Batching Experiment

Executive Summary

For the batching component of our project, we developed a discrete-event simulator of a single-GPU LLM serving system to evaluate fixed-size and dynamic batching strategies. In fixed batching, the system dispatches requests once the queued token work reaches a target batch size *b* (or a timeout), while dynamic batching adapts batch size in real time to queue depth and applies a latency guard to prevent excessive waiting. We assessed both methods on throughput and key user-facing latency metrics: Time to First Token (TTFT), completion latency, and Time Between Tokens (TBT), which together capture responsiveness and streaming smoothness.

Across our sweep of fixed batch sizes (b = 32-256), throughput remained within a narrow range and was comparable to the dynamic baseline. This suggests batching has a greater influence on latency behavior than on raw capacity. Fixed batching with moderate or large *b* yielded tight, predictable completion latency and highly stable streaming (TBT concentrated in a narrow band). Dynamic batching often reduced average completion latency under light load but introduced greater token-level variability, leading to less consistent streaming. Overall, fixed batching offers steadier, more predictable performance once *b* is sufficiently large, while dynamic batching improves perceived responsiveness but can add jitter if not carefully tuned.

Modeling Approach, Assumptions, Parameters

We model a **s**ingle-GPU LLM serving system with one shared request queue, capturing the two key inference phases that shape user experience: prefill, which processes the entire prompt, and decode, which generates tokens sequentially- each token beginning only after the previous one completes. This minimal structure captures all latency-driving behaviors needed to measure Time to First Token (TTFT), completion latency (Appendix A), and Time Between Tokens (TBT) (Appendix B).

Requests arrive as a Poisson process with rate λ, and each request is characterized by a prompt length L and an output-length budget B. The GPU executes work in batches, where each batch's service time includes a fixed setup cost plus additional load-dependent compute beyond a threshold b_0. Batching therefore trades off queueing delay against GPU efficiency. Simulation inputs include λ, (L,B), service parameters (c,a,b_0), and policy parameters (for fixed batching: b and timeout; for dynamic batching: queue-depth thresholds and latency target). Because real production traces were unavailable, we generated synthetic workloads under consistent load and service assumptions. Key aggregate outcomes are summarized in appendix (E: Throughput, F: P95 completion, and G: P95 TTFT)

Model Detail

The simulator operates as a discrete-event system with arrivals, batch dispatches, and batch completions. When a batch finishes, all jobs in that batch complete simultaneously: a completed prefill releases its first decode token, and each decode token completion triggers the next token for that request. This mechanism enforces decode dependencies and lets us compute TBT directly from observed token timestamps.

Two scheduler modules implement the benchmark policies: fixed batching dispatches once queued token work ≥ b or the oldest job exceeds the timeout; dynamic batching adjusts its target batch size with queue depth and dispatches early when the oldest job approaches its waiting-time threshold. We verified model correctness by enforcing invariants (one prefill + B decode tokens per request, no parallel decodes, and conservation of queued token work) and cross-checking expected performance patterns from analytical reasoning against our validation runs.

Model Analysis (See Appendix A-G)

We ran a fixed-b sweep over b ∈ {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. Recommendation: if streaming consistency and predictable tails matter most, use fixed batching in the stable region; if responsiveness under light load matters most, dynamic can help but should be tuned to avoid frequent small dispatches that increase TBT jitter.

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 ~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 delivers the most predictable completion latency and the smoothest token streaming. Dynamic batching is competitive on throughput and can lower typical completion times, but it introduces more token-level variability unless carefully tuned.

Chunked Prefills Experiment

Executive Summary

In this part, 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 (See Appendix H to J)

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.

Workload Sensitivity Experiment

Executive Summary

For the workload sensitivity component of our 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 (See Appendix K to M)

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.

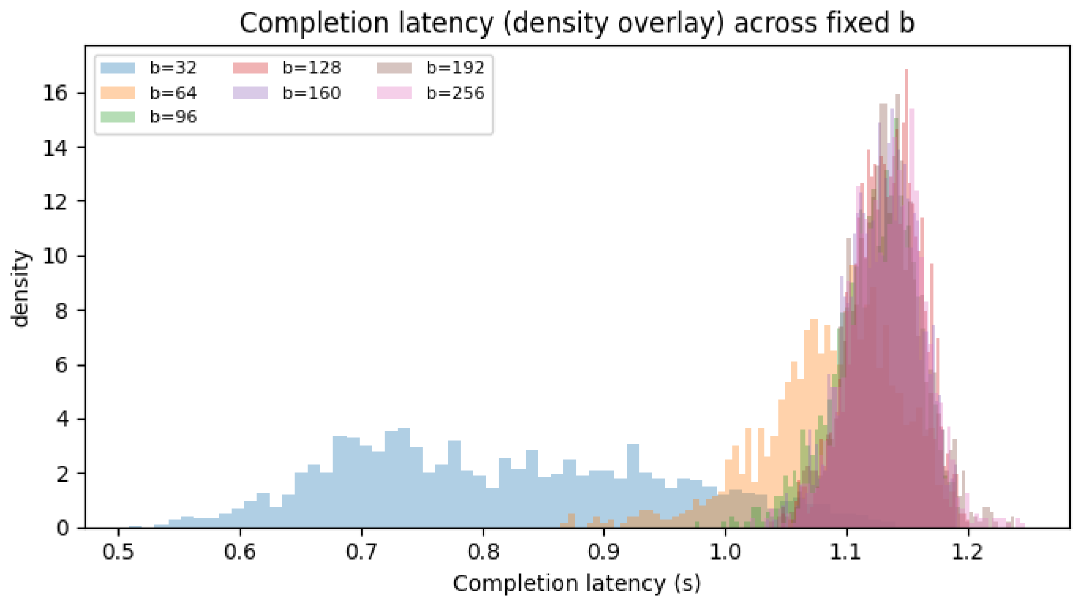
Conclusions

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.

Appendix

Appendix A.

Completion latency (arrival to last token) density overlay across fixed batch token targets b in {32, 64, 96, 128, 160, 192, 256}.

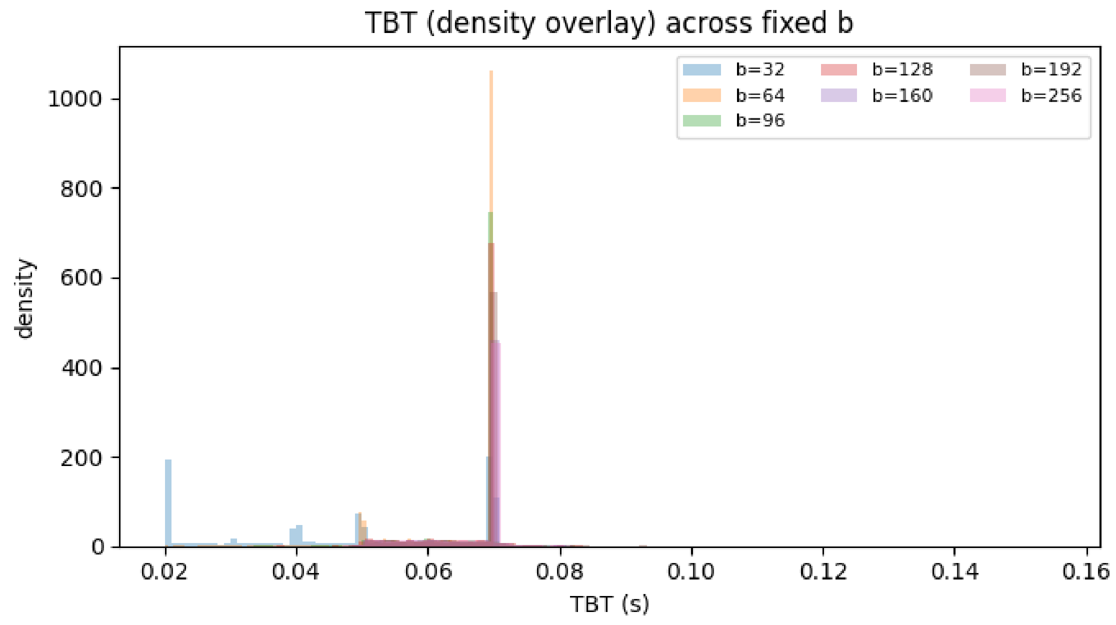


This figure overlays the empirical density of per-request completion latency for each fixed batching policy, where completion latency is measured from the request's arrival time to the completion of its final decode token. Each curve corresponds to a fixed token target b; the scheduler dispatches a batch once queued token work reaches b (or a timeout occurs). Because larger b typically increases batch formation waiting time but amortizes per-batch overhead, the distribution shifts provide a direct view of how batching aggressiveness impacts end-to-end latency under the same workload and service model.

In this run, the curves separate into two visible regimes: the smallest batch target b=32 is concentrated at lower completion latencies (roughly 0.6-0.9 seconds), while b at or above 96 clusters tightly around approximately 1.1-1.2 seconds with heavy overlap among b=128, 160, 192, and 256. The tight grouping for larger b suggests a stable operating region where increasing b further does not materially change completion latency, while the faster b=32 outcome indicates the system is likely not congested in this configuration (small batches reduce waiting and do not yet trigger backlog growth). Tail risk is better captured by P95/P99 metrics reported elsewhere, since density overlays can visually de-emphasize rare extreme delays.

Appendix B.

Time Between Tokens (TBT) density overlay across fixed batch token targets b in {32, 64, 96, 128, 160, 192, 256}.

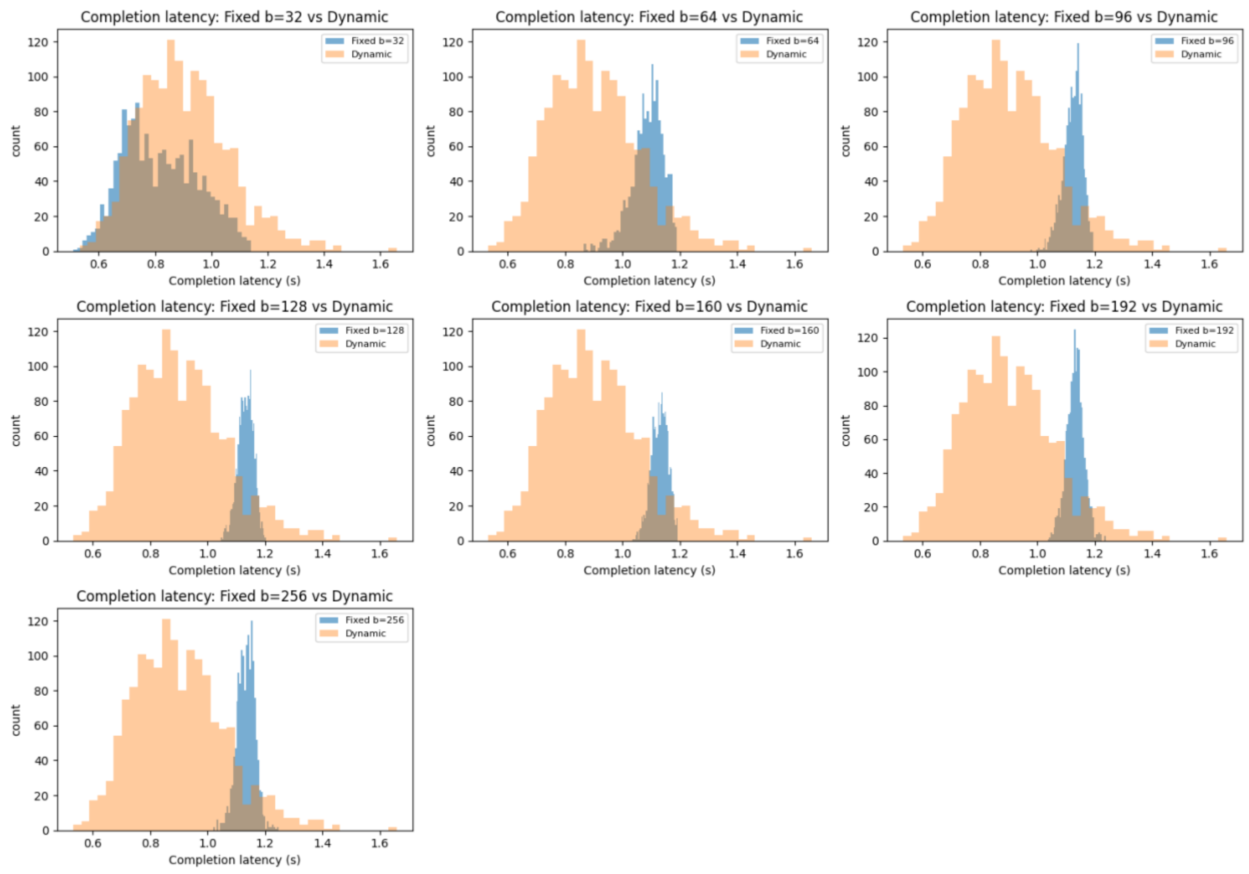


This figure overlays the empirical density of token-level Time Between Tokens (TBT) under different fixed batching targets. TBT measures the gap between consecutive token completions for the same request during the decode phase, so it is a direct proxy for "streaming smoothness" from a user perspective. Because decode tokens are produced sequentially, batching and queueing effects show up as variability in these inter-token gaps even when the underlying per-token service time distribution is unchanged.

In this run, most settings with b at or above 64 concentrate tightly around a narrow TBT band near roughly 0.06-0.07 seconds, indicating relatively steady token cadence once the system is in a stable regime. The b=32 curve is visibly more dispersed, with additional mass at smaller and larger gaps, which suggests higher jitter in token streaming when batches are very small, and the system cycles through more frequent dispatches and overhead. Overall, the plot implies that moderate-to-large fixed batch targets yield smoother and more predictable token streaming than the smallest batch target in this parameterization.

Appendix C.

Completion Latency Comparison: Fixed b vs Dynamic Batching

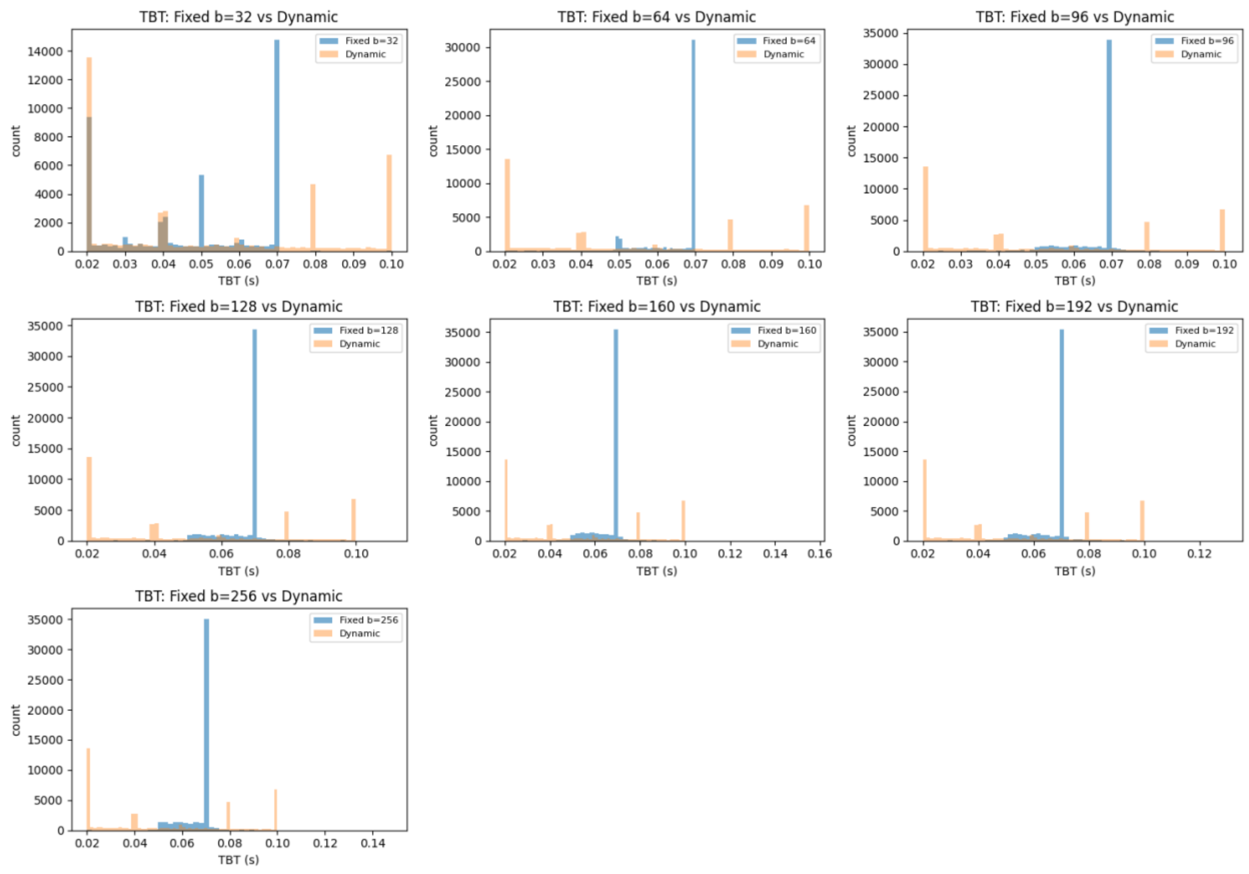


This set of panels compares the completion-latency distribution under each fixed batching target (blue) to the same workload under the dynamic batching policy (orange). The main pattern is that dynamic batching tends to place more probability mass at lower completion latencies (roughly 0.7-1.0 seconds), while the fixed policies with b at or above 64 concentrate later, around about 1.1-1.2 seconds, consistent with added waiting to accumulate larger batches before dispatch. At b=32, fixed batching overlaps more with dynamic, which aligns with the idea that small b reduces batching delay when the system is not congested.

Across $b \geq 64$, the fixed distributions are tight and shifted to the right relative to dynamic, indicating that in this parameterization dynamic's latency guard and smaller on-the-fly batches reduce end-to-end completion time for many requests. At the same time, the dynamic histograms appear broader with a longer right tail than the fixed ones in several panels, suggesting more variability: some requests benefit from early dispatch, while others still wait behind fluctuating batch formation decisions. Overall, these plots show a trade-off where dynamic improves typical completion latency but can introduce more dispersion, whereas fixed large-b batching yields more consistent completion times at the cost of a higher typical latency level.

Appendix D

Time Between Tokens Comparison: Fixed b vs Dynamic Batching

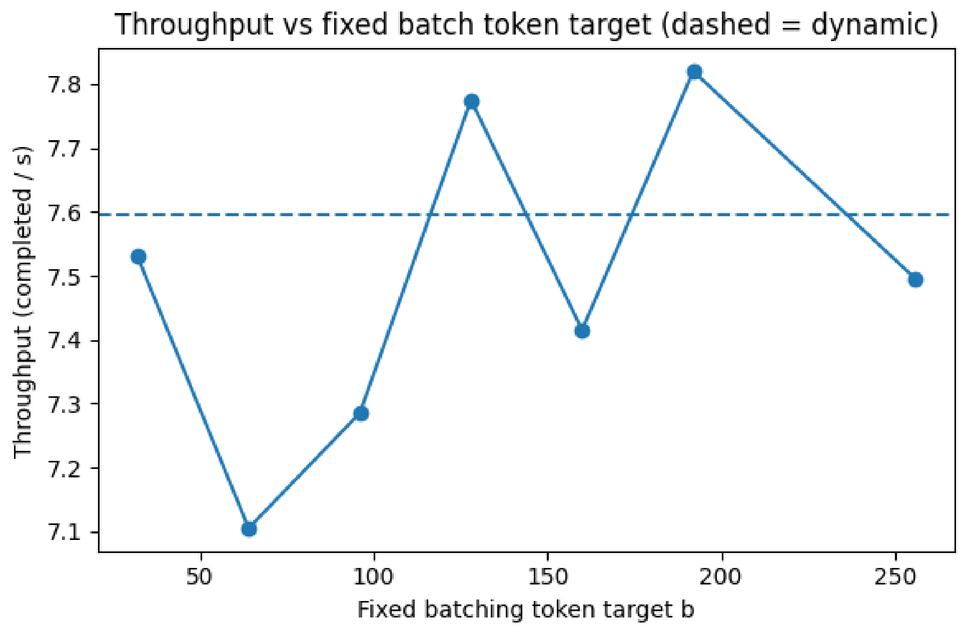


These panels compare the distribution of token-to-token gaps during decode under each fixed batching target (blue) versus the dynamic policy (orange). A consistent pattern across b values is that the fixed policies produce a very sharp concentration of TBT around a narrow band near roughly 0.06-0.07 seconds, meaning that once a request is streaming, tokens tend to arrive at a highly regular cadence under fixed batching in this parameterization. In contrast, the dynamic policy spreads probability mass more widely and shows more pronounced spikes at other gap values, indicating higher variability in the inter-token timing.

The interpretation is that dynamic batching's on-the-fly batch formation and latency-guard behavior introduces additional jitter in the token stream: some tokens are served quickly when the policy dispatches small or early batches, while other tokens experience longer gaps when they get caught behind variable batch composition or additional dispatch overhead. Meanwhile, fixed batching- especially for b at or above 64- behaves more like a steady "rhythm" in decode, with less dispersion in TBT, even if it may not always minimize end-to-end completion latency.

Appendix E.

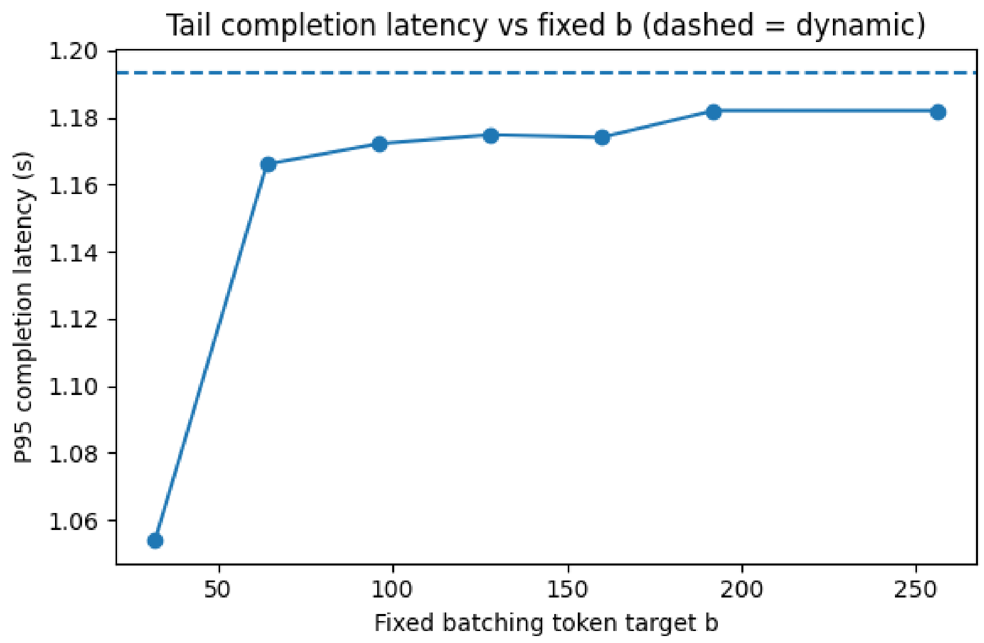
Throughput vs Fixed Batch Token Target (Dynamic Baseline)



Throughput (completed queries per second) as a function of the fixed batching token target b, with the dashed line showing the dynamic batching baseline. In this run, throughput varies only modestly across b and stays in a narrow band, indicating that the system is operating in a regime where changing the fixed token threshold mostly shifts waiting/latency behavior rather than fundamentally changing how many requests can be completed per unit time. The dashed dynamic baseline sits within the same range as the fixed policies, suggesting that dynamic batching is not providing a large throughput advantage under this parameterization and load, and that the primary differentiation between policies will show up in latency and streaming metrics rather than raw capacity.

Appendix F.

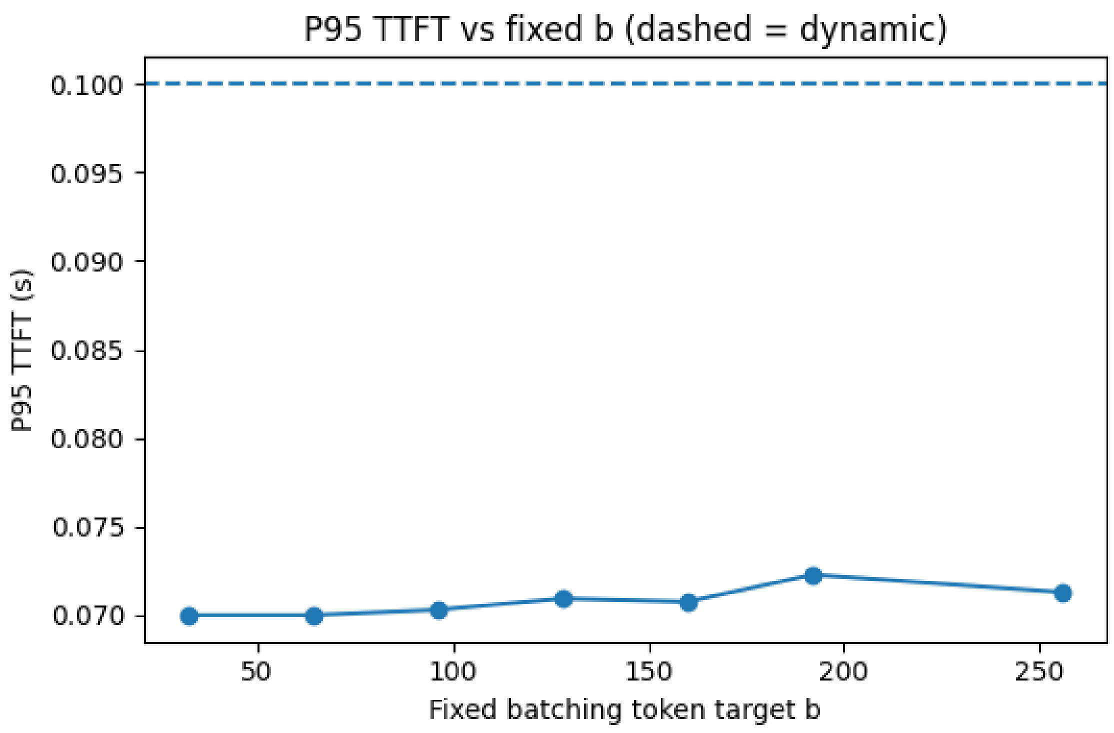
P95 Completion Latency vs Fixed Batch Token Target (Dynamic Baseline)



P95 completion latency (arrival to last token) versus fixed batching token target b, with the dashed line indicating the dynamic baseline. The plot shows that P95 completion latency increases as b grows from very small values into the moderate range, then largely plateaus for b around 64 and above, which is consistent with higher b introducing extra batch-formation waiting that becomes a stable part of the end-to-end completion time. The dynamic baseline is higher than most fixed settings in this run, implying that while dynamic may reduce typical completion latency for many requests (as seen in the histogram comparisons), it can still have a slightly heavier tail at the 95th percentile depending on how its latency guard triggers and how variable its batch sizes become.

Appendix G.

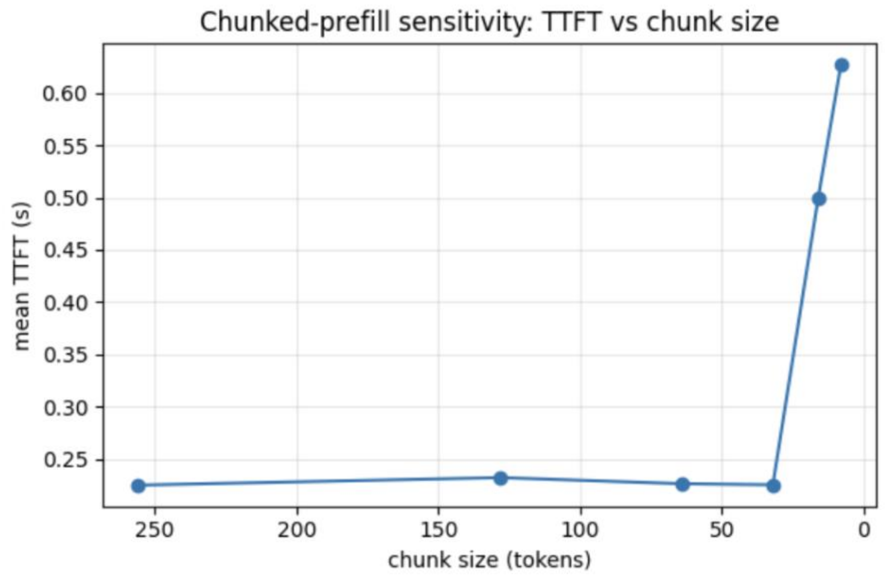
P95 TTFT vs Fixed Batch Token Target (Dynamic Baseline)



P95 time-to-first-token (TTFT) versus fixed batching token target b, with the dashed line indicating the dynamic baseline. Across all fixed b values shown, P95 TTFT remains low and relatively flat, which suggests that under these settings the prefill phase is not heavily backlogged and that batch formation does not dramatically delay the first token for most requests. Dynamic's dashed baseline is noticeably higher than the fixed values here, indicating that the specific dynamic configuration used in this run can sometimes delay prefill completion for a nontrivial fraction of requests, likely because the latency guard and variable batch targeting change when prefill jobs get dispatched relative to the fixed policy.

Appendix H.

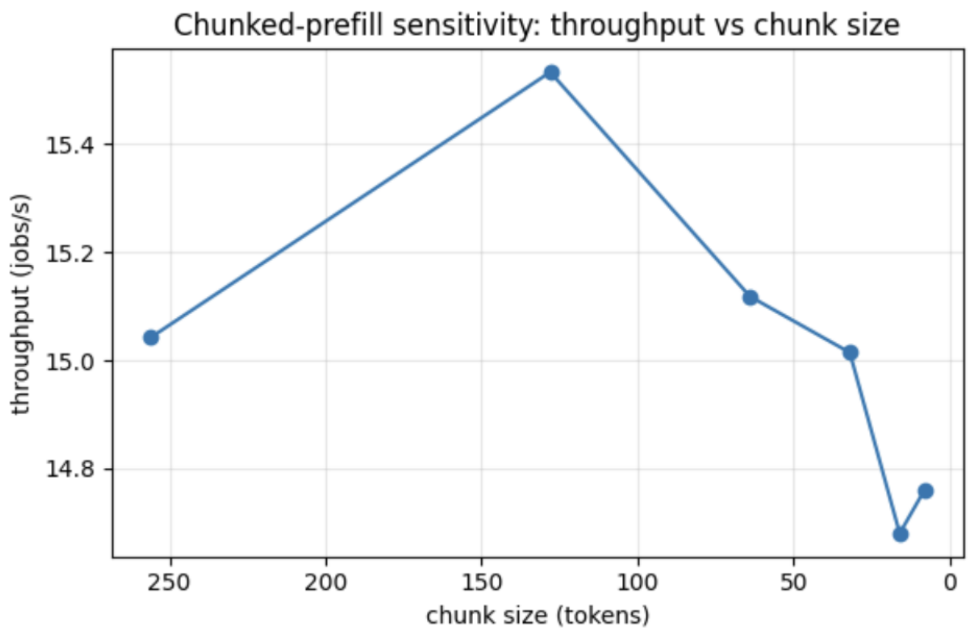
Chunked-prefill sensitivity: mean TTFT (arrival to first token) vs. chunk size (tokens) for chunk_size ∈{8, 16, 32, 64, 128, 256} under fixed batch token budget K=128.



This figure plots the average time-to-first-token (TTFT) across completed requests for each chunked-prefill setting. TTFT is measured from a request's arrival to the emission of its first decode token. For moderate chunk sizes (32-256), mean TTFT is nearly flat at ~0.225-0.232s (with p95 ~0.399-0.406s), indicating that chunking at these levels does not materially delay the first token. However, when chunk_size becomes very small, TTFT rises sharply: mean TTFT increases to ~0.500s at chunk_size=16 and ~0.627s at chunk_size=8 (p95 up to ~0.692s and ~0.884s). This pattern suggests that overly fine-grained prefill chunking introduces enough iteration/scheduling overhead (and/or more frequent prefill-decode alternation) that requests spend longer waiting before their first decode token is produced.

Appendix I.

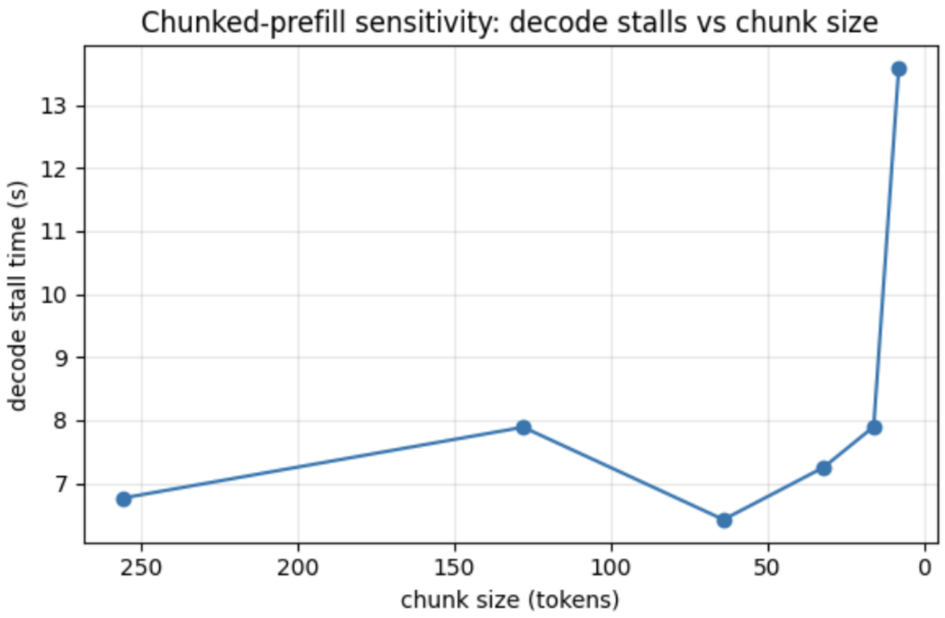
Chunked-prefill sensitivity: throughput (jobs/s) vs. chunk size (tokens) for chunk_size ∈{8, 16, 32, 64, 128, 256} under fixed batch token budget K=128.



This figure reports system throughput, measured as completed jobs per second, as chunk_size varies. Throughput is highest at chunk_size=128 (~15.53 jobs/s) and remains close to ~15.0-15.12 jobs/s for chunk_size in {32, 64, 256}. When chunk_size is reduced to 16 and 8, throughput drops to ~14.68-14.76 jobs/s. Since utilization stays ~1.0 across settings, the throughput reduction at small chunk sizes is consistent with lost efficiency from extra scheduling/iteration overhead and reduced batching effectiveness, rather than GPU idling.

Appendix J.

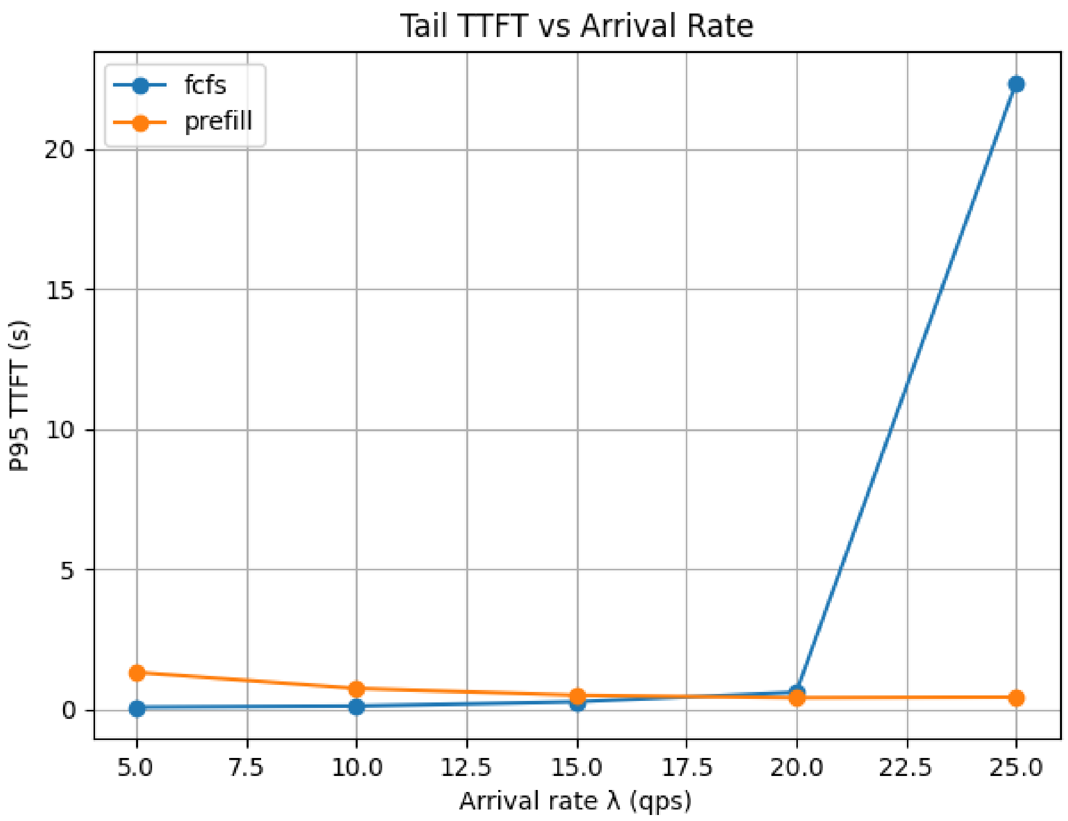
Chunked-prefill sensitivity: decode stall time (s) vs. chunk size (tokens) for chunk_size ∈{8, 16, 32, 64, 128, 256} under fixed batch token budget K=128.



This figure shows the simulator's decode stall time, an internal metric capturing time spent in states where decode work is delayed (e.g., decode-ready requests waiting while the system services prefill or cannot sustain smooth decode iterations). For chunk sizes 32-256, decode stall time stays in a relatively narrow band (~6.42-7.89s). In contrast, chunk_size=8 produces a large spike to ~13.58s, indicating substantially more decode disruption under extremely fine chunking. This aligns with the latency metrics: at chunk_size=8, TTFT and completion latency both worsen sharply, and p95 TBT also increases (from ~0.020s to ~0.040s), consistent with more intermittent decode progress.

Appendix K.

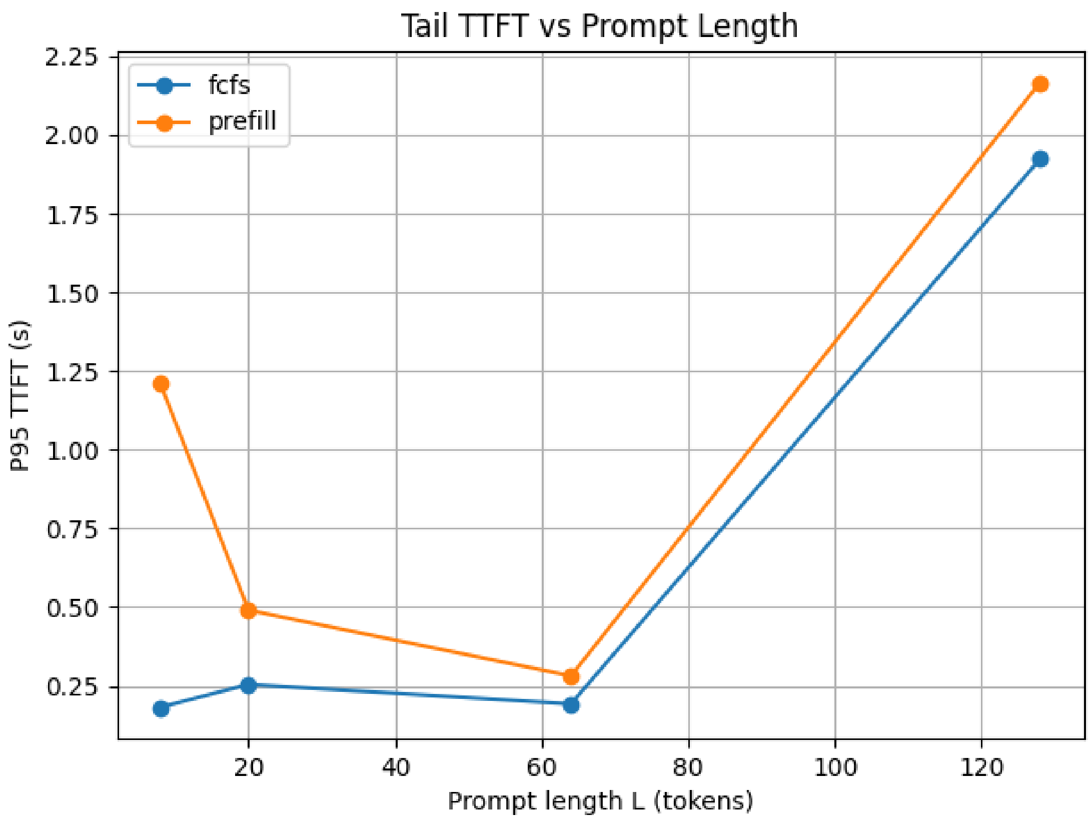
Tail TTFT vs Arrival Rate



Time To First Token (TTFT) at the 95th percentile as a function of the arrival rate λ. This plot illustrates how increasing system load impacts tail responsiveness under different scheduling policies. As arrival rate increases, tail TTFT grows slowly at first and then rises sharply once the system approaches saturation, reflecting queue buildup and head-of-line blocking effects. The transition is particularly pronounced under process-to-completion scheduling, where long-running decode phases delay newly arriving requests. Policies that prioritize or batch prefill work exhibit a smoother degradation, indicating improved robustness to load spikes. Overall, this figure highlights arrival rate as the dominant driver of tail latency collapse and motivates scheduling designs that protect TTFT under high utilization.

Appendix L.

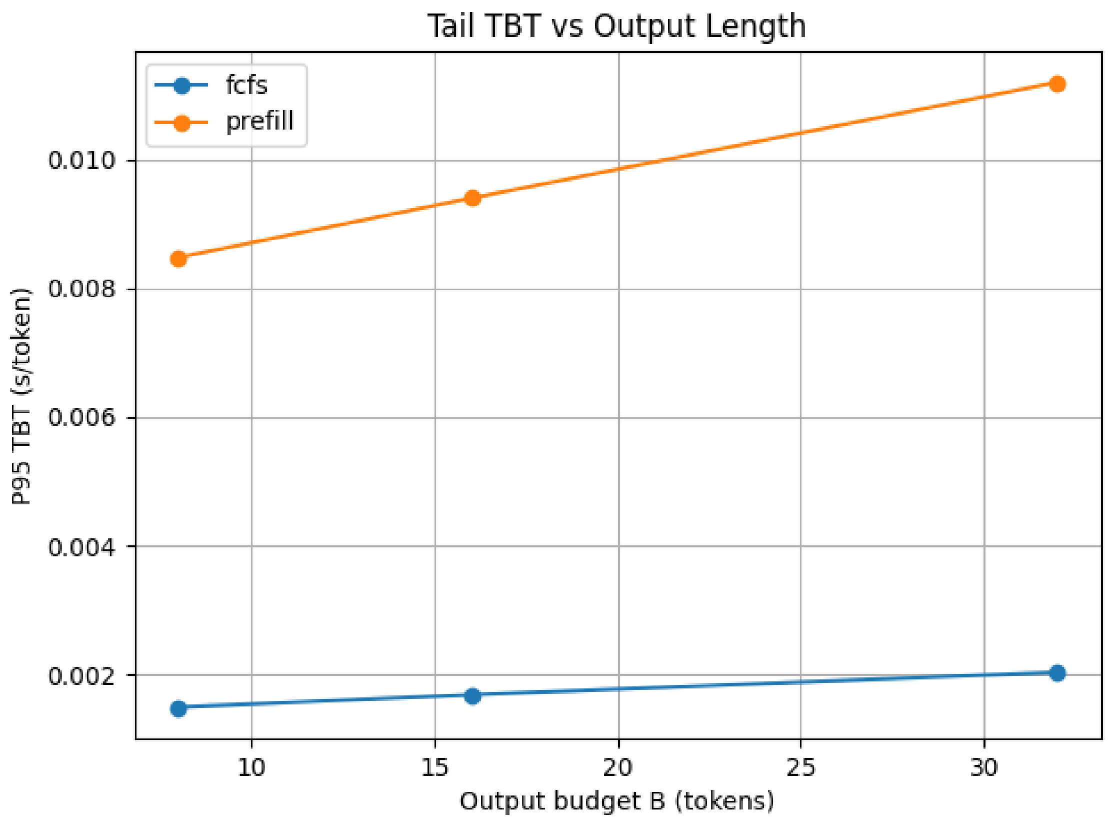
Tail TTFT vs Prompt Length



Tail TTFT as a function of prompt length L, holding arrival rate and output length fixed. Prompt length directly determines the amount of prefill computation required before the first token can be generated. As L increases, tail TTFT rises for all policies, reflecting increased GPU service time per request. Under fixed or FCFS-style scheduling, long prompts introduce significant head-of-line blocking, increasing waiting time for subsequent arrivals. Batching-based approaches mitigate this effect by amortizing prefill cost across multiple requests, though very large prompts can still inflate batch service time. This figure demonstrates that prompt length primarily impacts user-perceived responsiveness through prefill overhead rather than decode contention.

Appendix M.

Tail TBT vs Output Length



Time Between Tokens (TBT) at the 95th percentile as a function of output length B. Unlike TTFT, which is dominated by prefill behavior, TBT captures decode-phase smoothness and streaming quality. As output length increases, tail TBT rises due to prolonged decode activity and increased contention for GPU cycles. Under process-to-completion scheduling, TBT grows relatively slowly, as each request decodes without interleaving. In contrast, batching-based or prefill-prioritized policies introduce additional decode interference, leading to higher inter-token gaps at the tail. This figure highlights a key trade-off: policies that improve TTFT under load often shift congestion to the decode phase, degrading streaming smoothness for long outputs.