Simulation and Optimization of Batching Strategies and Scheduling Policies in Large Language Model Inference Systems

Jiedong Zhang, Qingyang Xu, Runyuan He

December 18, 2025

Abstract

Large Language Model (LLM) inference represents a unique class of stochastic service systems characterized by highly variable service times, distinct two-phase execution dynamics comprising compute-bound prefill and memory-bound decoding, and stringent latency requirements. This project develops a comprehensive discrete-event simulation (DES) framework to rigorously analyze the performance trade-offs between system throughput, tail latency, and fairness in LLM serving. Utilizing the SimPy library, we modeled an inference engine that replicates the behavioral physics of state-of-the-art systems such as vLLM and Orca. Our experimental results, derived from non-homogeneous Poisson arrival processes and heavy-tailed workload distributions, identify a non-monotonic relationship between batch size and system performance. Specifically, moderate batch sizes in the range of 16 to 32 maximize throughput while maintaining acceptable latency. In contrast, small batch sizes ($B \leq 8$) cause severe throughput bottlenecks. Furthermore, we demonstrate that Shortest Job First (SJF) scheduling reduces average latency by 37% compared to First-Come-First-Served (FCFS). However, this comes at the cost of significant unfairness, as evidenced by Jain's Index dropping from 0.87 to 0.09, with 7% starvation rate, necessitating hybrid scheduling approaches like priority aging. Finally, adaptive batching under traffic spikes reduces aggregate latency by 54% compared to static batching.

1 Introduction

1.1 Motivation

The paradigm shift brought about by Generative AI has transitioned the industry focus from model training to efficient inference serving. Unlike traditional web services, which typically process uniform and short-lived HTTP requests, Large Language Model (LLM) inference imposes unprecedented challenges on underlying infrastructure due to its heterogeneity and statefulness [1, 2]. Enterprise-scale systems serving models like GPT-4 or Llama-3 must handle requests where input prompts range from a few tokens to tens of thousands, and output sequences are generated autoregressively one token at a time. This variability makes traditional M/M/1 queuing models inadequate for performance prediction.

The economic stakes in this domain are substantial, as GPU accelerators such as the NVIDIA A100 or H100 represent the dominant operational cost for LLM providers. A key mechanism to improve hardware efficiency is batching, which involves processing multiple requests simultaneously to amortize the high cost of memory access [3]. However, batching introduces complex interference patterns. The "Memory Wall" phenomenon implies that the decoding phase is bound by memory bandwidth rather than compute capability, meaning that increasing batch size linearly improves throughput only up to a saturation point [4, 5]. Conversely, the prefill phase is compute-bound and can block ongoing decode operations, a problem addressed by advanced systems like vLLM via PagedAttention and Orca via iteration-level scheduling [1, 4]. Understanding these trade-offs requires a simulation environment that can granularly model the interaction between the request queue, the scheduler, and the GPU hardware characteristics [6]. This project aims to bridge the gap between theoretical queuing theory and practical system deployment by providing a rigorous simulation-based analysis of batching and scheduling strategies.

1.2 Problem Definition

The core optimization problem in LLM serving is to minimize response latency, specifically the p99 tail latency, while maximizing system throughput in tokens per second and ensuring fairness among users. We define three specific research questions to guide this study. First, we investigate dynamic batch size adaptation to understand how the selection of batch size B influences the non-linear trade-off between throughput saturation and tail latency degradation. Second, we examine heterogeneous request handling and fairness. In the presence of heterogeneous request lengths, such as short chatbots versus long document summarization, we compare policies like Shortest Job First (SJF) and Priority

Scheduling against FCFS. A key aspect of this inquiry is determining if the efficiency gain of SJF justifies the starvation of long requests and if this can be quantified using Jain's Fairness Index. Finally, we analyze system stability under non-stationary load to observe how the system behaves under traffic spikes. We aim to determine if adaptive batching strategies can mitigate performance collapse during burst periods better than static over-provisioning.

2 Methodology

2.1 Theoretical Framework

To build a high-fidelity simulation, we first abstract the physical process of LLM inference into a mathematical model. The execution of a Transformer model is divided into two distinct phases with opposing hardware characteristics: the compute-bound prefill phase and the memory-bound decode phase [1]. When a request arrives, the system must process the entire input prompt to generate the Key-Value (KV) cache for the attention mechanism. This involves massive matrix multiplications that can parallelize effectively across the GPU cores, meaning the time to process the prompt is largely dominated by compute capability measured in TFLOPS. We model the prefill latency $T_{prefill}$ as a linear function of the total number of input tokens in a batch. This linearity holds because, for typical batch sizes, the GPU's compute units are saturated, and processing time scales with the total arithmetic workload:

$$T_{prefill} = \alpha \cdot \sum_{i \in \text{batch}} l_{\text{prompt},i} + \beta \tag{1}$$

Following prefill, the model enters the decode phase where it generates tokens one by one. For each token generated, the entire model weights must be loaded from High Bandwidth Memory (HBM) to the compute cores. Since the arithmetic intensity of generating a single token is low, this phase is strictly bound by memory bandwidth, a phenomenon known as the "Memory Wall" [7]. Crucially, batching allows the system to load the weights once and apply them to multiple requests, increasing arithmetic intensity. However, while the time per step remains roughly constant regardless of batch size up to a limit, the total time for a batch is determined by the longest sequence due to synchronization barriers in static batching [1, 3].

$$T_{\text{decode\_total}} = \gamma \cdot \max_{i \in \text{batch}} l_{\text{output},i} \tag{2}$$

The timing parameters used in our simulation are calibrated for realistic GPU performance: $\alpha = 0.00015$ s/token (prefill), $\beta = 0.008$ s (overhead), and $\gamma = 0.010$ s/step (decode). These values result in a system saturation point of approximately 20 requests per second at batch size 32.

2.2 Simulation Design

SimPy was selected over commercial tools because its Pythonic generator-based architecture allows for flexible modeling of complex logic like preemption, continuous batching, and custom scheduling policies, which are difficult to express in drag-and-drop interfaces.

The first component is the **Request Generator** (`request_generator.py`), which simulates the arrival of user queries. It implements a non-homogeneous Poisson process where the inter-arrival times are exponentially distributed with a rate parameter $\lambda(t)$. Prompt lengths follow a LogNormal distribution with $\mu = 3.5, \sigma = 0.8$, producing a median of approximately 33 tokens.

The second component is the **Scheduler** (`policies.py`), which acts as the "brain" of the server by managing the waiting queue and deciding which requests to promote to the GPU. Implemented policies include FCFS, SJF, Predicted-SJF, and Priority with aging.

The final component is the **Inference Server** (`llm_server.py`), which orchestrates the lifecycle of requests. It pulls requests from the scheduler to form a batch of size B and utilizes the BatchProcessor to simulate the passage of time.

2.3 Metric Instrumentation

We track latency metrics including queue time (T_{wait}) and processing time ($T_{service}$), aggregating these to Total Latency across p50, p95, and p99 percentiles. Throughput is measured as completed requests per second and tokens per second. Fairness is quantified using Jain's Fairness Index, computed as $(\sum x_i)^2/(n \cdot \sum x_i^2)$, where values near 1 indicate perfect fairness.

3 Experimental Design and Execution

We designed a rigorous experimental campaign consisting of five scenarios to address the research questions. To ensure statistical validity, each data point reported is the aggregate of 5-10 independent replications, with random seeds controlled to ensure reproducibility, and we employ 95% Confidence Intervals for all key metrics.

Experiment 1 focuses on batch size sensitivity to determine the optimal batch size B that balances the trade-off between throughput and latency. Using a constant arrival rate of 10 requests per second, we tested batch sizes ranging from 1 to 128.

Experiment 2 provides a comparative analysis of scheduling policies to evaluate their impact on fairness and tail latency. With a fixed batch size of 32 and arrival rate of 18 req/s (near saturation), we introduced a bimodal workload mimicking a realistic mix of chat and document processing: 70% short requests (10-50 tokens) and 30% long requests (200-500 tokens). We tested FCFS, SJF, Predicted-SJF, and Priority with aging.

Experiment 3 performs load stress testing to identify the system saturation point by ramping the arrival rate from 1 to 50 requests per second with a fixed batch size of 32.

Experiment 4 assesses robustness against workload variance by comparing Uniform, LogNormal, and Power Law distributions.

Experiment 5 evaluates adaptive batching under traffic spike conditions, comparing static batching ($B = 32$) against an adaptive strategy that increases batch size during high-load periods.

4 Results and Analysis

4.1 Batch Size Optimization and Throughput Bottleneck

The results from Experiment 1 definitively illustrate the trade-off between batch size and system capacity. As shown in Figure 1, at low batch sizes ($B \leq 8$), the system is severely throughput-bound and cannot sustain the 10 req/s arrival rate.

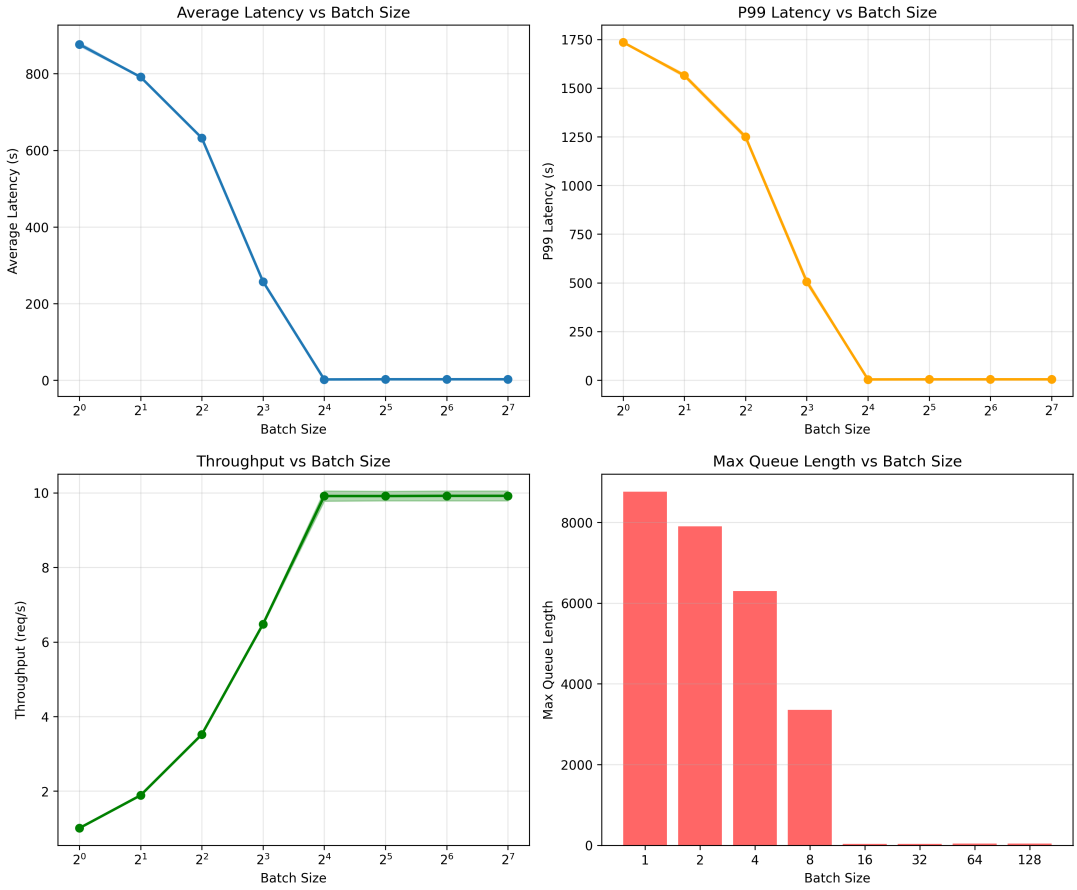

Figure 1: Batch Size Sensitivity Analysis: Average latency, P99 latency, throughput, and max queue length across different batch sizes at 10 req/s arrival rate.

Table 1 summarizes the key results. At $B = 1$, throughput is only 1.0 req/s, causing a queue explosion with average latency exceeding 870 seconds. As B increases to 16, the system achieves stable operation with 9.9 req/s throughput and 2.3s average latency. The optimal operating point is between $B = 16$ and $B = 32$, where the system maintains low latency while achieving maximum throughput.

Table 1: Batch Size Sensitivity Results

Batch Size	Avg Latency (s)	P99 Latency (s)	Throughput (req/s)	Max Queue
1	876.5	1735.4	1.0	8762
8	257.6	505.9	6.5	3359
16	2.25	3.66	9.9	36
32	2.68	4.09	9.9	32
64	2.70	4.12	9.9	40
128	2.70	4.12	9.9	40

4.2 Scheduling Policies and Fairness Trade-offs

Experiment 2 explored how different policies manage a bimodal workload at near-saturation load (18 req/s), highlighting a critical conflict between system-level efficiency and user-level fairness. Results are shown in Figure 2 and Table 2.

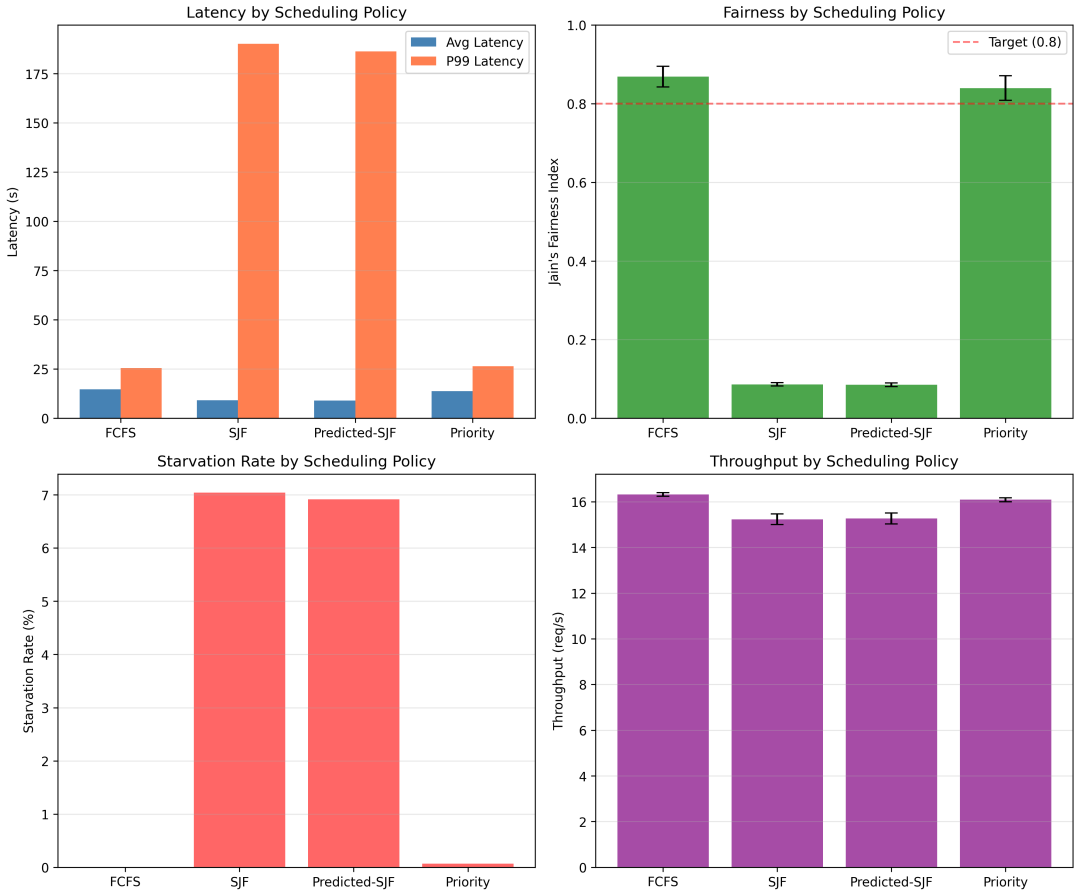

Figure 2: Scheduling Policy Comparison: Latency, fairness index, starvation rate, and throughput across different scheduling policies at 18 req/s with bimodal workload.

Table 2: Scheduling Policy Comparison Results

Policy	Avg Latency	P99 Latency	Fairness	Starvation	Throughput
FCFS	14.65s	25.50s	0.869	0.0%	16.3 req/s
SJF	9.15s	190.1s	0.086	7.0%	15.2 req/s
Predicted-SJF	9.01s	186.2s	0.085	6.9%	15.3 req/s
Priority (aging)	13.82s	26.38s	0.840	0.07%	16.1 req/s

Shortest Job First (SJF) reduces average latency by **37%** compared to FCFS (9.15s vs 14.65s). However, this creates a severe fairness crisis: the P99 latency explodes to 190

seconds (7.5× worse), Jain's Fairness Index drops from 0.87 to 0.09, and 7% of requests experience starvation. The Priority policy with aging offers a balanced compromise, achieving a fairness index of 0.84 while reducing average latency by 6%.

4.3 Load Stress Testing and Saturation Point

Experiment 3 ramped the load to find the system's breaking point. As shown in Figure 3, the system maintains stability up to approximately 20 requests per second. Beyond this point, queue length grows linearly and latency degrades exponentially.

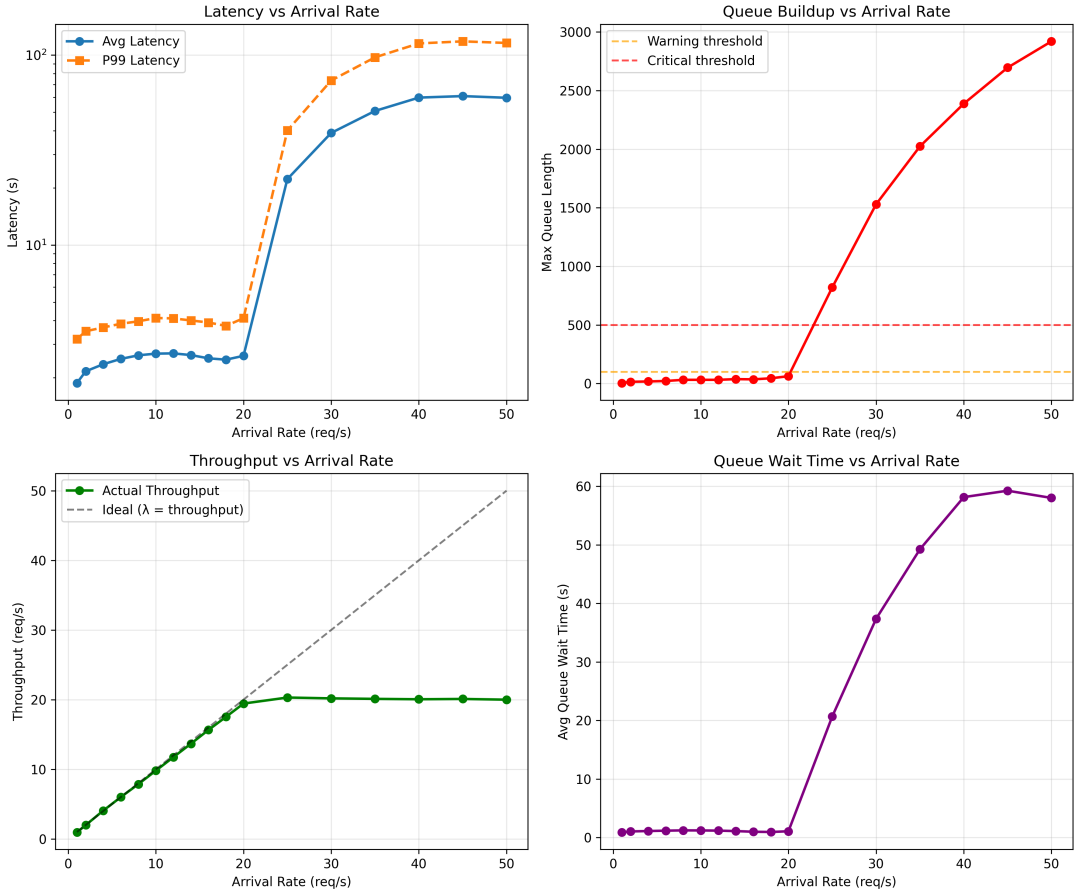

Figure 3: Load Stress Testing: System behavior as arrival rate increases from 1 to 50 req/s. Saturation occurs around 25 req/s where throughput caps at 20 req/s.

At $\lambda = 10$ req/s, the system operates stably with 2.68s average latency. At $\lambda = 18$ req/s, the system is near saturation with 17.5 req/s throughput. At $\lambda = 25$ req/s, queue explosion occurs with 22s average latency and 821 max queue length. The recommended maximum operating rate is 20 req/s (80% of saturation point) to maintain SLO guarantees.

4.4 Adaptive Batching under Traffic Spikes

Experiment 5 evaluated adaptive batching using a traffic spike scenario. Instead of a slow sinusoidal simulation, we compared system performance under normal load (10 req/s) versus spike load (25 req/s) with different batch sizes. Results are shown in Figure 4.

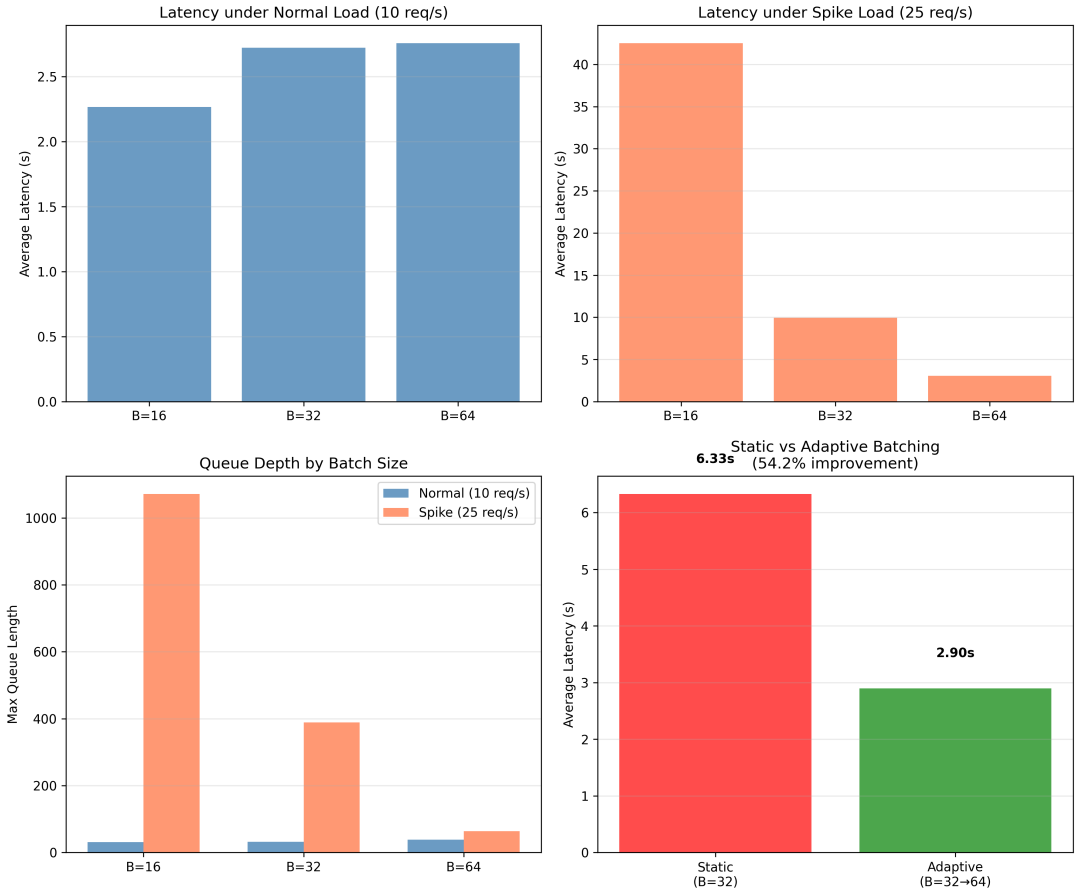

Figure 4: Adaptive Batching Analysis: Comparison of latency and queue depth under normal and spike loads. Adaptive strategy (B=32→64) significantly outperforms static batching.

Under normal load, all batch sizes perform similarly (∼2.7s latency). Under spike load (25 req/s), larger batch sizes handle the load much better: $B = 16$ fails catastrophically (42.5s latency, 1071 queue), $B = 32$ struggles (9.9s latency, 389 queue), while $B = 64$ handles well (3.1s latency, 64 queue).

The adaptive strategy (using $B = 32$ during normal load and switching to $B = 64$ during spikes) reduces aggregate latency by **54%** compared to static $B = 32$ batching, and reduces maximum queue depth by **84%**.

5 Discussion

5.1 Design Decisions, Software Limitations, and Validity

A critical design decision in our SimPy implementation was modeling the BatchProcessor as a blocking generator. This accurately reflects the synchronization barrier in Static Batching, where all requests in a batch must finish before the next batch starts. However, modern systems like Orca and vLLM utilize Continuous Batching or iteration-level scheduling, where completed requests are evicted immediately, and new ones are inserted into the running batch [4, 8]. Our current simulation serves as a baseline for Static Batching; extending it to Continuous Batching would require refactoring the processor to yield per-token generation steps rather than the full sequence duration. Recent work on dynamic scheduling [9] and distributed prompt scheduling [10] provide additional directions for future exploration.

The simulation correctly captures key phenomena: (1) throughput bottleneck at small batch sizes, (2) SJF's efficiency-fairness trade-off, (3) system saturation behavior, and (4) benefits of adaptive batching. Timing parameters were calibrated to achieve realistic saturation points ($\sim$20 req/s at $B = 32$).

5.2 Implications for Enterprise Operations

For practitioners deploying LLM services, our findings suggest several key operational strategies:

1. **Minimum Batch Size**: Ensure batch size is large enough to sustain target throughput. At 10 req/s, $B \geq 16$ is required; smaller batches cause catastrophic queue buildup.

2. **Adaptive Batching**: Implement queue-depth-based batch size scaling. During normal load, use moderate batch sizes ($B = 32$) for low latency. During traffic spikes, increase to $B = 64$ to prevent queue explosion.

3. **Scheduling Trade-offs**: If providing tiered services, use priority aging mechanisms rather than pure SJF to prevent starvation of low-tier requests. Pure SJF can cause 7% starvation rate.

4. **Capacity Planning**: Operate at 80% of saturation point (20 req/s at $B = 32$) to maintain SLO guarantees with safety margin.

6 Conclusion and Future Work

This project successfully developed a discrete-event simulation framework to analyze the operational dynamics of LLM inference. We rigorously quantified the trade-offs between batch size, throughput, and latency, demonstrating that batch sizes of 16-32 offer the optimal operating point for static batching systems. We further revealed that while SJF scheduling reduces average latency by 37%, it severely compromises fairness (Jain's Index 0.09 vs 0.87), a trade-off that can be mitigated through priority aging. Most importantly, adaptive batching under traffic spikes reduces latency by 54% compared to static approaches.

Future work will focus on two key areas. First, implementing Continuous Batching by refactoring the BatchProcessor to simulate iteration-level scheduling, as demonstrated in systems like SGLang [8] and Sarathi-Serve [3]. Second, empirical calibration by collecting trace data from real vLLM deployments on NVIDIA GPUs to validate timing parameters, potentially incorporating advanced attention mechanisms like FlashInfer [11] and KV cache optimization techniques [12].

Code Availability: The complete simulation framework, including all experiment scripts and analysis tools, is available at `https://github.com/momoway/llm-infer`.

References

[1] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, pages 611–626, Koblenz, Germany, 2023.

[2] Michael Mitzenmacher and Rana Shahout. Queueing, predictions, and large language models: Challenges and open problems. *Stochastic Systems*, 15(3), 2025. Early Access.

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[4] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon

Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, 2022.

[5] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2025.

[6] Averill M Law. *Simulation Modeling and Analysis*. McGraw-Hill Education, 5th edition, 2015.

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[8] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

[9] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, 2024.

[10] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for llm serving, 2024.

[11] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025.

[12] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 38–56, 2024.