ORIE 4580 Final Report

London Brown, Courtney Candy, Kayla Levy, Claire Wang, Katie Zelvin

Executive Summary

Large Language Models (LLMs) are an essential part of modern artificial intelligence that are exemplified by models such as ChatGPT, Claude, and Gemini. However, these serving systems present unique engineering challenges as they must handle variable workloads and complicated scheduling policies to balance latency, throughput, and efficiency. An LLM leverages the technique Key-Value caches in real systems. The idea is that when someone asks an LLM a query, the query must go through two steps: prefill and decode. The prefill phase processes the input prompt and emits a token, while the decode phase generates output tokens sequentially. The prefill phase is compute-heavy, occupying more GPU resources, and the decode phase is relatively quick. This project aimed to construct a simulation model of an LLM serving system to analyze its performance while focusing on the tradeoffs between improving latency, throughput, and resource utilization under different scheduling and batching configurations.

The main goal of this project was to identify strategies that maximize system throughput while maintaining low latency, which is typically measured using two Service-Level Objectives (SLOs): Time to First Token (TTFT) and Time Between Tokens (TBT). We used Python to develop a discrete event simulation that captures the arrival of user queries randomly over time, each requiring a full prefill of its prompt followed by sequential token decoding, as well as batching dynamics, and GPU service times based on cost.

Our results revealed several insights surrounding the design of LLM platforms. First, we found that large batches improve throughput. Each batch has a fixed overhead cost c. Small batches pay the setup cost c several times, but reasonable batch sizes pay c for many tokens at once. However, excessive batching spikes increase in TTFT as new query arrivals wait to join a batch. We also found that short decode tasks get stuck behind long prefill tasks, driving tail latency. This skewed latency is particularly present under First Come First Serve (FCFS) scheduling, providing a reasonable estimate of the system's capacity, underscoring the need for latency-aware scheduling. Finally, we found that employing chunked-prefill, which involves splitting the prefill phase into smaller token chunks, reduces latency by preventing large queries from taking over the GPU while maintaining acceptable throughput. These findings demonstrate

that effective LLM serving systems require dynamic scheduling policies rather than simple FCFS.

Our simulation model serves as a practical decision support tool for engineering teams responsible for large LLM services. Our model captures realistic workload variability, batching dynamics, scheduling, and GPU behavior and can therefore be used to evaluate new configurations before production. For example, these teams can use our simulation model to test the impact of additional GPUs on cost, alternative scheduling policies, and workload sensitivity without risking poor production results. Not only does our simulator highlight the tradeoffs present in current LLM designs, but it also provides a flexible starting point for ongoing performance optimization and capacity planning in LLM service systems.

Modeling Approach, Assumptions, and Parameters

We wanted to analyze LLM serving without the prohibitive cost of reserving actual GPU clusters. So, we employed a Discrete Event Simulation (DES) approach. This method is what allowed us to model the system as a sequence of events. This included query arrivals, batch formations, and service completions that change the system's state at discrete time points. Rather than modeling the GPU as a black box with a fixed service rate, the basis of our modeling approach was creating the GPU as a processor with state-dependent service times. These service times varied based on the specific "token load" of the current batch.

Our simulation assumes we are using a single GPU. The purpose of this was to isolate the dynamics of queueing, before considering distributed systems. We modeled the arrival of user queries as a Poisson process with a rate λ. These Poisson arrivals imply that inter-arrival times are exponentially distributed. This is a standard assumption for services where each user request is independent of the others. To make our simulation similar to real-world use cases, we do not assume fixed request sizes. The prompt length L_i (the number of input tokens) was modeled using a Poisson distribution. Most queries cluster around a mean length; however, there is a significant variance between queries. Similarly, the number of output tokens, $B_{i,}$ to be generated was modeled using a Geometric distribution. The distribution captures the variability in lengths of real conversations. Sometimes a query could result in a short "yes/no" answer, but other times the query may be answered with a long response.

An important aspect of our modeling approach is the distinction between the "prefill" and "decode" phases. In our model, a request must first complete a prefill step. The prefill step processes all L_i tokens in parallel. Next, the request enters the decode phase. In the decode phase, the request passes through the GPU B_i times. This generates one token per pass. The service time for any batch is modeled using the linear approximation as follows: S(b) = C + a * b. Here, "C" is the fixed overhead of launching a kernel on the GPU (latency bound) and "a" is the marginal cost per token (compute-bound). To reflect real-world stochastic processes, the actual service duration in our simulation is drawn from an exponential distribution. The exponential distribution has a mean equal to the calculated S(b).

We established baseline parameters to resemble a realistic Llama inference workload on A100 GPUs. We set a mean prompt length of 173 tokens and a mean output budget of 14 tokens. The fixed setup cost "C" and marginal cost "a" were calibrated to an average of the values typically seen in Llama, c = 35 ms and a = 0.3 ms/token. This reflects the high throughput potential that we can achieve through batching. By varying the arrival rate λ with respect to the service rate, we were able to shift through different utilization levels. This pushed the system from a stable state into saturation, allowing us to see how latencies and queue lengths degrade when under pressure.

Model Details and Implementation

The model is developed using Python. The model contains three classes:

- **Requests:** Class represents queries given to the LLM; tracks arrival, time to first token, and departure time of each query. Simulation adds prompt lengths and output length to each request.
- **SystemStateTracker**: Class enables engines by tracking the time of the system and the number of queries in the system.
- **LLMEngine:** Class allows simulation to take the next step based on current batching and scheduling parameters and server parameters.
- **ChunkedLLMEngine:** A child class of LLMEngine with adjustments in step functions to include Chunked Prefill. It contains the additional parameter chunk_size and tracks prefill progress to enable adjustments to the step functions.

Experiments and validation were repeated through these functions:

- **run_simulation:** Takes desired arrival rates of requests, total number of queries, and batching and scheduling mode to process as input. It assigns arrival times and prompt/output lengths based on random variables. It adds arrival to the Engine based on this information. It tracks key metrics in latency and throughput.

- **validation_mm1:** Validates the accuracy of the simulation by comparing it to M/M/1 queue, a well-behaved system. Accuracy contains limitations since service time is not exponential.
- **run_chunk_experiment:** Acts similarly to run_simulation, however, takes a parameter chunk size and uses ChunkedLLMEngine to step forward the simulation.

Additional Code acted as support in sensitivity, validation, and visualization:

- **Sensitivity Analyses:** Compare time to first token (TTFT), time between tokens (TBT), throughput, and average queries in the system for each batching, scheduling, and chunking strategy over a range of arrival rates.
- **Graphical Validation:** Look for phase transitions with throughput and latency across arrival rates.
- **Additional Repetition:** Build Confidence Intervals to understand the effect of randomness on policy performance.
- **Visualization:** Build graphs to show how different policies can impact customer experience and company costs.

Outside of code, models of queueing and service systems allowed calculations of expected metrics based on parameters:

- **Expected Throughput:** Calculate expected throughput for the given policy and compare it to the observed throughput.

The model was built iteratively, slowly increasing the number of parameters classes and functions take. Initially, the model defaulted to a simple, single-GPU, decode-prioritizing, first-come, first-served batching and scheduling. With additional iterations, batching, chunked prefill, and multiple GPUs were added. During each iteration, validation, including graphical phase transitions and expected throughput calculations, was completed.

Scheduling in LLM models is found in three key methods: Decode-Prioritizing, Prefill-Prioritizing, and Hybrid Scheduling. In the model, the baseline was considered decode-prioritizing. This means the queries in which the prefill, or prompt reading, is already complete are prioritized. New queries in which the prefill needs to be processed have to wait.

Baseline Scheduling: First-Come First-Serve
Process each query to completion before processing the next query in the queue. The simulation is moved through time with an associated scheduling step function:

FUNCTION STEP_FCFS():

    IF running_queue **is NOT** empty **THEN**

```
        req <- first request in running_queue
    ELSE
        req <- first request in waiting_queue
        running_queue.append(req)
        req.start_time <- current_time
    IF NOT req.prefill_complete THEN
        batch_tokens <- req.prompt_length
        service_time <- compute_batch_service_time(batch_tokens)
        current_time <- current_time + service_time
        req.tokens_generated <- 1
        req.first_token_time <- current_time
    ELSE
        batch_tokens <- 1
        service_time <- compute_batch_service_time(1)
        current_time <- current_time + service_time
        req.tokens_generated <- req.tokens_generated + 1
    END IF

    IF req.finished THEN
        req.end_time <- current_time
        complete_req_list.append(req)
        completion_times.append(current_time)
        running_queue <- empty
    END IF
```

The next method combined batching with a decode-prioritizing schedule, which used a variation of the above step function.

```
FUNCTION STEP_BATCHING():
    FOR req IN running_queue:
        IF current_batch_size + 1 ≤ max_batch_size THEN
            batch_requests.append(req)
            active_decodes.append(req)
            current_batch_size += 1
        END IF
    IF self.waiting_queue THEN
        req = first request IN waiting_queue
        IF current_batch_size + req.prefill_length ≤ max_batch_size THEN
            waiting_queue.pop(req)
            batch_requests.append(req)
```

```
                current_batch_size += req.prefill_length
            END IF
        END IF
```
[There are steps to service the batch that follow this code before the step function completes.]

The “step_batching” iterates through requests in the running queue. If the maximum batch size has not yet been reached, add the request to the batch for decoding. Then, look at the first request in the waiting queue. If the maximum batch size would not be exceeded by adding a request to prefill, then add it to the batch as a prefill request. After the batch is full, the batch receives service, and the step completes.

Validation was done through viewing the output metrics: Time Between Tokens, Time to First Token, and Throughput across Arrival Rates. Since the batch size is larger, there is an expected increase in throughput as the fixed cost to set up a batch is spread across more tokens. Then, the prioritization of decode requests will delay the first token of new requests, increasing the time to first token. These metrics behaved as expected in validation. To validate throughput, the calculations to support the baseline scheduling as available in the Appendix.

Model Analysis and Results

We performed various experiments to compare the FCFS and prefill prioritizing batching schedulers under an increase in workload intensity. In particular, we focused on differences in throughput, latency behavior (TTFT and TBT), queue evolution, and overall system stability with many arrival rates. These experiments show the tradeoffs between efficiency and responsiveness in LLM systems.

Under FCFS scheduling, the system processes each query to completion before beginning service on the next. Given our prompt, output length distributions, and service time function, the expected service time per query is approximately 627.64 milliseconds. This corresponds with a theoretical maximum throughput of about 1.59 queries per second when the GPU is fully utilized. Our simulation produced a measured throughput of roughly 1.7 queries per second, which is consistent with this expected value and reflects minor randomness in query lengths and interarrival times. At arrival rates below this capacity, the system remains stable. Most queries begin prefill shortly after arrival, and decode tokens are produced in a relatively timely manner. The system maintains stability, and TTFT and TBT remain modest.

However, as arrival rates approach the system’s maximum throughput or exceed 1.7 queries per second, performance degrades rapidly. Because FCFS processes each query to

completion, including all decode tokens, before starting any work on later arrivals, long prompts and long decode sequences block shorter queries behind them. This creates a significant head of line blocking. TTFT increases because prefill operations for later queries must wait until earlier queries are entirely finished. TBT also increases because decode tokens are processed one at a time with no overlapping across queries. The queue length grows quickly once the arrival rate surpasses the system's capacity, indicating instability, where the system is effectively falling behind, and the number of jobs in the system increases without bound over time. This behavior is typical of a single-server system operated above its theoretical capacity and confirms that FCFS leaves little room for handling bursts or variability.

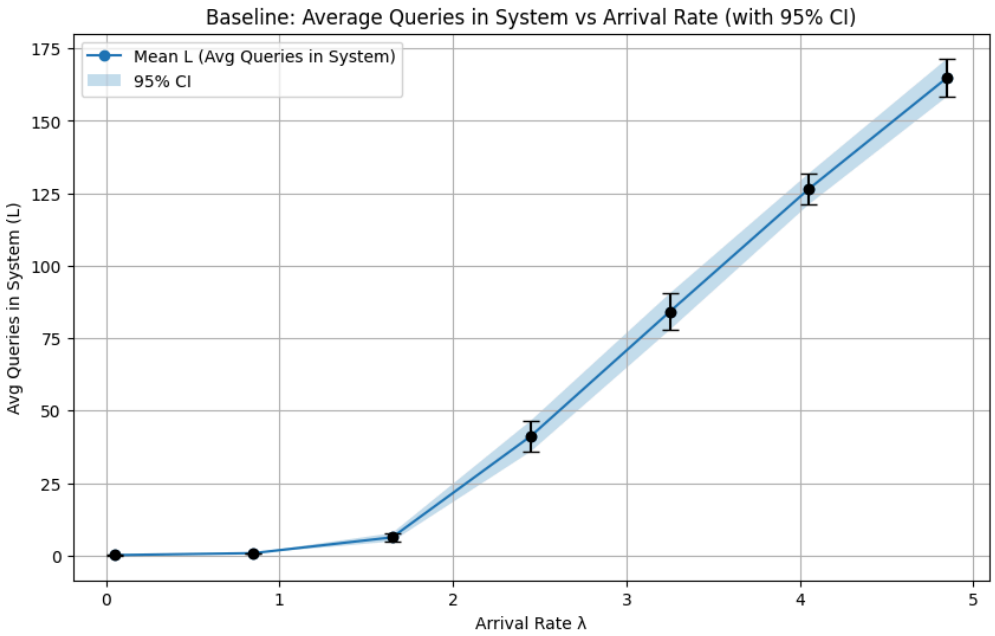

Figure 1: Baseline Simulation: First Come First Serve (FCFS): Average Queries vs. Arrival Rate

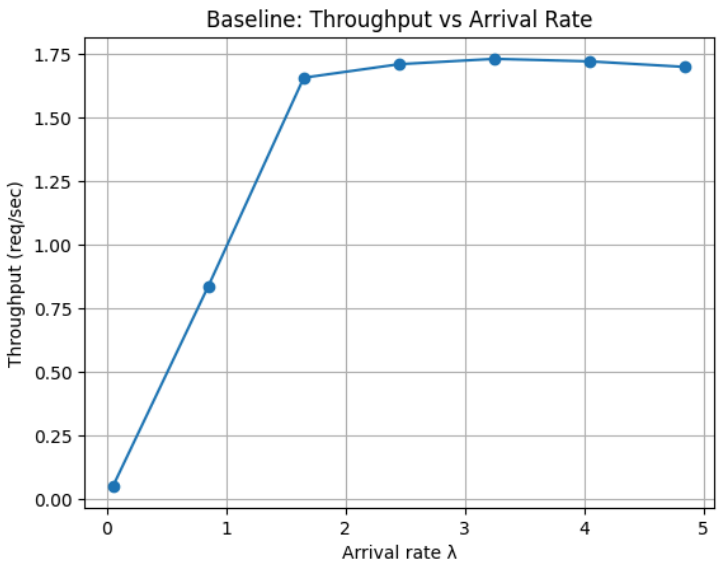

Figure 2: Baseline Simulation: Throughput vs. Arrival Rate

The decode-prioritizing scheduler with batching demonstrates far superior performance. In this configuration, the throughput reaches approximately 7 queries per second, meaning the system can handle nearly twice as many incoming queries before saturating. This improvement stems directly from batching decode operations, which allows the GPU to process many decode tokens simultaneously in a single iteration. Because the fixed setup cost is paid once per batch rather than once per decode token, the amortized cost per token decreases significantly. TTFT also improves because prefill requests are admitted immediately whenever the GPU is free, rather than waiting behind long decode sequences as in FCFS. Users begin seeing their first token sooner, which is especially important for perceived responsiveness.

TBT decreases because batches contain multiple decode tokens from different queries, resulting in a smoother and more parallel execution decode process. Instead of a stop-and-go pattern where each query monopolizes the GPU during its entire decode sequence, the system alternates decode tokens across many queries in shared iterations. Queuing behavior under batching is much more stable. While the system eventually saturates at around 7 queries per second, queue buildup occurs at a slower rate, and the system performs well across a much wider range of arrival rates. Tail latencies remain lower under batching at moderate loads, though at extreme loads, both schedulers experience rapidly increasing delays simply because the arrival rate exceeds the maximum processing capacity.

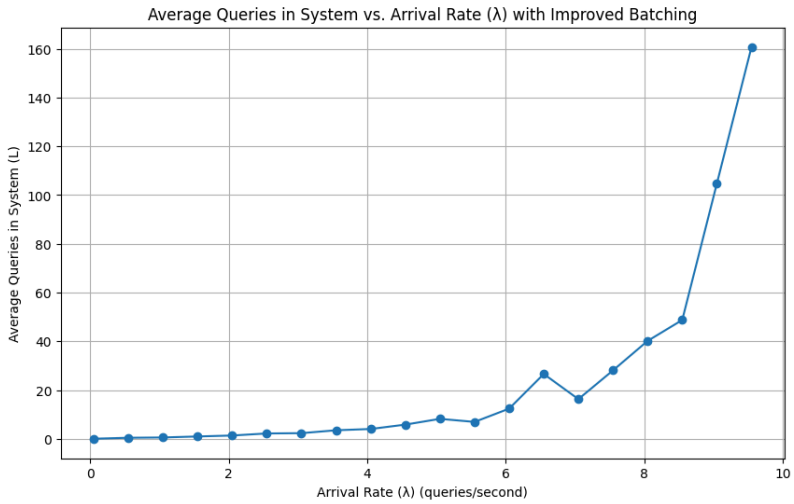

Figure 3: Improved Batching: Average Queries vs. Arrival Rate

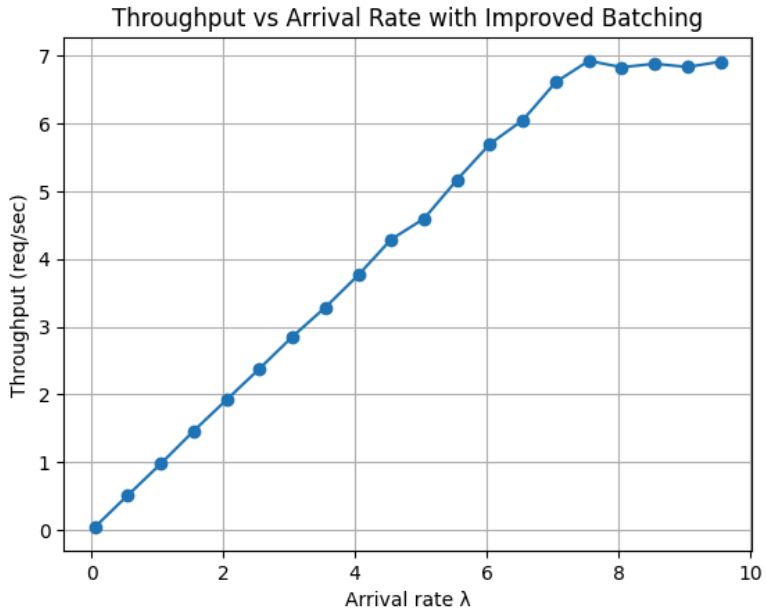

Figure 4: Improved Batching: Throughput vs. Arrival Rate

We also examined steady-state behavior across long-term simulation runs. For each policy, performance metrics stabilized after the warm-up period, confirming that our simulation length was sufficient to capture long-term system characteristics. The FCFS policy consistently showed sharp transitions in latency and queue length once the arrival rate exceeded its capacity, consistent with classic queueing theory. In contrast, the batching scheduler maintained predictability and stability throughout until the arrival rate approached its significantly higher

saturation limit. The contrast between the two policies confirms that batching is not a minor optimization but fundamental for high-performance LLM serving.

Conclusion

Our project successfully developed and utilized a stochastic simulation model to explore the performance characteristics of LLM serving systems. By capturing the unique two-phase execution nature of Generative AI workloads, our model provided a high-fidelity sandbox for testing scheduling and batching theories.

Our investigation led to three primary conclusions. First, we demonstrated that batching is fundamental to system scalability. While our baseline FCFS policy saturated at approximately 1.7 requests per second, the implementation of a batching scheduler significantly improved the system capacity to nearly 7 requests per second. However, we found that batching is not strictly "the more, the better," and there exists a critical threshold beyond which latency penalties (specifically TTFT) outweigh throughput gains.

Second, our saturation analysis highlighted the critical importance of capacity planning. We identified a distinct "hockey stick" latency curve under both policies, allowing us to pinpoint the maximum sustainable throughput before queuing delays dominate and the system becomes unstable.

Finally, the heterogeneity of prefill and decode tasks creates significant interference. Naive scheduling allows compute-heavy prefills to degrade the experience of latency-sensitive decodes. Our experiment showed that advanced scheduling mechanisms, especially Chunked Prefill, are highly effective at mitigating this interference, preventing large queries from monopolizing the GPU while maintaining acceptable throughput.

For future work, the model could be extended to simulate multi-GPU environments with pipeline parallelism or tensor parallelism. This could introduce network interconnect latency as a new variable, adding another layer of complexity to the scheduling problem. Additionally, modeling the memory capacity of the KV-cache would allow for the study of eviction and recomputation policies, which are critical for long context models. Nevertheless, our current simulation provides a robust foundation for understanding the first-order dynamics of the modern AI infrastructure stack.

Appendix

A. Key Notation: From Project Description

λ: Arrival Rate (queries per second)

L_i: prompt length of query *i* (number of prefill tokens)

B_i: output budget (max response length) of query *i* (number of decode tokens)

K: Maximum Batch Size (number of jobs per batch)

b: token load in a batch (total number of tokens across all jobs)

S(b): service time for a batch with token load *b* (milliseconds)

c: setup/fixed cost per batch (ms)

a: marginal cost per token beyond b_0 (milliseconds per token)

b_0: minimum batch size threshold (tokens)

B. Baseline Simulation Throughput Calculation

Observed Baseline Max Throughput: 1.71 queries/second

Time to Complete 1 Query: Time for Process Prefill Tokens + Time to Generate Decode Tokens

Time (ms) to Process Prefill Tokens = $c + a*E[L_i]$

Time (ms) for Decode Tokens = $E[B_i]*c - a*b_0$

Time (ms) to Complete 1 query: $35 + 0.3*(172.8) + 14*35 - 0.3*64 = 575.04$

Convert to Seconds: 575.04/1000 = 0.575 seconds

Maximum Rate: 1/(Time per Query) = 1/0.575 = 1.739 queries per second

Expected Throughput: 1.739 queries per second

Actual Throughput: 1.71 queries per second

C. Batching Simulation Throughput Calculation

Observed Batching Max Throughput: 6.8-7 queries per second

Maximum Batch Size: 128 queries

Setup Time: 35 ms
Time per Token: 0.3 ms
Expectation Prefill Tokens: 172.8
Expectation Decode Tokens: 14
Expected Set-up Time per Job: 35/128 = 0.273 ms
Time to Process Tokens = (172.8 + 14)*0.3 = 56.04 ms
Time to Complete Query: 56.04 + 0.273 = 56.313 ms
Convert to Seconds: 56.313 /1000 = 0.056313 seconds
Maximum Rate: 1/(Time per Query) = 1/ 0.056313 = 17.758 queries per second

Theoretical Throughput: 17.758 queries per second
Observed Throughput: 6.8-7 queries per second

Cause of Difference: The step function only looks to add one prefill job during each iteration, so the actual batch size is far smaller than the maximum batch size, leading to a smaller throughput than theoretically possible. While this appears non-optimal, increasing the batch size can negatively affect the latency, so this batching style provides a balance in these two metrics.

D. Validation through M/M/1 Queue

To validate the code, the simulation is compared to a well-behaved M/M/1 Queue. However, since the service times are not exponentially distributed, there is expected to be some deviation during the validation.

Code:

```
run_simulation(arrival_rate=0.1, total_queries=2000, verbose=True,
warmup_time=0, analysis_time=float('inf') )
```

The simulation defaults to 1 GPU or 1 server. First-come, first-served scheduling mimics the scheduling pattern of an M/M/1 queue. The arrival rate is 0.1 with Inter-arrival times exponentially distributed with the parameter 10. So, the expected time between arrivals (10

seconds) is greater than the expected service time per query (0.575 seconds). So, the throughput is expected to be the arrival rate, 0.1.

```
Throughput: 0.10
Avg TTFT: 0.1485 s
Avg TBT: 0.0350 s
95th Percentile TTFT: 0.4011 s
95th Percentile TBT: 0.0350 s
```

Figure 1a: Output of M/M/1 Queue Validation Testing

E. Hardware Scalability: Scaling Number of GPUs

For the simple assumed case, service time is divided by the number of GPUs. As the arrival rate increases, the number of queries grows faster with more GPUs. Increasing the number of GPUs is shown to increase the number of arrivals the system is able to handle.

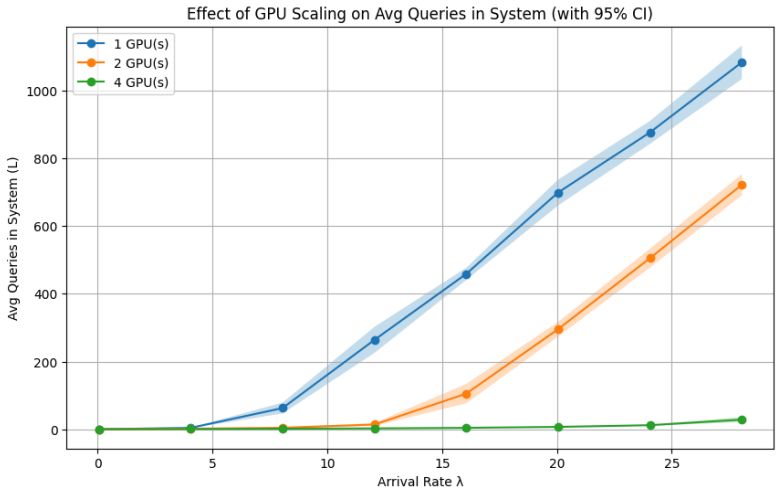

Figure 2a: GPU Scaling enables the successful handling of a faster arrival rate

F. Chunked Prefill Effect on Latency and Throughput

Chunked Prefill allows prefill tokens to be chunked into different batches. Here are the effects of Chunked Prefill on Tail Latency, time between tokens, and throughput. The simulation varies the chunk size from 32 to 2048 while keeping a fixed arrival rate.

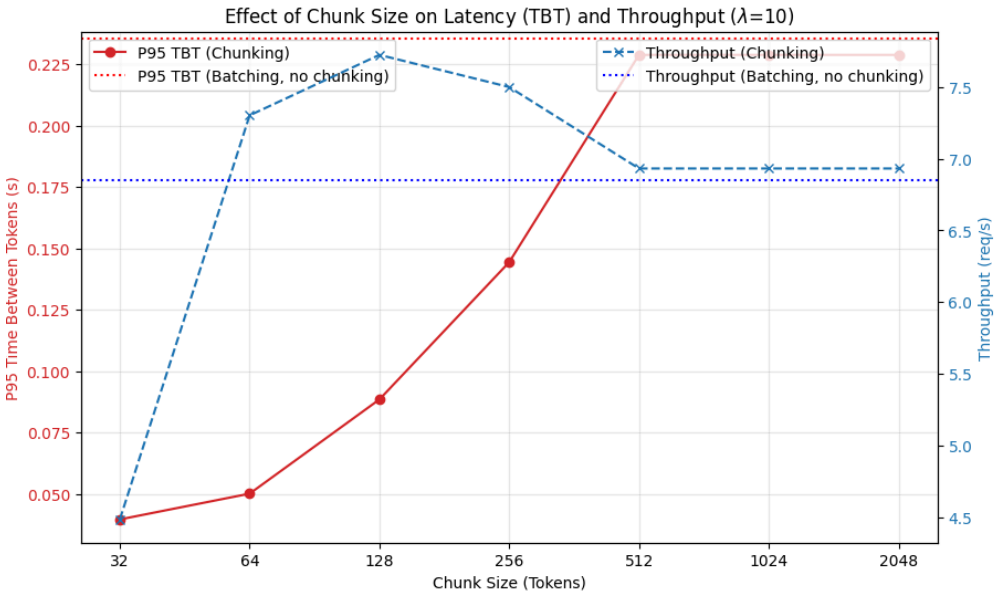

Figure 3a: Effects of Chunked Prefill

For the small chunk size of 32 tokens, there is a sacrifice of throughput for a better tail latency. However, the Chunked Prefill throughput is above the previously used Batching throughput for all other chunk sizes tested. An optimal chunk size is 64-128 tokens, depending on desired tradeoffs between tail latency and throughput.

G. Chunked Prefill Code

```
FUNCTION step_batching():
    FOR req IN running_queue:
        IF batch_actions.len() + 1 <= max_batch_size THEN
            current_batch_tokens += 1
            batch_actions.append(req)
        END IF
    IF waiting_queue IS NOT empty THEN
        Req = first request IN waiting_queue
        Processed = prefill_progress(req)
        Remaining_prompt = req.prompt_len - processed
        IF len(batch_actions) + 1 < max_batch_size:
```

```
                tokens_to_process = min(remaining_prompt, chunk_size)
                current_batch_tokens += tokens_to_process
            END IF
        END IF
```

Then, process batch tokens the same as in other batch functions, recording the number of prefill tokens processed for the request.

The key difference between this and the previous step function is that it adds no more than the chunk size of tokens.