Batching in LLM Inference

Techniques, Strategies, and Applications

February 2026

Technical Report

Table of Contents

1. Introduction

Large language model (LLM) inference is one of the most computationally expensive workloads in modern AI systems. As organizations deploy models with billions or even trillions of parameters to serve millions of concurrent users, the efficiency of inference pipelines becomes a critical factor in both cost management and user experience. Batching, the practice of grouping multiple inference requests together for simultaneous processing, has emerged as one of the most important optimization strategies in this domain.

The fundamental insight behind batching is that GPU hardware is designed for massively parallel computation. When processing a single request, much of the GPU's compute capacity sits idle. By combining multiple requests into a single batch, the system can utilize more of the available compute resources per operation, amortizing fixed overheads such as memory transfers and kernel launch costs across many requests. This report examines the various batching techniques used in LLM inference, their trade-offs, and their practical applications across different deployment scenarios.

2. Background: Why Batching Matters

2.1 The Memory-Bound Nature of LLM Inference

LLM inference, particularly the autoregressive decoding phase, is predominantly memory-bandwidth bound rather than compute bound. During token generation, each forward pass reads the full model weights from GPU memory but performs relatively little arithmetic per byte loaded. This results in extremely low arithmetic intensity, meaning the GPU's floating-point units are underutilized. Batching multiple sequences together increases the amount of computation performed per memory access, shifting the workload toward a more compute-bound regime where the GPU hardware can be used more efficiently.

2.2 The Two Phases of LLM Inference

LLM inference consists of two distinct phases with different computational profiles. The **prefill phase** (also called prompt processing) processes all input tokens in parallel and is compute-intensive, resembling a training forward pass. The **decode phase** generates tokens one at a time autoregressively and is memory-bandwidth bound because each step reads the entire model's weights to produce a single token per sequence. Effective batching strategies must account for these fundamentally different computational characteristics.

3. Batching Strategies

3.1 Static (Naive) Batching

The simplest form of batching collects a fixed number of requests and processes them together as a single batch. All sequences in the batch begin and end processing together. While straightforward to implement, static batching suffers from significant inefficiency: when sequences in a batch have different lengths, shorter sequences must be padded to match the longest sequence, wasting compute on padding tokens. Furthermore, the entire batch is blocked until the longest-running sequence completes, leaving GPU resources idle as individual sequences finish generating.

3.2 Continuous Batching (In-Flight Batching)

Continuous batching, introduced by the Orca system and adopted by frameworks such as vLLM and TensorRT-LLM, addresses the limitations of static batching by allowing individual sequences to enter and leave the batch dynamically at each iteration. When one sequence finishes generating, a new request can immediately take its slot without waiting for the rest of the batch to complete. This dramatically improves GPU utilization and throughput because the system no longer wastes cycles on completed sequences. Continuous batching has become the de facto standard in production LLM serving systems.

3.3 Disaggregated Prefill and Decode

Because the prefill and decode phases have very different computational profiles, some systems separate them onto different hardware or processing pools. Prefill, being compute-heavy, benefits from high-FLOPS accelerators, while decode, being memory-bandwidth bound, benefits from high-bandwidth memory configurations. By disaggregating these phases, each can be batched and optimized independently, avoiding interference between compute-heavy prefill operations and latency-sensitive decode operations. Systems like Splitwise and DistServe have demonstrated significant efficiency gains with this approach.

3.4 Speculative Decoding and Batched Verification

Speculative decoding uses a small, fast draft model to propose multiple candidate tokens, which are then verified in a single batched forward pass through the full target model. This effectively converts multiple sequential decode steps into a single parallel verification step, reducing the number of expensive forward passes required. The verification can reject incorrect tokens and fall back to the target model's distribution, preserving output quality while significantly improving throughput, particularly for sequences where the draft model has high acceptance rates.

4. Key Enabling Techniques

4.1 PagedAttention and Memory Management

One of the primary challenges of batching is managing the key-value (KV) cache, which stores intermediate attention states for each sequence. Traditional approaches pre-allocate contiguous memory for each sequence's maximum possible length, leading to severe memory fragmentation and waste. PagedAttention, introduced by vLLM, borrows concepts from operating system virtual memory to store KV cache in non-contiguous memory pages. This allows the system to allocate memory on demand and share pages across sequences (for example, when multiple requests share a common system prompt), enabling much larger effective batch sizes.

4.2 Dynamic Scheduling and Priority Queues

Sophisticated serving systems implement request schedulers that dynamically decide which requests to include in each batch based on factors like arrival time, priority level, estimated generation length, and current system load. These schedulers balance throughput (processing as many requests as possible) against latency (keeping response times low for individual requests). Advanced techniques include preemption, where low-priority or long-running requests can be temporarily evicted from the batch to make room for higher-priority requests.

4.3 Quantization and Reduced-Precision Batching

Quantization reduces model weight and activation precision from FP16 or BF16 to formats like INT8, INT4, or FP8, shrinking memory requirements and increasing the effective memory bandwidth. Smaller memory footprints mean more sequences can fit in the KV cache simultaneously, directly increasing achievable batch sizes. This is particularly impactful for the decode phase, where the bottleneck is memory bandwidth. Modern inference engines combine quantization with batching to maximize throughput on a given hardware budget.

5. Comparison of Batching Strategies

The following table summarizes the key characteristics and trade-offs of the major batching strategies discussed in this report.

Strategy	Throughput	Latency	Complexity	Memory Eff.
Static Batching	Low–Medium	High (padding waste)	Low	Poor
Continuous Batching	High	Low–Medium	Medium	Good

Disaggregated Prefill/Decode	Very High	Low	High	Very Good
Speculative Decoding	High	Very Low	High	Good

6. Practical Applications

6.1 API Serving and Cloud Inference

Commercial LLM API providers such as OpenAI, Anthropic, and Google serve millions of concurrent users and rely heavily on continuous batching to maximize throughput per GPU. In this context, the ability to dynamically adjust batch composition in real time is essential for maintaining consistent latency SLAs while keeping infrastructure costs manageable. Providers typically combine continuous batching with PagedAttention, KV cache compression, and multi-level scheduling to handle heterogeneous workloads that mix short chat messages with long document-processing tasks.

6.2 Offline and Batch Processing

Many enterprise use cases involve processing large volumes of text offline, such as document summarization, data extraction, classification, or embedding generation. In these scenarios, latency is less critical than throughput and cost efficiency. Large static or semi-static batches can be used to maximize GPU utilization, and requests can be sorted by length to minimize padding overhead. Anthropic's Message Batches API and similar offerings from other providers give significant cost discounts for batch workloads, reflecting the efficiency gains possible when latency constraints are relaxed.

6.3 Edge and On-Device Inference

On edge devices with limited memory and compute, batching plays a different role. While concurrent user requests are rare on a single device, internal batching techniques such as speculative decoding and chunked prefill are used to improve single-user latency. These techniques batch multiple speculative tokens or prompt chunks together to better utilize the available hardware, achieving faster time-to-first-token and higher tokens-per-second even for individual users.

6.4 Training and Evaluation Pipelines

Batched inference is also critical in LLM training pipelines, particularly for reinforcement learning from human feedback (RLHF) and other preference optimization methods that require generating completions from a policy model as part of the training loop. Efficient batched generation allows training systems to generate large numbers of completions quickly, keeping the training process moving. Similarly, model evaluation suites rely on

batched inference to run thousands of benchmark examples across multiple models within practical time budgets.

7. Challenges and Trade-Offs

While batching delivers substantial benefits, it introduces several engineering and operational challenges. Larger batches require more GPU memory for storing KV caches, creating tension between batch size and the maximum sequence length that can be supported. Systems must implement sophisticated memory management (such as PagedAttention or KV cache eviction policies) to navigate this trade-off. Additionally, batching can increase latency for individual requests when the system prioritizes throughput, since a request may need to wait for a batch to be assembled or may share compute resources with many other sequences.

Another challenge is fairness and quality of service. In a multi-tenant serving environment, different users and applications have different latency and throughput requirements. A batch-oriented system must implement priority scheduling and admission control to ensure that high-priority or latency-sensitive requests are not starved by large batch workloads. Finally, the interaction between batching and other optimizations like tensor parallelism, pipeline parallelism, and expert parallelism (in mixture-of-experts models) adds significant system complexity.

8. Future Directions

The evolution of batching techniques continues to be driven by both hardware trends and model architecture innovations. The growing adoption of mixture-of-experts (MoE) architectures introduces new batching challenges and opportunities, since different tokens in a batch may be routed to different expert subnetworks. Efficient batching for MoE models requires load-balancing across experts and managing the sparse compute patterns that result.

Hardware advances such as higher-bandwidth memory (HBM3E and beyond), increased on-chip SRAM, and dedicated inference accelerators will shift the compute-versus-memory-bandwidth balance, potentially enabling even larger batch sizes. At the software level, ongoing research into prefix caching, radix-tree-based KV cache sharing, and adaptive scheduling algorithms promises to further improve batching efficiency. The trend toward longer context windows (from 4K to 128K tokens and beyond) also creates new pressure on KV cache memory and makes memory-efficient batching techniques increasingly essential.

9. Conclusion

Batching is a foundational optimization in LLM inference that transforms underutilized, memory-bandwidth-bound GPU workloads into efficient, high-throughput serving systems. From naive static batching to sophisticated continuous batching with

PagedAttention and disaggregated serving, the field has developed a rich set of techniques that collectively enable the practical deployment of large language models at scale. As models grow larger, context windows extend further, and user demand continues to increase, batching strategies will remain at the core of efficient LLM inference infrastructure, evolving alongside advances in hardware, model architectures, and systems research.