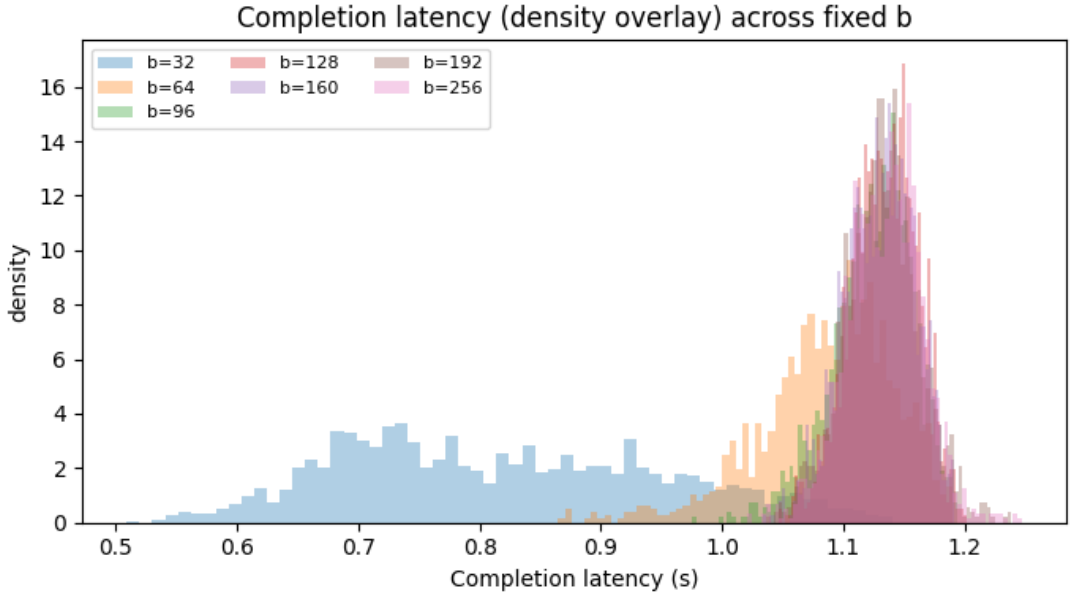
# Appendix

## Appendix A.

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

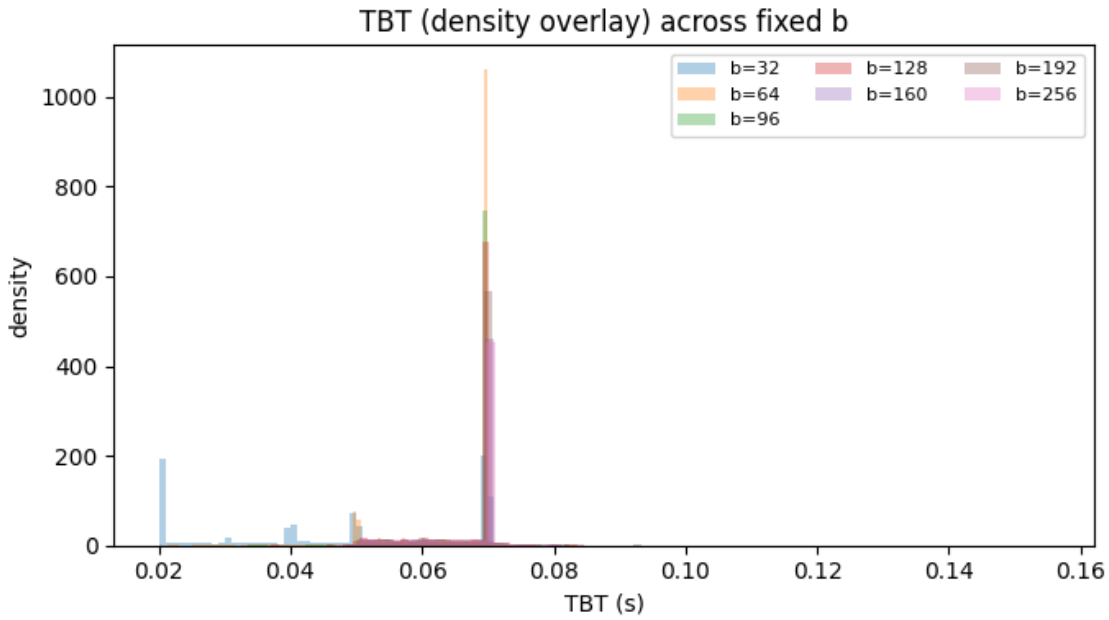



This figure overlays the empirical density of per-request completion latency for each fixed batching policy, where completion latency is measured from the request's arrival time to the completion of its final decode token. Each curve corresponds to a fixed token target b; the scheduler dispatches a batch 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}.*

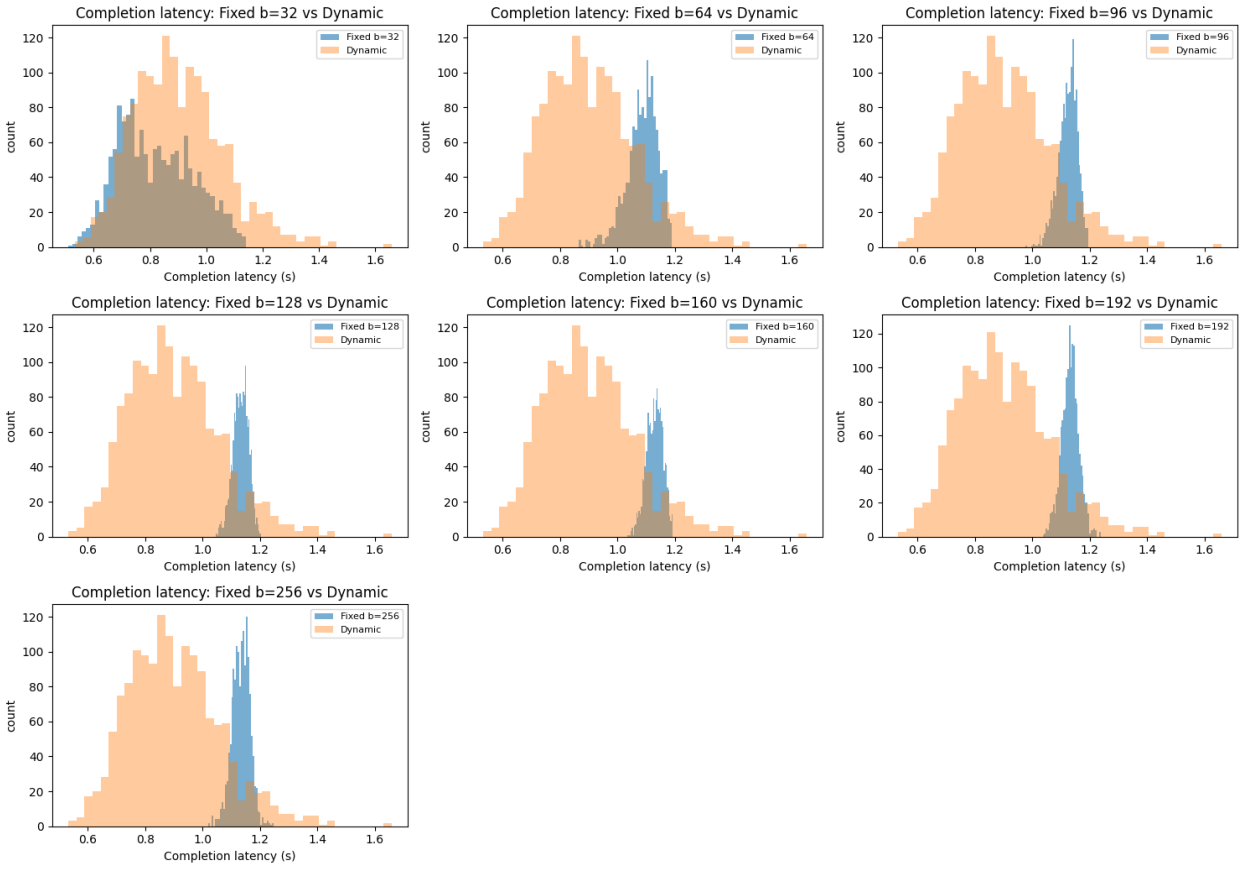



This figure overlays the empirical density of token-level Time Between Tokens (TBT) under different fixed batching targets. TBT measures the gap between consecutive token completions for the same request during the decode phase, so it is a direct proxy for "streaming smoothness" from a user perspective. 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*

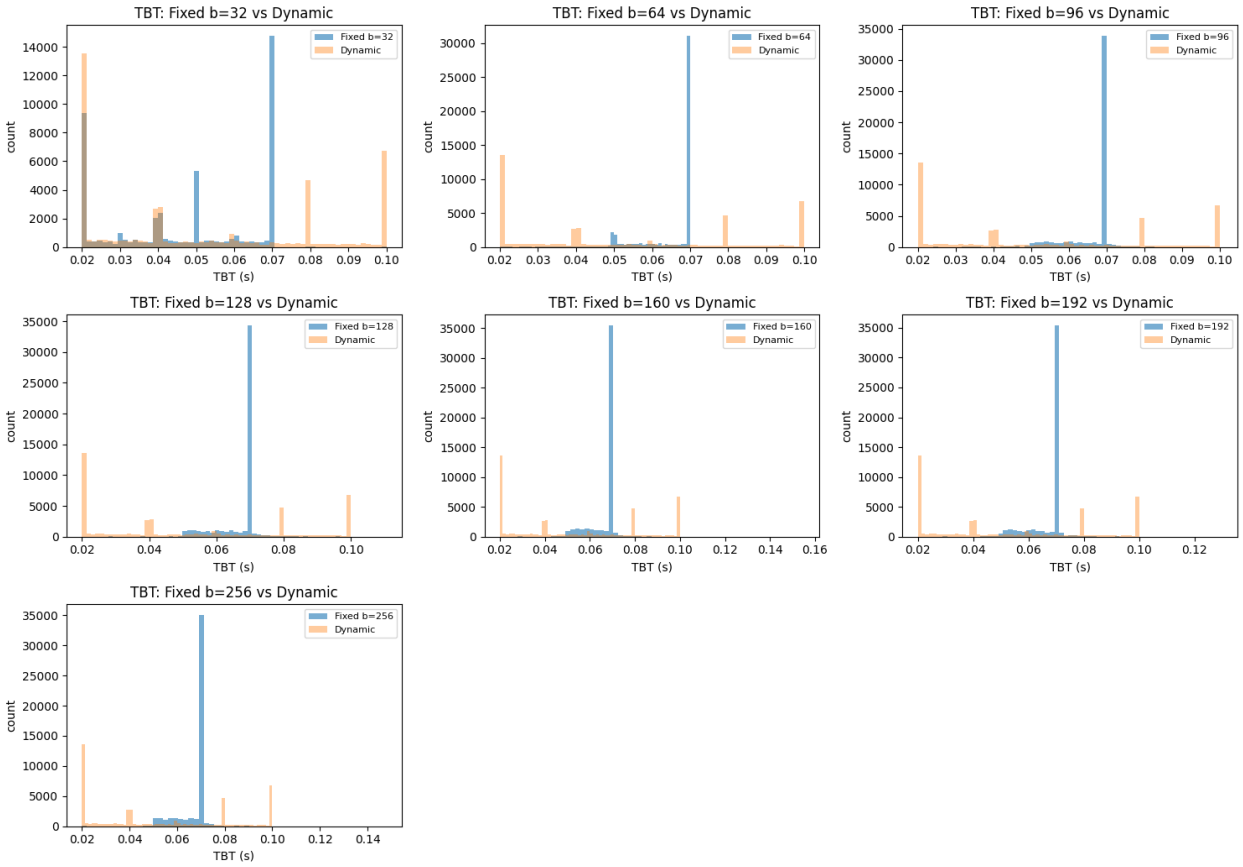



This set of panels compares the completion-latency distribution under each fixed batching target (blue) to the same workload under the dynamic batching policy (orange). The main pattern is that dynamic batching tends to place more probability mass at lower completion latencies (roughly 0.7-1.0 seconds), 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*

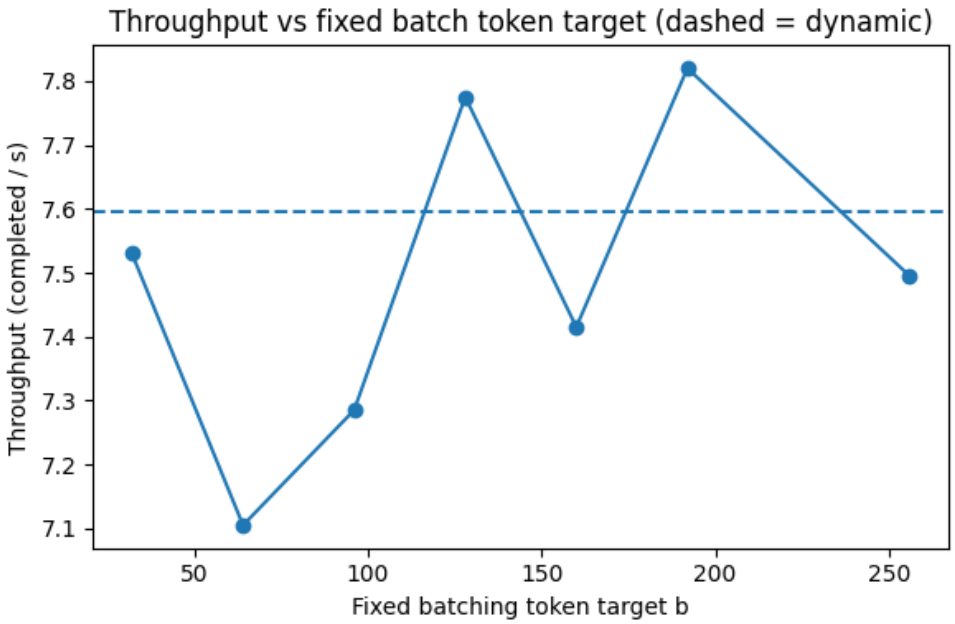



These panels compare the distribution of token-to-token gaps during decode under each fixed batching target (blue) versus the dynamic policy (orange). A consistent pattern across b values is that the fixed policies produce a very sharp concentration of TBT around a narrow band near roughly 0.06-0.07 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.

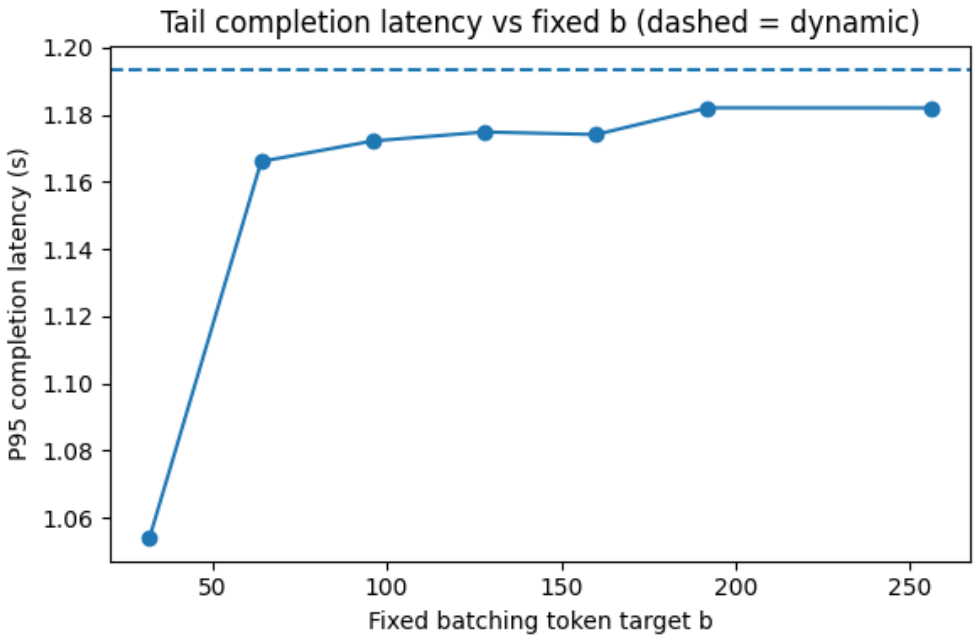
*Throughput vs Fixed Batch Token Target (Dynamic Baseline)*




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.

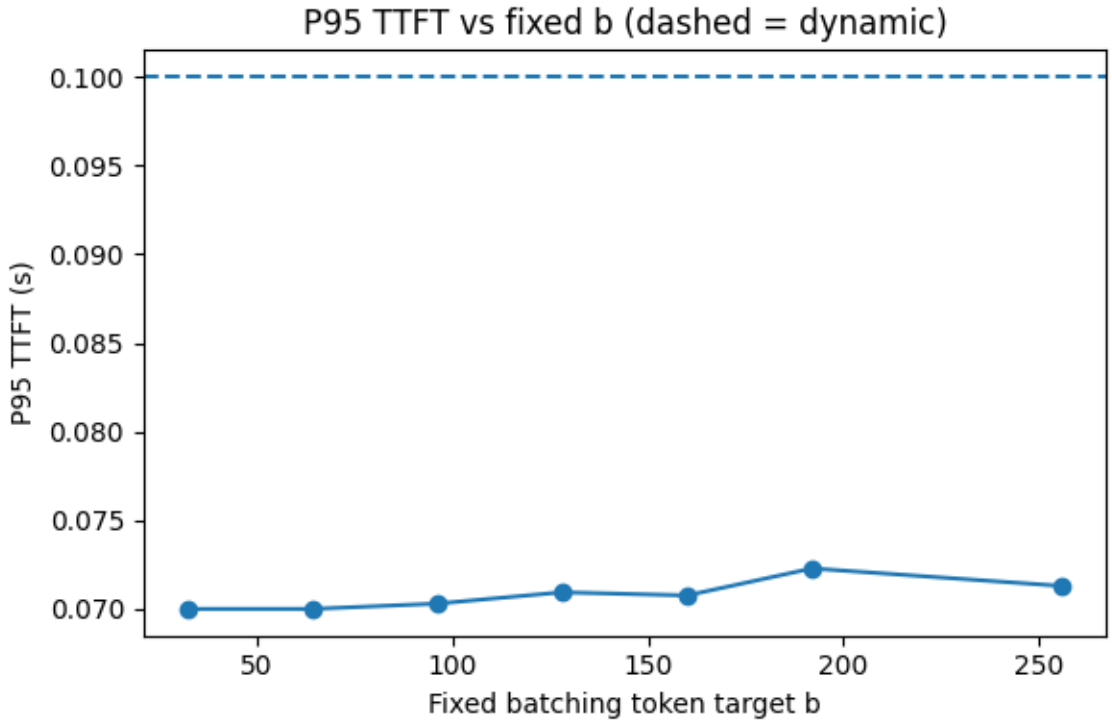
*P95 Completion Latency vs Fixed Batch Token Target (Dynamic Baseline)*




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.

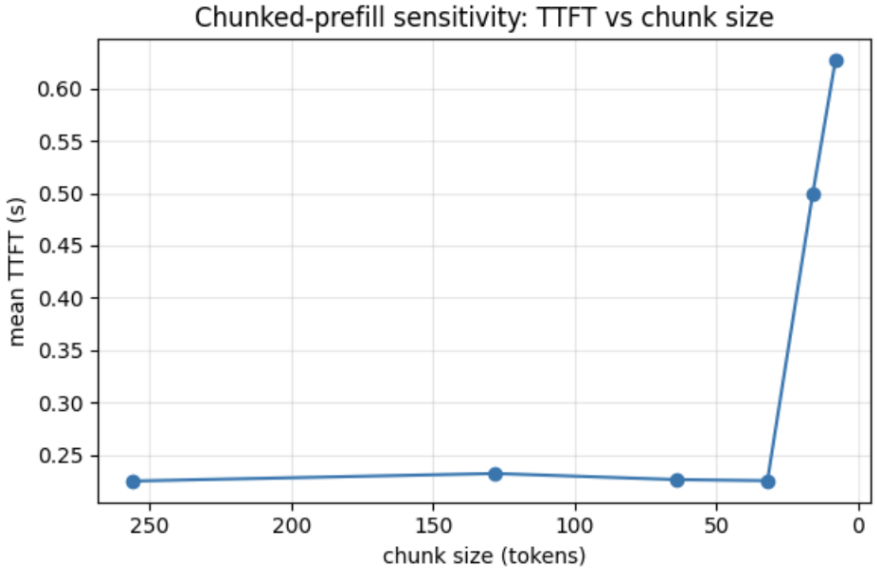
*P95 TTFT vs Fixed Batch Token Target (Dynamic Baseline)*




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.

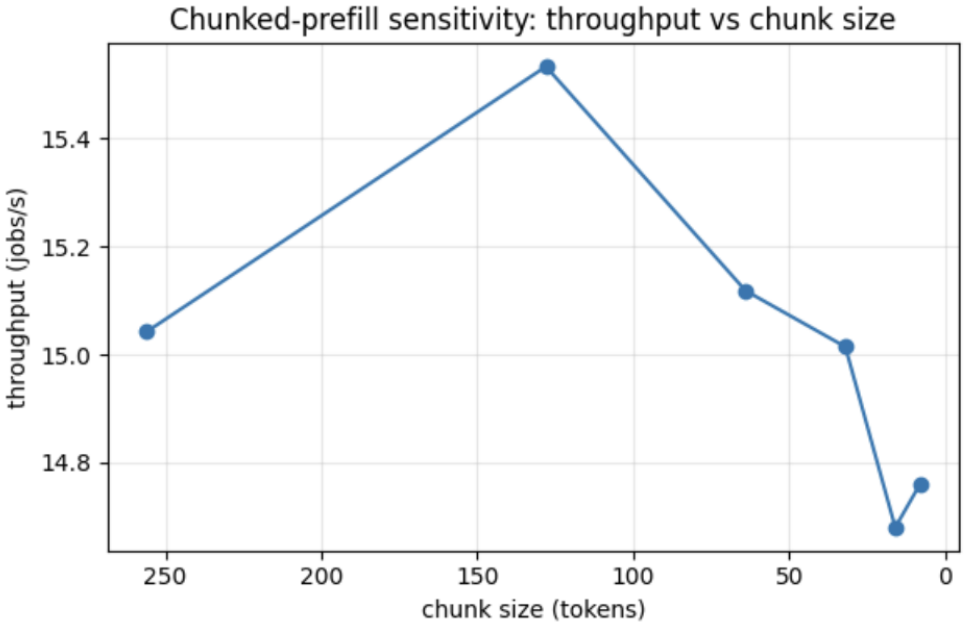
*Chunked-prefill sensitivity: mean TTFT (arrival to first token) vs. chunk size (tokens) for chunk_size ∈{8, 16, 32, 64, 128, 256} under fixed batch token budget K=128.*




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.

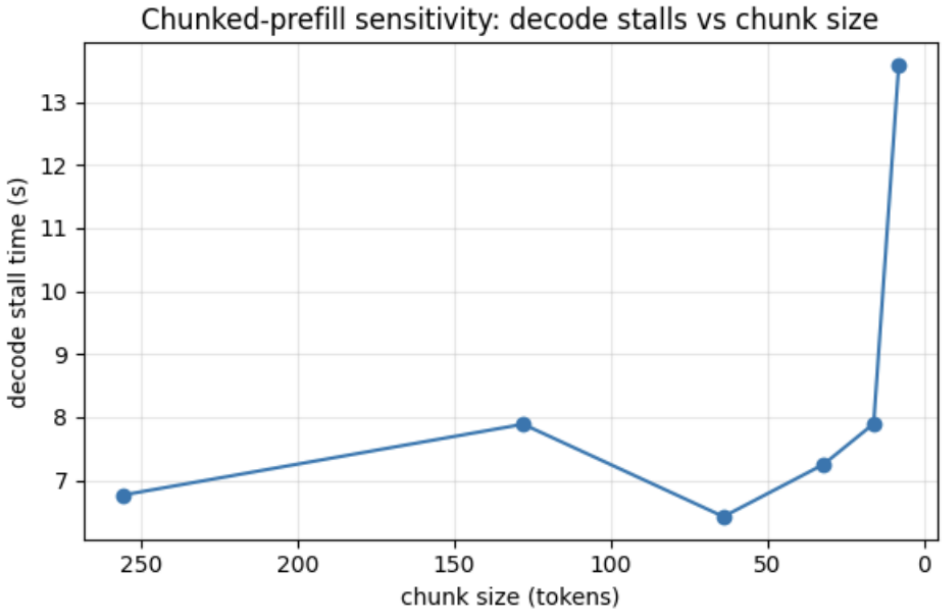
*Chunked-prefill sensitivity: throughput (jobs/s) vs. chunk size (tokens) for chunk_size ∈{8, 16, 32, 64, 128, 256} under fixed batch token budget K=128.*




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.

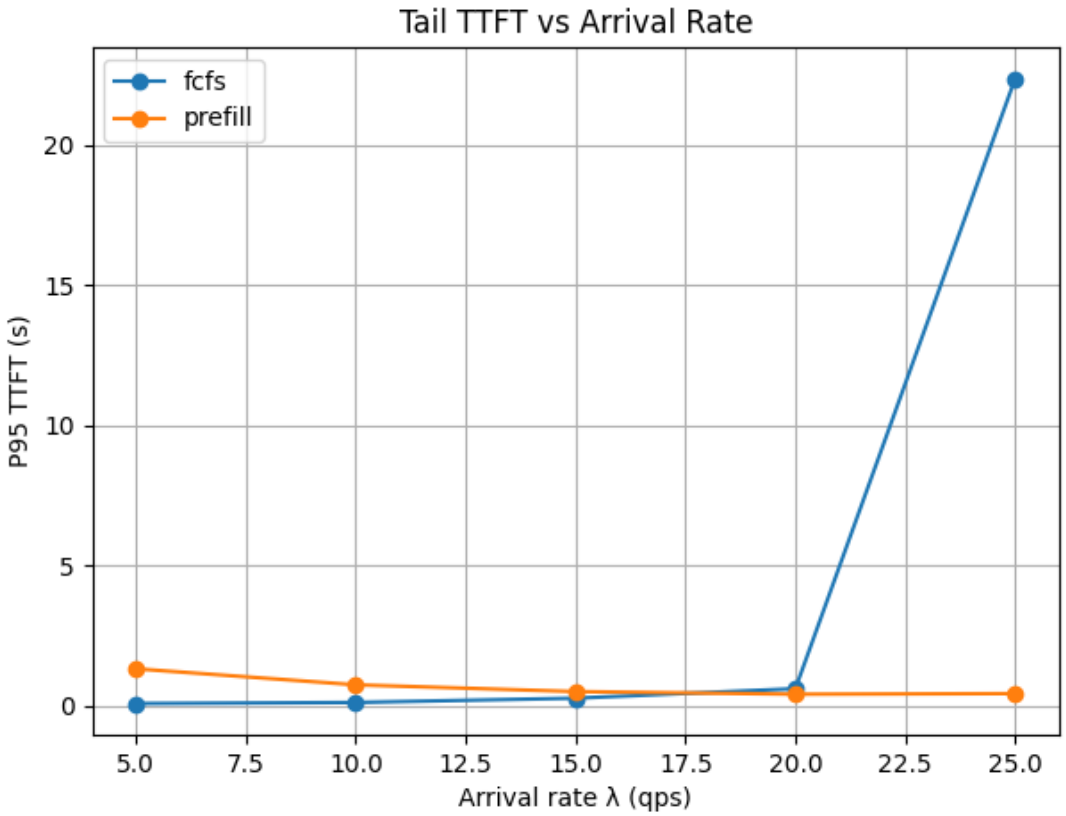
*Chunked-prefill sensitivity: decode stall time (s) vs. chunk size (tokens) for chunk_size ∈{8, 16, 32, 64, 128, 256} under fixed batch token budget K=128.*




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.

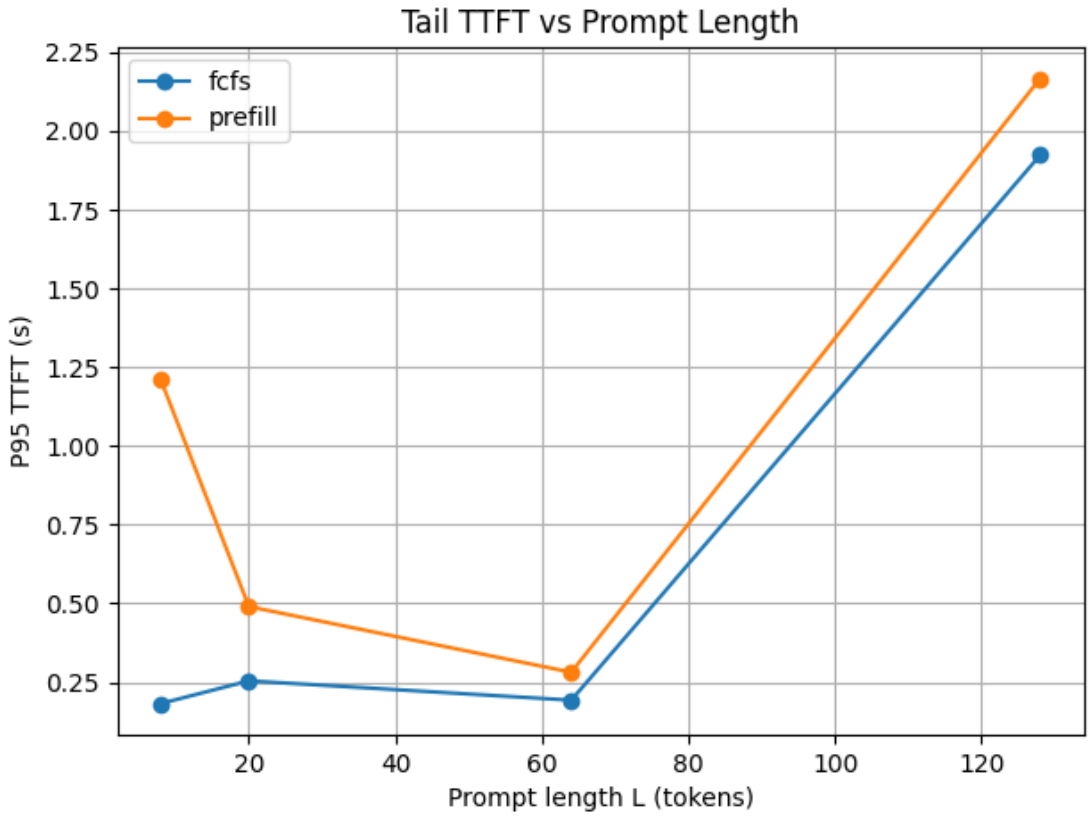
*Tail TTFT vs Arrival Rate*




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.

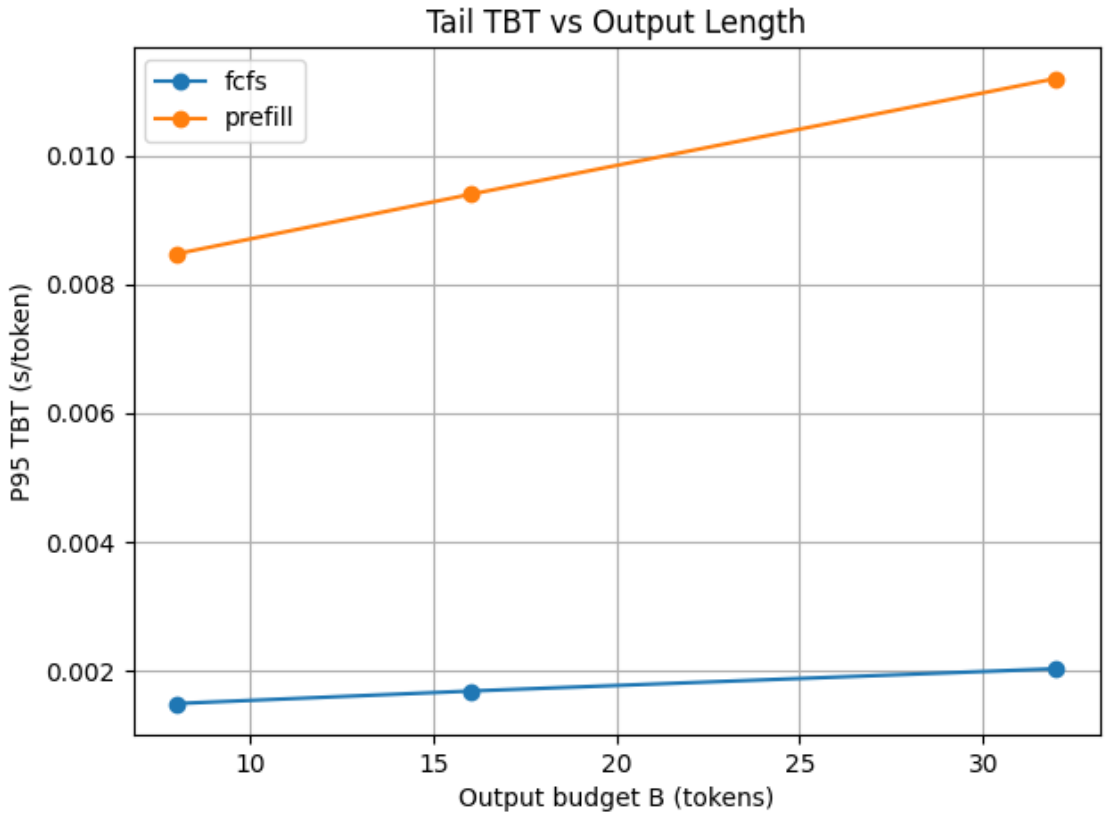
*Tail TTFT vs Prompt Length*




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.