STREAM2LLM: OVERLAP CONTEXT STREAMING AND PREFILL FOR REDUCED TIME-TO-FIRST-TOKEN

Rajveer Bachkaniwala [1] **Chengqi Luo** [1] **Richard So** [1] **Divya Mahajan** [1] **Kexin Rong** [1]

ABSTRACT

Context retrieval systems for LLM inference face a critical challenge: high retrieval latency creates a fundamental tension between waiting for complete context (poor time-to-first-token) and proceeding without it (reduced quality). Streaming context incrementally–overlapping retrieval with inference–can mitigate this latency, but doing so with concurrent requests introduces new challenges: requests contend for GPU compute and memory, and scheduling must adapt to dynamic context arrivals.

We present STREAM2LLM, a streaming-aware LLM serving system for concurrent prefill-decode disaggregated deployments. STREAM2LLM introduces adaptive scheduling and preemption for two distinct retrieval patterns: *append-mode* (progressive context accumulation) and *update-mode* (iterative refinement with cache invalidation). It decouples scheduling decisions from resource acquisition, enabling flexible preemption strategies guided by hardware-specific cost models, and uses longest common prefix matching to minimize redundant computation when input changes dynamically. To evaluate STREAM2LLM, we collect two large-scale, real-world streaming workloads based on web crawling and approximate nearest neighbor search. Our evaluation demonstrates that streaming architecture delivers up to 11× TTFT improvements, with cost-aware scheduling providing critical benefits under memory pressure, all while maintaining throughput parity with non-streaming baselines.

Code: https://github.com/rajveerb/stream2llm/

1 INTRODUCTION

Modern large language models (LLMs) increasingly rely on external context retrieval to provide accurate, up-to-date responses (Gao et al., 2024; Lewis et al., 2020; Xu et al., 2023). Context retrieval mechanisms, such as fetching relevant pages from web search or vector search (ANNS) over large vector databases, can take hundreds of milliseconds to seconds. This creates a fundamental tension between two undesirable outcomes: the system can either wait for all context to arrive, which increases time-to-first-token and degrades user interactivity, or begin generation early with partial context, which can reduce response quality.

A natural approach is to stream context incrementally, feeding chunks to the model as they arrive rather than waiting for retrieval to complete (Figure 1). In single-request settings, this can significantly reduce latency (Jiang et al., 2024; Yu et al., 2025). Production deployments, however, must balance per-user interactivity against serving cost: batching concurrent requests onto shared GPU compute and memory improves throughput and profit margins but directly increases time-to-first-token (key metric for responsiveness). Moreover, context arrives asynchronously and at varying rates across requests, requiring the system to dynamically adapt scheduling while managing competition for shared GPU resources.

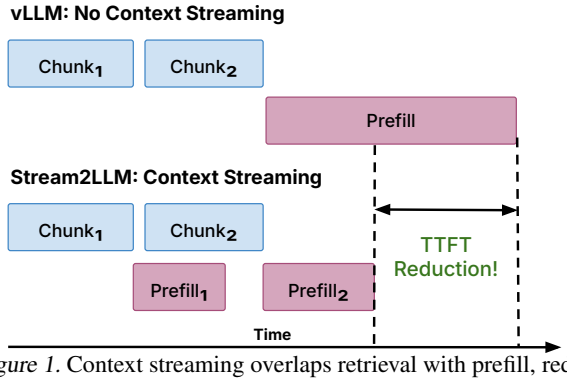

Figure 1. Context streaming overlaps retrieval with prefill, reducing TTFT by beginning inference as chunks arrive.

Streaming with concurrent requests introduces three interrelated challenges. First, GPU memory becomes contested: each request maintains KV cache blocks for its growing input, and memory exhaustion forces preemption—either swapping cache to CPU or discarding it for later recomputation. Second, the scheduler must decide which requests

[1]Georgia Tech, Atlanta, USA. Correspondence to: Rajveer Bachkaniwala <rr@gatech.edu OR @rajveerbach on X.com>.

Proceedings of the 9th MLSys Conference, Seattle, WA, USA, 2026. Copyright 2026 by the author(s).

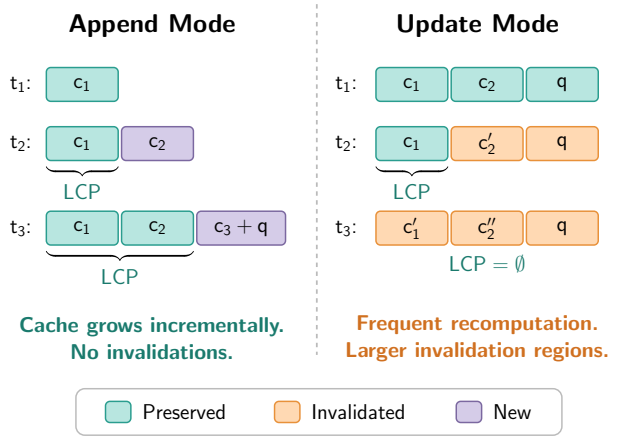

Figure 2. STREAM2LLM supports two retrieval patterns–append-mode and update-mode–and uses longest common prefix (LCP) matching to minimize cache invalidation by preserving shared prefixes across input updates.

to prioritize as context chunks arrive at different times, balancing responsiveness, fairness, and cache locality. Third, context retrieval itself exhibits two distinct patterns (Figure 2): *append-mode*, where chunks progressively extend the input, and *update-mode*, where the context set changes iteratively as search algorithms refine results. Naïve approaches that treat all streaming requests uniformly suffer from poor scheduling or excessive cache recomputation.

We present STREAM2LLM, a system that extends the vLLM inference engine (Kwon et al., 2023) to support streaming inputs with concurrent requests. STREAM2LLM targets the prefill stage in a prefill-decode disaggregated deployment (Patel et al., 2024; Qin et al., 2024; Zhong et al., 2024), following industry practice. This architecture enables optimization for time-to-first-token (TTFT) and throughput, where streaming decisions matter most, without impacting decode-stage latency, i.e., time-per-output-token (TPOT), which is handled by separate decode instances.

Specifically, STREAM2LLM introduces a two-phase scheduling architecture that decouples priority-based request ordering from resource acquisition. This separation of concerns–deciding what to run versus how to allocate resources–enables flexible preemption strategies without hardcoded policies. When memory is exhausted, the system preempts low-priority requests using cost-based decisions guided by hardware-specific performance models, choosing between recomputing invalidated cache and swapping cache blocks to CPU. To minimize redundant computation when inputs change dynamically, STREAM2LLM uses longest common prefix (LCP)-based cache invalidation, preserving cache for unchanged prefix tokens while invalidating only changed portions. This mechanism uses a single cache management strategy for both append-mode and update-mode retrieval patterns.

We conduct a comprehensive evaluation using real-world streaming traces collected from web crawling and ANNS-based retrieval systems, capturing both append-mode and update-mode retrieval patterns. Experiments on H100 and H200 GPUs with Llama-3.1 models show that *while streaming architecture provides substantial benefits across different schedulers, intelligent scheduling becomes increasingly important as load increases*. At low loads, all streaming schedulers converge to similar performance, improving time-to-first-token by up to 3.9-11.0× over non-streaming baselines. However, as concurrency and memory contention increase, scheduler choice becomes critical: proper request prioritization strategies enable efficient cache reuse and maintain responsiveness, while naïve scheduling causes catastrophic tail latency (up to 10× worse under extreme memory pressure). These findings hold across GPU architectures and workload types, demonstrating that streaming is necessary but insufficient; intelligent scheduling and cache management is essential for making streaming viable in production deployments.

In summary, this paper makes the following contributions:

- We present STREAM2LLM, a system that extends vLLM to support concurrent streaming inputs in LLM inference, handling both *append-mode* (progressive context accumulation) and *update-mode* (iterative context refinement) workloads.
- We introduce a two-phase scheduling architecture that decouples request prioritization from GPU memory allocation and preemption, enabling flexible policies that improve KV cache reuse. This is combined with cost-based adaptive preemption (recomputation vs. swapping) and longest common prefix-based cache invalidation to reduce redundant computation for dynamically changing inputs.
- We evaluate STREAM2LLM using real-world, streaming traces from web crawling and ANNS-based retrieval systems. Results show that streaming with intelligent scheduling and cache management delivers up to 11× improvements in TTFT.

2 BACKGROUND

2.1 LLM Inference and KV Cache Management

Large language model inference follows an autoregressive generation process consisting of two distinct phases. Given an input with n tokens $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, the *prefill phase* processes all input tokens in parallel, computing attention scores and generating key-value pairs (K_i, V_i) for each token position $i \in [1, n]$. These KV pairs are stored in GPU memory as the KV cache. The *decode phase* then generates output tokens autoregressively: at each step t, the model attends to all previously computed

KV pairs $\{(K_1, V_1), \ldots, (K_{n+t-1}, V_{n+t-1})\}$ to generate the next token x_{n+t}. The newly generated token's KV pair (K_{n+t}, V_{n+t}) is appended to the cache, avoiding recomputation of attention for tokens processed in earlier steps.

For a transformer model with L layers, hidden dimension d, total attention heads h, and h_{kv} key-value heads (head dimension $d_h = d/h$), each token position stores $2Ld\frac{h_{\text{kv}}}{h}$ values across all layers (keys and values). The total KV cache memory for a sequence of length ℓ is $M_{\text{KV}} = 2L\ell d\frac{h_{\text{kv}}}{h} \cdot b$, where b is the number of bytes per parameter (typically 2 bytes for FP16). For Meta Llama-3.1-8B with $L = 32$, $d = 4096$, $h = 32$, $h_{\text{kv}} = 8$, and $\ell = 32$K, this amounts to $M_{\text{KV}} \approx 4.0$ GB per request in FP16 precision. When serving B concurrent requests through batching, total memory consumption becomes $B \cdot M_{\text{KV}}$. As B increases to improve throughput, available memory per request decreases, forcing systems to either limit batch size or preempt requests to free KV cache.

2.2 vLLM and PagedAttention

Traditional KV cache implementations allocate a single contiguous memory region of size M_{KV} for each request, leading to memory fragmentation as requests arrive and complete at different times. vLLM (Kwon et al., 2023) addresses this through PagedAttention, which divides the KV cache into fixed-size blocks. For a block size of k tokens, the KV cache for a sequence is stored as a sequence of blocks $\mathcal{B} = \{B_1, B_2, \ldots, B_{\lceil \ell/k \rceil}\}$, where each block B_j stores KV pairs for tokens $[(j-1)k+1, \min(jk, \ell)]$. These blocks need not be contiguous in physical memory, similar to virtual memory paging in operating systems.

The attention computation in PagedAttention operates blockwise. For computing attention at position t, the system retrieves all blocks $\{B_1, \ldots, B_{\lceil t/k \rceil}\}$ containing KV pairs for positions $[1, t]$, regardless of their physical memory locations. This design provides two key benefits: (1) reduced fragmentation by allocating blocks on-demand from a free block pool, and (2) efficient prefix sharing when multiple requests share common input prefixes, as blocks containing the shared prefix can be referenced by multiple requests without duplication (Gim et al., 2024).

vLLM supports continuous batching (Yu et al., 2022), maintaining a dynamic batch $\mathcal{R}_t = \{r_1, r_2, \ldots, r_{B_t}\}$ where requests can be added or removed at any scheduling step t. When total KV cache memory exceeds GPU capacity M_{GPU}, the system must preempt requests. For a request r with ℓ_r processed tokens occupying $\lceil \ell_r/k \rceil$ blocks, preemption follows one of two strategies:

- **Recomputation**: Discard all KV cache blocks for r, freeing $\lceil \ell_r/k \rceil \cdot M_{\text{block}}$ memory (where $M_{\text{block}} = 2Lkd\frac{h_{\text{kv}}}{h} \cdot b$). Upon resumption, recompute the prefill phase for all ℓ_r tokens, incurring compute cost $C_{\text{recomp}}(r) = \ell_r \cdot C_{\text{prefill}}$ where C_{prefill} is the per-token prefill latency.
- **Swapping**: Transfer all blocks $\{B_1, \ldots, B_{\lceil \ell_r/k \rceil}\}$ from GPU to CPU memory, incurring data transfer cost $C_{\text{swap}}(r) = \frac{\lceil \ell_r/k \rceil \cdot M_{\text{block}}}{BW_{\text{PCIe}}}$ where BW_{PCIe} is the PCIe bandwidth. Upon resumption, swap blocks back to GPU with symmetric cost. The blocks preserve computed KV values, avoiding recomputation.

Selecting between the two strategies requires comparing $C_{\text{recomp}}(r)$ versus $2 \cdot C_{\text{swap}}(r)$ (accounting for bidirectional transfer); STREAM2LLM uses this cost model to guide its adaptive preemption decisions, as described in §4.3.

2.3 Context Retrieval for LLMs

Modern LLM deployments require external context $\mathcal{C}$ to augment the model's parametric knowledge (Lewis et al., 2020). Given a query q, context retrieval systems produce a set of relevant documents $\mathcal{D} = \{d_1, d_2, \ldots, d_m\}$ that are concatenated with q to form the final input. Two retrieval mechanisms are commonly used:

Web crawler retrieval: Given a query q, the system fetches documents from the web or internal knowledge bases. Latency T_{web} includes network round-trip time, DNS resolution, content fetching, and parsing, typically ranging from 200ms to multiple seconds per document.

ANNS-based retrieval: Given a query embedding $\mathbf{e}_q \in \mathbb{R}^{d_e}$ and a corpus $\mathcal{C} = \{c_1, \ldots, c_N\}$ with embeddings $\{\mathbf{e}_{c_1}, \ldots, \mathbf{e}_{c_N}\}$, find the k-nearest neighbors $\mathcal{D} = \text{top-}k(\{c_i : \text{sim}(\mathbf{e}_q, \mathbf{e}_{c_i}) \mid c_i \in \mathcal{C}\})$, where $\text{sim}(\cdot, \cdot)$ is a similarity metric (*e.g.*, cosine similarity or L2 distance). While in-memory ANNS solutions such as HNSW (Malkov & Yashunin, 2018) achieve low retrieval latency, storing the full index in memory is infeasible at scale ($N \sim 10^9$). Disk-based ANNS solutions such as DiskANN (Jayaram Subramanya et al., 2019) and SPANN (Chen et al., 2021) address this by storing indexes on disk, where search performs $O(\log N)$ graph traversals with disk I/O at each step. They are widely deployed in production due to cost: SSDs are orders of magnitude cheaper per capacity than DRAM (Karsin et al., 2025; Li, 2025). End-to-end retrieval latency T_{ANNS} ranges from 100ms to several seconds depending on corpus size, index complexity parameter (e.g., search list size in DiskANN), disk bandwidth, and pipeline overhead (query embedding, network, post-processing).

2.4 Traditional vs Streaming Inference

Traditional inference systems construct the complete input $\mathbf{x} = [\mathcal{D}, q]$ only after retrieval finishes at time T_{retrieve}, then begin the prefill phase. This results in time-to-first-token (TTFT) of $T_{\text{TTFT}} = T_{\text{retrieve}} + T_{\text{prefill}}(|\mathcal{D}| + |q|)$, where

T_{retrieve} often dominates. Streaming context approaches reduce TTFT by sending context chunks incrementally. Instead of waiting for complete $\mathcal{D}$, documents arrive over time: d_1 at time t_1, d_2 at time t_2, etc. The LLM begins inference with partial context $\mathcal{D}_t = \{d_i : t_i \leq t\}$ and continues as new documents arrive.

We consider two complementary streaming modes that capture distinct classes of retrieval workloads. In *append mode*, context grows monotonically ($\mathcal{D}_t \subseteq \mathcal{D}_{t'}$ for $t < t'$), corresponding to pipelines that produce results incrementally without revisiting earlier results. For example, in web crawling pipelines, documents arrive as pages are retrieved and can be streamed immediately when no global post-processing (e.g., reranking) is required. More broadly, append mode applies whenever documents are emitted in final form upon arrival. In *update mode*, the context is refined over time, with $\mathcal{D}_t$ evolving as higher-quality candidates replace earlier ones, corresponding to pipelines that progressively refine retrieval results. For example, AquaPipe (Yu et al., 2025) enables early return of partial ANNS results before search completion; Vespa implements phased ranking where documents are progressively filtered and re-ranked across stages (Vespa Team, 2024); and distributed search engines such as Elasticsearch similarly emit partial results as individual shards complete. These pipelines naturally produce a sequence of refined top-k sets, which can be streamed to the inference system.

These modes have been explored separately in prior systems—PipeRAG (Jiang et al., 2024) for append and AquaPipe (Yu et al., 2025) for update—and have been evaluated in single-request settings ($B = 1$). In this work, we support both modes under a unified concurrent setting.

3 Challenges

Streaming concurrent inference workloads differ fundamentally from static batch inference. Requests stream context incrementally over time, evolve asynchronously, and compete for shared GPU resources. These unique properties create several system design challenges.

Dynamic Workload Variability. Each request expands dynamically as new chunks of context arrive and generates outputs of unpredictable length until termination tokens appear. This variability makes both compute demand and memory footprint time-dependent, and requires dynamic policies.

Chunk arrivals are asynchronous: some requests stall waiting for retrieval while others suddenly receive new context. Requests that just received a chunk should be prioritized to process it while the request's KV cache blocks remain allocated in GPU memory, but traditional scheduling policies (*e.g.*, FCFS or progress-based heuristics) cannot capture this temporal priority. Schedulers must dynamically re-rank requests as chunk arrival patterns shift, balancing freshness, fairness, and preemption cost.

All requests draw from the same finite pool of GPU memory for KV cache blocks. As input sequences grow, total cache usage can exceed GPU capacity, forcing the system to free memory through preemption. The optimal strategy–whether to recompute discarded cache or to swap it to CPU memory–depends on request progress, hardware bandwidth ratios, and expected resumption time.

Mode-Dependent Context Evolution. Streaming retrieval workloads differ in how input sequences evolve. In web-crawler-style retrievers (append mode), requests extend the existing sequences, whereas in ANNS-style retrievers (update mode), requests replace parts of the sequence as retrieval refines results. These modes require different scheduling and cache management behaviors.

In update mode, the system must determine which KV cache blocks remain valid after each update. If the scheduler naively invalidates all blocks, it wastes memory by discarding valid cached results and forces expensive recomputation. Conversely, if the scheduler reuses blocks without verification, it risks computing incorrect output based on stale cache, since subsequent attention computations would attend to outdated key-value pairs that no longer correspond to the current input sequence. Frequent replacements early in the sequence can lead to high recomputation costs.

Append and update workloads also differ in optimal preemption strategies. Append-mode requests favor swapping to preserve reusable cache, while update-mode requests often prefer recomputation to avoid retaining soon-invalid data.

4 System Design

Deployment Assumption. Stream2LLM targets the prefill instance in a prefill-decode disaggregated architecture (Qin et al., 2024), where prefill and decode run on separate GPU pools. This is standard practice in production LLM serving (Patel et al., 2024; Zhong et al., 2024). TTFT and throughput are the metrics relevant to prefill instances; token generation latency (TPOT) is handled by decode instances and is not evaluated.

Stream2LLM addresses the challenges with concurrent, streaming workloads with two key design principles:

- **Support for two distinct streaming modes:** Requests can either append new input chunks to the existing sequence or replace parts of the input sequence entirely. Each mode imposes different requirements on cache management, scheduling, and preemption.
- **Decoupling scheduling from resource acquisition:**

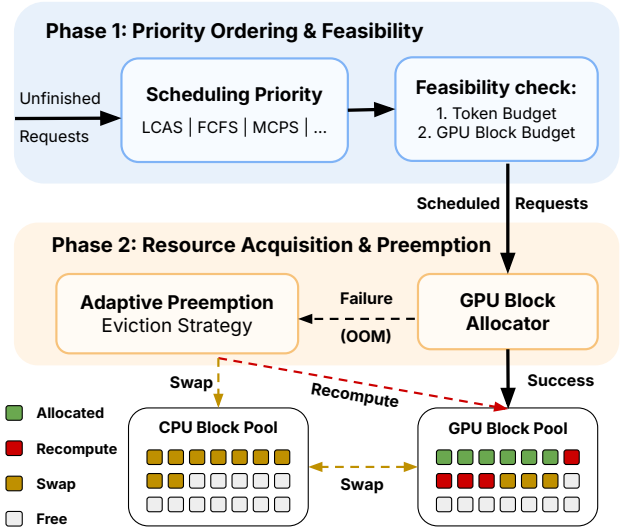

Figure 3. Overview of STREAM2LLM's two-phase scheduler that separates scheduling decisions from resource acquisition. Phase 1 determines request priority and feasibility, while Phase 2 allocates KV cache blocks and applies preemption under memory pressure.

STREAM2LLM separates the decision of which request to prioritize from the mechanics of allocating GPU memory and applying preemption. This decoupling enables sophisticated policies that maximize KV cache reuse while adapting to dynamic workloads.

The subsequent subsections introduce the two-phase scheduling design (§ 4.1), cache invalidation mechanisms (§ 4.2), cost-based preemption strategies (§ 4.3), and scheduling policies (§ 4.4).

4.1 Two-Phase Scheduling and Request Lifecycle

A key insight in STREAM2LLM is the separation of *scheduling decisions* from *resource acquisition*. Rather than embedding both in a single control loop, STREAM2LLM organizes them into two distinct phases, allowing scheduling policies to reason about priorities independently of allocation and preemption mechanics. Figure 3 illustrates this two-phase architecture.

When scheduling, allocation, and preemption are tightly coupled, several challenges arise:

- *Fixed Policy Coupling.* A monolithic loop forces the scheduler to preempt immediately when resources are exhausted, using a static rule that may contradict the policy's notion of priority. This prevents different scheduling algorithms from expressing distinct preemption behaviors.
- *Cache Invalidation Coupling.* When an input sequence is updated, two things must happen: the token count increases and cached KV blocks may become invalid. In a monolithic loop, these operations interleave with scheduling and allocation, making it unclear whether to invalidate before or after attempting allocation.

STREAM2LLM decouples scheduling and resource acquisition into two phases that explicitly separate concerns:

Phase 1: Priority Ordering and Feasibility Analysis. The scheduler invokes the selected scheduling algorithm (§ 4.4) to compute an ordered list of all unfinished requests ranked by priority. It then performs a feasibility analysis that determines which requests can theoretically be scheduled given the current token budget (i.e., the maximum number of tokens that can be processed in a scheduling step) and estimated GPU block requirements. Specifically, for each request in priority order, the scheduler estimates the number of KV cache blocks required (based on computed and new tokens) and checks whether sufficient free GPU blocks remain; if not, the request is marked infeasible. Crucially, this phase performs no resource allocation and modifies no request state—it only computes feasibility. Requests that cannot fit within current resources are added to a `not_scheduled_reqs` list while preserving their priority.

Phase 2: Resource Acquisition with Adaptive Preemption. The scheduler attempts to allocate GPU blocks for each request selected in Phase 1. If allocation fails, the scheduler selects a preemption candidate from the `not_scheduled_reqs` list in reverse priority order (i.e., lowest-priority first). Because requests in `not_scheduled_reqs` may still hold KV cache blocks from previous scheduling steps, preempting them releases those blocks, freeing GPU memory to proceed with allocation for higher-priority requests. The scheduler then applies the decision framework from § 4.3 to choose between recomputation and swapping as the eviction strategy.

This two-phase structure separates *what* to run from *how* to allocate resources and provides several benefits. The separation ensures that priority decisions are made independently of resource constraints, and preemption naturally follows the scheduling priority order without requiring policy-specific rules. Cache invalidation occurs at precise boundaries between analysis and allocation. The resource-acquisition phase can also flexibly integrate cost-, or latency-aware strategies without altering core scheduling logic.

Request Lifecycle. The scheduler tracks requests through three primary queues: `waiting` (arrived but not yet scheduled), `running` (currently in execution or scheduled in the current batch), and `finished` (completed). Transitions between these queues are governed by the two-phase design (Figure 4). Requests move from `waiting` to `running` when selected in Phase 1 and allocated in Phase 2. If a running request is preempted, it returns to `waiting`: recomputation resets its progress, while swapping preserves

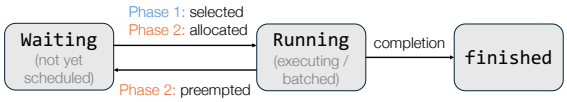

Figure 4. STREAM2LLM request state transitions.

state by offloading KV blocks to CPU memory. Upon completion, requests transition from `running` to `finished`.

4.2 KV Cache Invalidation for Streaming Inputs

STREAM2LLM supports two input sequence modification modes suited to different context retrieval patterns:

- **Append Mode:** New input chunks are appended to the existing sequences, typical in crawler-style workloads.
- **Update Mode:** The entire input sequence is updated dynamically, typical in ANNS-style workloads.

A unique challenge for streaming input workloads is that the input token sequence itself changes dynamically. When a new input chunk arrives, the scheduler must decide which cached KV blocks remain valid and which must be invalidated. Traditional LLM inference assumes a fixed input at request arrival, computing KV cache blocks once for the entire request lifetime. However, in streaming context retrieval workloads, the input grows or changes as new document chunks arrive. For example, the input may evolve from $[d_1, q]$ (document chunk 1 + query) to $[d_1, d_2, q]$ as document chunks are added, or from $[d_1, d_2, q]$ to $[d'_1, d_2, q]$ when chunks are replaced in ANNS-style systems. If the scheduler naively invalidates all KV cache blocks upon each input update, it wastes memory by discarding previously computed blocks that remain valid and forces expensive recomputation. Conversely, if the scheduler reuses blocks without verification, it risks computing incorrect output based on stale cache.

Longest Common Prefix Invalidation. STREAM2LLM addresses this by computing the longest common prefix (LCP) between the old and new input token sequences, as illustrated in Figure 2. When a streaming request receives a new input chunk, the engine computes the LCP and invalidates only the KV cache blocks corresponding to tokens *beyond* the LCP. Blocks for tokens within the LCP are preserved, avoiding redundant recomputation. The LCP approach is optimal when input updates involve appending new chunks or replacing suffix tokens, which is typical in context retrieval systems. It becomes less beneficial when updates are frequent or affect early tokens.

For example, suppose a request previously computed KV cache for tokens $[d_1, d_2, q, \text{output}_1, \text{output}_2]$, and an input update replaces the input with $[d_1, d'_2, q, \text{output}_1, \text{output}_2]$, where $d'_2 \neq d_2$. The LCP is then $[d_1]$ (length 1). The scheduler invalidates cache blocks for tokens 1 onward (corresponding to d_2, q, and output) while preserving the KV cache for token 0 (document chunk 1).

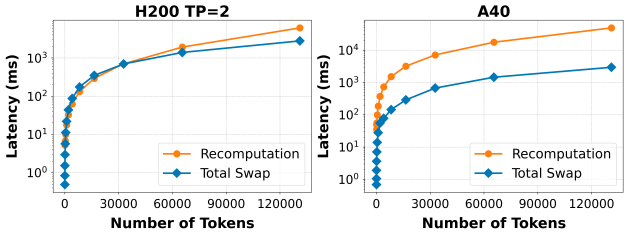

Figure 5. Performance models for recomputation vs. total swap latency costs across token counts on H200 and A40.

For analysis, STREAM2LLM tracks the total number of tokens invalidated across all input updates per request via `total_tokens_invalidated`. This metric is reported when the request finishes, allowing researchers to measure the cost of cache invalidation in context retrieval workloads. High invalidation counts indicate frequent or aggressive input updates, suggesting that the system is spending significant effort on recomputation.

Cache Invalidation Semantics. When k tokens are invalidated beyond the LCP, the scheduler frees the corresponding KV cache blocks and returns them to the free block pool. For CPU-swapped requests, the scheduler also frees the corresponding CPU blocks to reduce memory pressure. The scheduler then sets `num_computed_tokens` to the LCP length, forcing recomputation only for tokens that changed.

The interaction between cache invalidation and preemption requires careful sequencing to avoid freeing blocks that are about to be swapped back in. When a preempted request receives an input update, the scheduler invalidates blocks from the LCP onward on CPU and free these blocks from CPU memory. When the request resumes, it first swaps its remaining blocks (those before the LCP) back to GPU, then recomputes tokens from the LCP onward. This design ensures that input updates do not cause memory leaks or inconsistencies in the swapped state.

4.3 Cost-based Preemption

When GPU memory is exhausted, STREAM2LLM preempts requests and chooses between recomputation and swapping strategies based on cost models. STREAM2LLM profiles two hardware-specific latency cost functions offline on each target platform, such as shown in Figure 5. *recomputation_latency*(T) predicts the time to recompute T tokens via prefill. We measure prefill latency across varying token counts (1K-128K) on each GPU (A40, H100, H200) and fit a piecewise-linear model to account for memory bandwidth saturation. *swap_latency*(C) predicts the time to swap C KV cache blocks between GPU and CPU memory. We measure transfer latency for typical block sizes 2 MB (for the given model Llama 3.1 8B, and default block size of

Table 1. Scheduling algorithm performance on streaming workloads.

	Append Mode		Update Mode	
Policy	**Rating**	**Key Behavior**	**Rating**	**Key Behavior**
DEFAULT vLLM	Poor	Ignores arrival timing	Poor	Ignores recomputation cost
FCFS	Good	No recency awareness	Moderate	No update prioritization
MCPS	Good	Tracks progress	Poor	Unstable under updates
LCAS	Good	Prioritizes recent input	Good	Prioritizes recent updates

16) and account for PCIe bandwidth constraints. On systems with NVLink or faster interconnects (e.g., NVIDIA GB300 and GH200 has up to 450 GB/s host–device bandwidth), swap costs are correspondingly lower.

These cost functions are hardware-specific: a recomputation that takes 50ms on H100 may take 200ms on A40 due to differences in compute density and memory bandwidth. The scheduler evaluates both cost functions at preemption time and selects the strategy with lower predicted latency. The cost functions can be updated dynamically to adapt to changing system characteristics (e.g., contention for PCIe bandwidth), though our current implementation uses static profiles derived from idle-system benchmarks.

4.4 Scheduling and Eviction Policies

The scheduling algorithm determines request execution order. Different policies optimize different tradeoffs: throughput, latency, fairness. Each policy selects its lowest-priority request for eviction when GPU memory fills, and uses the cost model from § 4.3 to decide between recomputation and swapping strategies.

STREAM2LLM implements multiple scheduling algorithms with different trade-offs, selected via the `SCHEDULER_TYPE` variable. Table 1 summarizes each algorithm's performance under different streaming workloads.

4.4.1 DEFAULT vLLM (variant of FIFO)

The baseline scheduler uses arrival-time ordering for the waiting queue, processing new requests in FIFO order. For running requests, it maintains their current execution order without explicit arrival-time re-sorting. When GPU memory is exhausted, it preempts the last request in the running queue (LIFO eviction). Preempted requests are re-queued at the front of the waiting queue, bypassing newly arrived requests to maintain priority.

Default vLLM ignores both when input chunks arrive and when sequences are updated. A request that arrived early but has been idle receives higher priority than a request that arrived later but has freshly received input or updates, delaying progress on active requests. In update mode, DEFAULT vLLM prioritizes based on arrival time rather than recomputation cost. This causes requests with less recomputation work to be delayed behind requests requiring significant recomputation, extending their latency unnecessarily.

4.4.2 FCFS (First-Come-First-Served)

This scheduler separates requests into two tiers: *full requests* (input sequence complete) scheduled in arrival order, and *partial requests* (still receiving input) scheduled opportunistically. Eviction removes requests in reverse scheduling order when memory is exhausted.

The two-tier structure improves upon FIFO by separating complete from incomplete requests and prioritizing completed requests for output generation. However, FCFS shares FIFO's limitations: within each tier, scheduling uses only arrival order and ignores both chunk arrival recency and recomputation costs. Requests with older arrivals are prioritized over newer requests with fresh input or high-cost updates, delaying progress on active requests.

4.4.3 MCPS (Most Chunks Processed Scheduling)

Requests are prioritized by the number of tokens already computed (`num_computed_tokens`, highest first), with ties broken by arrival time. Eviction removes the request with the fewest computed tokens.

MCPS's progress-based metric works well in append mode, and naturally favors completing requests before starting new ones. Requests receiving chunks frequently maintain high priority and make steady progress. MCPS becomes problematic in update mode. When a request's input sequence is updated with a short LCP, `num_computed_tokens` resets to the LCP length (potentially near zero). A request that had computed many tokens suddenly drops to lowest priority, even though significant work has been done. This wastes prior computation and delays progress on requests undergoing frequent updates with short LCPs.

4.4.4 LCAS (Last Chunk Arrival Scheduling)

This scheduler combines two strategies: (1) separating complete and partial requests into two tiers, and (2) ordering both tiers by most recent chunk arrival time (`last_chunk_arrival_time`, most recent first). Eviction removes the request with the oldest chunk arrival.

LCAS provides strong performance in append mode by pri-

oritizing requests with recent chunk arrivals. When a chunk is appended at time t, the request is boosted to highest priority. The scheduler processes the expanded input sequence immediately while prior context is warm, maximizing efficiency. Requests transition from partial to complete tier as input finishes, and complete requests are prioritized for output generation. The approach naturally handles temporally clustered chunk arrivals well.

LCAS also performs well in update mode. When an input sequence is updated, the request is boosted to high priority. This enables immediate recomputation of invalidated tokens, making progress on high-cost updates quickly. The priority boost benefits requests with short LCPs.The main weakness in both modes is potential starvation: requests with infrequent chunk arrivals may be delayed indefinitely if other requests have more recent arrivals.

5 Implementation and Usage

We implement Stream2LLM on top of the vLLM v1 engine, extending the core scheduler to support streaming inputs and the two-phase scheduling architecture described in Section 4.1. The scheduler is modified to detect streaming request lifecycle events (initial request, input chunks, and completion) and invoke the KV cache manager to compute the longest common prefix (LCP) between old and new inputs, invalidating only the cache blocks corresponding to changed tokens. We implement the preemption decision framework as a cost comparison between recomputation and swapping strategies using performance models derived from hardware profiling. Multiple scheduling algorithms (Section 4.4) are implemented as priority sorting functions that can be selected at runtime via the `SCHEDULER_TYPE` environment variable.

The KV cache manager is extended with GPU and CPU block pools to support both swapping and recomputation preemption strategies. When a request is preempted via swapping, its blocks are transferred to CPU memory and later swapped back when the request resumes. When preempted via recomputation, blocks are freed and the request recomputes affected tokens upon resumption. The evaluation drivers (crawler and ANNS) implement the streaming request lifecycle, submitting initial requests with the `is_streaming_prompt` flag, appending or updating inputs as chunks arrive (via `is_prompt_update`), and signaling completion with `is_streaming_prompt_finished`. Event tracking records key state transitions (QUEUED, SCHEDULED, KV_ON_GPU, PREEMPTED_SWAP, PREEMPTED_RECOMPUTE, FINISHED) to enable detailed latency analysis and telemetry for streaming workloads.

```
# e is engine object
# append mode
e.new_stream(rid="q1", ids=[query],...)
for chunk in chunks:
    e.append(rid="q1", ids=[chunk],...)
e.append(rid="q1", ..., finished=True)
# update mode
e.new_stream(rid="q2", ids=docs1+[q],...)
e.update(rid="q2", ids=docs2+[q],...)
e.update(rid="q2", ..., finished=True)
```

Listing 1. Sample API usage in append and update modes.

5.1 Public Interface

Stream2LLM extends the vLLM engine interface with streaming-aware request objects and lifecycle management. The `EngineCoreRequest` object supports append and update modes via three key flags: `is_streaming_prompt` (incomplete input), `is_streaming_prompt_finished` (signals completion), and `is_prompt_update` (toggles update vs. append). Clients may append input chunks or replace the input sequence entirely, and the engine automatically computes longest-common-prefix matches for cache invalidation.

The driver provides convenient functions for both modes: `new_stream` (create initial request), `append` (add chunks), and `update` (replace input). Listing 1 shows sample usage in both modes.

Request Lifecycle. Append mode submits an initial request with `is_streaming_prompt=True`, appends additional chunks as they arrive, then signals completion with `is_streaming_prompt_finished=True`. Update mode follows the same pattern but sets `is_prompt_update=True` when replacing the entire input sequence. The engine tracks both modes transparently and manages KV cache invalidation accordingly.

Output and Telemetry. The engine returns `EngineCoreOutput` objects containing newly generated tokens, finish reason, and event timestamps. Events (QUEUED, SCHEDULED, KV_ON_GPU, PREEMPTED_SWAP, PREEMPTED_RECOMPUTE, FINISHED) enable latency analysis and scheduling diagnostics. The scheduler algorithm is selected via `SCHEDULER_TYPE` environment variable (`DEFAULT_VLLM`, `FCFS`, `MCPS`, or `LCAS`).

6 Evaluation

We evaluate Stream2LLM on two distinct streaming workloads: crawler-based context retrieval (append mode) and ANNS-based refinement (update mode). Our experiments demonstrate that scheduling policies designed for streaming inputs significantly improve responsiveness without sacrificing system throughput.

Metric	Mean	P50	P75	P95
ANNS (Update Mode, 500 queries)				
Total Tokens per Query	13K	10K	15K	31K
Retrieval Latency (s)	4.5	3.9	5.3	8.5
Crawler (Append Mode, 4,322 queries)				
Total Tokens per Query	9.1K	5.8K	11.3K	28.9K
Retrieval Latency (s)	9.9	9.3	11.6	16.7

Table 2. Summary of streaming workload characteristics.

6.1 Experiment Setup

Streaming Workloads. Due to the lack of large-scale, publicly available streaming workloads, we collected two realistic retriever traces with full response and timing information. These traces are replayed at varying query-per-second (QPS) rates to emulate moderate load conditions where scheduling decisions are most impactful. Table 2 summarizes their characteristics.

Update Mode. We use the Fineweb-edu corpus (index: 372 GB, vectors: 279 GB) with SQuAD queries (Rajpurkar et al., 2016), both encoded using `e5-base-v2` (Wang et al., 2022). We build a DiskANN index(Jayaram Subramanya et al., 2019) with L2 distance and configure search with beam width $W = 8$ and list size $L = 10000$. This large L increases retrieval accuracy but introduces significant latency, creating the high-latency, I/O-bound scenario STREAM2LLM targets. We implement AquaPipe's recall-aware prefetching (Yu et al., 2025), which progressively refines the top-k candidate list and enables early emission of partial results with sufficient recall.

Append Mode. We use the `crawl4ai` Python library for core web crawling and scraping functionalities, which includes content pruning and link filtering with the BM25 algorithm to extract content only relevant to the original query (UncleCode, 2024). Each crawl explores pages up to depth 2 and streams retrieved content to STREAM2LLM in arrival order. Queries are from OpenAI's SimpleQA dataset(Wei et al., 2024), which contains 4,327 fact-seeking questions requiring web-based evidence. Our crawler workload applies per-document filtering and deduplication inline as each page is retrieved, enabling incremental streaming without a global reranking step.

Chunk Arrival Patterns. The two workloads exhibit fundamentally different chunk arrival patterns, which drive the scheduling challenges described in the main paper.

Figure 6 shows the distribution of inter-chunk arrival times. ANNS arrivals are tightly concentrated around 36.7 ms (median), with the bulk of the distribution spanning 1–1,000 ms. Crawler arrivals are 19× slower (median 700.7 ms) and span approximately three orders of magnitude, from tens of milliseconds to over 30 seconds. This high variability

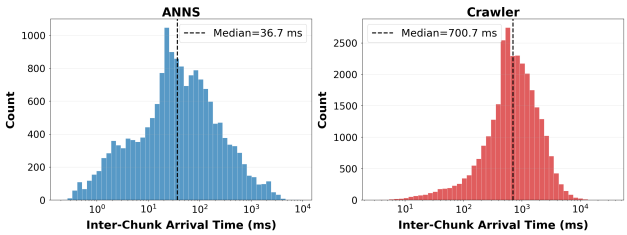

Figure 6. Distribution of inter-chunk arrival times for ANNS and Crawler workloads (log-scale). ANNS chunks arrive with a median of 36.7 ms, while crawler chunks arrive with a median of 700.7 ms, exhibiting significantly higher variability.

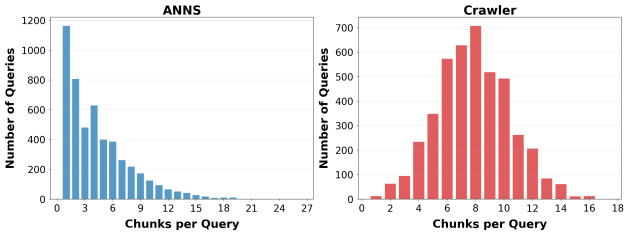

Figure 7. Distribution of chunks per query. ANNS queries are heavily skewed toward 1–3 chunks, while crawler queries are concentrated around 6–10 chunks per query.

in crawler chunk arrivals creates extended idle periods for individual requests, motivating scheduling policies that can dynamically re-prioritize requests as new chunks arrive.

Figure 7 shows the distribution of chunks per query. ANNS queries are heavily right-skewed, with the majority receiving 1–3 chunks (>50% of queries receive ≤3 chunks). Crawler queries follow a broader, roughly unimodal distribution centered around 6–10 chunks. The combination of more chunks per query and higher per-chunk variability makes the crawler workload substantially more demanding for streaming schedulers.

Hardware and Configuration. Experiments run on NVIDIA H200 (141GB) and NVIDIA H100 (80GB) GPUs. We use Llama-3.1-8B-Instruct as the primary model. Tensor parallelism is set to 2, and GPU memory utilization target is 80% (20% buffer to prevent OOM). Token budget per scheduling step varies from 2048 to 8192. STREAM2LLM extends capabilities of vLLM inference system.

Methods of Comparison. We compare STREAM2LLM against two baselines: (1) vLLM-NS: default vLLM scheduler without streaming support, and (2) vLLM-S: default vLLM scheduler with streaming support. We also evaluate STREAM2LLM under its three custom scheduling and preemption policies–FCFS, MCPS, and LCAS–to assess the impact of scheduling strategy on latency and throughput.

Metrics. We report two key performance metrics: (1) *TTFT*

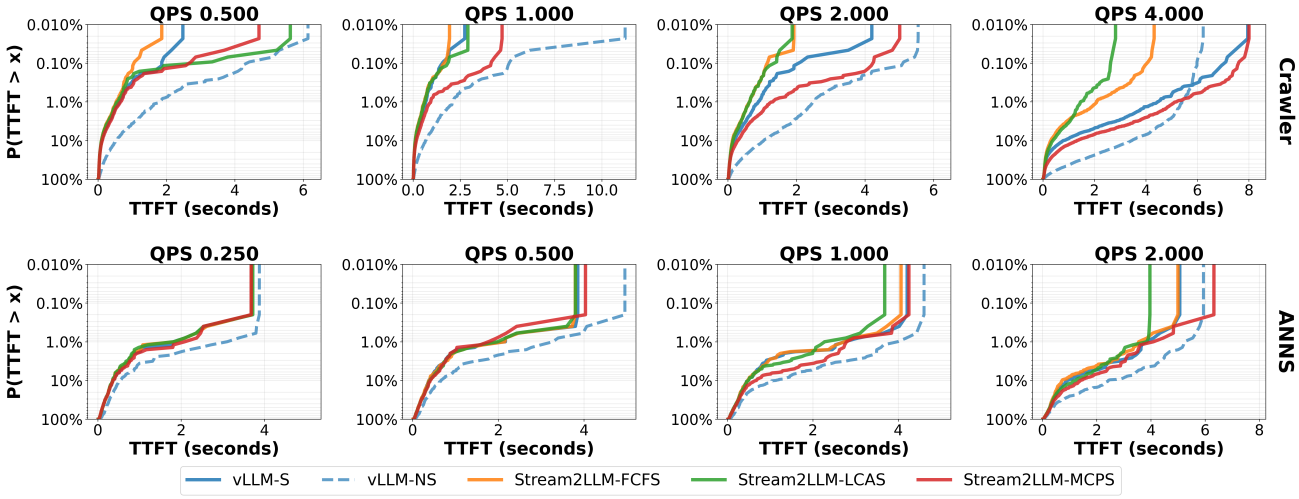

Figure 8. TTFT CCDF across load levels for the crawler and ANNS workloads on H200. Streaming achieves up to 10.8–11.0× faster median latencies than non-streaming on the crawler workload, and up to 2.49–2.63× P95 speedups on the ANNS workload.

(Time-To-First-Token), the time from request arrival to first generated token, which directly captures user-facing responsiveness, and (2) *Trace Completion Time*, the total wall-clock time to complete all requests in a workload, reflecting system throughput and end-to-end performance. All experiments target the prefill instance in a disaggregated serving configuration; decode latency (TPOT) is handled by a separate decode instance and is not evaluated.

6.2 Prefill Latency

Figure 8 shows TTFT performance on H200 across load levels for Crawler (top) and ANNS (bottom) workloads.

Crawler Workload (Append Mode): Streaming consistently improves TTFT across all loads. The performance gap between streaming and non-streaming widens with increasing load: at low loads (QPS 0.5–1.0), streaming achieves 3.9–4.3× faster median latencies over non-streaming, while at QPS 4.0, it delivers 10.8–11.0× faster median latencies by effectively overlapping retrieval and prefill.

Within the streaming approaches, scheduler differentiation becomes apparent only when the page arrival rate approaches the prefill rate. At QPS 2, FCFS's arrival-time ordering and LCAS's prioritization of recent page arrivals both achieve 1.44× and 1.32× P95 speedups over the default streaming baseline respectively, while MCPS's progress-based prioritization degrades to 0.73× as recent pages with fresh content are deprioritized. This effect amplifies at QPS 4.0, where FCFS and LCAS reach 3.16× and 2.64× P95 speedups over the default streaming baseline, whereas MCPS falls to 0.71×. Beyond QPS 4.0, GPU utilization saturates and queuing delay dominates scheduling decisions, causing all approaches to converge to similar performance.

ANNS Workload (Update Mode): Streaming also provides substantial and consistent benefits in update-mode workloads across all load levels. Similar to the crawler workload, at low loads (QPS 0.25–0.5) and high loads (>QPS 2.0), all schedulers perform similarly–prefill rate dominates at low loads while queuing dominates at high loads. At QPS 1.0, streaming schedulers converge tightly with 2.49–2.63× P95 speedups over non-streaming, indicating that prefill rate still dominates update arrival rate, and streaming architecture provides the dominant benefit regardless of scheduler choice. As load increases to QPS 2.0, scheduler differentiation emerges: FCFS achieves 2.26× P95 speedup over non-streaming, LCAS 1.81×, and MCPS 1.53× over non-streaming. MCPS slightly underperforms in this case because accumulated token progress becomes invalid when

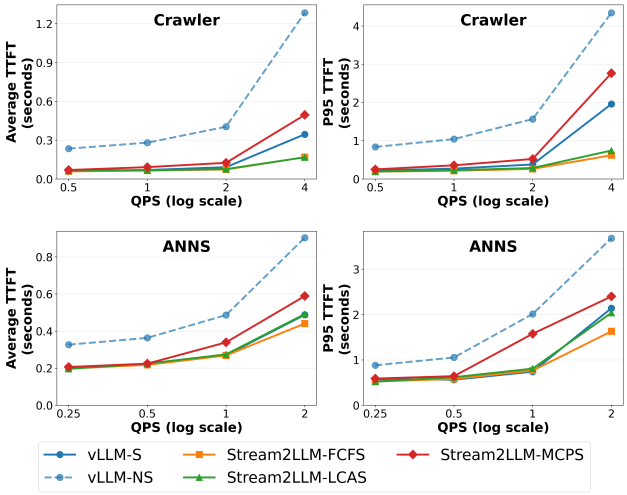

Figure 9. Average and P95 time-to-first-token across schedulers at increasing request rates: crawler workload (top, 0.5–4 QPS) and ANNS (bottom, 0.25–2 QPS).

Table 3. Scheduler and Eviction Strategy Ablation Study. Each cell shows TTFT speedup vs. Default vLLM non-streaming baseline (in secs) at different percentiles (P50/P99). red indicates degradation below the non-streaming baseline (<1×). green indicates best speedup per column. Crawler: 4.0 QPS, 10× delays. ANNS: 2.0 QPS, 30× delays.

	Crawler Workload						ANNS Workload					
	Recompute		Swap		Cost-Based		Recompute		Swap		Cost-Based	
Scheduler	P50	P99	P50	P99	P50	P99	P50	P99	P50	P99	P50	P99
vLLM-NS	0.64s	8.97s	0.64s	8.97s	0.64s	8.97s	0.72s	6.56s	0.72s	6.56s	0.72s	6.56s
vLLM-S	8.59×	0.78×	7.29×	0.66×	8.25×	0.71×	2.53×	0.19×	2.26×	0.18×	2.63×	0.19×
FCFS	8.77×	10.03×	7.78×	6.69×	8.30×	8.62×	2.70×	1.84×	2.37×	1.26×	2.70×	2.04×
LCAS	8.61×	9.23×	7.32×	4.80×	7.82×	9.14×	2.38×	1.79×	1.62×	0.97×	2.44×	1.79×
MCPS	5.92×	0.73×	3.86×	0.48×	4.96×	0.77×	2.20×	0.19×	1.47×	0.22×	2.23×	0.19×

Table 4. Preemption statistics across workloads. Crawler (4.0 QPS, 10× delays) shows higher preemption with balanced swap/recompute. ANNS (2.0 QPS, 30× delays) shows lower frequency with cost-based heavily favoring recompute.

Policy	**Recomp.**	**Swap**	**Cost-Based**
Crawler Workload			
vLLM-S	2,448	779	1,735 (17%/83%)
FCFS	3,552	709	1,575 (21%/79%)
LCAS	12,564	1,049	3,486 (21%/80%)
MCPS	3,316	721	1,741 (19%/81%)
ANNS Workload			
vLLM-S	130	35	125 (2%/98%)
FCFS	342	50	331 (1%/99%)
LCAS	385	48	373 (2%/98%)
MCPS	215	40	237 (0%/100%)

document sets change. All maintain 1.30–1.42× P50 advantage over non-streaming. Streaming incurs significant cache invalidation (Figure 11): more than 10% of requests at all loads invalidate over 10,000 tokens across all schedulers. Despite this cost, streaming achieves up-to $\sim$ 2× faster TTFT than non-streaming, proving the responsiveness gains justify the trade-off. Beyond QPS 2.0, queueing delay dominates scheduling decisions, making cache efficiency gains secondary to responsiveness.

6.3 Latency–Throughput Tradeoff

Figure 9 shows average and P95 TTFT as a function of QPS. As QPS increases, TTFT rises across all methods, but streaming schedulers consistently outperform the non-streaming baseline. In the crawler workload, FCFS and LCAS maintain the lowest TTFT at high QPS, while MCPS degrades. In the ANNS workload, all streaming schedulers track closely until QPS 2.0, where FCFS pulls ahead.

Figure 10 (Appendix) confirms that streaming and scheduling optimizations do not affect aggregate throughput. Trace completion times decrease hyperbolically with increasing QPS, and all scheduler variants (including non-streaming) produce indistinguishable curves on both workloads, demonstrating that the TTFT improvements come at no cost to system-level throughput.

6.4 Performance under Memory Pressure

Our default H200 setup (141GB memory, 80% utilization) has sufficient memory that eliminates preemption. To evaluate scheduler behavior under memory pressure, we modified the workloads to saturate the GPU KV cache pool by increasing chunk delays: a factor of 10× for the crawler workload (append-mode) and 30× for ANNS (update-mode), where the delay multiplier is defined as $\frac{total\ KV\ tokens\ capacity}{avg\ tokens\ per\ query \times avg\ query\ duration \times QPS}$.

Scheduler and Eviction Strategy Ablation: We vary scheduler and eviction policy independently on both workloads. Table 3 shows median (P50) and tail (P99) latency speedups versus the non-streaming baseline. The study shows two key findings: (1) DEFAULT vLLM streaming exhibits catastrophic tail latency degradation under memory pressure—at P99, it performs 0.71× (crawler) and 0.19× (ANNS) vs. non-streaming baseline, demonstrating that streaming without proper scheduling is harmful under contention. (2) Eviction strategy choice creates substantial performance variation: in the crawler workload, FCFS achieves 10.03× speedup with recompute-only but only 6.69× with swap-only at P99, while LCAS shows similar sensitivity (9.23× vs. 4.80×). The cost-based approach balances these extremes, achieving 8.62× (FCFS) and 9.14× (LCAS). For ANNS, the pattern repeats with smaller absolute speedups due to update-mode characteristics: FCFS achieves 1.84× (recompute) vs. 1.26× (swap) vs. 2.04× (cost-based) at P99. These results confirm that hardware-aware eviction is essential for performance under memory pressure, and proper scheduling (FCFS/LCAS) prevents catastrophic tail latency.

Preemption Statistics: Table 4 reports preemption frequencies across configurations. The crawler workload triggers 779–12,564 total preemptions depending on scheduler aggressiveness (LCAS highest at 12.5K for recompute-only). Cost-based eviction allocates 17–21% to SWAP and 79–83%

to RECOMPUTE for crawler, confirming balanced strategy selection. The ANNS workload exhibits lower preemption frequency (35–385 total) with cost-based heavily favoring RECOMPUTE (98–100%), validating the earlier claim that update-mode characteristics make recomputation nearly always cost-optimal due to smaller effective KV caches after context invalidation.

6.5 Overhead Analysis

Cache Efficiency in Update Mode. Figure 11 (Appendix) shows the CCDF of tokens invalidated per request across QPS levels for the ANNS workload. All three streaming schedulers (FCFS, LCAS, MCPS) exhibit nearly overlapping invalidation curves, indicating that cache invalidation behavior is driven by the retrieval workload rather than the scheduling policy. The non-streaming baseline (vLLM-NS) shows zero invalidation by design, as it waits for complete retrieval before beginning inference. Invalidation patterns remain stable across load levels: at all QPS values, more than 10% of requests invalidate over 10,000 tokens, reflecting the iterative document replacement in update mode.

Scheduling Overhead. Scheduler sorting and budget allocation add sub-millisecond overhead per scheduling step. With 50 concurrent requests (representative of peak QPS), sorting latency is 15–16 μs for FCFS/LCAS and 12–13 μs for MCPS. Even at 500 requests, P99 latency remains below 165 μs—negligible compared to prefill computation. The one-time offline profiling for hardware-specific cost models takes approximately five minutes per GPU configuration and is stored as a JSON file, imposing no runtime overhead.

7 Related Work

Retrieval–Inference Co-Design. Recent work has explored overlapping retrieval and inference to reduce end-to-end latency. AquaPipe (Yu et al., 2025) improves time-to-first-token via recall-aware prefetching of intermediate ANNS results, but is limited to single-request settings and does not support scheduling with dynamic priorities. It also relies on interrupting and restarting prefill when retrieval context changes, which is not supported in production systems such as vLLM. PipeRAG (Jiang et al., 2024) uses pipeline parallelism with flexible retrieval intervals for iterative retrieval-generation, but evaluates with fixed batch sizes and does not address variable sequence lengths in deployments. Notably, PipeRAG targets encoder-decoder architectures with cross-attention, which is architecturally incompatible with the causal self-attention models used by Stream2LLM. Both systems leave open questions about scheduling policies for concurrency, adaptive preemption, and cache invalidation mechanisms across requests with different retrieval patterns. Stream2LLM addresses these by introducing a two-phase scheduling architecture supporting both append-mode and update-mode retrieval patterns in concurrent environments.

LLM Inference Serving. vLLM (Kwon et al., 2023) provides PagedAttention for efficient KV cache storage and supports preemption via recomputation or swapping, but its scheduler assumes static input sequences. Orca (Yu et al., 2022), Sarathi (Agrawal et al., 2024), and FastServe (Wu et al., 2023) improve batch efficiency and throughput through iteration-level scheduling, chunked prefills, and preemptive prioritization, respectively. Mooncake (Qin et al., 2024), DistServe (Zhong et al., 2024) and Splitwise (Patel et al., 2024) use disaggregated GPU pools for prefill and decode. These systems assume static input sequences and do not support dynamic updates or cache invalidation when sequences change dynamically.

Scheduling and Cache Optimization. Scheduling policies range from FCFS to SJF variants (Patel et al., 2024) and sophisticated approaches like Sarathi-Serve (Agrawal et al., 2024) that minimize pipeline stalls. However, existing policies assume static input sequences and do not account for dynamic priority shifts from streaming context arrival. For cache optimization, PagedAttention divides KV cache into fixed blocks, while RadixAttention (Zheng et al., 2024) enables fine-grained sharing via radix trees. These optimize memory for static input sequences but lack mechanisms for cache invalidation when sequences change. Stream2LLM addresses this with longest common prefix (LCP)-based cache invalidation that preserves valid blocks while selectively invalidating only affected blocks, minimizing recomputation across dynamic retrieval patterns.

8 Conclusion

Stream2LLM bridges the latency gap in context retrieval systems by enabling streaming input processing with concurrent requests. Its decoupled scheduling architecture, LCP-based cache invalidation, and hardware-aware preemption models extend vLLM to handle dynamically evolving contexts efficiently. Our evaluation shows that streaming delivers order-of-magnitude improvements in time-to-first-token latency without sacrificing throughput. Scheduler choice has limited impact when memory is abundant but becomes critical under pressure, improving append-mode performance and preventing degradation in update-mode workloads. Stream2LLM establishes streaming as a core architectural primitive for low-latency, high-throughput context retrieval in large-scale LLM deployments.

Acknowledgments

This work was supported in part by the National Science Foundation under grant IIS-2335881.

REFERENCES

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024.

Chen, Q., Zhao, B., Wang, H., Li, M., Liu, C., Li, Z., Yang, M., and Wang, J. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems*, 34:5199–5212, 2021.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., and Wang, H. Retrieval-augmented generation for large language models: A survey, 2024. URL https://arxiv.org/abs/2312.10997.

Gim, I., Chen, G., Lee, S.-s., Sarda, N., Khandelwal, A., and Zhong, L. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Jayaram Subramanya, S., Devvrit, F., Kadekodi, R., Krishnaswamy, R., and Simhadri, H. V. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems*, 32, 2019.

Jiang, W., Zhang, S., Han, B., Wang, J., Wang, B., and Kraska, T. Piperag: Fast retrieval-augmented generation via algorithm-system co-design. *arXiv preprint arXiv:2403.05676*, 2024.

Karsin, B., Santana, R., and Li, X. GPU DiskANN and Beyond: Accelerating Microsoft Vector Search with NVIDIA cuVS. NVIDIA GTC session, 2025. Accessed: 2026-03-23.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

Li, M. Introducing AISAQ in Milvus: Billion-Scale Vector Search Just Got 3,200x Cheaper on Memory. Milvus blog post, 2025. Accessed: 2026-03-23.

Malkov, Y. A. and Yashunin, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Qin, R., Li, Z., He, W., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. SQuAD: 100,000+ questions for machine comprehension of text. In Su, J., Duh, K., and Carreras, X. (eds.), *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264.

UncleCode. Crawl4ai: Open-source llm friendly web crawler & scraper. https://github.com/unclecode/crawl4ai, 2024.

Vespa Team. Perplexity: Show what great rag takes. https://blog.vespa.ai/perplexity-show-what-great-rag-takes, 2024. Accessed: 2025-12-01.

Wang, L., Yang, N., Huang, X., Jiao, B., Yang, L., Jiang, D., Majumder, R., and Wei, F. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*, 2022.

Wei, J., Karina, N., Chung, H. W., Jiao, Y. J., Papay, S., Glaese, A., Schulman, J., and Fedus, W. Measuring short-form factuality in large language models. *arXiv preprint arXiv:2411.04368*, 2024.

Wu, B., Zhong, Y., Zhang, Z., Liu, S., Liu, F., Sun, Y., Huang, G., Liu, X., and Jin, X. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

Xu, P., Ping, W., Wu, X., McAfee, L., Zhu, C., Liu, Z., Subramanian, S., Bakhturina, E., Shoeybi, M., and Catanzaro, B. Retrieval meets long context large language models. In *The Twelfth international conference on learning representations*, 2023.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for transformer-based generative models. *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Yu, R., Huang, W., Bai, S., Zhou, J., and Wu, F. Aquapipe: A quality-aware pipeline for knowledge retrieval and large language models. *Proceedings of the ACM on Management of Data*, 3(1):1–26, 2025.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

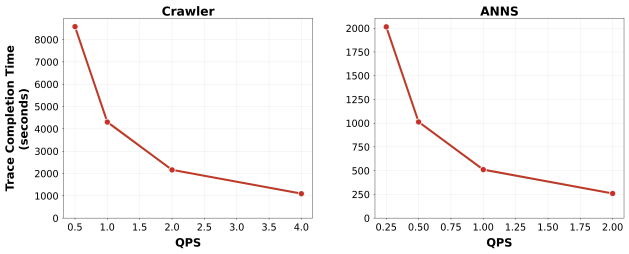

Figure 10. Trace completion time across QPS levels for both workloads. All scheduler variants achieve near-identical completion times, confirming throughput parity.

A Additional Evaluation

A.1 Cache Invalidation in Update Mode

Figure 11 shows the CCDF of tokens invalidated per request across QPS levels for the ANNS workload. All three Stream2LLM schedulers (Stream2LLM-FCFS, Stream2LLM-LCAS, Stream2LLM-MCPS) exhibit nearly overlapping invalidation curves, indicating that cache invalidation behavior is driven by the retrieval workload rather than the scheduling policy. The non-streaming baseline (vLLM-NS) shows zero invalidation by design, as it waits for complete retrieval before beginning inference. Invalidation patterns remain stable across load levels: at all QPS values, more than 10% of requests invalidate over 10,000 tokens, reflecting the update-mode characteristic of iterative document replacement.

A.2 Throughput Parity

Figure 10 confirms that streaming and scheduling optimizations do not affect aggregate throughput. Trace completion times decrease hyperbolically with increasing QPS, and all scheduler variants (including non-streaming) produce indistinguishable curves—appearing as a single overlapping line—on both workloads. This demonstrates that the TTFT improvements reported in the main evaluation come at no cost to system-level throughput.

B Artifact Appendix

B.1 Abstract

This artifact contains the open-source Stream2LLM system: a modified vLLM 0.8.1 engine with streaming input support, along with all scripts, pre-computed run logs, and workload traces needed to reproduce every figure, table, and inline number in the paper. The repository includes two evaluation paths: (1) an *artifact-only* path that regenerates all paper artifacts from pre-computed data in approximately five minutes without GPU hardware, and (2) a *full re-run* path that re-executes the experiments end-to-end on NVIDIA GPUs. For a detailed version of this artifact appendix, see the repository `README.md`.

B.2 Artifact check-list (meta-information)

- **Algorithm:** vLLM-NS, vLLM-S, Stream2LLM-FCFS, Stream2LLM-LCAS, Stream2LLM-MCPS scheduling; cost-based preemption
- **Program:** Python scripts; modified vLLM 0.8.1 engine
- **Data set:** Web-crawler traces; ANNS pipeline traces; performance-model JSONs
- **Run-time environment:** Linux, Python 3.10.9, CUDA, conda, pip
- **Hardware:** Artifact-only: any machine. Full re-run: 2× H200 or 2× H100 GPUs
- **Metrics:** TTFT, trace completion time, preemption counts, scheduler sorting latency
- **Output:** Figures in `figures/`, tables in `tables/`, logs in `data/run_log/`
- **How much disk space required (approximately)?:** ∼5 GB
- **How much time is needed to prepare workflow (approximately)?:** ∼15 minutes
- **How much time is needed to complete experiments (approximately)?:** ∼5 minutes (artifact-only); ∼48 hours (full re-run)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Data licenses (if publicly available)?:** CC-BY-4.0
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.18906769

B.3 Description

B.3.1 How delivered

The artifact is delivered as a GitHub repository on the `mlsys_artifact` branch:

https://github.com/rajveerb/stream2llm

All large data (run logs, workload traces, performance-model JSONs) is stored in a git submodule hosted on HuggingFace:

https://huggingface.co/datasets/rbachkaniwala3/stream2llm-data

B.3.2 Hardware dependencies

No GPU is required for artifact-only evaluation. For full experiment re-runs, the paper's experiments used NVIDIA 2×H200 and 2×H100 GPUs. See `README.md` for details.

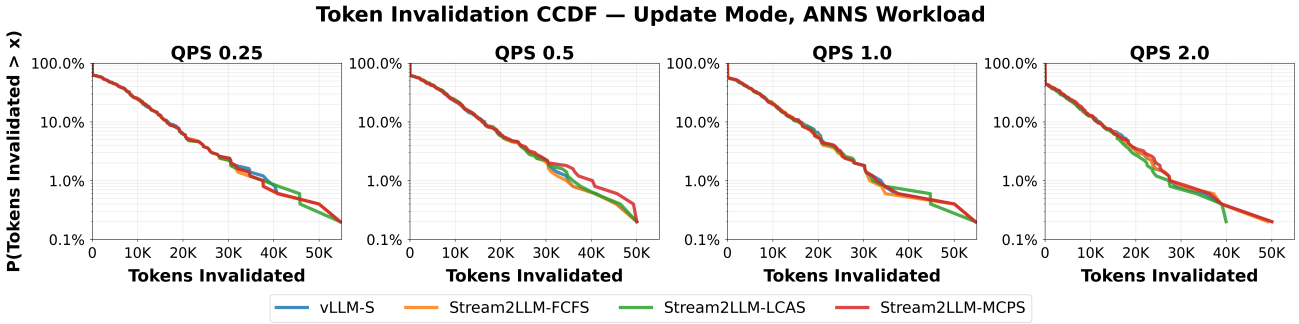

Figure 11. Tokens invalidated per request (CCDF) across QPS levels (0.25–2.0) for the ANNS workload. All STREAM2LLM schedulers show similar cache invalidation behavior. vLLM-NS has zero invalidation as it waits for complete retrieval.

B.3.3 Software dependencies

Dependencies are installed in a `conda` environment via `pip`. See `README.md` for setup instructions and `requirements.txt` for the full package list.

B.3.4 Data sets

All data is hosted as a HuggingFace dataset (`rbachkaniwala3/stream2llm-data`) and fetched automatically via the git submodule. See the "Data Organization" section of `README.md` for details.

B.4 Installation

Refer to the `README.md`.

B.5 Experiment workflow

B.5.1 Artifact-only (no GPU required)

```
bash reproduce_artifacts.sh
```

This generates all figures in `figures/` and analysis tables in `tables/` from pre-computed run logs. Takes approximately five minutes.

B.5.2 Full re-run (GPU required)

The experiment drivers run each scheduler variant (`default_vllm`, `fcfs`, `lcas`, `mcps`) across all arrival-time configurations and write logs to `data/run_log/`. See `README.md` ("Building Stream2LLM Engine" and "Running Experiments") and `experiments/README.md` for the full list of experiment configurations and commands.

B.6 Evaluation and expected result

Figures and tables. Generated figures in `figures/` should visually match the pre-built reference copies in `figures/reference/`. Analysis scripts produce `.txt` files in `tables/`. The mapping from paper artifacts to scripts is:

- Figure 5 → `plot_recomp_vs_swap_clean.py`
- Figures 6 and 7 → `chunk_arrival_characterization.py`
- Figure 8 → `plot_ttft_ccdf_stacked_2x4.py`
- Figure 9 → `crawler/plotter_utils/plot_ttft_qps_comparison.py` (Crawler half) and `anns/plotter_utils/plot_ttft_qps_comparison.py` (ANNS half)
- Figure 10 → `plot_trace_completion_combined.py`
- Figure 11 → `plot_tokens_invalidated_aggregated.py`
- Table 3 → `compute_scheduler_improvements.py`
- Table 4 → `analyze_preemptions.py`
- Scheduler latency → `benchmark_scheduler_latency.py`

B.7 Experiment customization

- **Scheduler selection:** per-scheduler YAML configs in `experiments/{crawler,anns}/configs/`
- **Arrival-time sweeps:** supported via the driver script's `scheduler` and `compare` modes
- **Hardware adaptation:** modify YAML configs and performance-model JSONs in `data/perf_model/`

See `experiments/README.md` for full details.

B.8 Methodology

Submission, reviewing, and badging methodology:

- http://cTuning.org/ae/submission-20190109.html

- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging