

Batching Experiment

Executive Summary

For the batching part of our ORIE simulation project, we built a discrete-event simulator of a single-GPU LLM serving system and compared fixed-size batching against dynamic batching. Fixed batching dispatches when queued token work reaches a target b (or a timeout), while dynamic batching adapts batch size to queue depth and uses a latency guard to avoid waiting too long. We evaluated throughput and user-facing latency metrics: TTFT, completion latency, and TBT (streaming smoothness).

In our fixed- b sweep ($b = 32, 64, 96, 128, 160, 192, 256$), throughput stayed in a relatively tight range and was similar to the dynamic baseline, so the main effect of batching here is latency/variability rather than raw capacity. Fixed batching with moderate/large b produced very tight and predictable completion latency and the most stable token streaming (TBT concentrated in a narrow band). Dynamic batching often reduced typical completion latency, but it introduced more token-level variability (more dispersed TBT), meaning streaming can feel less consistent. The takeaway is that fixed batching behaves more “steady” once b is large enough, while dynamic batching improves responsiveness but can add jitter depending on tuning.

Modeling Approach, Assumptions, Parameters

We model a single GPU with a shared queue and two phases per request: prefill and decode. Requests arrive as a Poisson process with rate λ . Each request has prompt length L and output length budget B . Prefill is one L -token job; decode is B sequential 1-token jobs (the next token is only released after the previous completes). This is the minimum structure needed to measure TTFT, completion latency, and TBT.

The GPU executes batches. Each batch has a token load equal to the sum of tokens in the jobs it contains, and service time includes a setup component plus additional work beyond a threshold b_0 . Inputs are $\lambda, (L, B)$, service parameters (c, a, b_0) , and batching policy parameters (b and timeout for fixed; queue-depth thresholds and a latency target for dynamic). We use synthetic workloads because we do not have real production arrival traces.

Model Detail

The simulator is discrete-event with arrivals, batch dispatches, and batch completions. When a batch completes, all jobs finish at the same time; prefill completion releases the first decode token, and each decode token completion releases the next token for that request. This enforces the decode dependency and lets us compute TBT from realized token completion timestamps.

Policy logic is simple: fixed batching dispatches when queued tokens $\geq b$ or the oldest job exceeds the timeout; dynamic batching chooses a batch target based on queue depth and dispatches early if the oldest queued job nears a waiting-time threshold. We checked basic invariants: one prefill + B decode tokens per request, no parallel decode tokens for the same request, and conservation of queued token load under dispatch.

Model Analysis

We ran a fixed- b sweep over $b \in \{32, 64, 96, 128, 160, 192, 256\}$ and compared each case to one dynamic configuration. We report throughput, P95 TTFT, P95 completion latency, and TBT distributions, using density overlays across b and per- b histograms versus dynamic. In our runs, throughput differed only slightly across b and was close to dynamic, indicating we were not in a strongly overloaded regime.

Latency behavior showed clearer trade-offs. For fixed batching, moderate/large b produced tightly clustered completion latency distributions, while small b behaved differently, reflecting reduced batching wait but less stable behavior. Token-level results were more decisive: fixed policies (especially $b \geq 64$) generated the most consistent streaming cadence, while dynamic batching had a more dispersed TBT distribution, i.e., more jitter in inter-token gaps. Practically, fixed batching in the stable region gives predictable performance, while dynamic batching can reduce typical completion times but may worsen streaming smoothness unless tuned.

Steady-State Considerations

To reduce startup bias, we discard an initial warm-up set of completed requests before computing metrics. For final reporting, the simulation should be run long enough to obtain at least $\sim 10,000$ completed queries and repeated across multiple random seeds, with confidence intervals for P95 metrics and throughput.

Conclusions

Under our configuration, fixed batching with moderate/large b provides the most consistent completion latency and the smoothest token streaming, while dynamic batching is competitive on throughput and can lower typical completion times but introduces more token-level variability. If dynamic batching is preferred for robustness under changing load, it should be tuned to avoid overly frequent small dispatches (which tends to increase TBT jitter).

Workload Sensitivity Experiment

Executive Summary

For the workload sensitivity component of our ORIE simulation project, we studied how different workload characteristics influence tail latency in a single-GPU LLM serving system. Using a discrete-event simulator, we compared two scheduling policies: process-to-completion FCFS (decode-prioritized) and prefill-prioritized batching with a fixed token budget. Our evaluation focused on user-facing tail metrics—P95 Time-to-First-Token (TTFT) and P95 Time-Between-Tokens (TBT)—which are particularly sensitive to congestion and scheduling decisions.

We varied three workload dimensions: arrival rate λ , prompt length L , and output budget B . The results show that arrival rate is the dominant driver of tail latency phase transitions, with FCFS exhibiting a sharp collapse once system utilization approaches saturation. Prefill batching significantly smooths TTFT under high load, delaying this phase transition. Prompt length primarily affects TTFT through prefill blocking, while output length primarily affects TBT through decode congestion. Overall, batching improves robustness of TTFT under load but shifts bottlenecks to the decode phase, highlighting a fundamental trade-off between responsiveness and streaming smoothness.

Modeling Approach, Assumptions, Parameters

We model a single GPU serving LLM inference requests, each consisting of two phases: prefill and decode. Requests arrive according to a Poisson process with rate λ . Each request has a fixed prompt length L and an output budget B . Prefill processes the entire prompt as a single batchable unit, while decode generates B tokens sequentially, enforcing a strict dependency: the next token cannot be generated until the previous one completes.

GPU service time includes a setup cost c and a per-token marginal cost governed by an exponential distribution with mean a . To capture batching effects, batch service time depends on total token load relative to a threshold b_0 , beyond which additional work is incurred. We assume no preemption and a single shared queue. All workloads are synthetic, as real production traces are unavailable, allowing controlled sensitivity analysis across workload dimensions.

Model Detail

The simulator is discrete-event, tracking request arrivals, GPU service start times, and completion events. Under FCFS scheduling, each request executes prefill and decode to completion before the next request is served. This maximizes locality but introduces head-of-line blocking when long requests arrive.

Under prefill-prioritized batching, incoming requests are grouped into batches up to a maximum token budget K during the prefill phase. All jobs in a batch complete prefill simultaneously, releasing their first token at the same time. Decode then proceeds sequentially per request in

arrival order. This structure allows us to measure TTFT precisely as the time from arrival to prefill completion, and TBT as the realized inter-token gaps during decode.

We verified basic invariants: each request executes exactly one prefill and B decode tokens; decode tokens for a request never overlap; and GPU service time is conserved across batching decisions.

Model Analysis

The arrival-rate sweep reveals a clear phase transition under FCFS scheduling. P95 TTFT remains low at moderate arrival rates but increases sharply once λ exceeds approximately 20 qps, indicating that the system has crossed its effective capacity. This sudden tail explosion is characteristic of non-preemptive FCFS queues near saturation, where long jobs block all subsequent arrivals. In contrast, prefill batching maintains bounded P95 TTFT across the entire arrival-rate range, demonstrating significantly improved robustness under heavy load.

Prompt length primarily affects TTFT. As L increases, both schedulers experience higher tail TTFT due to longer prefill times. Prefill batching performs worse than FCFS for very small prompts, where batching delay dominates, but becomes competitive at moderate prompt sizes. For very large prompts, batching can again increase TTFT by inflating batch service time, causing all jobs in the batch to wait together.

Output length has minimal impact on TTFT but strongly influences TBT. Under FCFS, P95 TBT remains low and grows slowly with B , since each request decodes to completion without interruption. Under prefill batching, P95 TBT is substantially higher and increases steadily with B , reflecting decode-phase congestion and interference from other jobs' prefill work. This shows that batching shifts the system bottleneck from prefill to decode.

Steady-State Considerations

All metrics are computed after the system reaches steady operation, implicitly discarding initial transient behavior. To improve statistical reliability, longer simulation horizons and multiple random seeds should be used, with confidence intervals reported for tail metrics. In particular, tail latency estimates near the phase transition are sensitive to variance and require sufficient sample sizes to stabilize.

Conclusions

Our results demonstrate that workload characteristics influence different latency metrics through distinct mechanisms. Arrival rate is the primary driver of phase transitions and tail collapse, while prompt length dominates TTFT through prefill blocking, and output length dominates TBT through decode congestion. Prefill batching substantially improves TTFT robustness under high load but increases token-level latency variability, revealing a fundamental trade-off between responsiveness and streaming smoothness. These findings suggest that adaptive or hybrid scheduling policies may be needed to balance TTFT and TBT across diverse workload regimes.

Chunked Prefills Experiment

Executive Summary

In this work, we model an LLM-serving system using a stochastic, event-driven queueing simulator and study the performance impact of chunked-prefill scheduling, an extension of prefill-prioritized batching. The system consists of one or more identical GPU workers, an input queue of queries, and a scheduler that batches and dispatches prefill and decode work. Each query undergoes a heavy prefill phase to construct the KV cache, followed by a decode phase that generates output tokens sequentially.

Our key contribution is evaluating how prefill chunk size affects latency, throughput, and decode stalls. By sweeping chunk sizes from 256 down to 8 tokens, we identify a clear “knee” in system behavior. Moderate chunk sizes (32–128 tokens) preserve high throughput and stable TTFT, while overly aggressive chunking significantly degrades TTFT, completion latency, and throughput. The results demonstrate that chunked-prefill improves fairness and head-of-line blocking only within a limited regime; excessively small chunks introduce repeated overheads and increased decode stalls that dominate performance.

Modeling Approach, Assumptions, Parameters

We model an LLM-serving system as a discrete-event simulator with a shared input queue and a scheduler controlling GPU execution. Query arrivals follow a Poisson process with rate λ . Each query i is characterized by a prompt length L_i and an output budget B_i , drawn either from simple distributions or fixed values for controlled experiments.

Each query consists of two distinct stages. The prefill stage processes all prompt tokens and enables generation of the first output token, while the decode stage generates output tokens one at a time, with a strict dependency between successive tokens. These stages have different performance characteristics: prefill is compute-heavy and benefits from batching in token load, whereas decode consists of many short iterations where batching across queries can improve throughput but may increase streaming delay.

GPU execution is modeled at the batch level. A batch’s service time depends on its total token load b through a piecewise-linear model $S(b) = c + a \max(0, b - b_0)$, capturing a fixed per-batch setup cost and a marginal per-token cost beyond a threshold. Parameter ranges are chosen based on prior systems literature and adjusted to operate near saturation, where scheduling trade-offs are most visible.

Model Detail

The simulator explicitly tracks arrivals, batch dispatches, and batch completions. Prefill and decode are scheduled at the granularity of GPU iterations. In baseline prefill-prioritized batching, a request must complete all L_i prefill tokens before becoming eligible for decoding.

Chunked-prefill modifies this behavior by splitting each request’s prefill into smaller chunks. In each prefill iteration, a request can contribute at most `chunk_size` tokens, and the total token load

of the batch is capped by a global token budget K (set to 128). Prefill iterations may therefore include chunks from multiple requests. Once a request’s remaining prefill tokens reach zero, it becomes decode-ready. Decode iterations are also batch-based, with each active request contributing at most one decode token per iteration.

This design allows the scheduler to interleave prefill work across requests and alternate between prefill and decode iterations, reducing head-of-line blocking from long prompts. The simulator tracks user-facing SLOs—TTFT, completion latency, and TBT—as well as throughput and internal metrics such as decode stall time.

Model Analysis

We conducted a chunk-size sweep over $\{256, 128, 64, 32, 16, 8\}$ and evaluated TTFT, completion time, TBT, throughput, and decode stall time. The results reveal a clear phase transition driven by chunk size.

For moderate chunk sizes (32–256), system performance is stable. Mean TTFT remains approximately 0.225 seconds and P95 TTFT stays near 0.40 seconds. Throughput remains high at roughly 15.0–15.5 jobs per second, indicating near-saturated but efficient GPU utilization. In this regime, chunking successfully avoids head-of-line blocking without materially increasing overhead or decode interference.

In contrast, aggressive chunking (16 and 8 tokens) leads to sharp performance degradation. Mean TTFT increases to 0.50 seconds at chunk size 16 and 0.63 seconds at chunk size 8, with corresponding increases in P95 TTFT. Completion latency rises substantially, and throughput drops to 14.7–14.8 jobs per second, signaling a real loss in system efficiency rather than a simple fairness trade-off.

Decode stall analysis explains this behavior. With small chunks, long prompts require many more prefill iterations to complete, repeatedly incurring setup-like overhead and keeping prefill backlog persistently nontrivial. Under the prefill-prioritizing scheduler, this causes more frequent situations where decode work is available but delayed by ongoing prefill iterations. Empirically, total decode stall time roughly doubles at chunk size 8 compared to the moderate-chunk regime.

Conclusions

Chunked-prefill is effective only within a moderate operating range. Chunk sizes around 32–128 tokens successfully balance fairness and efficiency, reducing head-of-line blocking while preserving high throughput and stable TTFT. However, overly small chunks introduce excessive iteration overhead and significantly increase decode stall time, degrading both latency and throughput.

These results highlight an important design insight for LLM-serving systems: mechanisms intended to improve responsiveness can backfire if applied too aggressively. Effective schedulers must balance granularity against overhead, and chunked-prefill should be carefully tuned rather than minimized indiscriminately.