BRIDGESTOP: BRIDGING STREAMING TOKENIZATION AND LLM PREFILL

Anonymous authors
Paper under double-blind review

ABSTRACT

Streaming computation has been increasingly adopted across LLM serving to improve efficiency. However, existing streaming methods mainly optimize individual serving stages, while the tokenization-to-prefill path remains limited by batch-bounded inter-request execution, insufficient bridging between the two stages, and transient efficiency mismatch across them. We propose **BridgeSTOP**, a framework that bridges streaming tokenization and LLM prefill to improve, preserve, and propagate streaming tokenization efficiency downstream and ultimately translate it into end-to-end serving gains. **BridgeSTOP** first introduces batch-free asynchronous parallel tokenization, which extends streaming tokenization from intra-request processing to inter-request execution to reduce batch-bounded synchronization waiting and improve tokenization efficiency. We then establish a token pool that continuously collects streaming token outputs and controls when and how many available tokens are released to downstream prefill, preserving and propagating streaming tokenization efficiency. When upstream tokenization temporarily falls behind, we introduce speculative tokenization as a fast replenishment mechanism to replenish transient token deficits and prevent downstream prefill starvation, while the longest-match speculation strategy improves speculation accuracy and reduces rollback overhead. Experiments on Qwen3-8B across five diverse workloads show that **BridgeSTOP** consistently improves sustainable serving capacity and reduces latency, effectively translating streaming tokenization efficiency into end-to-end serving gains.

1 INTRODUCTION

Streaming is a computation paradigm that incrementally processes fine-grained data as they become available, enabling computation and communication to overlap without waiting for complete data (Fragkoulis et al., 2024). Streaming principles have been embodied in TPU dataflow architectures (Jouppi et al., 2017), real-time GPU pipelines (NVIDIA, 2026), and neuromorphic processors (Davies et al., 2021; Ma et al., 2024), and have also been widely explored algorithmically to reduce latency and computation (Shi et al., 2021; Qian et al., 2024). Together, these advances establish streaming as a paradigm with substantial potential for end-to-end system efficiency.

A growing body of work exploits streaming to improve LLM serving efficiency across its three core stages. For *tokenization*, streaming methods incrementally process the input sequence and expose available tokenization results before the complete request is tokenized, with output granularity ranging from groups of tokens to individual tokens (Jiang & Gong, 2026; Chen et al., 2026). For *prefill*, streaming methods partition progressively available inputs into smaller chunks and incrementally execute prefill, overlapping upstream input processing with downstream computation (Yu et al., 2025; Bachkaniwala et al., 2026; vLLM Team et al., 2026). For *decode*, streaming methods divide decoding into smaller token blocks or execution units and progressively process or expose their results (Wei et al., 2026; Bhendawade et al., 2024). Despite targeting different stages, these methods share the principle of decomposing request-level data or computation into progressively available fine-grained units for incremental processing. These advances demonstrate the potential of streaming to improve efficiency throughout the LLM serving pipeline.

However, isolated streaming optimization at each stage cannot guarantee end-to-end serving efficiency gains because tokenization, prefill, and decode form a sequential pipeline with explicit de-

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.

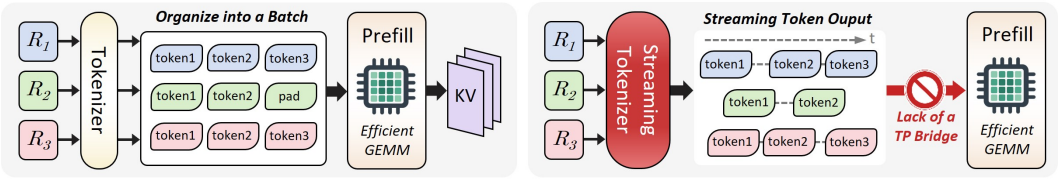

Figure 1: Existing prefill operates on complete token sequences from multiple requests, which can be organized into a regular batched matrix for efficient GEMM. In contrast, streaming tokenization progressively produces fine-grained token outputs with variable release time, but the resulting efficiency cannot be exploited by existing prefill.

pendencies. Such dependencies have been extensively considered between prefill and decode (Yu et al., 2022; Agrawal et al., 2024; Patel et al., 2024), whereas the bridge between tokenization and prefill (TP) remains largely underexplored. We therefore examine streaming along the tokenization-to-prefill path, and identify three problems:

1) Lack of inter-request streaming tokenization: Current streaming tokenization methods focus on intra-request processing by incrementally processing and releasing fine-grained units within each request (Jiang & Gong, 2026; Chen et al., 2026), while execution across requests still commonly follows batch-bounded scheduling with fixed synchronization boundaries. This introduces synchronization waiting across and within batches, as later requests must wait for the current batch to complete, while heterogeneous request lengths cause workers finishing shorter requests to wait for longer ones, reducing tokenization efficiency (Yu et al., 2022; Cong et al., 2025). Therefore, streaming tokenization should be extended beyond individual requests to support inter-request streaming.

2) Lack of a TP bridge: Isolated optimization cannot guarantee end-to-end serving gains because current tokenization methods optimize tokenization in isolation from downstream prefill, making their efficiency gains difficult for prefill to exploit (Feher et al., 2025; Minixhofer et al., 2024). Meanwhile, prefill methods commonly assume that token sequences become available only after a request is fully tokenized or multiple tokenized requests form a batch, incompatible with the streaming paradigm (Kwon et al., 2023; Agrawal et al., 2024). Therefore, a TP bridge is required to connect upstream tokenization with downstream prefill, which can preserve and propagate streaming tokenization efficiency across TP.

3) Lack of transient efficiency mismatch handling across TP: Even under a fixed serving configuration, tokenization and prefill efficiency can vary with transient requests and runtime conditions (Shao et al., 2026; Bachkaniwala et al., 2026). Such dynamic changes in their relative efficiency can disrupt streaming efficiency propagation. When upstream tokenization is temporarily faster, a transient token surplus arises, which supports downstream prefill to continue with sufficient tokens. When upstream tokenization falls behind, however, a transient token deficit can cause prefill starvation and leave compute resources idle, degrading end-to-end serving efficiency (Chung et al., 2026). Therefore, a fast replenishment mechanism is required when upstream tokenization temporarily falls behind, allowing downstream prefill to continue without prolonged starvation.

To address these problems, we develop **BridgeSTOP** to **Bridge** **S**treaming **TO**kenization and LLM **P**refill, which improves, preserves, and propagates streaming tokenization efficiency to downstream prefill and ultimately translates it into end-to-end serving gains. Our method contains three key components. We propose batch-free asynchronous parallel tokenization (BAPT), which removes fixed batch boundaries and allows requests to be independently processed and released, extending streaming tokenization across requests to reduce synchronization waiting and improve tokenization efficiency. We establish a token pool that continuously collects streaming token outputs and controls when and how many available tokens are released to downstream prefill, preserving and propagating streaming tokenization efficiency. We propose speculative tokenization as a fast replenishment mechanism that generates speculative tokens to replenish transient token deficits and prevent prolonged prefill starvation. Together, these designs extend streaming from intra-request tokenization to inter-request execution, bridge streaming tokenization and prefill, and maintain efficient TP execution under transient efficiency mismatch.

To sum up, our contributions are as follows:

- We extend streaming tokenization from intra-request processing to inter-request execution via BAPT, which removes fixed batch boundaries and reduces synchronization waiting, thereby improving tokenization efficiency.
- We formulate the problem of bridging streaming tokenization and prefill in LLM serving, and address it with a token pool that preserves and propagates upstream tokenization efficiency to downstream prefill.
- We identify transient efficiency mismatch across TP as a critical challenge for streaming efficiency propagation, and propose speculative tokenization as a fast replenishment mechanism to prevent transient token deficits from disrupting this propagation.
- We conduct comprehensive experiments with Qwen3-8B across diverse workloads, demonstrating that **BridgeSTOP** effectively translates the efficiency achieved by streaming tokenization into end-to-end serving gains.

2 Related Work

Tokenization. Existing tokenization methods can be broadly categorized as non-streaming or streaming according to their token output behavior. Most existing tokenizers are non-streaming, including subword tokenizers such as BPE and Unigram (Sennrich et al., 2016; Kudo, 2018), rule-based tokenizers (RToks) using handcrafted or heuristic designs (Feher et al., 2025; Oh et al., 2025), and learnable tokenizers (LToks) that predict adaptive token boundaries (Nawrot et al., 2023; Ahia et al., 2024; Zheng et al., 2024). System-oriented methods further improve tokenization execution efficiency (You, 2025; Shao et al., 2026). Recently, streaming tokenizers have emerged with finer-grained token outputs, such as Incremental BPE that emits stable tokens in advance (Jiang & Gong, 2026) and SPEAK that supports causal token boundary prediction and token-by-token output (Chen et al., 2026). However, both non-streaming and streaming tokenization methods primarily focus on processing within individual requests, while execution across multiple requests still commonly remains constrained by batch-bounded scheduling. In contrast, our BAPT extends streaming tokenization from intra-request processing to inter-request execution. Moreover, existing tokenization methods mainly optimize tokenization itself in isolation from downstream prefill, preventing tokenization-side efficiency improvements from being fully translated into end-to-end serving gains (details in Appendix A). In contrast, **BridgeSTOP** bridges streaming tokenization and prefill, preserving and propagating upstream tokenization efficiency to downstream prefill.

Prefill. Existing prefill methods can also be broadly categorized as non-streaming or streaming according to how input tokens become available and are processed. Most existing prefill optimizations are non-streaming, including algorithmic methods that reduce prefill computation (Jiang et al., 2024; Lai et al., 2025) and system-oriented methods that improve execution through memory management, scheduling, workload partitioning, and resource allocation (Agrawal et al., 2024; Zhong et al., 2024; Patel et al., 2024). Despite different optimization targets, these methods generally assume that a complete and fixed token sequence is available before prefill begins, requiring prefill to wait for complete tokenization. Recent streaming prefill methods partially relax this assumption by processing progressively available input chunks and incrementally executing prefill before the complete input becomes available (Yu et al., 2025; Bachkaniwala et al., 2026; vLLM Team et al., 2026). However, these methods do not explicitly bridge efficient prefill with upstream streaming tokenization, making it difficult to preserve and exploit tokenization-side streaming efficiency. In contrast, **BridgeSTOP** explicitly bridges streaming tokenization and prefill, allowing progressively produced token outputs and their efficiency benefits to be preserved and propagated into downstream prefill execution.

3 Methodology

3.1 Batch-Free Asynchronous Parallel Tokenization

Batch-Free Scheduling. BAPT removes fixed batch boundaries and treats each tokenization request as an independent scheduling, completion, and release unit. Let $\mathcal{R} = \{R_i\}_{i=1}^{N}$ denote the tokenization requests and $\mathcal{W} = \{W_m\}_{m=1}^{M}$ denote M tokenizer workers. For an execution mode $x \in \{\text{batch}, \text{BAPT}\}$, let τ_i^{adm}, $\tau_i^{\text{sta},x}$, $\tau_i^{\text{cmp},x}$, and $\tau_i^{\text{rel},x}$ denote the admission, start, completion,

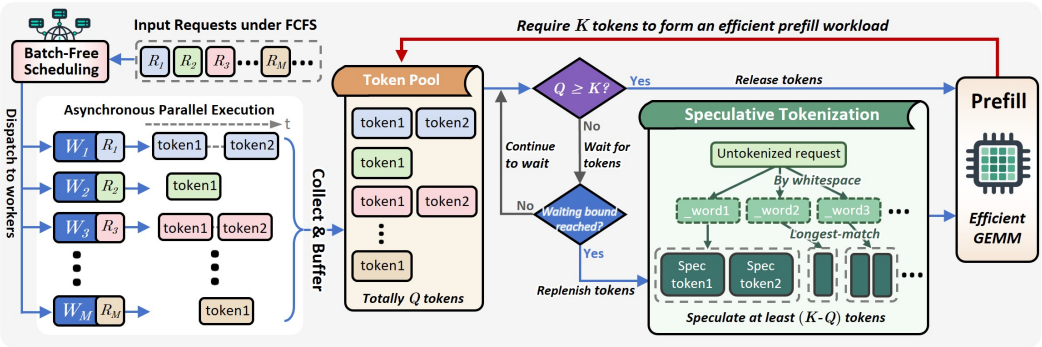

Figure 2: Overview of **BridgeSTOP**. BAPT processes input requests with asynchronous parallel tokenizer workers, while the token pool collects and buffers their streaming token outputs. When prefill requires K tokens, the pool releases sufficient exact tokens if available. Otherwise, it waits until either K tokens are collected or the waiting bound is reached, in which case speculative tokenization replenishes the remaining deficit to maintain an efficient prefill workload.

and release timestamps of R_i, respectively. We define its synchronization waiting as

$$w_i^x = \tau_i^{\mathrm{rel},x} - \tau_i^{\mathrm{cmp},x}. \tag{1}$$

Conventional batch-bounded execution partitions requests into fixed batches $\{\mathcal{B}_k\}$ and releases a request only when all requests in the same batch complete. Hence, we calculate the waiting time as:

$$w_i^{\mathrm{batch}} = \max_{R_j \in \mathcal{B}_k} \tau_j^{\mathrm{cmp,batch}} - \tau_i^{\mathrm{cmp,batch}} \geqslant 0. \tag{2}$$

This equation means that heterogeneous request lengths cause requests that finish earlier to wait for the longest request in the batch before release. Instead, BAPT employs a global FCFS scheduler to manage all admitted requests without fixed batch boundaries and releases each request immediately after completion. Hence, we have its waiting time as:

$$\tau_i^{\mathrm{rel,BAPT}} = \tau_i^{\mathrm{cmp,BAPT}} \Longrightarrow w_i^{\mathrm{BAPT}} = 0. \tag{3}$$

This result demonstrates that removing the batch boundary therefore eliminates release-side synchronization waiting, but it also changes when requests from subsequent batches become eligible for execution. We therefore compare the two scheduling policies under the same request order and worker configuration.

Proposition 1 (Batch-Boundary Removal)**.** *Consider batch-bounded scheduling and BAPT processing the same request stream with M identical workers. Both follow the same FCFS request order, and each request has the same tokenization duration under the two scheduling policies. Then removing batch boundaries cannot delay the start or release of any request:*

$$\tau_i^{\mathrm{sta,BAPT}} \leqslant \tau_i^{\mathrm{sta,batch}}, \qquad \tau_i^{\mathrm{rel,BAPT}} \leqslant \tau_i^{\mathrm{rel,batch}}, \qquad \forall R_i \in \mathcal{R}. \tag{4}$$

Detailed proof is displayed in Appendix B.2. Proposition 1 shows that removing fixed batch boundaries does not delay request execution and allows completed requests to be released independently, extending streaming tokenization from intra-request processing to inter-request execution while reducing synchronization waiting and exposing tokenization results earlier to downstream prefill.

However, removing batch boundaries alone does not eliminate worker-level synchronization, where heterogeneous tokenization durations can still cause faster workers to wait for slower ones.

Asynchronous Parallel Execution. To remove this remaining worker-level synchronization, BAPT allows each tokenizer worker to independently retrieve a new request immediately after completing its current one, enabling asynchronous parallel execution.

Consider a synchronized execution round starting at τ_r with active workers $\mathcal{W}_r \subseteq \mathcal{W}$. Let $t_{r,m}$ denote the tokenization duration of the request processed by worker W_m, and let $t_r^{\max} = \max_{W_m \in \mathcal{W}_r} t_{r,m}$ denote the longest tokenization duration in the round. Workers that complete their requests earlier must wait for the slower workers before the next round can begin, remaining idle for

$$\iota_{r,m}^{\mathrm{sync}} = t_r^{\max} - t_{r,m}. \tag{5}$$

The total synchronization-induced idle time is therefore

$$I_r^{\text{sync}} = \sum_{W_m \in \mathcal{W}_r} \iota_{r,m}^{\text{sync}} = |\mathcal{W}_r| t_r^{\max} - \sum_{W_m \in \mathcal{W}_r} t_{r,m} \geqslant 0. \tag{6}$$

This equation demonstrates that whenever pending requests are available, asynchronous execution allows each early-finishing worker to immediately start a new request rather than remaining idle until $t_r^{\max}$, thereby eliminating the synchronization-induced idle time I_r^{sync}.

Together, batch-free scheduling and asynchronous parallel execution allow requests to continuously enter, execute, and release without waiting for fixed batch boundaries or other workers, reducing both request synchronization waiting and worker idle time while supplying token outputs to downstream prefill more efficiently.

3.2 Token Pool for Bridging Streaming Tokenization and Prefill

Token Collection and Buffering. We establish a token pool between streaming tokenization and downstream prefill to continuously collect available token outputs and bridge their different execution behaviors. For request R_i, let $\tau_{i,j}^{\text{rel}}$ denote when its j-th exact token becomes available from upstream tokenization, and let $\tau_{i,j}^{\text{pool}}$ denote when this token is released from the pool to prefill. At timestamp τ, we denote the exact tokens currently buffered in the pool by $\mathcal{P}(\tau)$ and their number by $Q(\tau)$:

$$\mathcal{P}(\tau) = \{(i,j) \mid \tau_{i,j}^{\text{rel}} \leqslant \tau < \tau_{i,j}^{\text{pool}}\}, \qquad Q(\tau) = |\mathcal{P}(\tau)|. \tag{7}$$

The pool preserves token order within each request and buffers only contiguous token prefixes ready for downstream prefill. It can therefore accommodate different token arrival granularities and patterns, including fine-grained streaming outputs and asynchronous outputs produced by BAPT, providing a unified interface to downstream prefill.

Token Release. Immediately forwarding every available token can fragment downstream prefill into excessively small workloads, leading to inefficient GEMM execution. The pool therefore aggregates available tokens before releasing them to prefill. We denote by K the target token budget for each prefill execution. When downstream prefill requests tokens at timestamp τ^{pre}, the pool immediately releases K exact tokens if $Q(\tau^{\text{pre}}) \geqslant K$. When fewer than K exact tokens are available, the pool temporarily waits for additional streaming outputs. To prevent this accumulation from offsetting the latency benefit of streaming tokenization, we use $w_{\max}$ to bound how long the earliest available token can remain buffered. The pool waits until the first timestamp $\tau \geqslant \tau^{\text{pre}}$ satisfying

$$Q(\tau) \geqslant K \quad \text{or} \quad \tau - \min_{(i,j)\in\mathcal{P}(\tau)} \tau_{i,j}^{\text{rel}} \geqslant w_{\max}. \tag{8}$$

The first condition indicates that sufficient exact tokens have accumulated to form the target prefill workload. The second prevents available streaming outputs from being retained excessively.

Tokens from the same request may be released across multiple prefill iterations. This can be easily resolved by modern chunked engines such as vLLM (Kwon et al., 2023) and TensorRT-LLM (NVIDIA, 2023), which preserve request identity, token positions, and previously computed KV states, allowing later tokens to continue prefill from the cached prefix without recomputation.

Streaming Efficiency Preservation and Propagation. By separating when tokens become available from when they are consumed by prefill, the token pool bridges heterogeneous streaming tokenization outputs with downstream prefill execution. Upstream improvements that produce tokens earlier or more efficiently can therefore increase the tokens available in the pool, while downstream prefill continues to receive workloads of appropriate size. The pool thus preserves tokenization-side efficiency and propagates it to prefill.

Nevertheless, this propagation depends on sufficient exact tokens being available. When upstream tokenization temporarily falls behind and $Q(\tau) < K$ after the waiting bound is reached, continuing to wait delays downstream prefill, while releasing an undersized workload can reduce prefill efficiency. Such transient token deficits can therefore interrupt streaming efficiency propagation and degrade end-to-end serving efficiency.

3.3 Speculative Tokenization for Fast Replenishment

Replenishment Trigger. To maintain the TP bridge under temporary token deficits, we introduce speculative tokenization as a fast replenishment mechanism to supply tokens. At the trigger timestamp τ, the number of missing tokens is

$$M(\tau) = K - Q(\tau). \tag{9}$$

Instead of waiting for upstream exact tokenization to produce these tokens, speculative tokenization rapidly generates $M(\tau)$ speculative tokens from the remaining unprocessed input and appends them after the exact tokens already available in the pool. The replenished tokens restore the target prefill workload, allowing downstream prefill to continue without prolonged starvation and preventing transient TP mismatch from interrupting streaming efficiency propagation.

Verification and Rollback. Speculative tokens may differ from the exact tokens subsequently produced by upstream tokenization. We therefore verify each speculative sequence against the corresponding exact outputs once they become available. For request R_i, let $\widehat{\mathbf{z}}_i = (\hat{z}_{i,1}, \ldots, \hat{z}_{i,m_i})$ denote its speculative token sequence and $\mathbf{z}_i = (z_{i,1}, \ldots, z_{i,m_i})$ denote the later exact token sequence at the same positions. We define the length of their matched prefix as

$$c_i = \max\left\{c \in \{0, \ldots, m_i\} \mid \hat{z}_{i,j} = z_{i,j},\ \forall 1 \leqslant j \leqslant c\right\}. \tag{10}$$

If $c_i = m_i$, all speculative tokens are verified and their prefill results are retained. Otherwise, the first mismatch invalidates the remaining speculative suffix. The system rolls back to the mismatched token and re-prefills the invalidated positions using the exact tokens. Therefore, the overhead introduced by replenishment is mainly determined by speculation accuracy and the resulting rollback frequency.

Prefill results involving unverified speculative tokens remain provisional and are not used for downstream decoding until verification completes. Although this verification barrier may partially dilute the benefit of speculation, exact tokenization proceeds concurrently with speculative prefill, so the corresponding exact outputs are typically available by the time prefill completes. Especially with high speculation accuracy, verification further incurs limited rollback and re-prefill, thereby avoiding prolonged post-prefill stalling before decoding.

Longest-Match Speculation. To improve speculation accuracy and reduce rollback and re-prefill, we adopt a longest-match strategy. We split the remaining untokenized text by whitespace and process each word sequentially. Whitespace is retained with the following word, while punctuation remains part of the word during prefix matching. Starting from the current position of a word, we identify vocabulary tokens that match its prefix and select the longest valid candidate. Let w denote the remaining text of the current word, $\mathcal{V}$ denote the tokenizer vocabulary, and $\mathcal{C}(w) \subseteq \mathcal{V}$ denote the candidate tokens matching prefixes of w. The speculative token is selected as

$$\hat{z} = \arg\max_{v \in \mathcal{C}(w)} |v|, \tag{11}$$

where $|v|$ denotes the length of token v. After selecting $\hat{z}$, the procedure continues from the first unmatched character and repeats until $M(\tau)$ speculative tokens are generated. For example, `wording` can be speculated as

$$\texttt{wording} \rightarrow \{\texttt{w}, \texttt{wo}, \textbf{\texttt{word}}\} \rightarrow \texttt{ing} \rightarrow \{\texttt{i}, \texttt{in}, \textbf{\texttt{ing}}\}, \qquad \widehat{\mathbf{z}} = (\texttt{word}, \texttt{ing}). \tag{12}$$

By rapidly replenishing temporarily insufficient exact tokens, speculative tokenization prevents downstream prefill from being stalled by short-term tokenization slowdown. Together with verification and longest-match speculation, it maintains the TP bridge under transient efficiency mismatch while limiting the rollback overhead introduced by speculative execution.

4 Experimental Setup

Language Model and Datasets. We use Qwen3-8B (Yang et al., 2025) as the target LLM for all serving experiments. We select five datasets covering conversational workloads, general knowledge, mathematical reasoning, and long-context inputs. Specifically, ShareGPT4 (Wang et al., 2023) provides real-world conversational requests, while MMLU-English (Hendrycks et al., 2021) and

Table 1: LLM serving capacity (unit: QPS, **Best**, **SecondBest**)

Tokenization	Prefill	ShareGPT4	MMLU	GSM8K	LongBench	CLongEval
LoPT	Sarathi	0.75	40.50	8.5	0.0825	0.1025
zip2zip	Sarathi	0.60	42.75	9.5	0.0875	0.1000
MAGNET	Sarathi	0.55	32.50	7.5	0.1075	0.1275
IncreBPE	Sarathi	0.90	55.0	15.0	0.1125	0.1525
SPEAK	Sarathi	0.80	47.5	14.0	0.1400	0.1725
LoPT	Stream2LLM	0.90	45.75	9.0	0.1050	0.1225
zip2zip	Stream2LLM	0.85	48.25	11.0	0.1150	0.1400
MAGNET	Stream2LLM	0.65	40.25	9.5	0.1225	0.1575
IncreBPE	Stream2LLM	1.05	67.5	16.5	0.1275	0.1775
SPEAK	Stream2LLM	0.90	54.0	15.0	0.1750	0.2125
IncreBPE	**BridgeSTOP** w/o BAPT	**1.15**	**69.0**	**19.0**	0.1400	0.1800
SPEAK	**BridgeSTOP** w/o BAPT	1.00	60.5	16.5	**0.1800**	**0.2275**
IncreBPE	**BridgeSTOP**	**1.25**	**71.5**	**19.5**	0.1525	0.2025
SPEAK	**BridgeSTOP**	1.10	62.5	18.0	**0.1950**	**0.2450**

GSM8K (Cobbe et al., 2021) represent general-knowledge and mathematical-reasoning workloads. We further use LongBench (Bai et al., 2024) and CLongEval (Small) (Qiu et al., 2024) to evaluate efficiency on long-context requests. Detailed dataset descriptions are displayed in Appendix C.1.

Baselines. For tokenization, we include advanced RToks and LToks. For RToks, we select zip2zip (Geng et al., 2025), and IncreBPE (Jiang & Gong, 2026). For LToks, we select MAGNET (Ahia et al., 2024), and SPEAK (Chen et al., 2026). We further include LoPT (Shao et al., 2026) as a system-oriented tokenization baseline. IncreBPE and SPEAK support streaming tokenization and serve as the base tokenizers for applying our method. For prefill, we use Sarathi (Agrawal et al., 2024) and Stream2LLM (Bachkaniwala et al., 2026) as prefill baselines. Implementation details of all baselines are provided in Appendix C.2.

Metrics. To evaluate end-to-end serving, we evaluate maximum sustainable serving capacity, P50 time to first token (TTFT), and P99 time between tokens (TBT). Maximum serving capacity is defined as the highest request rate that satisfies P50 TTFT $\leqslant$ 400 ms, P99 TBT $\leqslant$ 200 ms, and median scheduling delay $\leqslant$ 2000 ms. We remove the TTFT constraint for LongBench and CLongEval because their long-context requests naturally incur substantially longer TTFT, and remove the TBT constraint for MMLU because each request generates only one output token. To evaluate serving latency, we vary the request rate and report P50 TTFT and P99 TBT at each rate, allowing all methods to be compared under identical serving loads. All serving methods use the same request traces, model configuration, generation settings, and resource limits.

Implementation. All experiments run on an NVIDIA RTX 5090 GPU with 32 GB memory and an AMD Ryzen 9 9950X CPU with 32 threads. BAPT is implemented in Rust using a Rayon worker pool, where we establish 32 asynchronous parallel tokenizer workers that continuously obtain requests from a global queue, while each request writes its outputs to an independent preallocated result slot. Our speculative tokenization reuses the same worker pool as BAPT, with replenishment tasks dispatched to available workers when triggered. The target prefill token budget K is determined through one-time offline profiling as the smallest token count that achieves near-saturated prefill efficiency under the given model and hardware configuration. We set $K = 512$ for all experiments. The maximum waiting bound $w_{\max}$ is set to 10 ms according to the token production efficiency measured on our hardware platform. Detailed selection criteria and sensitivity analyses for both hyperparameters are provided in Appendix C.3.

5 Results

5.1 Serving Capacity Evaluation

We evaluate the maximum sustainable serving capacity of **BridgeSTOP** built on SOTA streaming tokenizers under the specified SLO constraints, with results reported in Table 1 and more details in Appendix D.1. Overall, **BridgeSTOP** consistently improves serving capacity across all workloads and both RTok- and LTok-based configurations, achieving up to $1.22\times$ improvement over the corresponding baselines and up to $2.60\times$ over other baseline configurations. Existing methods ei-

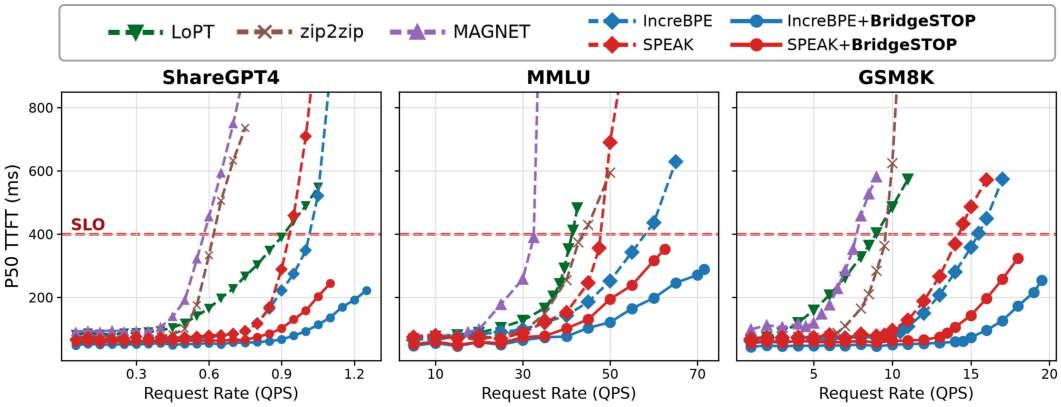

Figure 3: Serving latency across request rates. All baseline tokenizers are paired with Sarathi as the prefill method. Dashed red lines indicate the 400 ms TTFT SLO.

ther retain sequential tokenization-to-prefill execution or bridge streaming inputs and prefill only at fixed chunk granularity. In contrast, **BridgeSTOP** sustains higher request rates before violating the serving SLOs, translating streaming tokenization efficiency into higher serving capacity.

From the tokenization perspective, the relative advantage of RToks and LToks changes with request length. On the shorter ShareGPT4, MMLU, and GSM8K workloads, IncreBPE + **BridgeSTOP** achieves the highest capacity. For LongBench and CLongEval, longer requests substantially increase RTok execution cost, whereas LToks exhibit slower latency growth. Consequently, SPEAK + **BridgeSTOP** achieves the highest capacity on both long-context workloads. These results show that **BridgeSTOP** preserves and propagates tokenization-side efficiency across different tokenizer backends and translates it into end-to-end serving gains.

From the prefill perspective, Stream2LLM consistently improves capacity over Sarathi under the same tokenizer configuration, demonstrating the benefit of bridging streaming inputs and downstream prefill. However, its execution remains constrained by coarse-grained input chunks. **BridgeSTOP** further improves capacity beyond chunk-level streaming by preserving and propagating efficiency derived from finer-grained streaming tokenization.

5.2 Serving Latency Evaluation

We evaluate serving latency using P50 TTFT across increasing request rates, with all baseline tokenizers paired with Sarathi as the prefill method. The TTFT results are displayed in Figure 3 and Appendix Figure 4, with P99 TBT results and additional discussions in Appendix D.2. Overall, **BridgeSTOP** consistently maintains lower TTFT over a wider range of request rates across different streaming tokenizer configurations. At low request rates, TTFT remains relatively stable and the differences among methods are small because queueing and resource contention remain low. As the request rate increases, TTFT gradually rises as synchronization waiting, queueing, and downstream contention become more pronounced. Near serving capacity, these effects accumulate rapidly and cause a sharp increase in TTFT. In contrast, **BridgeSTOP** delays such latency degradation to higher request rates.

The same trend with increasing request rate is observed across different workloads. On ShareGPT4, TTFT begins to increase at relatively low request rates as the system approaches saturation earlier, whereas MMLU and GSM8K exhibit a broader region of gradual latency growth before the sharp increase near capacity. Across these workloads, **BridgeSTOP** consistently shifts the TTFT increase toward higher request rates, showing that improved streaming tokenization efficiency is preserved and propagated to downstream prefill as the request rate increases. The LongBench and CLongEval results in Figure 4 show a similar pattern, further demonstrating that the latency advantage persists across different workloads.

Table 2: Tokenization efficiency with batch size 64. (**Best**, **Second Best**)

Dataset	Metric	LoPT	zip2zip	MAGNET	IncreBPE	SPEAK	IncreBPE + BAPT	SPEAK + BAPT
ShareGPT4	*Throughput (MiB/s)*	15.25	18.79	5.90	**20.97**	13.39	**34.06**	18.92
	Avg. Latency (ms)	3.21	2.65	8.23	2.39	3.67	**0.73**	**1.31**
	P95 Latency (ms)	17.83	15.41	12.59	**7.59**	9.81	**3.69**	7.67
MMLU	*Throughput (MiB/s)*	13.45	15.50	6.65	16.38	15.72	**28.93**	**20.31**
	Avg. Latency (ms)	2.88	2.51	5.73	2.33	2.43	**0.67**	**0.96**
	P95 Latency (ms)	8.07	6.83	8.17	6.29	3.46	**1.62**	**2.13**
GSM8K	*Throughput (MiB/s)*	8.73	10.63	4.01	12.41	11.87	**23.34**	**17.32**
	Avg. Latency (ms)	3.00	2.48	6.48	2.15	2.14	**0.55**	**0.75**
	P95 Latency (ms)	4.46	3.70	7.56	3.46	3.18	**0.81**	**1.04**
LongBench	*Throughput (MiB/s)*	16.46	14.31	18.92	17.70	**22.61**	16.62	**28.92**
	Avg. Latency (ms)	162.60	186.99	135.56	152.59	113.53	**78.64**	**44.41**
	P95 Latency (ms)	332.74	379.32	216.78	311.04	203.12	**130.04**	**97.88**
CLongEval	*Throughput (MiB/s)*	16.56	14.53	17.22	18.85	**40.80**	20.56	**47.89**
	Avg. Latency (ms)	484.31	561.80	475.42	424.83	207.10	**204.57**	**87.82**
	P95 Latency (ms)	1154.29	1350.52	1071.32	1022.43	475.94	**319.74**	**203.57**

5.3 Ablation Analysis

We ablate two key components of **BridgeSTOP** to validate their respective effectiveness.

BAPT. We validate whether BAPT improves tokenization efficiency via tokenizer-only experiments. Table 2 reports the batch-size-64 results, where BAPT generally improves throughput while consistently reducing latency. We further ablate BAPT from **BridgeSTOP** in Table 1, where removing BAPT consistently reduces serving capacity, demonstrating that improved tokenization efficiency is translated into end-to-end serving gains. Additionally, Appendix D.3 extends the evaluation from batch size 64 to 128–512 and analyzes tokenization efficiency across different request lengths. The advantage of BAPT becomes more pronounced as batch size increases because batch-bounded execution suffers greater synchronization waiting, while BAPT removes predefined batch boundaries. BAPT also maintains consistent efficiency improvements across request lengths. Overall, all these results demonstrate that BAPT effectively improves tokenization efficiency and that **BridgeSTOP** can successfully translate this improvement into end-to-end serving gains.

Table 3: Ablation of speculative tokenization.

Metric	Waiting	Random	Frequency	Longest
Overhead Time (ms)	0.750	0.012	0.014	**0.006**
Speculation Acc. (%)	–	17.45	8.79	**89.13**
Rollback Rate (%)	–	98.33	98.89	**39.67**
Rollback Tokens	–	94.60	95.11	**38.82**
Serving Capacity (QPS)	70.0	68.5	68.0	**71.5**

Speculative Tokenization. We evaluate speculative tokenization on MMLU with IncreBPE as the base tokenizer by comparing different strategies: waiting indicates waiting for exact token outputs; longest selects the longest matching token; random samples a candidate uniformly; and frequency selects the candidate with the highest Qwen BPE token frequency following (Zhao et al., 2025). The results are reported in Table 3. Around 5% of requests trigger replenishment, and the speculation-related metrics are averaged over these requests. Speculation accuracy reflects whether each speculative token matches the corresponding exact token, while rollback rate reflects how many speculative tokens are actually rolled back under our verification policy, where one mismatched token invalidates all subsequent speculative tokens. Our longest-match strategy substantially outperforms others. The serving capacity results further show that direct waiting reduces capacity by stalling downstream prefill, while the low speculation accuracy of random and frequency-based strategies leads to frequent rollback and re-prefill, resulting in even lower capacity. In contrast, longest-match speculation rapidly replenishes transient token deficits while substantially reducing rollback and re-prefill overhead, achieving the highest serving capacity.

6 Conclusion

We propose **BridgeSTOP**, a framework that bridges streaming tokenization and LLM prefill to improve, preserve, and propagate efficiency toward end-to-end serving gains. By combining BAPT, a token pool, and speculative tokenization, **BridgeSTOP** extends streaming tokenization across requests, preserves and propagates tokenization-side efficiency across TP, and handles transient efficiency mismatch. Experiments across diverse workloads and tokenizer types show consistent improvements in sustainable serving capacity and latency.

AI USE STATEMENT

In this work, we used generative AI tools to aid or polish writing, for research ideation or execution, and for proving mathematical claims.

For writing, generative AI tools were used for grammar checking and improving overall language fluency. The specific wording, sentence structures, writing ideas, and logical organization of the paper were developed by us.

For research ideation or execution, generative AI tools were used only to assist with code implementation. The research ideation was conducted entirely by us, including identifying the research idea and topic, conducting and understanding the literature review, identifying the significance, objectives, and motivations of the research problem, and determining the contributions and novelty with respect to the broader community and research field. All AI-assisted code was manually checked by us for correctness.

For proving mathematical claims, generative AI tools were used to assist with the formal mathematical presentation of the proposition in BAPT. The mathematical notation, variable definitions, practical meanings of the variables, derivations, target results of the proposition, and the main proof ideas and reasoning steps were developed by us. Based on these author-provided definitions, target results, and proof procedures described in natural language, generative AI tools were used to express the proof procedures in mathematical form. All AI-assisted mathematical derivations and proofs were manually checked by us for correctness.

We did not use generative AI tools for retrieval and discovery, to draft sections of the paper, or for generating synthetic datasets. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, we provide the mathematical formulation and implementation-relevant details of all three components of **BridgeSTOP**: BAPT in Section 3.1, the token pool in Section 3.2, and speculative tokenization in Section 3.3. The assumptions and complete proof of Proposition 1 are provided in Appendix B.2, with additional analysis of BAPT under different tokenizer execution modes in Appendix B.1. The main experimental setup specifies the target model, hardware platform, serving metrics and SLOs, implementation details, worker configuration, and the hyperparameters used throughout evaluation. Appendix C.1 provides the evaluated datasets and their input statistics; Appendix C.2 details the integration and configuration of all tokenization and prefill baselines; and Appendix C.3 describes the selection criteria and sensitivity analysis for the target prefill token budget K and maximum token retention bound $w_{\max}$. All serving methods are evaluated using the same request traces, model configuration, generation settings, and resource limits. Additional serving-capacity analyses are provided in Appendix D.1, additional TTFT and TBT results in Appendix D.2, and complete tokenizer-only evaluations of BAPT across different batch sizes and request lengths in Appendix D.3. Together, these materials document the algorithmic assumptions, system configurations, baseline construction, hyperparameter choices, and evaluation conditions required to reproduce the reported comparisons. To further support reproducibility, we will release the implementation code upon acceptance.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, 2024.

Orevaoghene Ahia, Sachin Kumar, Hila Gonen, Valentin Hofmann, Tomasz Limisiewicz, Yulia Tsvetkov, and Noah A. Smith. MAGNET: Improving the multilingual fairness of language models with adaptive gradient-based tokenization. In *Advances in Neural Information Processing Systems*, 2024.

Rajveer Bachkaniwala, Chengqi Luo, Richard So, Divya Mahajan, and Kexin Rong. Stream2LLM: Overlap context streaming and prefill for reduced time-to-first-token. In *Proceedings of Machine Learning and Systems*, 2026.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

Nikhil Bhendawade, Irina Belousova, Qichen Fu, Henry Mason, Mohammad Rastegari, and Mahyar Najibi. Speculative streaming: Fast llm inference without auxiliary models. In *Proceedings of the 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, 2024.

Ming Chen, Wenyao Li, Chao Liang, Shi Gu, Peng Lin, De Ma, Huajin Tang, Qian Zheng, and Gang Pan. SPEAK: Spiking neurons as an entropy-aware tokenizer for large language models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, 2026.

Euijun Chung, Yuxiao Jia, Aaron Jezghani, and Hyesoon Kim. Characterizing cpu-induced slowdowns in multi-gpu llm inference. *arXiv preprint arXiv:2603.22774*, 2026.

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

Peizhuang Cong, Qizhi Chen, Haochen Zhao, and Tong Yang. BATON: Enhancing batch-wise inference efficiency for large language models via dynamic re-batching. In *Proceedings of the ACM on Web Conference*, 2025.

Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A. Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R. Risbud. Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proceedings of the IEEE*, 2021.

Darius Feher, Ivan Vulić, and Benjamin Minixhofer. Retrofitting large language models with dynamic tokenization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *The VLDB Journal*, 2024.

Saibo Geng, Nathan Ranchin, Yunzhen Yao, Maxime Peyrard, Chris Wendler, Michael Gastpar, and Robert West. zip2zip: Inference-time adaptive tokenization via online compression. In *Advances in Neural Information Processing Systems*, 2025.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *Proceedings of International Conference on Learning Representations*, 2021.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *Advances in Neural Information Processing Systems*, 2024.

Shenghu Jiang and Ruihao Gong. Incremental BPE tokenization. In *Proceedings of International Conference on Machine Learning*, 2026.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 2018.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. FlexPrefill: A context-aware sparse attention mechanism for efficient long-sequence inference. In *Proceedings of International Conference on Learning Representations*, 2025.

De Ma, Xiaofei Jin, Shichun Sun, Yitao Li, Xundong Wu, Youneng Hu, Fangchao Yang, Huajin Tang, Xiaolei Zhu, Peng Lin, and Gang Pan. Darwin3: A large-scale neuromorphic chip with a novel isa and on-chip learning. *National Science Review*, 2024.

Benjamin Minixhofer, Edoardo Maria Ponti, and Ivan Vulić. Zero-shot tokenizer transfer. In *Advances in Neural Information Processing Systems*, 2024.

Piotr Nawrot, Jan Chorowski, Adrian Łańcucki, and Edoardo Maria Ponti. Efficient transformers with dynamic token pooling. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2023.

NVIDIA. TensorRT-LLM, 2023. URL https://docs.nvidia.com/tensorrt-llm.

NVIDIA. DeepStream SDK. https://developer.nvidia.com/deepstream-sdk, 2026.

Yerim Oh, Jun-Hyung Park, Junho Kim, SungHo Kim, and SangKeun Lee. Incorporating domain knowledge into materials tokenization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 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 *Proceedings of the ACM/IEEE 51st Annual International Symposium on Computer Architecture*, 2024.

Rui Qian, Xiaoyi Dong, Pan Zhang, Yuhang Zang, Shuangrui Ding, Dahua Lin, and Jiaqi Wang. Streaming long video understanding with large language models. In *Advances in Neural Information Processing Systems*, 2024.

Zexuan Qiu, Jingjing Li, Shijue Huang, Xiaoqi Jiao, Wanjun Zhong, and Irwin King. CLongEval: A chinese benchmark for evaluating long-context large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016.

Wei Shao, Lingchao Zheng, Pengyu Wang, Peizhen Zheng, Jun Li, and Yuwei Fan. LoPT: Lossless parallel tokenization acceleration for long context inference of large language model. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, 2026.

Yangyang Shi, Yongqiang Wang, Chunyang Wu, Ching-Feng Yeh, Julian Chan, Frank Zhang, Duc Le, and Mike Seltzer. Emformer: Efficient memory transformer based acoustic model for low latency streaming speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2021.

vLLM Team, Meta, and Mistral AI. Streaming requests & realtime api in vllm. https://vllm.ai/blog/2026-01-31-streaming-realtime, 2026.

Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. OpenChat: Advancing open-source language models with mixed-quality data. *arXiv preprint*, 2023. arXiv:2309.11235.

Chiyue Wei, Cong Guo, Bowen Duan, Junyao Zhang, Haoxuan Shan, Yifei Wang, Yangjie Zhou, Hai Li, Danyang Zhuo, and Yiran Chen. Optimus: Elastic decoding for efficient diffusion llm serving. *arXiv preprint arXiv:2605.24832*, 2026.

An Yang et al. Qwen3 technical report. *arXiv preprint*, 2025. arXiv:2505.09388.

Amos You. BlockBPE: Parallel BPE tokenization. *arXiv preprint*, 2025. arXiv:2507.11941.

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 *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation*, 2022.

Runjie Yu, Weizhou Huang, Shuhan Bai, Jian Zhou, and Fei Wu. AquaPipe: A quality-aware pipeline for knowledge retrieval and large language models. *Proceedings of the ACM on Management of Data: SIGMOD*, 2025.

Weilin Zhao, Tengyu Pan, Xu Han, Yudi Zhang, Ao Sun, Yuxiang Huang, Kaihuo Zhang, Weilun Zhao, Yuxuan Li, Jie Zhou, Hao Zhou, Jianyong Wang, Zhiyuan Liu, and Maosong Sun. FR-Spec: Accelerating large-vocabulary language models via frequency-ranked speculative sampling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

Mengyu Zheng, Hanting Chen, Tianyu Guo, Chong Zhu, Binfan Zheng, Chang Xu, and Yunhe Wang. Enhancing large language models through adaptive tokenizers. In *Advances in Neural Information Processing Systems*, 2024.

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 *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*, 2024.

APPENDIX CONTENTS

A Why Existing Tokenizers Cannot Be Directly Bridged with Downstream Prefill 15

B Detailed Analysis of Batch-Free Asynchronous Parallel Tokenization 16

B.1 Analysis under Different Tokenizer Execution Modes 16

B.2 Proof of Proposition 1 16

C Additional Experimental Setup Details 18

C.1 Dataset Details 18

C.2 Baseline Implementation Details 18

C.3 Hyperparameter Selection and Sensitivity Analysis 19

D Additional Experimental Results and Discussions 21

D.1 Additional Serving Capacity Results 21

D.2 Additional Serving Latency Results 21

D.3 Additional Tokenization Efficiency Results for BAPT 23

A Why Existing Tokenizers Cannot Be Directly Bridged with Downstream Prefill

Existing tokenizer optimizations improve tokenization efficiency in substantially different ways, but their gains do not necessarily translate into end-to-end LLM serving efficiency. DyTok shortens token sequences by dynamically merging existing subwords, while zip2zip further compresses recurring token sequences into larger hypertokens (Feher et al., 2025; Geng et al., 2025). Although fewer and coarser tokens reduce downstream sequence length and computation, they also change the representations consumed by the LLM and require the model to adapt to these merged tokens. Consequently, both methods introduce model-side adaptation and can incur downstream performance degradation, which is undesirable for large-scale serving where the behavior of an already deployed LLM should ideally remain unchanged. ZeTT alleviates this problem by using a pretrained hypernetwork to predict embeddings for tokens introduced by a new tokenizer (Minixhofer et al., 2024). However, the hypernetwork is trained for the embedding space of a particular base LLM and must be separately prepared for different base models or embedding configurations, substantially limiting its plug-and-play applicability across heterogeneous serving deployments. Gradient-based tokenizers such as DTP and MAGNET take a more tightly coupled approach by integrating token-boundary prediction and learned segment representations into the language model itself (Nawrot et al., 2023; Ahia et al., 2024). Their tokenization and language modeling components are jointly optimized, replacing the conventional vocabulary-token-ID interface with model-integrated learned representations and consequently requiring joint model training. Their reported implementations therefore rely on specially trained lightweight language models rather than being directly deployable with existing mainstream LLMs. SPEAK instead predicts discrete token boundaries over character spans and retains an explicit tokenization interface compatible with vocabulary token IDs (Chen et al., 2026). Although SPEAK requires tokenizer-side training and proposes an adaptation procedure for mainstream LLMs, the trained tokenizer can subsequently be fixed and integrated with existing LLM serving without jointly retraining the target LLM. From a system perspective, IncreBPE preserves the original BPE vocabulary and introduces eager output to expose stable tokens before complete tokenization, providing a natural basis for streaming tokenization (Jiang & Gong, 2026). However, it does not address how these progressively available tokens should be bridged to downstream prefill, and its evaluation consequently focuses on tokenization rather than end-to-end LLM serving. Similarly, LoPT losslessly accelerates conventional tokenization through parallel text partitioning and result merging (Shao et al., 2026), but remains a tokenizer-side execution optimization whose evaluation does not study how the resulting efficiency improvement propagates through prefill and the complete serving pipeline. These limitations show that improving tokenization alone, whether through shorter token sequences, learned tokenization, or faster execution, is insufficient to guarantee end-to-end serving gains without an explicit bridge that preserves and propagates tokenization-side efficiency to downstream prefill.

B Detailed Analysis of Batch-Free Asynchronous Parallel Tokenization

This section further analyzes BAPT under different tokenizer execution modes and provides the proof of Propositions 1.

B.1 Analysis under Different Tokenizer Execution Modes

The exact source of synchronization waiting differs across tokenizer execution modes. For rule-based tokenizers (RToks), CPU workers commonly follow static parallel execution or tokenizer-pool execution. For learnable tokenizers (LToks), parallel workers can be abstracted as independently schedulable GPU execution units, where multiple execution groups share the available GPU resources. We discuss how BAPT applies to these representative execution modes below.

RTok Static Parallel Execution. Static parallel execution partitions requests among fixed worker-local queues. A request must wait for preceding requests assigned to the same worker even when another worker becomes available earlier. Consequently, some workers can remain idle while tokenization requests are still waiting in other local queues. BAPT instead places admitted requests in a global queue and allows any available worker to retrieve the next request. This removes waiting caused by fixed worker-local assignment and allows requests to continuously progress across workers. Because global scheduling may change request-to-worker assignment and execution order, however, the exact latency improvement of an individual request can depend on the runtime workload.

RTok Tokenizer-Pool Execution. A tokenizer pool already uses a shared queue and allows available workers to retrieve requests from the current batch. Its limitation lies in the batch boundary. Requests belonging to later batches remain unavailable until the current batch reaches its synchronization boundary, even when a worker has already become available. BAPT extends the shared queue across all admitted requests and removes this batch-level eligibility restriction. Under identical workers, the same FCFS order, and unchanged per-request tokenization duration, this execution mode satisfies Proposition 1, so BAPT cannot delay the start or release of any request.

LTok GPU Execution. For LToks, multiple requests can be processed by independently schedulable GPU execution units, while variable-length inputs may additionally introduce padding and shared-resource interference. Different execution-group compositions can therefore change the actual tokenization duration of a request. BAPT still removes fixed batch boundaries, independently releases completed requests, and allows available execution units to accept new requests without waiting for other groups. However, because the tokenization duration itself can vary with GPU grouping and resource sharing, Proposition 1 is not used to claim universal per-request latency dominance in this setting. Instead, the principal benefit of BAPT is the removal of synchronization constraints that would otherwise prevent available execution resources and completed tokenization results from progressing independently.

Overall, these execution modes exhibit different sources of inefficiency. Static RTok execution can suffer from fixed worker-local assignment, tokenizer-pool execution remains bounded by batch-level request eligibility, and LTok execution additionally involves GPU grouping and shared-resource effects. Across these settings, BAPT consistently removes predefined batch boundaries and allows requests and available workers to progress independently, extending streaming tokenization from individual requests to execution across requests.

B.2 Proof of Proposition 1

Proof. We compare batch-bounded scheduling and BAPT under the assumptions of Proposition 1. Both process the same request stream using the same M identical workers, follow the same FCFS request order, and preserve the tokenization duration of each request. Their difference is that batch-bounded execution prevents requests in a later batch from becoming eligible until the previous batch completes, while BAPT removes this restriction.

Consider requests in their common FCFS order. For the initially eligible requests, removing a batch boundary cannot postpone their start. Suppose all requests preceding R_i start no later under BAPT

than under batch-bounded execution. Because their tokenization durations remain unchanged, these requests also complete no later under BAPT. A worker therefore becomes available for R_i no later under BAPT than under batch-bounded execution.

After R_i is admitted, BAPT only requires the preceding requests in the FCFS queue to be processed first. Batch-bounded execution follows the same request order but may additionally require R_i to wait for the preceding batch to complete. Removing this additional restriction therefore gives

$$\tau_i^{\mathrm{sta,BAPT}} \leqslant \tau_i^{\mathrm{sta,batch}}. \tag{13}$$

Applying this argument sequentially to the common FCFS order establishes the relation for every request.

Since the tokenization duration of each request is unchanged,

$$\tau_i^{\mathrm{cmp,BAPT}} \leqslant \tau_i^{\mathrm{cmp,batch}}. \tag{14}$$

BAPT releases each request immediately after completion, whereas batch-bounded execution releases $R_i \in \mathcal{B}_k$ only after the last request in $\mathcal{B}_k$ completes. Therefore,

$$\tau_i^{\mathrm{rel,BAPT}} = \tau_i^{\mathrm{cmp,BAPT}} \leqslant \tau_i^{\mathrm{cmp,batch}} \leqslant \max_{R_j \in \mathcal{B}_k} \tau_j^{\mathrm{cmp,batch}} = \tau_i^{\mathrm{rel,batch}}. \tag{15}$$

Thus, BAPT cannot delay either the start or release of any request.

If R_i completes before the last request in its batch under batch-bounded execution, then

$$\tau_i^{\mathrm{cmp,batch}} < \max_{R_j \in \mathcal{B}_k} \tau_j^{\mathrm{cmp,batch}} = \tau_i^{\mathrm{rel,batch}}. \tag{16}$$

Together with $\tau_i^{\mathrm{rel,BAPT}} \leqslant \tau_i^{\mathrm{cmp,batch}}$, this yields

$$\tau_i^{\mathrm{rel,BAPT}} < \tau_i^{\mathrm{rel,batch}}, \tag{17}$$

which establishes the strict release-time improvement. □

C Additional Experimental Setup Details

C.1 Dataset Details

We evaluate five datasets covering conversational workloads, general knowledge, mathematical reasoning, and long-context inputs. Table 4 reports the complete input statistics of each dataset used in our experiments. Since different tokenizers may produce different token sequences for the same input, we additionally tokenize all requests with the original Qwen3 BPE tokenizer and report the resulting prompt token lengths only as a unified reference for input scale.

Table 4: Input statistics of the evaluated datasets. Token lengths are measured using the original Qwen3 BPE tokenizer as a unified reference.

Dataset	# Requests	Character Length				Qwen3 BPE Token Length			
		Mean	Std.	Min	Max	Mean	Std.	Min	Max
ShareGPT4	6,205	964.27	1,876.44	70	15,531	253.84	527.34	13	8,634
MMLU-English	14,042	639.13	459.51	214	5,041	136.50	93.36	52	1,026
GSM8K	1,319	424.87	97.57	258	1,033	96.03	22.30	58	223
LongBench	4,750	37,768.02	32,218.44	813	257,342	10,130.46	8,020.38	177	65,412
CLongEval (Small)	2,605	15,915.44	10,330.98	1,357	52,196	10,307.41	5,822.91	882	27,506

ShareGPT4 provides real-world conversational requests. We select ShareGPT4 following Sarathi-Serve, allowing our serving evaluation to use a conversational workload consistent with this established LLM serving baseline. **MMLU-English** and **GSM8K** represent general-knowledge and mathematical-reasoning workloads, respectively. We further use **LongBench** and **CLongEval** to evaluate long-context serving.

For CLongEval, we use the Small tier. Under our Qwen3-8B serving configuration on a 32 GB RTX 5090, requests from the Middle and Large tiers can exceed the available GPU memory after loading the LLM and maintaining the required serving states, making stable serving evaluation infeasible. In contrast, the Small tier contains long-context requests with an average Qwen3 BPE length of 10,307.41 tokens, close to the 10,130.46-token average of LongBench, and can utilize the available GPU memory without causing out-of-memory failures. We therefore use CLongEval-Small as a practical long-context workload for our hardware configuration.

C.2 Baseline Implementation Details

This subsection describes how the tokenization and prefill baselines are integrated into the Qwen3-8B serving pipeline.

LoPT and **IncreBPE** do not require additional training of the target LLM. LoPT operates on the original tokenizer implementation, while IncreBPE directly uses an existing BPE vocabulary. We therefore execute both methods with the Qwen3 BPE vocabulary without additional tokenizer or LLM training.

zip2zip introduces new tokens whose embeddings must be learned by the target LLM. Following its original training procedure, we apply LoRA-based PEFT to Qwen3 with zip2zip on eight NVIDIA H100 GPUs. The resulting model weights are then transferred to our RTX 5090 serving platform. Therefore, the additional tokens introduced by zip2zip are explicitly adapted to Qwen3 before serving evaluation.

MAGNET and **SPEAK** can produce segmentation spans that do not directly correspond to a single token in the original Qwen3 vocabulary. To integrate their outputs with the unchanged Qwen3-8B model, we introduce a lightweight vocabulary adapter before downstream prefill. For each tokenizer-generated span, the adapter first checks whether the complete span corresponds to a valid Qwen3 BPE token. Otherwise, starting from its first UTF-8 byte, the adapter repeatedly selects the longest valid Qwen3 token whose byte representation matches the remaining prefix. Because Qwen3 uses byte-level BPE, the procedure can always fall back to individual byte tokens and therefore preserves the original text losslessly. The final sequence contains only valid Qwen3 token IDs and can be directly consumed by downstream prefill. Across all five datasets, fewer than 0.1% of tokens require

this adaptation, which means a request tokenized into 1000 token outputs only has about 1 token out of the BPE vocabulary. Such a negligible fraction is unlikely to materially alter the tokenization behavior or efficiency of the tokenizers and introduces only a subtle, practically unobservable degradation in tokenization latency.

Sarathi-Serve introduces chunked prefill to improve LLM serving execution. We use its chunked-prefill mechanism as the non-streaming prefill baseline. The complete request is still tokenized before chunked prefill begins, so its tokenization and prefill stages retain a request-level interface rather than directly bridging streaming token outputs with downstream prefill.

Stream2LLM. Sarathi performs chunked prefill after the complete request has been tokenized, where the full token sequence is already available and chunking only controls the granularity of downstream prefill execution. Thus, it does not introduce tokenization–prefill overlap. In contrast, Stream2LLM incrementally processes progressively available input chunks. For our static-request workloads, we instantiate this behavior by partitioning each arriving raw input request into fixed-size chunks while preserving the same request identity and execution state across chunks. Each chunk is tokenized and incrementally prefetched as part of the same logical request, allowing the tokenization of later chunks to overlap with the prefill of earlier ones. This provides a stronger baseline than conventional chunked prefill by introducing coarse-grained tokenization–prefill overlap.

We use an input chunk size of 2048 UTF-8 bytes. After tokenization, these input chunks produce token-sequence segments whose lengths are approximately centered around 512 tokens across our workloads, yielding a downstream execution granularity comparable to the target prefill token budget $K = 512$ used by **BridgeSTOP**. Additionally, to avoid favoring our method through the choice of input granularity, we evaluate chunk sizes of 1024, 2048, 3072, and 4096 bytes. The 2048-byte configuration achieves the highest overall sustainable serving capacity on our hardware platform and is therefore used for all reported Stream2LLM results.

C.3 Hyperparameter Selection and Sensitivity Analysis

The token pool in **BridgeSTOP** involves two main hyperparameters: the target prefill token budget K and the maximum token retention bound $w_{\max}$. K determines the target amount of tokenized input delivered to each downstream prefill execution, while $w_{\max}$ independently limits how long an available token can remain buffered in the pool. Their roles are therefore complementary: K helps preserve downstream prefill efficiency, while $w_{\max}$ prevents progressively available streaming tokens from being retained for excessive time.

Effects of K and $w_{\max}$. The target token budget K determines how streaming token outputs are grouped before being delivered to downstream prefill. A larger K forms larger prefill workloads and generally improves GEMM efficiency, but requires more exact tokens to become available before the target workload is formed and can increase buffering or trigger replenishment when exact token supply is temporarily insufficient. A smaller K allows available streaming tokens to be consumed earlier, but may fragment prefill into small workloads with lower GPU execution efficiency. Importantly, K does not change the computational efficiency of the tokenizer itself. Instead, it determines how effectively the efficiency obtained from upstream streaming tokenization can be preserved when its outputs are consumed by downstream prefill. An excessively small K can lose upstream gains through inefficient prefill execution, whereas an excessively large K can retain progressively available tokens for unnecessarily long periods.

The retention bound $w_{\max}$ serves a different purpose. It directly limits the residence time of available exact tokens in the pool, preventing early streaming outputs from being retained indefinitely while the pool continues collecting tokens. A larger $w_{\max}$ permits longer token accumulation and can reduce the frequency of speculative replenishment, but increases token retention and may delay downstream prefill. A smaller $w_{\max}$ releases the retention constraint earlier, but can terminate accumulation before K exact tokens are available and therefore trigger speculative replenishment more frequently. This increases speculation, verification, and possible rollback overhead. Together, K and $w_{\max}$ determine how efficiently streaming token outputs are transformed into downstream prefill workloads while preserving both prefill execution efficiency and the early availability provided by streaming tokenization.

Selection of the Target Prefill Token Budget. Our selection of K is inspired by the token-budget mechanism of Sarathi-Serve (Agrawal et al., 2024). Sarathi-Serve limits the number of tokens processed in each serving iteration, where an excessively small token budget reduces GPU utilization and introduces repeated execution overhead, while a larger budget provides more efficient prefill computation at the cost of increased iteration latency. Although **BridgeSTOP** uses K for a different purpose, both mechanisms rely on the same underlying property that prefill efficiency strongly depends on the amount of token computation grouped into one execution.

Sarathi-Serve demonstrates that an appropriate token budget depends on the deployed model, hardware, and latency requirement. Its microbenchmarks use a token budget of 256 for Mistral-7B on a single A100 GPU and 512 for LLaMA2-70B on four A100 GPUs. Its serving experiments further use 512 tokens under strict latency requirements and generally use 2048 under more relaxed settings. These observations suggest that K should be determined for the target deployment rather than fixed universally.

We therefore perform one-time offline profiling using Qwen3-8B on our RTX 5090 platform and measure prefill execution efficiency as the processed token workload increases. We select the smallest token budget that reaches near-saturated prefill efficiency, which gives $K = 512$. This value allows the token pool to form efficient downstream workloads without requiring unnecessarily large token accumulation, and we use $K = 512$ throughout all experiments.

Selection of the Maximum Token Retention Bound. The purpose of $w_{\max}$ is to prevent available streaming tokens from remaining in the token pool for excessive time. Its value should therefore be sufficiently large to avoid prematurely terminating normal token accumulation, while remaining small enough to preserve the early token availability provided by streaming tokenization.

With $K = 512$, we profile the token arrival behavior under our hardware configuration and set $w_{\max} = 10$ ms. This value accommodates normal token accumulation in most cases while preventing unusually long token retention. We additionally evaluate neighboring values of 5–15 ms and observe limited sensitivity around 10 ms, indicating that **BridgeSTOP** does not rely on precise tuning within this range.

Extremely small values such as 0.1–0.2 ms frequently reach the retention bound before sufficient exact tokens are available, causing repeated speculative replenishment and introducing unnecessary speculation, verification, and rollback overhead. In contrast, extremely large values such as 50–100 ms allow already available streaming tokens to remain buffered for excessive time, delaying downstream prefill and increasing TTFT. Our experiments show that the former reduces sustainable serving capacity because of excessive replenishment overhead, while the latter reaches the TTFT constraint earlier because of prolonged token retention. We therefore use $w_{\max} = 10$ ms as a fixed value throughout all experiments.

D Additional Experimental Results and Discussions

D.1 Additional Serving Capacity Results

We further analyze the maximum sustainable serving capacity by examining which SLO constraint is first violated when the request rate increases to the next tested point above the maximum capacity. For ShareGPT4 and GSM8K, sustainable serving requires P50 TTFT $\leqslant 400$ ms, P99 TBT $\leqslant 200$ ms, and median scheduling delay $\leqslant 2000$ ms. MMLU excludes the TBT constraint because each request generates only one output token, while LongBench and CLongEval exclude the TTFT constraint because their long-context requests naturally incur substantially longer TTFT. The resulting failure modes are summarized in Table 5, where Sch denotes the median scheduling-delay constraint and multiple entries indicate that multiple SLO constraints are violated at the same tested request rate.

Table 5: SLO violation at the first tested request rate above the maximum sustainable serving capacity. Sch denotes scheduling delay.

Tokenization	Prefill	ShareGPT4	MMLU	GSM8K	LongBench	CLongEval
LoPT	Sarathi	TBT	TTFT	TTFT & TBT	Sch	TBT
zip2zip	Sarathi	TBT	TTFT	TTFT & TBT	TBT	Sch
MAGNET	Sarathi	TBT	TTFT	TBT	Sch	Sch
IncreBPE	Sarathi	TBT	TTFT	TTFT & TBT	Sch	TBT
SPEAK	Sarathi	TBT	TTFT	TTFT & TBT	TBT	Sch
LoPT	Stream2LLM	TTFT	TTFT	TTFT	Sch	Sch
zip2zip	Stream2LLM	TTFT	TTFT	TTFT	TBT	Sch
MAGNET	Stream2LLM	TTFT	TTFT	TTFT	TBT	Sch
IncreBPE	Stream2LLM	TTFT	TTFT	TTFT	Sch	Sch
SPEAK	Stream2LLM	TTFT	TTFT	TTFT	TBT	Sch
IncreBPE	**BridgeSTOP** w/o BAPT	TBT	TTFT	TTFT & TBT	Sch	TBT
SPEAK	**BridgeSTOP** w/o BAPT	TBT	TTFT	TTFT	TBT & Sch	Sch
IncreBPE	**BridgeSTOP**	TBT	TTFT	TTFT	Sch	TBT
SPEAK	**BridgeSTOP**	TBT	TTFT	TTFT	Sch	TBT & Sch

Overall, the limiting SLO varies substantially across workloads and serving configurations. On MMLU, all configurations are limited by TTFT because its single-token output eliminates inter-token decode latency from capacity determination. On ShareGPT4, TBT is the dominant limiting factor for Sarathi and **BridgeSTOP** configurations, showing that decode-side latency becomes critical near saturation, while Stream2LLM configurations first reach the TTFT constraint. GSM8K exhibits both TTFT and TBT limitations. Sarathi frequently violates both constraints simultaneously, whereas Stream2LLM and **BridgeSTOP** are more often limited by TTFT at their higher sustainable request rates. For the long-context LongBench and CLongEval workloads, TBT and scheduling delay become the primary limiting factors, reflecting the greater execution and scheduling pressure introduced by long requests.

These failure modes further show that improving tokenization and TP execution shifts the serving-capacity boundary rather than eliminating the remaining system bottlenecks. As tokenization and prefill become more efficient, the system can sustain higher request rates until another SLO constraint becomes dominant. For **BridgeSTOP**, the first violation at the next tested request rate is frequently determined by TBT or scheduling delay, which are not directly optimized along the tokenization-to-prefill path. This indicates that **BridgeSTOP** effectively reduces tokenization-to-prefill inefficiency and pushes the sustainable operating point toward downstream decode or system-level scheduling bottlenecks. Together with the maximum serving capacity results in Table 1, these results demonstrate that the serving gains of **BridgeSTOP** are achieved under the same SLO constraints across different workload-specific bottlenecks.

D.2 Additional Serving Latency Results

We provide additional serving latency results in Figure 4. Specifically, we report P50 TTFT on LongBench and CLongEval to complement the results in the main paper, and P99 TBT on ShareGPT4, GSM8K, LongBench, and CLongEval. MMLU is excluded from the TBT evaluation because each request generates only one output token and therefore has no inter-token latency.

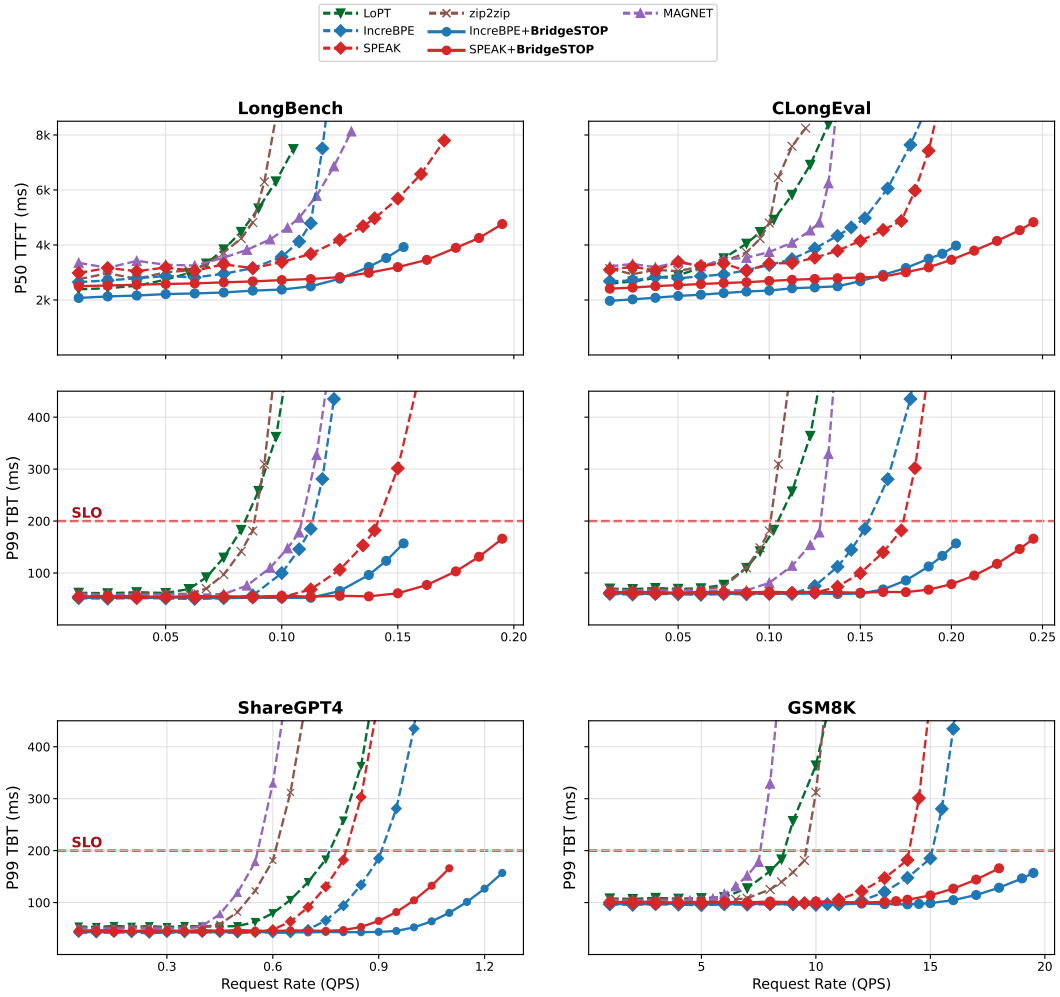

Figure 4: Additional serving latency results across request rates. The top row reports P50 TTFT on LongBench and CLongEval, while the remaining panels report P99 TBT on LongBench, CLongEval, ShareGPT4, and GSM8K. Dashed red lines indicate the 200 ms TBT SLO.

P50 TTFT. The LongBench and CLongEval results exhibit the same trend observed in the main paper. At low request rates, TTFT remains relatively stable because queueing and resource contention are limited. As the request rate increases, TTFT gradually rises and then increases sharply as the system approaches saturation. Compared with the baseline tokenizers paired with Sarathi, **BridgeSTOP** shifts this rapid TTFT increase to substantially higher request rates for both IncreBPE and SPEAK. This trend is particularly clear on the two long-context workloads, where tokenization and prefill require substantially more computation. The results therefore further show that the streaming tokenization efficiency achieved upstream can be preserved and propagated to downstream prefill even for long requests.

P99 TBT. Unlike TTFT, TBT mainly reflects decode-side scheduling, execution, and interference rather than tokenization latency itself. Accordingly, the differences among methods remain small at low request rates, where P99 TBT stays nearly constant across all configurations. As the request rate increases, however, baseline configurations eventually exhibit a sharp increase in P99 TBT and cross the 200 ms SLO as the system approaches saturation. **BridgeSTOP** consistently delays this degradation to higher request rates across ShareGPT4, GSM8K, LongBench, and CLongEval.

Importantly, **BridgeSTOP** does not directly optimize the decode stage. Its TBT advantage emerges indirectly as improved tokenization and TP execution allow the system to sustain higher request rates before queueing, prefill–decode interference, and resource contention become severe. The substantially extended low-TBT region therefore shows that the serving-capacity improvement of

BridgeSTOP is not obtained by sacrificing decode-side latency. Together with the TTFT results, these results demonstrate that **BridgeSTOP** translates streaming tokenization efficiency into higher end-to-end serving efficiency while maintaining both prefill-side responsiveness and decode-side latency under increasing request rates.

D.3 Additional Tokenization Efficiency Results for BAPT

This section provides a detailed ablation of BAPT and complete tokenizer-only results to complement the main evaluation, examining the effects of batch size and request length beyond the batch-size-64 setting reported in the main paper.

Scalability across Batch Sizes. The complete results across batch sizes are reported in Table 6, which further show that the latency advantage of BAPT becomes increasingly pronounced as the batch size grows. Larger batches improve throughput and hardware utilization but are more likely to contain requests with greater length variation, which increases idle computation from worker imbalance, padding, and batch-level synchronization. In contrast, BAPT removes batch formation and continuously assigns pending requests to available workers, making its execution independent of predefined batch size. For example, IncreBPE latency on ShareGPT4 increases from 2.39 ms at batch size 64 to 15.68 ms at batch size 512, whereas IncreBPE + BAPT remains at 0.73 ms, increasing its latency advantage from $3.3\times$ to $21.5\times$. Although larger batches can improve throughput for batch-bounded methods, they incur rapidly increasing idle computation and latency, while BAPT avoids this tradeoff.

Scalability across Request Lengths. Across all three metrics, a clear pattern emerges as request length increases: on the three shorter-request datasets, RToks achieve the best results, with LToks generally ranking second; whereas on the two longer-request datasets, LongBench and CLongEval, LToks achieve the best results, with RToks generally ranking second. Figure 5 quantifies this behavior using GSM8K and LongBench as representative short- and long-request datasets. On GSM8K, RTok latency remains low and stable as request length increases, while BAPT consistently reduces latency without affecting this stability. On LongBench, RTok latency increases substantially with request length, whereas LTok latency grows much more slowly. This difference arises because RTok computation scales more directly with input length, while LToks exhibit slower latency growth with request length, benefiting from GPU-parallel tensor computation. BAPT further removes waiting and idle computation caused by heterogeneous request lengths, particularly flattening SPEAK's latency curve. Overall, BAPT improves tokenization efficiency while maintaining latency stability for short requests and improving scalability as request length increases.

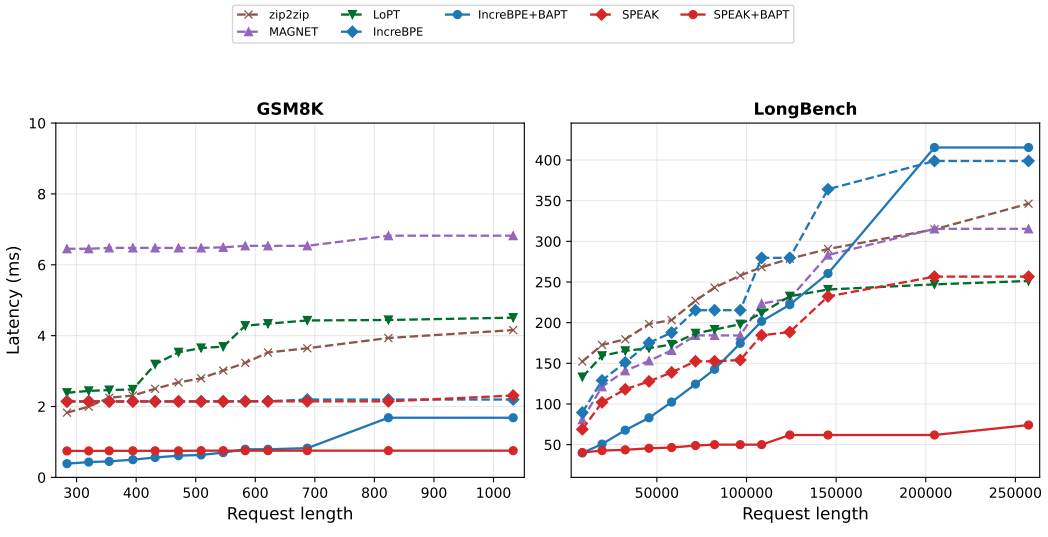

Figure 5: Average tokenization latency as request length increases on GSM8K and LongBench. The left panel shows the short-request regime, while the right panel shows the long-request regime.

Table 6: Tokenization efficiency with different batch sizes. (**Best**, **Second Best**)

(a) Batch size 128.

Dataset	Metric	LoPT	zip2zip	MAGNET	IncreBPE	SPEAK	IncreBPE + BAPT	SPEAK + BAPT
ShareGPT4	*Throughput (MiB/s)*	16.13	19.90	6.58	**24.32**	16.31	**34.06**	18.92
	Avg. Latency (ms)	5.84	4.76	14.22	4.27	6.25	**0.73**	**1.31**
	P95 Latency (ms)	28.54	26.79	20.58	12.02	15.09	**3.69**	**7.67**
MMLU	*Throughput (MiB/s)*	16.18	18.59	8.70	19.19	17.13	**28.93**	**20.31**
	Avg. Latency (ms)	4.93	4.31	9.18	4.10	4.26	**0.67**	**0.96**
	P95 Latency (ms)	11.76	10.03	12.89	9.56	5.93	**1.62**	**2.13**
GSM8K	*Throughput (MiB/s)*	10.73	12.87	4.49	14.96	14.53	**23.34**	**17.32**
	Avg. Latency (ms)	5.06	4.20	11.22	3.72	3.65	**0.55**	**0.75**
	P95 Latency (ms)	7.13	5.70	12.69	5.76	5.27	**0.81**	**1.04**
LongBench	*Throughput (MiB/s)*	17.82	15.36	20.98	19.40	**27.51**	16.62	**28.92**
	Avg. Latency (ms)	288.25	333.02	235.84	270.49	192.49	**78.64**	**44.41**
	P95 Latency (ms)	553.49	646.61	353.07	483.24	312.39	**130.04**	**97.88**
CLongEval	*Throughput (MiB/s)*	18.51	17.88	21.17	20.64	**45.21**	20.56	**47.89**
	Avg. Latency (ms)	843.34	940.43	802.62	747.72	363.43	**204.57**	**87.82**
	P95 Latency (ms)	1705.65	1962.05	1658.92	1630.21	717.42	**319.74**	**203.57**

(b) Batch size 256.

Dataset	Metric	LoPT	zip2zip	MAGNET	IncreBPE	SPEAK	IncreBPE + BAPT	SPEAK + BAPT
ShareGPT4	*Throughput (MiB/s)*	19.08	21.71	7.18	**23.61**	15.87	**34.06**	18.92
	Avg. Latency (ms)	10.76	8.93	26.00	8.12	11.83	**0.73**	**1.31**
	P95 Latency (ms)	50.57	35.86	36.04	21.75	27.32	**3.69**	**7.67**
MMLU	*Throughput (MiB/s)*	17.81	19.10	9.40	**21.01**	18.71	**28.93**	20.31
	Avg. Latency (ms)	9.10	7.93	16.64	7.56	7.92	**0.67**	**0.96**
	P95 Latency (ms)	22.08	16.44	21.07	17.07	10.12	**1.62**	**2.13**
GSM8K	*Throughput (MiB/s)*	10.61	12.95	5.38	15.81	14.08	**23.34**	17.32
	Avg. Latency (ms)	9.50	7.80	19.99	6.93	6.89	**0.55**	**0.75**
	P95 Latency (ms)	12.81	10.71	22.39	9.56	9.52	**0.81**	**1.04**
LongBench	*Throughput (MiB/s)*	18.89	16.66	22.47	20.38	**31.20**	16.62	**28.92**
	Avg. Latency (ms)	539.05	615.13	438.05	507.92	345.25	**78.64**	**44.41**
	P95 Latency (ms)	998.93	1094.44	633.43	904.65	567.45	**130.04**	**97.88**
CLongEval	*Throughput (MiB/s)*	19.71	20.11	21.06	24.35	**47.99**	20.56	**47.89**
	Avg. Latency (ms)	1553.54	1695.32	1488.50	1360.18	677.06	**204.57**	**87.82**
	P95 Latency (ms)	3241.69	3541.02	2906.05	2675.39	1316.49	**319.74**	**203.57**

(c) Batch size 512.

Dataset	Metric	LoPT	zip2zip	MAGNET	IncreBPE	SPEAK	IncreBPE + BAPT	SPEAK + BAPT
ShareGPT4	*Throughput (MiB/s)*	18.50	22.57	8.30	**24.52**	16.47	**34.06**	18.92
	Avg. Latency (ms)	20.62	16.98	48.56	15.68	22.83	**0.73**	**1.31**
	P95 Latency (ms)	70.65	67.61	64.34	37.58	48.11	**3.69**	**7.67**
MMLU	*Throughput (MiB/s)*	17.01	**21.89**	9.17	21.82	19.58	**28.93**	20.31
	Avg. Latency (ms)	17.42	14.85	31.99	14.37	15.14	**0.67**	**0.96**
	P95 Latency (ms)	38.20	31.89	40.43	27.57	18.74	**1.62**	**2.13**
GSM8K	*Throughput (MiB/s)*	10.99	13.42	5.73	16.72	16.27	**23.34**	17.32
	Avg. Latency (ms)	18.20	14.98	37.39	13.18	12.95	**0.55**	**0.75**
	P95 Latency (ms)	24.18	20.03	41.88	17.81	16.79	**0.81**	**1.04**
LongBench	*Throughput (MiB/s)*	21.78	17.54	25.66	23.10	**29.44**	16.62	**28.92**
	Avg. Latency (ms)	1018.61	1170.40	823.70	962.87	663.93	**78.64**	**44.41**
	P95 Latency (ms)	1664.55	1941.20	1100.72	1518.72	1027.19	**130.04**	**97.88**
CLongEval	*Throughput (MiB/s)*	20.95	21.33	24.49	23.91	**54.74**	20.56	**47.89**
	Avg. Latency (ms)	2981.48	3183.99	2797.69	2604.24	1275.42	**204.57**	**87.82**
	P95 Latency (ms)	5195.91	6026.60	5182.43	4610.15	2238.32	**319.74**	**203.57**