arXiv preprint arXiv:2502.01976*, 2025.

[13] Tianyu Fu, Zihan Min, Hanling Zhang, Jichao Yan, Guohao Dai, Wanli Ouyang, and Yu Wang. Cache-to-cache: Direct semantic communication between large language models. *arXiv preprint arXiv:2510.03215*, 2025.

[14] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[15] 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*, pages 611–626, 2023.

[16] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance, 2023. URL `https://arxiv.org/abs/2305.05176`.

[17] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing, 2024. URL https://arxiv.org/abs/2404.14618.

[18] Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam Mausam. Automix: Automatically mixing language models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 131000–131034. Curran Associates, Inc., 2024. doi: 10.52202/079017-4164. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/ecda225cb187b40ea8edc1f46b03ffda-Paper-Conference.pdf.

[19] Zesen Zhao, Shuowei Jin, and Z. Morley Mao. Eagle: Efficient training-free router for multi-llm inference, 2024. URL https://arxiv.org/abs/2409.15518.

[20] Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. Routerbench: A benchmark for multi-llm routing system, 2024. URL https://arxiv.org/abs/2403.12031.

[21] Zhongzhan Huang, Guoming Ling, Yupei Lin, Yandong Chen, Shanshan Zhong, Hefeng Wu, and Liang Lin. Routereval: A comprehensive benchmark for routing llms to explore model-level scaling up in llms, 2025. URL https://arxiv.org/abs/2503.10657.

[22] Keming Lu, Hongyi Yuan, Runji Lin, Junyang Lin, Zheng Yuan, Chang Zhou, and Jingren Zhou. Routing to the expert: Efficient reward-guided ensemble of large language models, 2023. URL https://arxiv.org/abs/2311.08692.

[23] Tao Feng, Yanzhen Shen, and Jiaxuan You. Graphrouter: A graph-based router for llm selections, 2025. URL https://arxiv.org/abs/2410.03834.

[24] Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities, 2024. URL https://arxiv.org/abs/2406.04692.

[25] Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, Yi Dong, Evelina Bakhturina, Tao Yu, Yejin Choi, Jan Kautz, and Pavlo Molchanov. Toolorchestra: Elevating intelligence via efficient model and tool orchestration, 2025. URL https://arxiv.org/abs/2511.21689.

[26] Yash Akhauri, Anthony Fei, Chi-Chih Chang, Ahmed F AbouElhamayed, Yueying Li, and Mohamed S Abdelfattah. Splitreason: Learning to offload reasoning. *arXiv preprint arXiv:2504.16379*, 2025.

[27] Zhuokun Chen, Zeren Chen, Jiahao He, Lu Sheng, Mingkui Tan, Jianfei Cai, and Bohan Zhuang. R-stitch: Dynamic trajectory stitching for efficient reasoning. *arXiv preprint arXiv:2507.17307*, 2025.

[28] Wenhao Zeng, Xuteng Zhang, Yuling Shi, Chao Hu, Yuting Chen, Beijun Shen, and Xiaodong Gu. Glimprouter: Efficient collaborative inference by glimpsing one token of thoughts. *arXiv preprint arXiv:2601.05110*, 2026.

[29] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

[30] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023. URL https://arxiv.org/abs/2211.17192.

[31] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024. URL https://arxiv.org/abs/2401.10774.

[32] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty, 2025. URL https://arxiv.org/abs/2401.15077.

[33] Jiale Fu, Yuchu Jiang, Peijun Wu, Chonghan Liu, Joey Tianyi Zhou, and Xu Yang. Rethinking llm ensembling from the perspective of mixture models. *arXiv preprint arXiv:2605.00419*, 2026.

[34] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[35] AIME. American invitational mathematics examination, 2025. URL `https://artofproblemsolving.com/`.

[36] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4149–4158, 2019.

[37] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[38] John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

# A Additional Experiment Results

## A.1 Additional Concurrency

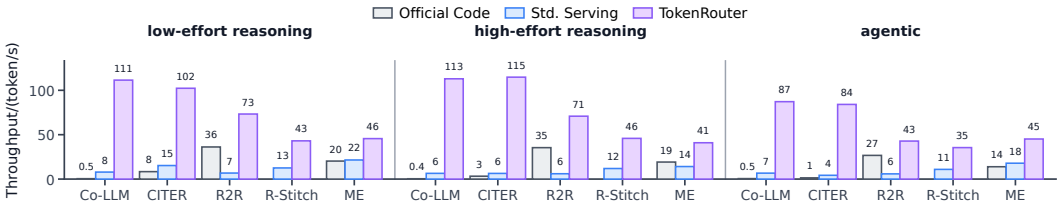



Figure 9: Throughput (token/s) comparison across serving systems at concurrency 1.

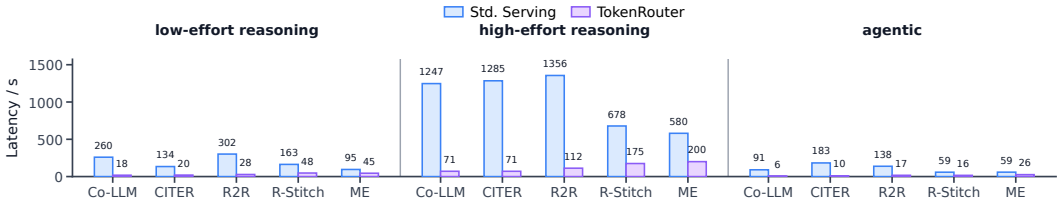



Figure 10: Latency (s) comparison across serving systems at concurrency 1.

Figure 9 and 10 compare different implementations of five token-level routing algorithms across three workloads at concurrency 1. TokenRouter consistently outperforms standard serving in all settings, achieving 2.12–19.74× higher throughput and a 2.12–18.83× latency speedup; it also provides 1.60–19.74× higher throughput than the stronger available baseline between the official code and standard serving.

As shown in Table 7, under the original settings of each token-level routing algorithm at concurrency 1, TokenRouter consistently improves serving efficiency over the official implementations. It achieves 1.93–10.20× higher throughput and reduces end-to-end latency by 2.30–10.09× for R2R, CITER, and Co-LLM, while maintaining comparable TTFT.

Table 7: TTFT, throughput, and end-to-end latency at concurrency 1, comparing across serving systems under original settings.

| Algorithm | Implementation | Throughput/(token/s) | TTFT/s | Latency/s |
|---|---|---|---|---|
| R2R | LLM-only | 40.17 | 0.074 | 388.12 |
| | Official Code | 54.77 | 0.12 | 330.69 |
| | TokenRouter | 105.81 | 0.11 | **144.10** |
| CITER | LLM-only | 34.40 | 0.055 | 2.42 |
| | Official Code | 14.26 | 0.031 | 2.41 |
| | TokenRouter | 50.63 | 0.031 | **0.36** |
| Co-LLM | LLM-only | 36.06 | 0.049 | 12.66 |
| | Official Code | 2.80 | 0.34 | 76.68 |
| | TokenRouter | 28.57 | 0.068 | **7.60** |
| R-Stitch | LLM-only | 40.82 | 0.072 | 341.84 |
| | Official Code | N/A | N/A | N/A |
| | TokenRouter | 55.78 | 0.102 | **119.29** |

### A.2 Additional Deployment Configurations

#### A.2.1 Non-overlapping GPU Deployment

We deploy Qwen3-0.6B (1 GPU) and Qwen3-32B (TP=2, 2 GPUs) on separate GPUs, so that the LLM in token-routing uses the same resources as the LLM-only baseline. This configuration improves the throughput of TokenRouter by 6.43% on average compared with the original overlapping setting.

Table 8: Throughput (token/s) comparison across routing strategies under different concurrency and overlap settings.

| Setup | | Routing Algorithm | | | |
|---|---|---|---|---|---|
| $N$ | **Deployment** | **R2R** | **R-Stitch** | **Co-LLM** | **CITER** |
| 1 | Overlapping | 73.27 | 43.06 | 47.46 | 102.29 |
| | Non-overlapping | 74.90 | 47.53 | 48.25 | 108.65 |
| 4 | Overlapping | 205.05 | 108.42 | 165.96 | 281.27 |
| | Non-overlapping | 228.94 | 116.14 | 170.20 | 308.36 |

#### A.2.2 Cross-node Deployment

We evaluate TokenRouter under cross-node deployment setups. Specifically, we deploy Qwen3-0.6B (1 GPU) and Qwen3-32B (2 GPUs, TP=2) on one or two nodes and test the throughput under the low-effort reasoning setup. The two nodes are connected via RoCE at 3.7 GB/s. As shown in Table 9, cross-node throughput is comparable to single-node throughput with the same computational resources, translating to 1.93–32.81$\times$ improvements over the respective official implementations.

Table 9: Throughput comparison across single-node and multi-node settings.

| Setup | | Routing Algorithm | | |
|---|---|---|---|---|
| $N$ | **Deployment** | **R-Stitch** | **R2R** | **CITER** |
| 1 | Single-node | 47.53 | 74.90 | **108.65** |
| | Multi-node | 46.19 | 69.92 | **93.99** |
| 4 | Single-node | 116.14 | 228.94 | **308.36** |
| | Multi-node | 104.20 | 202.43 | **277.56** |

### A.3 Latency Breakdown of the Std. Serving Baseline

To understand the gap over the CUDA-graph-enabled Std. Serving baseline, we profile the average SLM-side runtime per decoding step over the final 128 steps of a 2,048-token generation. As shown in Table 10, the majority of latency is attributed to prefix matching and RadixCache maintenance. This overhead stems from the single-model assumption in Std. Serving, where each returning request is treated as a new extend operation from the user side, incurring repeated prefix matching and KV-cache bookkeeping. In contrast, TokenRouter preserves the request-level KV-cache state across model handoffs and directly resumes execution when peer results return, thereby substantially reducing such overhead.

## B Token-Routing-Specific Optimizations

Beyond the core mechanisms in Section 4, TokenRouter adapts several single-LLM serving assumptions that token-level routing breaks. Each subsection first identifies the broken assumption and then describes TokenRouter's fix.

**Automatic KV-cache memory allocation**. Single-LLM servers [14, 15] assume one model owns the entire KV-cache budget on the GPU. Token-level routing co-locates cooperating LLMs on the

Table 10: Latency breakdown of the SLM-side execution per decoding step in Std. Serving.

| Operation | Latency (ms) | Ratio |
|---|---|---|
| Prefix matching | 7.78 | 20.94% |
| Locking cache nodes | 6.49 | 17.47% |
| SLM inference | 1.56 | 4.20% |
| Releasing locks | 5.13 | 13.81% |
| Update radix cache | 12.12 | 32.62% |
| Others | 4.08 | 10.98% |

same GPU and forces them to contend for this budget. Over-reserving per model then causes out-of-memory at full concurrency, while under-reserving idles capacity. TokenRouter resolves this by treating per-model KV reservations as a joint allocation. Given each model's weight footprint and per-token KV-cell size, the runtime computes the largest common KV-token capacity that fits within a configurable shared GPU budget and assigns each subserver its share before launch. This eliminates manual reservation tuning.

**CUDA-graph capture for extend mode**. Single-LLM servers capture CUDA graphs only for decode batches, where the per-step shape is stable. Under token-level routing, however, every peer return arrives as an extend batch, so extend mode (rather than decode) dominates the routing-handoff path. TokenRouter therefore captures CUDA graphs over a configurable range of extend batch sizes and token lengths and falls back to the standard non-graph extend path when a batch exceeds this range. Since the common case stays within the captured range, per-handoff kernel-launch overhead is eliminated on the dominant path while long extends remain correctly served.

**In-process router**. In single-LLM and query-level serving, the router (when present) runs once per request, so a cross-process invocation is acceptable. Token-level routing inverts this scaling, since the **route** function runs after every forward pass on every active request, orders of magnitude more frequently. A cross-process router would add an IPC round-trip per token and serialize routing against the next decoding step. TokenRouter therefore co-locates the router with the scheduler on the same GPU as the LLM runner, so each route decision becomes an in-process tensor op.

**Global state tracking**. The delayed-batching threshold (Section 4.3) depends on the system-wide active request count, so all subservers must agree on this count. Otherwise $B^*$ would be selected against stale concurrency and drift from optimal. Rather than adding a central coordinator, TokenRouter piggybacks status updates on the peer-message channel. Every newly admitted or finished request is announced as a control message alongside data tokens, keeping each subserver's view consistent without separate coordination infrastructure.

## C Implementation Details

TokenRouter is implemented on top of SGLang [14] 0.5.1. The Server Interface is adapted from SGLang's FastAPI front-end to support multi-model port arguments, and inter-subserver messages travel over local ZeroMQ IPC. Each subserver runs its three-stage iteration as a single event loop in a dedicated subprocess, and the only inter-subserver coordination is the peer-message channel of Section 4.2. All evaluated routing algorithms (Section 5) share this runtime, each implemented as a scheduler subclass over the three functions of Section 3.

## D Mathematical Modeling for Delayed-Batching Optimization

We develop a mathematical model to optimize the delayed-batching threshold introduced in Section 4.3. We first formulate the problem in Section D.1, then derive the steady-state system throughput in Section D.2 and optimize the batching threshold in Section D.3. Finally, we use R2R [5] as a representative case study to validate the model in Section 5.5. The key notation used throughout the analysis is summarized in Table 11.

Table 11: Key notations used throughout the analysis.

| Notation | Definition |
|---|---|
| $M$ | Number of cooperating models. |
| $N$ | Number of concurrent requests in the system. |
| $L_i$ | Latency of one decoding step on model $i$. |
| $p_{ij}$ | Probability that a request is routed from model $i$ to model $j$ after a decoding step. |
| $c_{ij}$ | Number of output tokens committed when a request is routed from model $i$ to model $j$. |
| $B_i$ | Delayed-batching threshold of model $i$. |
| $k_i$ | Number of requests waiting at model $i$. |
| $b_i$ | Size of the currently executing batch on model $i$. |
| $r_i$ | Residual execution time of the currently executing batch on model $i$. |
| $\mathcal{R}_{\mathbf{B}}$ | Set of states reachable under delayed-batching threshold $\mathbf{B}$. |
| $\tau(\mathbf{s})$ | Holding time of state $\mathbf{s}$. |
| $\mathcal{A}(\mathbf{s})$ | whose current batches complete at the next state transition. |
| $\mathbf{m_i}$ | Destination-count vector for the batch completed on model $i$. |
| $\mathbf{T_B}, \boldsymbol{\pi}_{\mathbf{B}}$ | Transition matrix and stationary distribution of the DTMC model under $\mathbf{B}$. |
| $g(\mathbf{s}, \mathbf{M})$ | Number of tokens committed during a state transition from $\mathbf{s}$ with routing outcome $\mathbf{M}$. |
| $T(\mathbf{B})$ | Steady-state aggregate committed-token throughput under delayed-batching threshold $\mathbf{B}$. |

### D.1 Problem Formulation

Consider a token-level routing system consisting of $M$ models, indexed by $i \in \{0, \ldots, M-1\}$, serving $N$ concurrent requests. Let $L_i$ denote the latency of one decoding step on model $i$.

After completing a decoding step on model $i$, a request is routed to model $j$ with probability $p_{ij}$ and commits $c_{ij}$ output tokens. We define the routing transition matrix and the token-commit matrix as

$$\mathbf{P} = [p_{ij}]_{M \times M}, \qquad \mathbf{C} = [c_{ij}]_{M \times M}, \tag{2}$$

where

$$p_{ij} \geq 0, \qquad \sum_{j=0}^{M-1} p_{ij} = 1. \tag{3}$$

In particular, $p_{ii}$ denotes the probability that a request remains on model $i$ for the next decoding step. We assume that the routing probabilities are stationary across decoding steps. If the routing probabilities vary over time, the same analysis applies over a short time interval during which they can be approximated as constant. Under this assumption, for $p_{ii} < 1$, the number of consecutive decoding steps a request executes on model $i$ before transitioning to another model follows a geometric distribution with parameter $1 - p_{ii}$. We further assume that routing decisions are independent across requests.

To improve efficiency, each model $i$ employs delayed batching with threshold $B_i$: a new batch is launched only when at least $B_i$ requests are available in its waiting queue. We denote the threshold vector by

$$\mathbf{B} = (B_0, \ldots, B_{M-1}). \tag{4}$$

Our objective is to characterize the steady-state throughput of TokenRouter as a function of $\mathbf{B}$ and determine the throughput-optimal threshold vector.

### D.2 Derivation of System Throughput

#### D.2.1 Problem Modeling: Discrete-Time Markov Chain

We represent the system state as

$$\mathbf{s} := (k_0, b_0, r_0, \ldots, k_{M-1}, b_{M-1}, r_{M-1}), \tag{5}$$

where $k_i$ is the number of requests waiting at model $i$, $b_i$ is the size of model $i$'s currently running batch, and $r_i$ the residual execution time of that batch, measured immediately after the most recent

batch-completion event in the system. If model $i$ has no running batch, we set $b_i = r_i = 0$. Under a delayed-batching threshold vector $\mathbf{B}$, the state space is

$$\begin{aligned}\mathcal{S}_{\mathbf{B}} := \Big\{ \mathbf{s} \in \mathbb{R}^{3M} : \sum_{i=0}^{M-1} (k_i + b_i) = N,\ k_i \in \mathbb{N},\ b_i \in \mathbb{N}, \\ b_i = 0 \Rightarrow k_i < B_i,\ r_i = 0, \\ b_i > 0 \Rightarrow b_i \geq B_i,\ 0 < r_i \leq L_i \Big\}.\end{aligned} \tag{6}$$

Since the residual execution times are continuous, $\mathcal{S}_{\mathbf{B}}$ is generally infinite. To obtain a finite-state model, we introduce a common time quantum $\delta$ and assume that the decoding-step latency of each model is an integer multiple of $\delta$:

$$L_i = d_i \delta, \qquad d_i \in \mathbb{N},\ i = 0, 1, \ldots, M-1. \tag{7}$$

We further assume that all initial residual times are integer multiples of $\delta$. Then all subsequent residual times remain on the same discrete grid, yielding the finite candidate state space

$$\begin{aligned}\tilde{\mathcal{S}}_{\mathbf{B}} := \Big\{ \mathbf{s} \in \mathbb{R}^{3M} : \sum_{i=0}^{M-1} (k_i + b_i) = N,\ k_i \in \mathbb{N},\ b_i \in \mathbb{N}, \\ b_i = 0 \Rightarrow k_i < B_i,\ r_i = 0, \\ b_i > 0 \Rightarrow b_i \geq B_i,\ r_i \in \{\delta, 2\delta, \ldots, d_i\delta\} \Big\}.\end{aligned} \tag{8}$$

We define a state transition at each batch-completion event, i.e., whenever one or more models complete their current batches. Because the state records the queue length, running batch size, and residual execution time of every model, the distribution of the next state depends only on the current state. Therefore, the system observed at batch-completion epochs forms a finite-state discrete-time Markov chain (DTMC).

For a fixed $\mathbf{B}$ and an initial state $\mathbf{s}_0 \in \tilde{\mathcal{S}}_{\mathbf{B}}$, let $\mathcal{R}_{\mathbf{B}}$ denote the set of states reachable from $\mathbf{s}0$ with positive probability under the routing and scheduling dynamics. In practice, $\mathcal{R}_{\mathbf{B}}$ can be enumerated using breadth-first search (BFS). By construction, $\mathcal{R}_{\mathbf{B}} \subseteq \tilde{\mathcal{S}}_{\mathbf{B}}$ and is therefore finite.

### D.2.2 Transition Matrix and Stationary Distribution

We next characterize the state-transition dynamics of the DTMC and derive its stationary distribution, which will be used to compute the steady-state system throughput.

For a state $\mathbf{s}$, define its holding time as

$$\tau(\mathbf{s}) := \min_{i: b_i > 0} r_i. \tag{9}$$

The set of models whose current batches complete at the next transition is therefore

$$\mathcal{A}(\mathbf{s}) := \{ i : b_i > 0,\ r_i = \tau(\mathbf{s}) \}. \tag{10}$$

For each model $i \in \mathcal{A}(\mathbf{s})$, its batch of $b_i$ requests completes one decoding step and is subsequently routed to the next models. Let $\mathbf{m}_i = (m_{i0}, \ldots, m_{i,M-1})$ be the corresponding destination-count vector, where $m_{ij}$ is the number of requests routed from model $i$ to model $j$. By definition,

$$\sum_{j=0}^{M-1} m_{ij} = b_i. \tag{11}$$

Conditioned on $\mathbf{s}$, $\mathbf{m}_i$ follows a multinomial distribution:

$$P_i(\mathbf{m}_i \mid \mathbf{s}) = \frac{b_i!}{\prod_{j=0}^{M-1} m_{ij}!} \prod_{j=0}^{M-1} p_{ij}^{m_{ij}}, \tag{12}$$

with

$$\mathbb{E}[m_{ij} \mid \mathbf{s}] = p_{ij} b_i. \tag{13}$$

We collect the routing outcomes of all completing models sas $\mathbf{M}(\mathbf{s}) = (\mathbf{m}_i)_{i \in \mathcal{A}(\mathbf{s})}$. Under the independence assumption across requests, their joint probability is

$$P(\mathbf{M} \mid \mathbf{s}) = \prod_{i \in \mathcal{A}(\mathbf{s})} P_i(\mathbf{m}_i \mid \mathbf{s}). \tag{14}$$

After the completed requests are routed and all batches that satisfy their delayed-batching thresholds are launched, the system transitions to

$$\mathbf{s}' = \Phi(\mathbf{s}, \mathbf{M}, \mathbf{B}), \tag{15}$$

where $\Phi$ denotes the deterministic state-update function induced by the routing outcome and the delayed-batching policy.

We define the transition matrix $\mathbf{T_B}$ over the reachable state space $\mathcal{R}_\mathbf{B}$. For any $\mathbf{s}, \mathbf{s}' \in \mathcal{R}_\mathbf{B}$,

$$\mathbf{T_B}(\mathbf{s}, \mathbf{s}') = P(\mathbf{s}_{t+1} = \mathbf{s}' \mid \mathbf{s}_t = \mathbf{s}) = \sum_{\mathbf{M}: \Phi(\mathbf{s}, \mathbf{M}, \mathbf{B}) = \mathbf{s}'} P(\mathbf{M} \mid \mathbf{s}). \tag{16}$$

If the finite-state chain on $\mathcal{R}_\mathbf{B}$ is irreducible, it has a unique stationary distribution $\boldsymbol{\pi}_\mathbf{B}$ satisfying

$$(\mathbf{T}_\mathbf{B}^\top - \mathbf{I})\boldsymbol{\pi}_\mathbf{B} = \mathbf{0}, \qquad \sum_{\mathbf{s} \in \mathcal{R}_\mathbf{B}} \pi_\mathbf{B}(\mathbf{s}) = 1. \tag{17}$$

Empirically, we observe that the reachable-state chain is irreducible for all configurations encountered during optimization.

### D.2.3 Throughput Derivation

We now derive the steady-state aggregate committed-token throughput from the stationary distribution of the DTMC.

For a transition from state $\mathbf{s}$ with routing outcome $\mathbf{M}$, the number of committed tokens is

$$g(\mathbf{s}, \mathbf{M}) := \sum_{i \in \mathcal{A}(\mathbf{s})} \sum_{j=0}^{M-1} c_{ij} m_{ij}. \tag{18}$$

The long-run aggregate committed-token throughput is therefore

$$\begin{aligned} T(\mathbf{B}) &= \lim_{H \to \infty} \frac{\sum_{n=0}^{H-1} g(\mathbf{s}_n, \mathbf{M}_n)}{\sum_{n=0}^{H-1} \tau(\mathbf{s}_n)} \\ &= \frac{\lim_{H \to \infty} \frac{1}{H} \sum_{n=0}^{H-1} g(\mathbf{s}_n, \mathbf{M}_n)}{\lim_{H \to \infty} \frac{1}{H} \sum_{n=0}^{H-1} \tau(\mathbf{s}_n)} \\ &= \frac{\mathbb{E}[g(\mathbf{s}, \mathbf{M})]}{\mathbb{E}[\tau(\mathbf{s})]}. \end{aligned} \tag{19}$$

We first derive the expected number of committed tokens per state transition. Taking the expectation of Equation 18 and applying Equation 13, we obtain

$$\begin{aligned}\mathbb{E}[g(\mathbf{s},\mathbf{M})] &= \mathbb{E}_{\mathbf{s}}[\mathbb{E}_{\mathbf{M}}[g(\mathbf{s},\mathbf{M}) \mid \mathbf{s}]] \\ &= \mathbb{E}_{\mathbf{s}}\left[\sum_{i\in\mathcal{A}(\mathbf{s})}\sum_{j=0}^{M-1} c_{ij}\mathbb{E}_{\mathbf{M}}[m_{ij} \mid \mathbf{s}]\right] \\ &= \mathbb{E}_{\mathbf{s}}\left[\sum_{i\in\mathcal{A}(\mathbf{s})}\sum_{j=0}^{M-1} b_i c_{ij} p_{ij}\right] \\ &= \sum_{\mathbf{s}\in\mathcal{R}_{\mathbf{B}}}\sum_{i\in\mathcal{A}(\mathbf{s})}\sum_{j=0}^{M-1} \pi_{\mathbf{B}}(\mathbf{s}) b_i c_{ij} p_{ij}.\end{aligned} \tag{20}$$

Similarly, the expected time between consecutive state transitions is

$$\mathbb{E}[\tau(\mathbf{s})] = \sum_{\mathbf{s}\in\mathcal{R}_{\mathbf{B}}} \pi_{\mathbf{B}}(\mathbf{s})\tau(\mathbf{s}). \tag{21}$$

Substituting Equation 20 and 21 into Equation 19 gives

$$T(\mathbf{B}) = \frac{\displaystyle\sum_{\mathbf{s}\in\mathcal{R}_{\mathbf{B}}}\sum_{i\in\mathcal{A}(\mathbf{s})}\sum_{j=0}^{M-1} \pi_{\mathbf{B}}(\mathbf{s}) b_i c_{ij} p_{ij}}{\displaystyle\sum_{\mathbf{s}\in\mathcal{R}_{\mathbf{B}}} \pi_{\mathbf{B}}(\mathbf{s})\tau(\mathbf{s})}. \tag{22}$$

### D.3 Delayed-Batching Threshold Optimization

We now optimize the delayed-batching threshold vector $\mathbf{B}$ using the throughput model derived above.

To avoid deadlock states in which no model has enough queued requests to launch a batch, we impose the feasibility condition

$$\sum_{i=0}^{M-1} (B_i - 1) < N. \tag{23}$$

This condition defines the finite feasible set

$$\mathcal{B}_M := \left\{ \mathbf{B} \in \{1,\dots,N\}^M : \sum_{i=0}^{M-1} (B_i - 1) < N \right\}. \tag{24}$$

The throughput-optimal delayed-batching threshold vector is then given by

$$\mathbf{B}^\star = \arg\max_{\mathbf{B}\in\mathcal{B}_M} \frac{\displaystyle\sum_{\mathbf{s}\in\mathcal{R}_{\mathbf{B}}}\sum_{i\in\mathcal{A}(\mathbf{s})}\sum_{j=0}^{M-1} \pi_{\mathbf{B}}(\mathbf{s}) b_i c_{ij} p_{ij}}{\displaystyle\sum_{\mathbf{s}\in\mathcal{R}_{\mathbf{B}}} \pi_{\mathbf{B}}(\mathbf{s}) \min_{i:b_i>0} r_i}, \tag{25}$$

where $\boldsymbol{\pi}_{\mathbf{B}}$is the stationary distribution obtained by solving Equation 17.

## NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

**The checklist answers are an integral part of your paper submission.** They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claimed model-centric design and throughput gain are supported by Section 4 and Section 5.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: Section 6 states the assumptions (geometric distribution approximation) under which the throughput model and optimal $\mathbf{B}^*$ are derived.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: Appendix D provides the formulation and the derivation of $\mathbf{B}^*$.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 4, Section 3 (with Table 1), and Appendix C fully specify the architecture, the four routing algorithms, and the experimental setup.

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The paper includes code in supplemental material.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).

- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Workloads, baselines, and algorithms are listed in Section 5; the delayed-batching threshold $\mathbf{B}^*$ is derived in Appendix D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: The observed speedups exceed measurement noise by a wide margin, rendering statistical significance testing unnecessary.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 5 reports the GPU configurations (2-GPU and 8-GPU setups) and the memory budget governing KV-cache allocation.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work is a serving-system optimization with no human subjects or sensitive data; it conforms to the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: TokenRouter has no direct societal impact.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper releases a serving-system framework with no high-risk pretrained models or datasets.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All used assets (SGLang, vLLM, CITER, R2R, R-Stitch, Co-LLM) are cited at first use and used under their open-source licenses.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [N/A]

Justification: No new datasets or models are released.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This work does not involve crowdsourcing or human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This work does not involve human subjects, so no IRB approval is required.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: The core method development does not involve LLMs as important components.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.