# VIDUR: A LARGE-SCALE SIMULATION FRAMEWORK FOR LLM INFERENCE

**Amey Agrawal** [1 2] **Nitin Kedia** [3] **Jayashree Mohan** [3] **Ashish Panwar** [3] **Nipun Kwatra** [3]
**Bhargav S. Gulavani** [3] **Ramachandran Ramjee** [3] **Alexey Tumanov** [1]

## ABSTRACT

Optimizing the deployment of Large language models (LLMs) is expensive today since it requires experimentally running an application workload against an LLM implementation while exploring large configuration space formed by system knobs such as parallelization strategies, batching techniques, and scheduling policies. To address this challenge, we present Vidur – a large-scale, high-fidelity, easily-extensible simulation framework for LLM inference performance. Vidur models the performance of LLM operators using a combination of experimental profiling and predictive modeling, and evaluates the end-to-end inference performance for different workloads by estimating several metrics of interest such as latency and throughput. We validate the fidelity of Vidur on several LLMs and show that it estimates inference latency with less than 9% error across the range. Further, we present Vidur-Search, a configuration search tool that helps optimize LLM deployment. Vidur-Search uses Vidur to automatically identify the most cost-effective deployment configuration that meets application performance constraints. For example, Vidur-Search finds the best deployment configuration for LLaMA2-70B in one hour on a CPU machine, in contrast to a deployment-based exploration which would require 42K GPU hours – costing 218K dollars. Source code for Vidur is available at `https://github.com/microsoft/vidur`.

## 1 INTRODUCTION

Large language models (LLMs) can learn from and generate natural language texts on a massive scale. LLMs such as GPT-3/4 (Brown et al., 2020; Bubeck et al., 2023), LLaMA (Touvron et al., 2023a), and Phi (Li et al., 2023) have demonstrated impressive performance on various natural language processing (NLP) tasks. However, LLM inference – the process of using an LLM to produce natural language outputs based on some input – is expensive. For example, the cost of serving ChatGPT is estimated to be $694K per day (Patel & Ahmed, 2023).

An LLM inference provider faces several challenges in optimizing LLM deployment. First, the provider has to choose a model parallelization strategy such as the number of tensor parallel dimensions, number of pipeline stages, number of replicas, etc. Second, the operator has to choose between different scheduling algorithms (e.g., Orca (Yu et al., 2022), vLLM (Kwon et al., 2023), Sarathi-Serve (Agrawal et al., 2024)). Third, the provider has to determine several configuration parameters, such as maximum batch size (BS), wait time for batching, as well as algorithm specific parameters (e.g., chunk size in Sarathi, watermark fraction in vLLM) to satisfy the desired throughput and latency constraints. Finally, they have to generate representative workload traffic to test out each of their models on an experimental testbed with each of the different combinations above. *Systematically optimizing deployment of tens of models with hundreds of configuration options is expensive and impractical.*

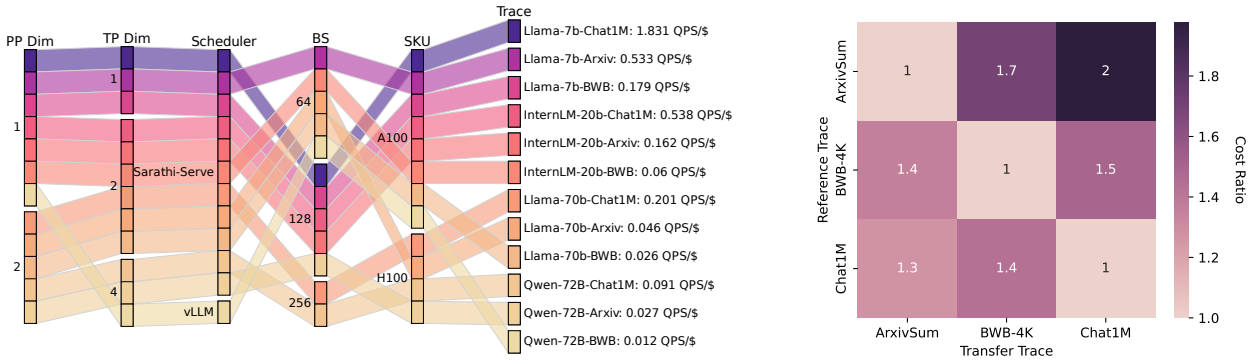
This cost is further exacerbated by our observation that optimal configuration is a function of a model-trace pair, i.e., optimal configuration also depends on application workload characteristics (Figure 1a). In fact, an optimal config obtained on one trace could be sub-optimal by a factor of up to 2× (Figure 1b) when applied to the same model on a different trace. With both new models and new traces being released almost daily, the cost of identifying the optimal deployment configuration becomes prohibitively expensive.

To tackle this challenge, we present Vidur – a large-scale, high-fidelity and extensible LLM inference performance simulator, and Vidur-Search – a configuration search tool. Together, they enable *fast* and *inexpensive* exploration of LLM inference performance under a variety of deployment scenarios.

Simulating LLM inference poses several unique challenges that are not addressed in prior work that simulate the performance of deep neural network (DNN) training (Zhu et al., 2020; Yu et al., 2021; Lin et al., 2022). First, LLM inference predictions have to be accurate at much finer time granu-

[1]Georgia Institute of Technology, USA. [2]Part of work done as an intern at Microsoft Research India. [3]Microsoft Research India. Correspondence to: Amey Agrawal <ameya-grawal@gatech.edu>.

*Proceedings of the $7^{th}$ MLSys Conference*, Santa Clara, CA, USA, 2024. 




(a) **Optimal configurations:** Color bands correspond to the optimal config for each of the 12 model-trace pairs with corresponding throughput achieved per dollar.

(b) **Cost of mis-configuration**: the optimal config on one trace used for another results in up to 2× cost difference (LLaMA2-70B).

*Figure 1.* **Both the model and workload matter for the optimal deployment configuration**. Optimal configurations for each model-trace pair is shown in (a). Throughput/cost can differ significantly for the same model if the workload is changed as shown in (b).

larity compared to training jobs where each iteration runs for hundreds of milliseconds. Second, unlike training where batch sizes are typically fixed, the input sizes during inference can vary drastically. The difference in input sizes stems from varying sequences lengths of different requests, as well as the interleaving of prefill and decode stages depending on the scheduling strategy, resulting in significant variations in iteration latency. Since it is infeasible to experimentally profile the performance of the model for all possible input sizes, the simulator has to rely on a mixture of careful profiling and a prediction strategy for unprofiled input sizes. Third, small errors in predictions lead to cascading effect due to the dynamic and stateful nature of inference workloads, thus inference simulators need to provide extremely accurate per-iteration predictions to get good fidelity at high request arrival rates.

**Vidur**. To address these challenges, Vidur uses the key insight that the large majority of LLMs share similar architectures that can be decomposed into a small set of *token-level, sequence-level and communication operators*. Thus, Vidur takes in a model specification and first identifies various operators and a minimal set of input sizes that need to be profiled experimentally. Vidur then builds a fine-grained runtime estimator that accurately predicts kernel performance on input sizes that might not have been profiled. Using the estimator, Vidur takes a specification of deployment configuration and workload, and predicts a variety of request-level metrics such as Time to First Token (TTFT), Time Between Tokens (TBT), latency, throughput, as well as cluster-level metrics such as Model Flops Utilization (MFU) and memory utilization.

We demonstrate the fidelity of Vidur across a range of models, hardware and cluster configurations. Vidur accurately predicts request-level LLM inference performance with under 9% error rate, and mimics overall cluster metrics for large-scale workloads and traces with high fidelity.

**Vidur-Bench**. We find that the workload has a considerable impact on output metrics of interest in LLM inference. For example, variations in the number of input tokens, number of decode tokens and batch size can impact performance dramatically (Agrawal et al., 2023). We observe that there is no standardized benchmark suite available today to comprehensively evaluate LLM inference performance. Thus, we introduce Vidur-Bench to address this gap. Vidur-Bench is an easily extensible collection of workload traces along with several existing batching and scheduling policies such as vLLM (Kwon et al., 2023), Orca (Yu et al., 2022), FasterTransformer (fas) and Sarathi-Serve (Agrawal et al., 2024).

**Vidur-Search**. Finally, we present Vidur-Search to help LLM inference providers optimize their deployment. Vidur-Search uses Vidur to automatically search over hundreds of deployment configurations to identify the highest throughput/cost configuration for a given model, workload pair. For example, for LLaMA2-70B, across a pool of A100 / H100 GPUs, Vidur-Search is able to identify the best configuration about one hour on a 96-core CPU cores that costs $9.93 per hour on Microsoft Azure, as opposed to an actual deployment-based exploration that would have taken 42K GPU hours, costing approximately $218K.

In summary, this paper makes the following contributions.

- Vidur: an LLM inference simulator that predicts key performance metrics of interest with high-fidelity (§4)
- Vidur-Bench: a benchmark suite comprising of various workload patterns, schedulers and serving frameworks, along with profiling information for popular hardware

like A100 and H100 GPUs (§5).
- Vidur-Search: a configuration search tool that helps optimize deployment by identifying the highest throughput per dollar configuration (§6).

## 2 Background and Motivation

### 2.1 Overview of LLMs

LLMs utilize the transformer architecture based on the self-attention mechanism (Vaswani et al., 2017) as their core building block. The self-attention mechanism helps a language model learn the relationship between different elements of an input sequence and subsequently produce the output sequence. An LLM consists of two dominant sub-modules, self-attention and multilayer perceptron (MLP). Various LLMs have been developed in recent years using a variation of these modules (e.g., GPTs, LLaMAs, Falcons). Primarily, these models differ only in terms of the embedding size, the number of transformer blocks, and the attention mechanism used by the model.

### 2.2 LLM Inference Efficiency Optimizations

LLM inference request processing consists of two distinct phases – prefill and decode. The prefill phase processes the entire user input prompt and produces the first output token. Subsequently, output tokens are generated one at a time in an autoregressive manner. During this decode phase, the token generated in the previous step is passed through the model to generate the next token until a special *end-of-sequence* token is generated at which point the request processing completes. The decode process requires access to the key and value activations of the previously processed tokens to perform the attention operation. To avoid repeated computation, contemporary LLM inference systems store them in *KV-Cache*.

Given the immense cost of LLM inference, LLM inference efficiency has become an active area of systems research. To this end, multiple optimization mechanisms have been proposed recently. Each of these techniques make different tradeoffs. For cost effective inference, right set of optimizations should be used be composed based on the specific application requirements. For example, Tensor Parallelism (TP) is a common strategy to parallelize LLM inference (Shoeybi et al., 2019; Pope et al., 2022). TP shards each layer across the participating GPUs by splitting the model weights and *KV-Cache* equally across GPU workers. TP (1) improves inference throughput with higher batch sizes, (2) lowers the latency of inference by splitting each operator across multiple GPUs. However, TP involves frequent blocking communication between workers, and thus requires expensive hardware with specialized high bandwidth interconnects like NVLINK. Alternatively, Pipeline Parallelism (PP) is another parallelization strategy in which the model is partitioned into stages of consecutive transformer blocks. Each GPU is responsible for computing a stage and output activations are transferred across GPU boundaries via send/recv operations. PP has a much more favorable compute-communication ratio compared to TP, but can suffer from pipeline bubbles (stalls due to imbalance between stages).

Recently, Agrawal et al. 2024 identified an inherent tradeoff in LLM inference scheduler design and proposed classification of existing LLM inference schedulers into two categories – prefill prioritizing (Yu et al., 2022; Kwon et al., 2023) and decode prioritizing (fas). Prefill prioritizing schedules achieve higher throughput, by generating schedules with higher batch sizes, but suffer higher latency cost. Decode prioritizing schedulers can achieve low latency but at the cost of lower throughput (Kwon et al., 2023). Sarathi-Serve (Agrawal et al., 2024) tries to mitigate this tradeoff by utilizing the computational slack in decode phase. Another set of recent works, Splitwise (Patel et al., 2023) and DistServe (Zhong et al., 2024) tackle this latency-throughput tradeoff by splitting the computation of prefill and decodes on separate devices.

**Takeaway:** *Various systems optimizations provide a rich cost-latency tradoff. The right techniques to use depend on the application requirements and hardware availability.*

### 2.3 LLM Inference Configuration Space

Control knobs like parallelism strategy, choice of scheduler, chunk size, batch size, SKU, etc. induce a large configuration space (Figure 1a) for LLM deployment. Furthermore, we make an important observation (Figure 1) that the optimal configuration (defined as a combination of specific choices for each control knob) is not simply a function of a specific model. But rather, the optimal configuration varies as a function of both the model $m$ and the trace $t$ evaluated on that model. Thus the complexity of configuration search is $O(|M| \cdot |T|)$, where $M$ is a set of all models of interest and $T$ is a set of workloads. With a rapid increase in both the number of models and downstream applications, the cost of optimal configuration search simply doesn't scale. And yet, misconfiguration is prohibitively expensive. For example, Figure 1b shows that using the optimal configuration of one trace can have up to $2\times$ cost differential on a different trace.

**Takeaway:** *There is no single best deployment configuration for a model – rather the choice of configuration should be made in a workload-aware fashion.*

With the cost of obtaining a single point in Figure 1 as high as \$97k, the high cost of misconfiguration, and the size of the search space growing with both models and traces, this begs a fundamental research question: *is it possible to*

*find a performant configuration without requiring access to expensive experimental resources at a fraction of the cost?* We explore this question in depth by proposing a simulation-based approach for LLM configuration search with Vidur, reducing the cost by several orders of magnitude.

## 3 Challenges in Simulating LLM Inference

State-of-the-art DNN simulation frameworks (Daydream (Zhu et al., 2020), Habitat (Yu et al., 2021) and Proteus (Duan et al., 2023)) focus on training jobs. Building a large-scale inference simulator, especially for LLMs, involves multiple challenges that are not addressed by the existing simulators. We enumerate them in detail below.

**Time Scale.** Conventional DNN training workloads are typically compute-bound workload where each iteration executes for 100s of milliseconds (Zhu et al., 2020). In comparison, LLM inference is a far more latency-sensitive task where iterations can be much shorter (a few milliseconds each) (Yu et al., 2022; Kwon et al., 2023). Therefore, simulating LLM inference requires predicting iteration times at a much finer granularity.

**Varying Iteration Times.** Compared to traditional DL workloads where each iteration performs the same amount of compute and has predictable minibatch latency (Xiao et al., 2018), latency of different iterations can vary significantly during LLM inference. The variation in inference runtimes come from multiple sources. First, LLM inference consists of different phases – prefill and decode, each with a different compute characteristic and runtime. Second, the requests being processed may have a large variation in their sequence length (due to varying prompt lengths or number of decode tokens generated), resulting in varying runtimes. Third, the batch size during online inference keeps varying depending on the system load and workload characteristics. Moreover, the composition of a batch can accommodate requests from both prefill and/or decode phases, again adding to the runtime variation.

**Cascading Errors.** In training workloads, the batch composition is uniform across all batches, and the execution of each batch is independent. However, during inference, requests arrive in the system dynamically, and if the runtime prediction of any batch has significant errors, that can change in the batching pattern. Thus small errors in individual batch predictions cascade over time and lead to aggregate errors.

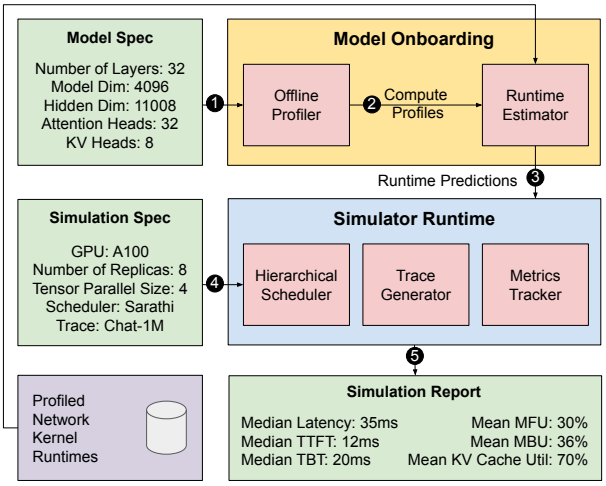



*Figure 2.* Vidur Simulator High Level Architecture.

## 4 Vidur

Vidur leverages domain knowledge to provide high-fidelity performance estimations of LLM inference. It emulates the behavior of all layers of the inference stack, including both the model execution and the various tiers of request scheduling, at both replica as well as the cluster level.

### 4.1 Key Insights

**LLMs Share Key Architectural Properties.** The large majority of LLMs share fundamentally similar architectures with small differences in the choice of activation functions, normalization layers, residual connections, etc. This allows us to use a common declarative model specification format that captures the essential architectural choices of various models. Another consequence of this architectural uniformity is that Vidur only needs to model a small number of compute operators that are shared across all model families.

**Operation Triaging for Runtime Prediction.** In a running batch, each request may be associated with varying numbers of *KV-Cache* and *query* tokens, leading to a vast combinatorial input space. Consequently, profiling every possible combination to predict operation runtimes is not feasible. Instead, we observe that LLM operators can be classified into different categories. For instance, execution time of some operations depend on the total context length of all the requests in the batch whereas for others, it depends only on the number of tokens in the current iteration. This classification allows us to design tailored runtime prediction strategies for each operator type.

For example, we observe that apart from the attention kernel, all other operations are independent of request history. During the decode phase, the MLP layer would take the same amount of compute irrespective of the number of input or

output tokens processed previously. Profiling the attention kernel requires modeling history of each request. However, since the attention operation during decode is largely a memory-bound operation (Dao et al., 2022; Agrawal et al., 2023), we find that it is sufficient to model the total amount of *KV-Cache* to be fetched in a batch of requests to determine the kernel runtime (§4.3).

**Automatic Profiling for Parallelism Strategies.** Each model parallel configuration has different memory, compute, and network communication characteristics. A naive profile and replay approach would require a separate profiling run for each parallelism configuration, which can be expensive. In contrast, Vidur incorporates the domain knowledge about LLM parallelism strategies, which allows it to identify the subset of computation that is performed on each device. During the profiling phase, we automatically identify the tensor sharding configurations for each operator from a declarative specification of the model. Consequently, Vidur can simulate various parallelization schemes with minimal profiling performed on a single GPU.

### 4.2 System Overview

Vidur primarily has two phases of processing. First is the model onboarding phase wherein the model specification is used to generate a set of compute operators to be profiled. The Vidur profiler (§4.3) collects the runtime characteristics for the identified operators and feeds them to the runtime estimator. To minimize the cost barrier of adding new models to the system, we collect minimal data during the profiling phase and then train small machine-learning models to generate predictions over a large range of parameters that these operation could be triggered on during simulation. This phase is handled by Vidur's runtime estimator (§4.4), which produces operation-wise runtime lookup tables that can be later used during simulation.

Once the model is onboarded, the user can perform simulations using various scheduling policies, and parallelism strategies, across a wide range of workloads supported by Vidur-Bench (§5). At the core of our event-driven simulator is a pluggable *Hierarchical Scheduler* (§4.5), which supports several popular batching strategies alongside memory planning and management capabilities. The simulator provides detailed metrics that capture both the request (normalized latency, time-to-first-token, time-between-tokens, etc.) and cluster (Model FLOPs utilization, *KV-Cache* utilization, etc.) performance metrics. The end-to-end process flow in Vidur is illustrated in Figure 2.

### 4.3 *Profiler*

To efficiently profile the runtime characteristics of LLMs, we leverage the insight that the large majority of LLMs share fundamentally similar architectures with small differences in the choice of activation functions, normalization layers, residual connections, etc.

**Operator Triaging.** The profiler analyzes different operators to identify their input dependencies. We find that all the operators can be placed on one of the three buckets:

- *Token-level Operators:* The operand dimensions for operations like linear, and activation functions depend on model architecture, however, their runtime only depends on the total number of tokens being processed (prefill plus decode) in the batch.
- *Sequence-level Operators:* The attention operation depends not only on the number of tokens in the current batch but also the context length of each request.
- *Communication Operators:* The runtime of communication operations like *all-reduce* and *all-gather* depend only on the amount of data to be transferred, independently of the model architecture.

**Profiling Token-level Operators.** There are two broad categories of token-level operators - matrix multiplications and simple point-wise apply or reduction operations, like addition, normalization, and activation functions. Based on the model specification, we generate all the different tensor parallel sharding configurations and profile each combination. This approach allows us to obtain traces for different parallelism configurations while profiling on a single GPU. We use standard PyTorch kernels for profiling these operations and measure their performance using CUPTI (cup).

**Profiling Sequence-level Operators.** Batching sequence-level operators such as the attention kernels is sensitive to the context length of the requests in the batch, thereby exploding the state space of inputs to profile.We use several techniques to address this problem. First, we separately profile the attention kernels for prefill and decode phases due to their difference in compute characteristics.

While processing the prefill attention, we observe that the attention time for each prefill is quadratic in its length. Suppose we have a batch of $P$ prefills of length $p_i$, where $i$ varies from 1 to $P$. The cost of prefill attention for the whole batch is therefore proportional to $\Sigma_{i=1}^{P} p_i^2$. To approximate the runtime of this batch we predict the runtime of an *equivalent* batch of a single prefill of length $\sqrt{\Sigma_{i=1}^{P} p_i^2}$.

In contrast to prefill, we notice that the attention decode operation is largely memory-bound (Dao et al., 2022; Agrawal et al., 2023). As a result, the runtime of this operation is mainly determined by the total data volume that needs to be fetched from the *KV-Cache* and not the exact split of context lengths between different requests in the batch. In practice, the attention kernel might not be able to effectively parallelize *KV-Cache* fetch operation when there is a large

skew between the context length of different requests in a batch. However, we observe that sequence parallel attention kernels such as PagedAttention v2 (Kwon et al., 2023), and FlashDecoding (Dao et al., 2023) can effectively handle such skews, and thus it is sufficient to model decode based on total *KV-Cache* reads.

**Profiling Communication Operators.** There are three collective operations that are frequently used in LLM inference, namely, *all-reduce*, *all-gather* (used for tensor parallelism) and *send-recv* (used for pipeline parallelism). Since these operations don't depend on model-specific characteristics, we independently profile these kernels ahead of time in a model-agnostic manner for different topologies.

### 4.4 *Runtime Estimator*

Collecting profiling data for every possible input combination across all the operators is prohibitively expensive. Therefore, we collect a limited set of data points and rely on small machine-learning models to interpolate the runtimes. *Runtime Estimator* first trains these models using the profiled data, and then generates runtime estimates for a large range of input tensor dimensions which it encounters in end-to-end simulation.

Prior DL training simulators (Yu et al., 2021; Lin et al., 2022) train Multi-layer Perceptron (MLP) models for opaque operations like matrix multiplications which are provided by closed-source third-party libraries like CUBLAS (NVIDIA Corporation, a) and cuDNN (Chetlur et al., 2014). However, training MLPs requires a large amount of data and results. On the other hand, simple polynomial regression does not capture the non-linear runtime characteristics of CUDA kernels due to phenomenons like tile and wave quantization (NVIDIA Corporation, b). For our scenario, we find that random forest (RF) regression models achieve the right balance between data frugality and fidelity.

### 4.5 *Hierarchical Scheduler*

In Vidur we adopt a three-tier hierarchical scheduler architecture, that provides a powerful and extensible interface. First is the global scheduler, that is responsible for request routing in Vidur. In addition to standard load balancing policies like round-robin and least outstanding requests, we also support stateful scheduling policies, where routing decisions can be deferred to a later point in time, which can be helpful under busty workloads where early binding routing decisions can hurt performance.

Second is the replica scheduler that encapsulates two key responsibilities; batching and memory management. The replica scheduler contains a memory planner, which uses the model specification and parallelism configuration to compute the memory available for *KV-Cache*. This information is then used by the memory manager to provide high-level management APIs that are used to implement custom batching policies. Vidur currently supports five batching policies, FasterTransformers (fas), Orca (Yu et al., 2022), Sarathi-Serve (Agrawal et al., 2024), vLLM (Kwon et al., 2023) and LightLLM (lig, 2023). The high-level API support provided by Vidur makes it extremely simple to implement new batching policies; all the aforementioned policies have been implemented each in less than 150 lines of Python code in our simulator

The final component of our scheduling stack is the replica stage scheduler, which handles the scheduling of micro-batches within a pipeline stage. While we currently only support synchronous pipeline parallel scheduling policy, in the future, we aim to extend the replica stage scheduler to emulate various optimizations like asynchronous communication, sequence parallelism (Li et al., 2021) and speculative pipelined decoding (Hooper et al., 2023).

## 5 VIDUR-BENCH

Vidur-Bench is a benchmark suite for easy evaluation performance evaluation of LLM inference systems that comprises of plug-and-play support for a variety of (a) workload patterns, (b) scheduling, batching, and routing policies, and (c) serving frameworks.

### 5.1 Datasets and workloads

The overall performance of LLM inference is highly sensitive to the type of workloads such as the number of input and output tokens in a given query e.g., the decode phase can be as high as $200\times$ more expensive than the prefill phase (Agrawal et al., 2023). Different workload patterns can therefore influence system performance in complex ways. For instance, vLLM incrementally allocates physical memory for the *KV-Cache* in order to fit a large batch size on the GPU. This works well when the number of decode tokens is high e.g., in chat applications (Zheng et al., 2023). In contrast, incremental memory allocation is less useful if the prompt length is much higher than the number of output tokens as in summarization tasks.

Vidur-Bench provides a set of workloads curated from publicly available datasets (see Table 1). These can be used to evaluate system performance for varying request types, arrival rates etc. or to tune the performance sensitive parameters of various components in the serving system.

### 5.2 Performance metrics

Vidur-Bench provides a comprehensive set of system-level performance metrics as discussed below:

**Operator-level metrics.** This includes each operator's input

| Dataset | Content | # queries | # prefill tokens | | | # decode tokens | | | P:D Ratio | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | mean | median | p90 | mean | median | p90 | median | std dev |
| LMSys-Chat-1M [ Zheng et al. 2023] | Natural language conversations | 2M | 786 | 417 | 1678 | 215 | 141 | 491 | 2.3 | 236 |
| (Chat-1M) | LMSys-Chat-1M with max 4k total tokens | 2M | 686 | 417 | 1678 | 197 | 139 | 484 | 2.3 | 228 |
| Arxiv-Summarization [ Cohan et al. 2018] | Summarization of arxiv papers | 203k | 9882 | 7827 | 18549 | 411 | 228 | 475 | 35.4 | 81 |
| (Arxiv-4K) | Arxiv-Summarization with max 4k total tokens | 28k | 2588 | 2730 | 3702 | 291 | 167 | 372 | 15.7 | 16 |
| Bilingual-Web-Book [ Jiang et al. 2023] | Document-level English–Chinese parallel dataset | 195k | 2418 | 2396 | 3441 | 3654 | 3589 | 5090 | 0.66 | 0.23 |
| (BWB-4K) | Bilingual-Web-Book with max 4k total tokens | 33k | 1067 | 1037 | 1453 | 1612 | 1601 | 2149 | 0.65 | 0.37 |

*Table 1.* Details of the workloads curated from open-source datasets.

size and execution time which can be used to identify and optimize the heavy-duty operators eg. *attn_prefill*, *mlp_up_proj* etc.

**Request-level metrics.** These include per-request metrics such as the scheduling delay, prefill completion time, time-to-first-token (TTFT), and time-between-tokens (TBT). Furthermore, any additional metrics of interest can be easily added, e.g., we added support to track how many times vLLM preempts or restarts each request when it runs out of GPU memory for *KV-Cache*.

**Replica-level metrics.** These include metrics such as the batch size, the number of tokens processed in each iteration, busy and idle times as well as the memory and compute utilization of each replica.

**Hardware metrics.** These capture cluster-wide GPU FLOPs and memory utilization. We plan to extend these to also capture the cluster's energy consumption.

## 6 VIDUR-SEARCH

When deploying an inference system, the system operator needs to take into account various aspects. For example, there may be SLOs on latency metrics such as TTFT and TBT or minimum QPS that needs to be supported. At the same time, the operator can try multiple configurations such as the GPU SKU (e.g. A100 vs H100) to use for deployment, the parallelization strategy (TP vs PP), scheduling policy (Orca, vLLM, Sarathi-Serve, etc.), replication degree, etc. Vidur-Search is a tool which helps find the optimal cost configurations to deploy an inference system while satisfying the desired SLO constraints. Vidur-Search leverages our simulator to compute the optimal configuration in an efficient manner. Along with the optimal configuration, Vidur-Search also gives detailed visualizations of how changes in configurations impact cost, TTFT, TBT, etc.

Vidur-Search has the following main components:

**Input.** The input to the search tool consists of the LLM model, the workload (request characteristics can significantly affect inference performance), available GPU SKUs, and maximum number of GPUs in a replica.

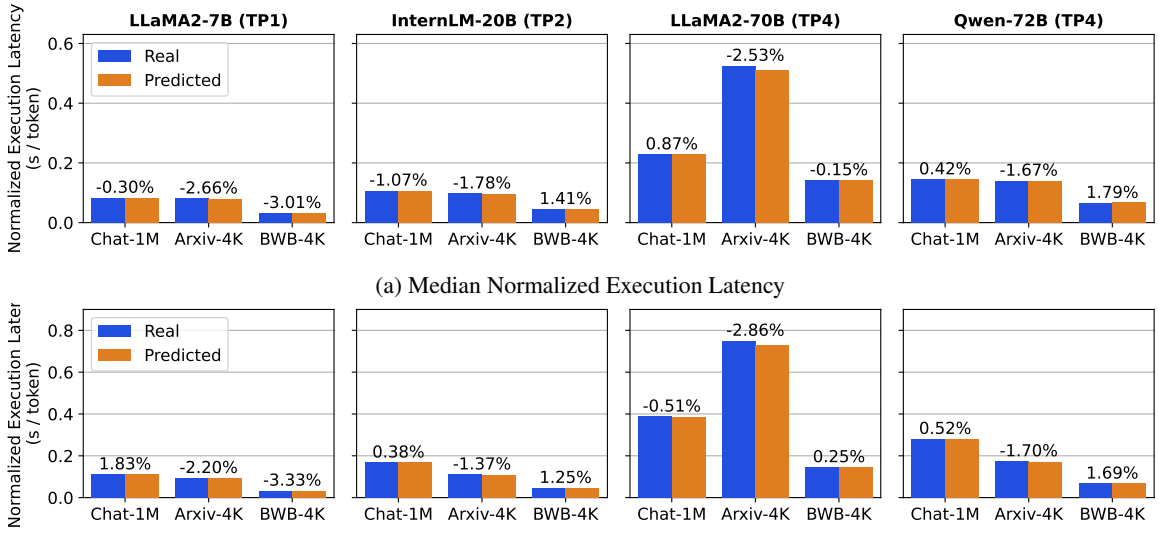
**Constraints.** SLOs on metrics such as TTFT and TBT.

**Search space.** The search tool has the freedom to configure the parallelism strategy (TP vs PP), parallelism degree, scheduling policy, scheduler specific parameters (e.g. chunk size in Sarathi), batch size, choice of GPU, SKU, etc.

**Optimization objective.** Vidur-Search helps the operator maximize QPS per dollar. Consider a deployment with 16 A100 GPUs. *Capacity* of the system is defined as the maximum queries per second that it can support without the queuing delay blowing up. Specifically we constrain the P99 scheduling delay to be under 5 seconds. This QPS value is divided by the cost of renting 16 A100 GPUs per hour to get the QPS per dollar value.

Given the above, Vidur-Search needs to solve a constrained optimization problem to find the optimal configuration in the search space. Vidur-Search starts with first enumerating all possible deployment configurations of the system. For each configuration, we can run our simulator on the input workload at a specified QPS and predict the metrics such as TTFT and TBT. Note, however, that the possible QPS values to pass to the simulator can be infinite. To get around this, we instead target to find the maximum QPS that a given configuration can support. We do this by tracking the scheduling delay of requests for a given configuration and QPS. Note that any system configuration will have a maximum QPS capacity for a given workload at which it can process the input requests without accumulating the request queue. We use this property to find the maximum QPS supported by a system via a simple binary search which searches for the maximum QPS which does not increase the scheduling delay beyond a threshold. Each step of this binary search involves running our simulator for the corresponding configuration and QPS. We parallelize these runs by running each search on a separate core. After this search, we have for each configuration, the maximum QPS which is supported by the system. Finally, Vidur-Search analyzes this data to output the optimal configuration and also generates visualizations of how changes in configurations impact the various metrics.

Since the number of configurations that need to be evaluated can be very large (in 1000s), doing a naïve search on actual hardware will be extremely costly. At the same time, a suboptimal choice of configuration can be very costly in the long run. Moreover, since the optimal configuration depends on the input workload, and the workload can change over time; it may be prudent to repeat this search whenever the




(a) Median Normalized Execution Latency




(b) P95 Normalized Execution Latency

*Figure 3.* Fidelity of Vidur's request execution time prediction for four models and three *static* traces.

workload characteristics have diverged from the original workload. The use of simulator in Vidur-Search makes this practical, by reducing this search cost by many orders of magnitude. We leverage Vidur-Search for our *what-if* analysis in §7.3.

Note that while Vidur-Search is primarily designed for configuration optimization of online serving systems, it can be repurposed for offline inference scenarios by changing the objective function from QPS per Dollar to an alternate objective like the makespan metric.

## 7 Evaluation

In this section, we demonstrate the fidelity and usefulness of Vidur across a wide range of models, hardware configurations and workloads. We perform all our evaluations on an optimized version of the vLLM codebase, with support for different scheduling policies and CUDA graphs, which eliminates unnecessary CPU overheads. Our evaluation seeks to answer the following questions:

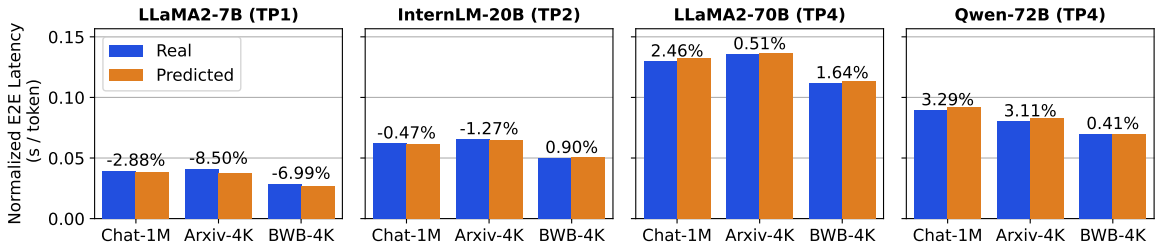
1. Can Vidur accurately predict the end-to-end performance metrics across models of different sizes, parallelization strategies and workload traces with varying request lengths and arrival patterns (§7.2)?
2. Can Vidur answer what-if questions related to LLM deployment challenges for a given hardware configuration (§7.3)?

### 7.1 Evaluation Setup

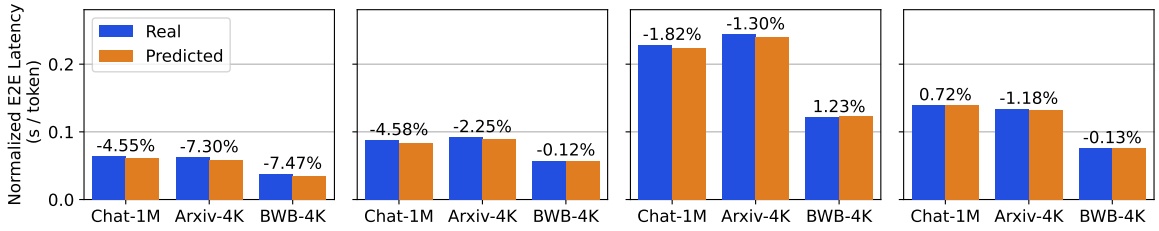
**Implementation.** As baseline, we use a fork of the open-source implementation of vLLM (Kwon et al., 2023; vLL). We extend the base vLLM codebase to support various scheduling policies, chunked prefills (Agrawal et al., 2024), and an extensive telemetry system.

**Models and Environment.** We evaluate Vidur across four models: LLaMA2 7/70B (Touvron et al., 2023b), InternLM-20B (Team, 2023), and Qwen-72B (Bai et al., 2023). We use Azure *Standard_NC96ads_A100_v4* VMs, each equipped with 4 NVIDIA 80GB A100 GPUs, connected with pairwise NVLink. Our H100 VMs have 4 NVIDIA H100s each with 80GB memory and connected with pairwise NVLink.

**Workloads.** In order to emulate the real-world serving scenarios, we generate traces by using the request length characteristics from LMSys-Chat-1M, Arxiv-Summarization and Bilingual-Web-Book. LMSys-Chat-1M contains one million real-world conversations with many state-of-the-art LLMs. A conversation may contain multiple rounds of interactions between the user and chatbot. Each such interaction round is performed as a separate request to the system. This multi-round nature leads to high relative variance in the prompt lengths. Arxiv-Summarization is a collection of scientific publications and their summaries (abstracts) on arXiv.org (arx). This dataset contains large prompts and lower variance in the number of output tokens, and is representative of LLM workloads such as Microsoft M365 Copilot (mic) and Google Duet AI (goo). Bilingual-Web-Book is a document-level Chinese–English parallel dataset. It consists of Chinese online novels across multiple genres and their corresponding English translations. The number of output tokens outweighs the number of prompt tokens in this dataset. This dataset also has a lower variance in number of prompt and decode tokens across requests. We restrict the




(a) Median normalized end-to-end latency.




(b) P95 normalized end-to-end latency.

*Figure 4.* Fidelity of Vidur's execution time predictions across four models and three *dynamic* workload traces, using request load at 85% of the maximum serving capacity for each scenario.

total request length to 4096 tokens based on the maximum context supported by the LLaMA2 family of models. We call these shortened traces Chat-1M, Arxiv-4K and BWB-4K respectively. Together, these traces represent varying workload characteristics, e.g., BWB-4K has 10× longer decodes and 2× longer prefills compared to Chat-1M; and a Prefill:Decode (P:D) ratio of 0.65 compared to 2.3. Further details for these workloads are present in Table 1.

### 7.2 Simulator Fidelity

In this section, we demonstrate Vidur's fidelity on end-to-end request-level predictions across the four models and three workloads detailed in §7.1. We use tensor parallel for InternLM-20B (TP2), LLaMA2-70B (TP4), and Qwen-72B (TP4). We use the default vLLM scheduler for all these experiments. We first evaluate Vidur using static (offline) workloads where all requests are assumed to have arrived before the system starts. We then evaluate Vidur using a dynamic (online) workload in which we assume requests arrive based on a Poisson distribution, with the arrival rate corresponding to the throughput of the system.

**Evaluation Metric.** For dynamic workloads, we compare the percentage error of Vidur predictions for normalized end-to-end latency, which captures the request's end-to-end latency divided by its output length (Yu et al., 2022; Kwon et al., 2023). We augment this metric slightly for static workload, and measure only the request execution time, excluding the scheduling delay – which would otherwise dominate the latency measurement. This allows us to perform more fine-grained analysis of Vidur's capability.

**Static Workloads.** We present the request latency fidelity evaluation in Figure 3. We observe that Vidur predicts even the tail latency (P95) with upto 3.33% error across the four models and three datasets. Note that we observe slightly higher average error rates for the 7B model, we attribute this to the higher CPU overhead for smaller models.

**Dynamic Workloads.** Next we present the evaluation of Vidur on dynamic workloads. In order to perform this evaluation, first we need to determine the request arrival rate at which we should perform this comparison. If the chosen arrival rate is too low, the system would have high idle time which is not an interesting scenario. On the other hand, if the request arrival rate is too high, the system would be overloaded where scheduling delay grows rapidly. Therefore, we evaluate Vidur's fidelity near the *capacity point*, which represents the maximum arrival rate the system can sustain without overloading (§6).

As shown in Figure 4, Vidur achieves high fidelity ($< 5\%$ error) in almost all scenarios with request rate set to 85% of the system capacity – which is reflective of real production scenarios. Note that, as we approach capacity point, any small deltas in prediction can lead to significant blow up of the errors. This is because at capacity, the system is at a tipping point – where even slight increase in the arrival rate or request processing time leads to a sharp increase in the request latency due to uncontrolled queue delays. If either the actual or simulated system runs into overload condition, the latency numbers become hard to reconcile due to large scheduling delay. However, production systems are provisioned with a buffer so that they don't tip over the critical

point due to sudden bursts. Since Vidur achieves high fidelity even at high arrival rates of up to 85% of capacity – making it valuable in QPS range of importance. We provide additional results at different arrival rates in Appendix A.

### 7.3 What-if Analysis

We leverage Vidur-Search for an extensive *what-if* analysis to understand how the performance of a configuration changes with the workload, and how the cost of serving is impacted by Service Level Objective (SLO) requirements.

**Inputs.** We find the optimal deployment configuration (one that maximizes QPS per dollar) for four models on three (dynamic) workloads described in §7.1. We allow choosing between the GPU SKUs of A100 and H100. The maximum number of GPUs available across replicas is set to 16.

**SLOs.** We put the following SLO constraints on the latency metrics: TTFT P90 $<$ 2s and TBT P99 $<$ 200ms. We use a more relaxed constraint of P90 for TTFT since it is a one time delay experienced by the user, as opposed to TBT which is recurrent for each output token.

**Deployment Configurations.** We experiment with TP and PP dimensions of 1, 2 and 4 for each, with three iteration-level schedulers vLLM, Orca+ and Sarathi-Serve that dynamically allocate memory for *KV-Cache* using paged attention. vLLM is a throughput-oriented scheduler that maximizes batch size by eagerly scheduling prefills while pausing on-going decodes. Orca+ is Orca (Yu et al., 2022) implemented over vLLM's paged attention. Sarathi-Serve creates hybrid batches with partial prefills to avoid pausing decodes while keeping GPU utilization high. We try these schedulers with batch size 32, 64, 128, 256 and 512. Note that the batch size gets divided by number of microbatches with PP. vLLM and Orca+ have a limit of maximum 4096 tokens per iteration while Sarathi-Serve has max 512, 1K and 2K tokens per iteration (also known as chunk size).

Figure 1a shows the optimal configuration for the three models for each of the workloads, and Figure 6 shows the QPS per dollar for the optimal configuration. We summarize the key takeaways below.

First, the *change in workload can drastically change the optimal configuration*. For example, for the LLama2-70B model, the optimal configuration for LMSys-Chat-1M uses batch size of 256, while for BWB it is 64. This is a consequence of the high *KV-Cache* load in BWB workload due to large decode sequences. Even the optimal GPU SKU changes from H100 for Chat-1M to A100 for BWB.

Second, even models with similar sizes can have very different performance characteristics due to variation in architectural details. For instance, LLaMA2-70B uses Group Query Attention (GQA), where as Qwen-72B employs Multi Head Attention (MHA) – which translates to 8$\times$ higher *KV-Cache* load. As a result, Qwen-72B is almost 2$\times$ more costly to serve and requires a different deployment configuration.

Finally, from Figure 6 it is clear that the capacity per dollar follows the expected trend. For example, larger models have lower capacity compared to smaller models. Also, Chat-1M has the least cost due to fewer prefill and decode tokens, while BWB has the highest cost due to larger number of tokens, especially decode tokens which are more expensive to compute compared to prefill. This complete exploration costs only 125 US dollars in simulation as opposed to actual execution which would have required 1.14 million dollars. We provide a detailed cost comparison in Table 2.

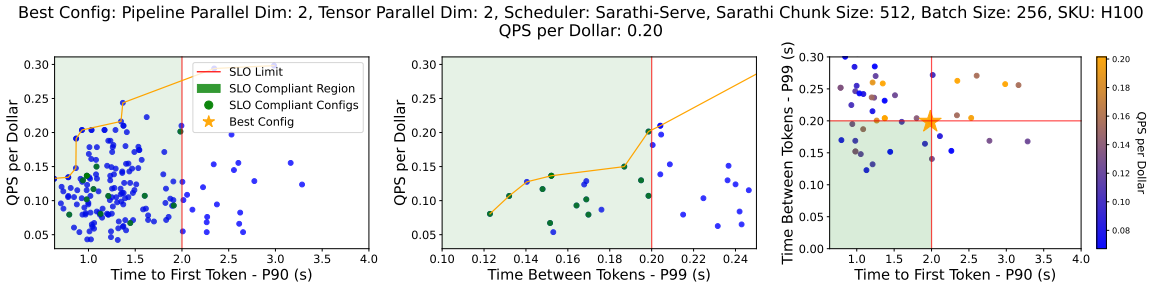
**Configuration Stability.** Figure 1b shows the overhead factor of using the optimal configuration for one workload, to serve a different workload on the LLaMA2-70B model. As shown, such a misconfiguration can result in a very high overhead, e.g., running LMSys-Chat-1M workload with the optimal configuration of Arxiv-Summarization-4K workload results in a 2$\times$ overhead! This shows that even for the same model, the cost of using a homogeneous deployment configuration can result in huge overheads, as the optimal configuration for one workload can be far from optimal for another workload.

**Pareto Frontier Analysis.** We next analyze the Pareto frontier produced by Vidur for LLaMA2-70B-LMSys-Chat-1M and Qwen-72B-Bilingual-Web-Book-4K workloads. Figure 5 shows the best QPS per dollar for different configurations and the corresponding TTFT-P90 (left), TBT-P99 metrics (middle) along with the SLO complient regions. The figures on the right plot both the latency metrics for these configuration, and visualize the QPS per dollar via a temperature colormap. We summarize the key takeaways.

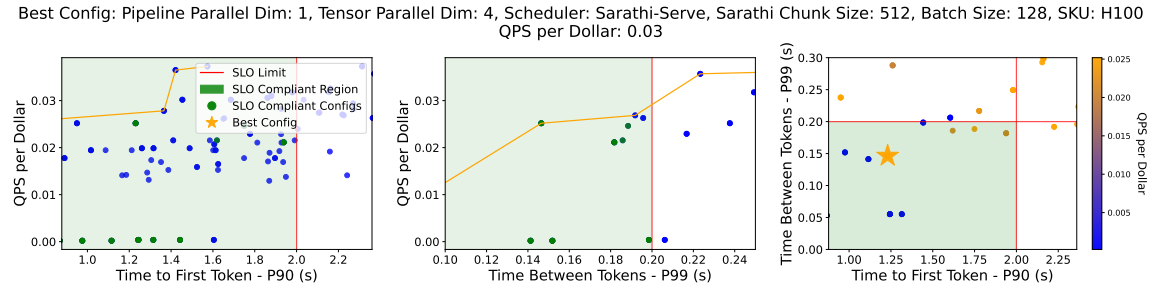
First, *configurations which are optimal on one metric may not satisfy the SLO constraint on the other metric* (these are the blue points on the Pareto curve). Second, *small changes in latency SLOs can result in a significant cost overhead*. For example, for the LLaMA2-70B-LMSys-Chat-1M workload, if the TBT SLO is changed from 0.12 seconds to 0.14 seconds (a difference of only 20ms), the Pareto curve point moves from approximately 0.07 to 0.13, $\sim$ 1.85$\times$ reduction in cost!

## 8 Related Work

Prior techniques leverage the predictability of DNN training iterations (Sivathanu et al., 2019; Xiao et al., 2018) to model the performance of the entire job. For example, Habitat (Yu et al., 2021) models the performance of a training job on different types of GPUs based on the runtime profile collected




(a) LLaMA2-70B– LMSys-Chat-1M




(b) Qwen-72B– Arxiv-4K

*Figure 5.* Capacity per dollar for different deployment configurations vs corresponding TTFT-P90 (left) and TBT-P99 (middle). Also show is the Pareto curve for these configurations. Shaded area corresponds to region where the corresponding SLO is satisfied. (right) Both latency metrics for these configuration, with capacity per dollar visualized via a temperature colormap. In the left and middle plots, green points correspond to configurations which satisfy SLOs for both metrics. Note that blue points on a Pareto curve show that, even Pareto curve points for one metric may not satisfy SLO for the other metric.

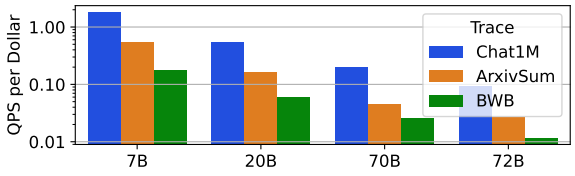



*Figure 6.* QPS per dollar for best configurations using P90 TTFT and P99 TBT SLOs of 2s and 200ms respectively.

of a few training iterations on a given GPU. In doing so, Habitat applies the roofline model (Williams et al., 2009) to estimate the performance of individual operators based on the compute and memory requirements of the operator along with the compute and memory bandwidth of a GPU. Daydream (Zhu et al., 2020) proposes a different approach focused on modeling the effect of various system optimizations on training performance across various deployment scenarios. Daydream can help answer questions like: what is the main performance bottleneck in my training job (e.g., memory or network bandwidth), how will optimizations like kernel-fusion, quantization or gradient compression help improve performance etc. To accurately model the effect of such optimizations, Daydream first constructs a computation graph of a training job and then applies optimizations via graph transformations (e.g., kernel-fusion can be applied by substituting individual kernel nodes with a single node that represents the fused kernels in the computation graph). Proteus (Duan et al., 2023) further enables simulating various parallelization strategies to identify the best partitioning and scheduling strategy for a given training job. It does so by first modeling a parallelization strategy with a unified representation called *Strategy Tree* and then compiling it into a distributed execution graph. In another approach (Lin et al., 2022), the authors propose a critical-path based strategy to predict the per-batch training time of deep learning recommendation models. Different from these training-based simulators, Vidur is the first simulator that accounts for the specific properties of LLM inference.

## 9 Conclusion

LLM inference efficiency depends on a large number of configuration knobs such as the type or degree of parallelism, scheduling strategy, GPU SKUs. It is impractical to run all possible configurations on actual hardware. In this paper, we present Vidur: a high fidelity and easily extensible simulator for LLM inference, along with a benchmark and search suite. Vidur answers deployment related what-if questions that identify efficient deployment strategies for production environments and helps in evaluating the efficacy of various systems optimizations at nominal cost.

## REFERENCES


arxiv.org e-print archive. https://arxiv.org/.

Cupti: Cuda toolkit documentation. https://docs.nvidia.com/cuda/cupti/index.html.

Faster Transformer. https://github.com/NVIDIA/FasterTransformer.

Google duet ai. https://workspace.google.com/solutions/ai/.

Microsoft copilot. https://www.microsoft.com/en-us/microsoft-copilot.

vllm: Easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm.

LightLLM: A python-based large language model inference and serving framework. https://github.com/ModelTC/lightllm, 2023.

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve. 2024.

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., Hui, B., Ji, L., Li, M., Lin, J., Lin, R., Liu, D., Liu, G., Lu, C., Lu, K., Ma, J., Men, R., Ren, X., Ren, X., Tan, C., Tan, S., Tu, J., Wang, P., Wang, S., Wang, W., Wu, S., Xu, B., Xu, J., Yang, A., Yang, H., Yang, J., Yang, S., Yao, Y., Yu, B., Yuan, H., Yuan, Z., Zhang, J., Zhang, X., Zhang, Y., Zhang, Z., Zhou, C., Zhou, J., Zhou, X., and Zhu, T. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T., and Zhang, Y. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning, 2014.

Cohan, A., Dernoncourt, F., Kim, D. S., Bui, T., Kim, S., Chang, W., and Goharian, N. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 615–621, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-2097. URL https://aclanthology.org/N18-2097.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

Dao, T., Haziza, D., Massa, F., and Sizov, G. Flash-decoding for long-context inference, 2023.

Duan, J., Li, X., Xu, P., Zhang, X., Yan, S., Liang, Y., and Lin, D. Proteus: Simulating the performance of distributed DNN training. *CoRR*, abs/2306.02267, 2023. doi: 10.48550/arXiv.2306.02267. URL https://doi.org/10.48550/arXiv.2306.02267.

Hooper, C., Kim, S., Mohammadzadeh, H., Genc, H., Keutzer, K., Gholami, A., and Shao, S. Speed: Speculative pipelined execution for efficient decoding, 2023.

Jiang, Y. E., Liu, T., Ma, S., Zhang, D., Cotterell, R., and Sachan, M. Discourse centric evaluation of machine translation with a densely annotated parallel corpus. In *Proceedings of the 2023 Conference of the Association for Computational Linguistics: Human Language Technologies*, pp. 1550–1565, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.main.111. URL https://aclanthology.org/2023.acl-main.111.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In Flinn, J., Seltzer, M. I., Druschel, P., Kaufmann, A., and Mace, J. (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Li, Y., Bubeck, S., Eldan, R., Giorno, A. D., Gunasekar, S., and Lee, Y. T. Textbooks are all you need ii: phi-1.5 technical report. September 2023. URL https://www.microsoft.com/en-us/research/publication/textbooks-are-all-you-need-ii-phi-1-5-technical-report/.

Li, Z., Zhuang, S., Guo, S., Zhuo, D., Zhang, H., Song, D., and Stoica, I. Terapipe: Token-level pipeline parallelism for training large-scale language models, 2021.

Lin, Z., Feng, L., Ardestani, E. K., Lee, J., Lundell, J., Kim, C., Kejariwal, A., and Owens, J. D. Building a performance model for deep learning recommendation model training on gpus. In *29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18-21, 2022*, pp. 48–58. IEEE, 2022. doi: 10.1109/HiPC56025.2022.00019. URL https://doi.org/10.1109/HiPC56025.2022.00019.

NVIDIA Corporation. CUBLAS library. https://docs.nvidia.com/cuda/cublas/index.html, a.

NVIDIA Corporation. Matrix multiplication background user's guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, b.

Patel, D. and Ahmed, A. The inference cost of search disruption – large language model cost analysis, 2023.

Patel, P., Choukse, E., Zhang, C., Goiri, Í., Shah, A., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference, 2022.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Sivathanu, M., Chugh, T., Singapuram, S. S., and Zhou, L. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pp. 909–923, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304072. URL https://doi.org/10.1145/3297858.3304072.

Team, I. Internlm: A multilingual language model with progressively enhanced capabilities, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023b.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Williams, S., Waterman, A., and Patterson, D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, apr 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., and Zhou, L. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 595–610, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/xiao.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Yu, G. X., Gao, Y., Golikov, P., and Pekhimenko, G. Habitat: A runtime-based computational per-

formance predictor for deep neural network training. In Calciu, I. and Kuenning, G. (eds.), *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pp. 503–521. USENIX Association, 2021. URL https://www.usenix.org/conference/atc21/presentation/yu.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E. P., Gonzalez, J. E., Stoica, I., and Zhang, H. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

Zhu, H., Phanishayee, A., and Pekhimenko, G. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In Gavrilovska, A. and Zadok, E. (eds.), *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pp. 337–352. USENIX Association, 2020. URL https://www.usenix.org/conference/atc20/presentation/zhu-hongyu.

# A APPENDIX

## A.1 Impact of Request Arrival Rate on Fidelity for Dynamic Workloads

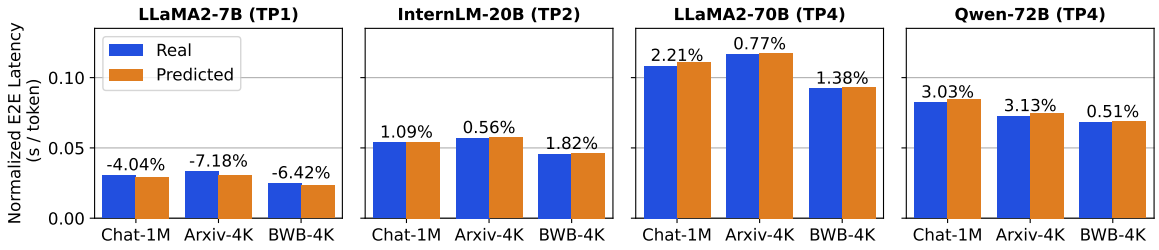
We present additional fidelity results for Vidur at different request arrival rates in Figure 7. We find that Vidur retains high fidelity even at 95% of maximum system capacity for larger models, however, for LLaMA2-7B, where we have slightly higher error due to CPU overheads, the errors cascade and we see up to 12.65% maximum error. We also provide the error trends in Figure 8.

## A.2 Cost Breakdown of What-if Analysis

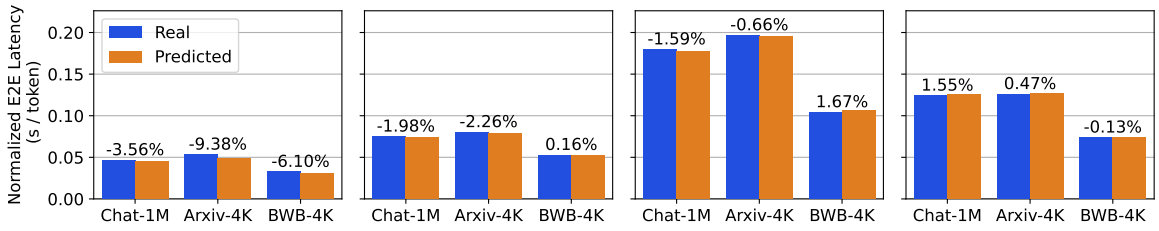
Vidur's ability to efficiently and accurate simulated complex deployment scenarios allows us to explore search configuration spaces at cheap negligible cost. The what-if analysis presented in Figure 1a, required a total of 35,565 runs, with a total projected GPU duration of 1,139,865 dollars. The same search completes takes only $\sim$ 12.5 hours on a 96-core CPU machine costing just $125. The breakdown of each task is presented in Table 2.

| Scenario | Time | | Cost($) | | |
|---|---|---|---|---|---|
| | Act | Sim | Act | Sim | Savings |
| 7B-Chat1M | 4K hrs | 31 min | 20K | 5 | 3837x |
| 7B-Arxiv | 10K hrs | 47 min | 52K | 8 | 6708x |
| 7B-BWB | 18K hrs | 136 min | 97K | 22 | 4324x |
| 20B-Chat1M | 6K hrs | 21 min | 33K | 3 | 9518x |
| 20B-Arxiv | 14K hrs | 25 min | 73K | 4 | 17746x |
| 20B-BWB | 16K hrs | 52 min | 84K | 9 | 9805x |
| 70B-Chat1M | 12K hrs | 21 min | 64K | 4 | 18151x |
| 70B-Arxiv | 15K hrs | 16 min | 78K | 3 | 30187x |
| 70B-BWB | 15K hrs | 27 min | 77K | 4 | 17333x |
| 72B-Chat1M | 14K hrs | 43 min | 76K | 7 | 10726x |
| 72B-Arxiv | 17K hrs | 16 min | 88K | 3 | 33354x |
| 72B-BWB | 18K hrs | 25 min | 93K | 4 | 22102x |

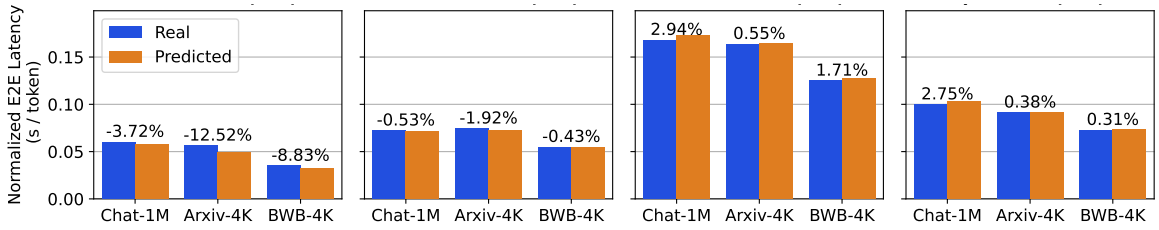
*Table 2.* Cost of finding the optimal deployment configuration.



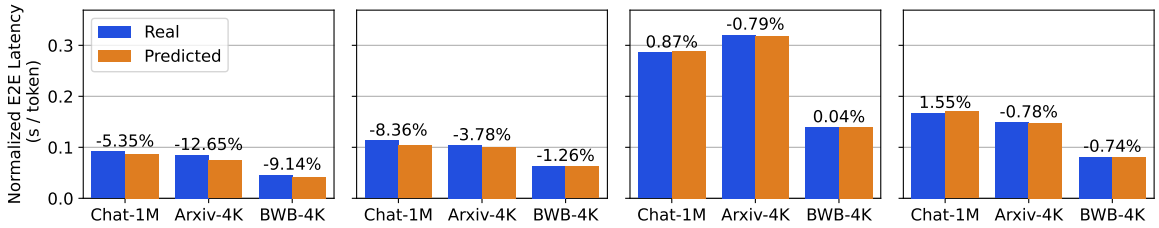

(a) Median normalized end-to-end latency at 75% of Maximum Capacity




(b) P95 normalized end-to-end latency at 75% of maximum capacity




(c) Median normalized end-to-end latency at 95% of maximum capacity




(d) P95 normalized end-to-end latency at 95% of Maximum Capacity

*Figure 7.* Fidelity of Vidur's execution time predictions across four models and three *dynamic* workload traces at different arrival rates.

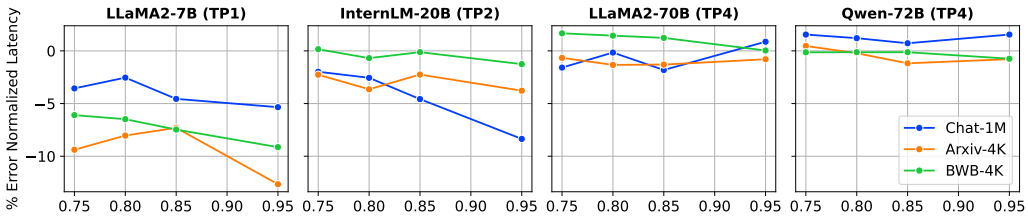



*Figure 8.* Prediction error for p95 normalized end-to-end latency at arrival rates between 0.75× and 0.95× of the maximum serving capacity.