Serving Models, Fast and Slow: Optimizing Heterogeneous LLM Inferencing Workloads at Scale

Shashwat Jaiswal[1]*, Kunal Jain[2]*, Yogesh Simmhan[3], Anjaly Parayil[2], Ankur Mallick[2], Rujia Wang[2], Renee St. Amant[2], Chetan Bansal[2], Victor Rühle[2], Anoop Kulkarni[2], Steve Kofsky[2] and Saravan Rajmohan[2]
[1]*University of Illinois Urbana-Champaign* [2]*Microsoft* [3]*Indian Institute of Science*

Abstract

Large Language Model (LLM) inference workloads handled by global cloud providers can include both latency-sensitive and insensitive tasks, creating a diverse range of Service Level Agreement (SLA) requirements. Managing these mixed workloads is challenging due to the complexity of the inference stack, which includes multiple LLMs, hardware configurations, and geographic distributions. Current optimization strategies often silo these tasks to ensure that SLAs are met for latency-sensitive tasks, but this leads to significant underutilization of expensive GPU resources despite the availability of spot and on-demand Virtual Machine (VM) provisioning. We propose SAGESERVE, a comprehensive LLM serving framework that employs adaptive control knobs at varying time scales, ensuring SLA compliance while maximizing the utilization of valuable GPU resources. Short-term optimizations include efficient request routing to data center regions, while long-term strategies involve scaling GPU VMs out/in and redeploying models to existing VMs to align with traffic patterns. These strategies are formulated as an optimization problem for resource allocation and solved using Integer Linear Programming (ILP). We perform empirical and simulation studies based on production workload traces with over 8M requests using four open-source models deployed across three regions. SAGESERVE achieves up to 25% savings in GPU-hours while maintaining tail latency and satisfying all SLOs, and it reduces the scaling overhead compared to baselines by up to 80%, confirming the effectiveness of our proposal. In terms of dollar cost, this can save cloud providers up to $2M over the course of a month.

1 Introduction

There is a push towards supporting intelligent features within enterprise products and services, and augmenting the behavior of clients through the use of Large Language Models (LLMs) and other ML/AI models. Such features may be unobtrusive and happening proactively in the background, or actively initiated by the users. There are rapid advances towards exploring the multitude of scenarios to which LLMs can be applied to and novel use-cases are being continuously developed. As their capabilities continue to grow, the use of LLMs for enterprise and consumer applications is increasing exponentially.

GPU-accelerated Virtual Machines (VMs) hosted on the cloud are at the vanguard of enabling inferencing over LLMs at massive scales. Cloud Service Providers (CSPs) and large Internet companies are investing heavily on GPU hardware, both for internal consumption by their products and services, and to lease them out as GPU VMs or as AI inferencing services. E.g., AWS UltraClusters offer 20,000 Nvidia H100 GPUs per region for training workloads [1], while Meta and Microsoft reportedly purchased 150,000 H100 GPUs in 2023 [2]. As the demand for these AI capabilities is growing, even governments across the world are expanding AI datacenters, such as the StarGate Project of the US government [21]. However, besides these substantial investments, it is equally critical to ensure these resources are utilized effectively to maximize return on investment. Inefficient utilization can arise when GPU provisioning across regions fails to align with actual traffic distribution. This can lead to missed SLOs when demand exceeds supply or resource wastage when supply exceeds demand. As most CSPs prioritize user experience due to business requirements, they typically over-provision GPU capacity which quickly escalates infrastructure costs. This can drive up the price of LLM services for end users and also divert resources away from other vital areas, such as research and development, potentially hindering long-term innovation.

While autoscaling CPU VMs is a long-studied problem in cloud computing [10, 29], scaling GPU VMs to meet LLM workload requirements comes with a unique set of challenges. Specifically, LLM inference requests across commercial LLM platforms like Gemini [3], Copilot [2], and ChatGPT [1] usually serve a mix of workloads that can broadly be categorized as interactive, non-interactive, and opportunistic. Interactive

*Equal contribution. Shashwat Jaiswal was an intern at Microsoft

[1]https://www.theregister.com/2024/01/15/cloud_providers_gpu_analysis/
[2]https://www.theregister.com/2023/11/27/server_shipments_fall_20_percent/

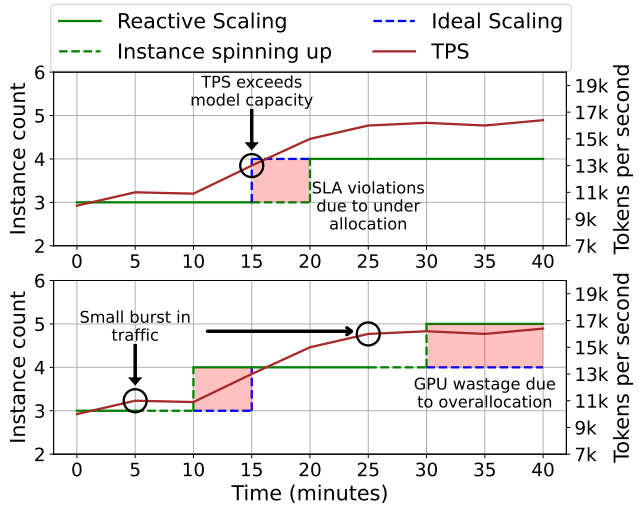

Figure 1: TPS based reactive scaling often under or over allocates hardware. The shaded region shows the difference in instance allocation between an ideal and a reactive scaling mechanism. *Blue curve:* An ideal scaling mechanism would follow the blue line and trigger scaling up at T=10 minutes, starting token processing by T=15 minutes. *Top:* A reactive scaling mechanism with correct TPS thresholds (4000 in this case) would trigger the scaling up at T=15 minutes, leading to SLA violations. *Bottom:* Lowering the threshold to 3500 TPS for the reactive mechanism would make it susceptible to bursts, leading to over allocation.

workload (IW) requests are latency-sensitive and require real-time responses, often within milliseconds or seconds, such as in chatbots, search query completions, real-time translations, and content moderation. In contrast, non-interactive workload (NIW) requests are less time-critical and focus on serving resource-intensive or batch processes such as large-scale content generation, data annotation, simulations, etc. at a higher throughput and lower cost. Finally, opportunistic workload (OW) requests are those guaranteed a lower tier of service such as those served by spot or pre-emptible instances [17,35]. They can be interactive or non-interactive but can be evicted to make way for higher priority requests. The exact mix of LLM workloads depends on several factors like time of day, region, tenant type (enterprise v/s consumer), etc. This wide variation in LLM workload characteristics and requirments makes it challenging to develop a unified autoscaling policy for all models and workloads.

Consider a reactive scaling mechanism which benchmarks the average Tokens Processed per Second (TPS) by a model instance and scales up/down based on the number of incoming tokens. As shown in Figure 1 over or under allocate instances due to variance in TPS and the problem is exasperated by the delays in spinning up LLM instances. When the model has a processing capability of 4000 TPS, the reactive mechanism would scale up the instance count at T=15 minutes, which will result in the instance being available at T=20 minutes, resulting in SLA violations due to under allocation for 5 minutes. If we keep the scaling threshold to 3500 incoming TPS to avoid under allocation, our system will become susceptible to over allocation, as illustrated in the bottom figure at T=25 minutes. Even though the incoming TPS becomes stable, reactive mechanisms would scale at T=25 minutes due to a small sudden burst in traffic. Scaling LLMs involves significant overhead due to the need to load large model weights (which can be in the order of gigabytes or more) into GPU memory. This process can block GPUs for several seconds to minutes, making frequent scaling costly. Thus, there is a critical need for a flexible and lightweight autoscaling policy that can seamlessly adapt to a varying mix of workloads, minimize delays associated with model loading and migration, and can reduce the overall infrastructure cost while meeting diverse workload requirements like low-latency SLAs, batch processing, and handling pre-emption.

We address these challenges with SAGESERVE. We start by examining a baseline siloed approach with separate GPU pools for Interactive Workload (IW) and Non-Interactive Workload (NIW) requests with Opportunistic Workloads (OW) requests of each type assigned to the corresponding pool if there is spare capacity. We empirically observe that this leads to significant underutilization of the IW pool in off-peak hours. To address this, SAGESERVE uses a unified pool with a reactive heuristic, routing NIW requests to GPUs with low IW load, saving GPU hours while maintaining SLAs. However, this approach uses a fixed number of instances of each model type which can lead to problems of demand-supply mismatch in settings with high fluctuation in traffic across each workload and LLM type. Therefore SAGESERVE optimizes GPU allocation by formulating a constrained optimization problem to dynamically scale model instances across regions based on ARIMA-based traffic forecasts, balancing short-term routing adjustments with long-term model scaling. This approach improves GPU utilization, meets SLAs, and allows surplus capacity to be rented for OW requests, reducing costs and enhancing overall efficiency. In summary, we make the following contributions:

1. We introduce the problem of designing LLM serving platforms for cloud service providers operating costly GPU VMs across regions, and receiving diverse workloads with variable SLA requirements. This motivates the need to holistically improve resource efficiency of GPU VMs by continuously maintaining the appropriate number of LLM model instances in various regions (section 3).
2. We present empirical studies on real-world workloads observed at a major cloud provider, Microsoft, that support the need for a systematic approach to instance scaling and highlight the pitfalls of siloed decision-making (section 4).
3. We propose SAGESERVE, a unified framework for serving requests while satisfying SLAs and maximizing capital ef-

ficiency by donating surplus instances to spot by defining this as an optimization problem, solved using ILP (section 5). SAGESERVE uses long-term aware reactive strategies that forecast the request arrival patterns, and optimize the scale out and in of GPU VMs and model instances to consider the token processing capability required for the interactive load and head-room for non-interactive ones (section 6). The overheads of dynamic provisioning are also considered.

4. We implement a prototype of SAGESERVE using a realistic simulation harness that extends from SplitWise [24]. We evaluate it on real-world workloads, for 3 regions and 4 LLM models handling over 10 million requests. We report results on the improvement in resource utilization and the scalability of the strategies across a week-long duration. We show a 25% reduction in GPU hours while processing a production trace with over 10 million requests over a week, reducing the scaling overhead by 80% and translating into savings of over $2M dollars over a month for cloud providers. We do this without compromising any SLO requirements and improving hardware utilization.
5. Finally, we plan to open-source our trace data and simulator. The harness can serve as a testbed for simulating LLM serving stacks that can be leveraged by the community, and encourage future research in this area.

2 Related Work

2.1 Autoscaling for cloud computing

There is a long history of forecast-based autoscaling of CPU VMs to serve cloud computing workloads [28, 33]. There are also works from commercial cloud owners on autoscaling [10, 29], and request routing [30] which illustrate the need for such mechanisms in all cloud services, the challenges associated with operating them at the cloud scale, and the enormous potential of cost reduction with these optimization. For e.g. in [10] the authors mention that even a 1% reduction in VM fragmentation can lead to $100M of cost savings in a year and we expect this number to be even higher for GPUs due to their higher cost. The main difference between our work and these prior works is that we consider autoscaling of LLMs on GPUs which encounters unique challenges due to the high latency sensitivity of interactive workloads, the significant variation in SLOs across workloads, and the high cost and latency of migrating and loading LLMs (hundreds of GB of data) on GPUs.

2.2 Efficient LLM Serving

Several recent works have focused on optimizing the latency or throughput of LLM requests at a single model instance. These include efficient computing approaches like PagedAttention [13], FlashAttention [6], and vAttention [26], as well as novel algorithms for batch processing of LLM requests like Orca [36], Sarathi [4], and Splitwise [25]. Additionally, [15] optimize Quality-of-Experience (QoE) for LLM serving systems, focusing on user-centric metrics to enhance individual experiences. However these works typically only consider a single LLM instance and do not take into account multiple model types and GPUs.

2.3 LLM Autoscaling

Since the launch of ChatGPT [1] and the advent of commercial LLM serving platforms, there have been some research works that have considered autoscaling LLM resources to meet workload requirments. [8, 14, 16] explored optimal placement of models on GPUs in a cluster by formulating different optimization problems. However, these optimizations are static and do not incroporate workload forecasts due to which they may not be as effective in handling the high fluctuations in workload traffic observed in commercial LLM serving platforms (Figure 4). There are also works that address specific challenges with LLM request serving such as handling preemption [17], addressing startup and migration delays due to large model sizes [7], and using a unified pool for serving online (IW) and offline (NIW) requests to reduce GPU fragmentation [27]. These works do not consider the general mix of workloads with different characteristics and models with different resource requirements as SAGESERVE does. Finally [11, 12, 20, 34] consider routing of LLM requests across different model instances with different goals like latency minimization and throughput maximization. While request routing is a crucial part of the LLM serving infrastructure for handling short-term fluctuations in traffic, it is important to combine it with dynamic model scaling as is done in SAGESERVE to handle longer-term traffic variations and prevent model assignments from becoming outdated.

3 System and Application Model

Our system and application models described here are motivated by real cloud systems, LLM deployments and workloads in Microsoft, a global public Cloud Service Provider (CSP). We will open source our trace data upon acceptance.

3.1 Cloud Regions and GPU VMs

Setup. Our system model for the CSP comprises of multiple data centers (*regions*) with captive GPU server capacity in each. To avoid issues of data sovereignty, we assume that these regions are in the United States (US), e.g., US-West, US-Central, etc. The regions are connected with high bandwidth network, with a typical network latency of $\approx 50ms$. Each region has servers with a mix of GPU generations that can be provisioned as GPU VMs with exclusive access to all underlying server resources, e.g., Azure's ND VM series that have

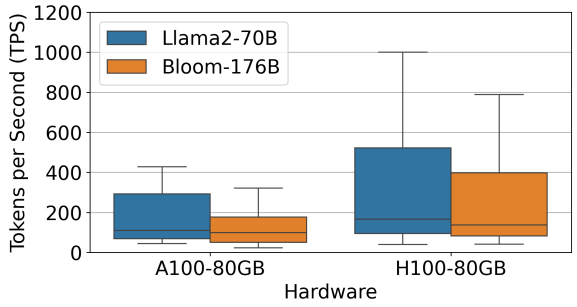

Figure 2: TPS seen for LLMs on VM with 8× GPUs each.

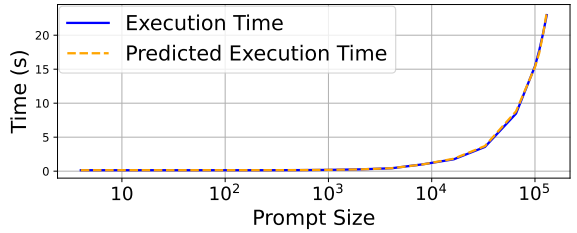

Figure 3: Comparison of batch execution time predicted by Splitwise vs real model instance.

8 Nvidia V100, A100 or H100 cards, or AWS's comparable P5/P4/P3 EC2 instances. Each region may have 1000s of such GPU VMs (hereafter referred to just as "VMs") that can be provisioned within the available capacity to host LLMs.

3.2 LLM Instances and Endpoints

LLM Architecture and Types. There are several standard pre-defined *LLM architectures* that are available, e.g., Llama 2, Bloom, GPT 3.5 turbo, etc. Further, each model architecture can either use default weights to give a standard behavior, or be fine-tuned for a specific application and have custom weights. The combination of model architecture and weights forms an *LLM type*.

Model Instance. A *model instance* is one instantiation of an *LLM model type* that can serve requests. Each instance may require one or more VMs, depending on the size of the LLM and capacity of the VM, e.g., a GPT3 model may require 9 H100 GPUs while a Llama-3 needs 4 H100s [16]. Each VM is exclusively used by one LLM instance. The VM type will determine the LLM instance's performance, defined in terms of TPS of throughput achieved at a certain target latency [16]. Figure 2 shows a boxplot of the TPS achieved on real deployments of the Llama2-70B (*Llama2*) [31] and Bloom-176B (*Bloom*) [5] models on VMs with 8× Nvidia A100 and H100 GPUs. Whiskers show the 5–95*%ile* range. The TPS drops with model size and improves on newer GPUs.

Instance Set. Each LLM type also has an *instance set* with a minimum and maximum range of model instances; having a minimum instance count provides redundancy in case of a VM failure. The VM type for all instances in the set are identical to ensure similar performance. Each instance set is exposed through an *endpoint* to receive incoming requests. There can be multiple LLM endpoints for the same LLM type, and they all have the identical semantic behavior.

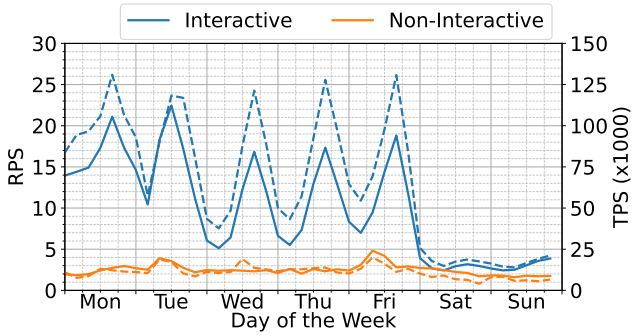

Figure 4: RPS *(solid line)* and TPS *(dashed line)* of IW and NIW requests summed across 4 LLM models and 3 cloud regions for 1 week in Nov, 2024.

3.3 Heterogeneous LLM Workloads

Workloads. The CSP needs to serve LLM inferencing requests to support several types of *production workloads*. One, is to support their own client-facing enterprise products that generate *Interactive Workloads* ***(IW)*** for specific LLM model types, with *low latency constraints (O(seconds))*, e.g., to produce and debug code snippets, generate email responses, chatbots, etc. These models require "fast" serving. Another is batch or *Non-Interactive Workloads* ***(NIW)*** with more *relaxed deadlines (O(hours))*, e.g., nightly summaries on documents in an enterprise repository, detailed content generation, etc. These batch requests expire if not completed within a relaxed deadline. So "slow" (or "no") serving is acceptable for NIW, in preference to IW.

Interactive Workloads. For IW, clients for each product/service may use one or more pre-defined LLM type, and this inferencing is initiated by the client while using the product. There can be 10,000s of clients during a day and we observe a diurnal pattern in the requests that are received. This is seen in Figure 4, which shows requests going to all models deployed in a US region, during one week in Nov, 2025. For simplicity, we assume all clients are US-based since the regions are in the US. These workloads have a *SLA latency limit* that ranges from a second to a minute for the *Time to First Token (TTFT)*, i.e., the time after the prompt request being received to the first response token being generated, for a certain percentile of request, e.g., 95%ile. Other quality of service factors include the end-to-end (E2E) time for the request to complete generating all output tokens, the number of input and output tokens that can range in the 1000s per request (Figure 4), etc.

Non-Interactive Workloads. NIW also uses a set of pre-

defined LLMs whose architectures often overlap with IW. The request rate for NIW, however, is lower and not periodic, staying stable through the week (Figure 4). Further, given their batch nature, their SLA is a *deadline* for completion before they expire, e.g., 24h to complete summarizing a document repository.

We assume that servicing each IW request within the latency SLA accrues a utility for the CSP, and servicing an NIW request before its deadline expires accrues a (lower) utility.

3.4 LLM Endpoint Provisioning and Routing

Scaling Delays. Creating a new endpoint for an LLM model by deploying an instance set to VMs has several *provisioning costs* that can vary based on the conditions. Allocating VMs to the instance set is an initial cost. Then, if these VMs do not have the LLM already deployed on them, the model architecture and weights need to be copied and instantiated on them. The time taken depends on the model size, and on whether they are available in a local repository in that region, e.g., taking ≈ 10*mins*, or need to be moved from a remote region, e.g., taking $\approx 2h$. If the VMs already have the LLM architecture deployed from a prior provisioning but with different weights, only the weights need to be updated and the cost reduces. When an instance set is being provisioned its instances are not available for use. So the model provisioning time constitutes *wasted GPU cycles*. Given that this time can run from mins–hours, frequent re-provisioning is inefficient. It should be noted that that in real world datacenter, acquiring a GPU, updating all upstream services such as load balancer, etc. would take time.

Spot Instances. The workloads are executed on LLM instances that are provisioned in a private network space. However, if the endpoints of common LLM models are idle, they can be leased to external users as (preemptible) *spot LLM instances* for inferencing at a lower cost, and reclaimed when the internal demand increases. Switching an instance from private to spot, and the reverse, is relatively fast, ≈ 1 mins. Typically, the utility benefits of leasing out spot instances is lower than that gained from executing the internal IW and NIW workloads. But this is still better than keeping the VMs idle. During some periods, 25% of instances in a region may be donated to spot; *this is a lost opportunity cost we aim to fix*.

Routing Mechanisms. All internal workload requests are directed to a common LLM API service [22] that redirects a request to one of several LLM endpoints that can service it (Figure 7). This *global routing* to one of a configured set of regions can be based on network latency, geographical proximity or the current load on the region's endpoints. Then, a *region router* sends requests to deployment endpoints for that model in that region, and further to instances within the selected deployment in a round robin manner to balance the load and address token skews.

We assume a managed network and trusted security environment. There are no other security constraints that limit the mapping of instances to VMs, or to internal or spot endpoints.

3.5 Simulating Datacenters

Experimenting with different scaling mechanisms and verifying their effectiveness with multiple model types and corresponding instances can prove quite costly. Therefore, to alleviate this pressure and enable easier research in this direction, we extend existing SOTA LLM simulator, Splitwise [24], to simulate a datacenter running multiple models on a variety of hardware.

We begin by independently verifying the accuracy of the Splitwise simulator (Figure 3), which reports $<$3% MAE on latest models and hardware. We further build upon this to launch multiple instances of these instance simulators and create a unified event queue for them, taking into account routing and iterations through the model. Different components of our simulator are presented in Figure 8.

Our simulator mimics multiple datacenters across the globe and is implemented at a granular level in order to make it flexible for different settings. Thus, it can be used for experimenting and evaluating new routing and batching algorithms as well, in order to study the impact of these changes across the entire inference serving stack. We will open source our work to give service providers holistic simulations and enable work on full stack optimizations.

4 Motivation: Effective Resource Scaling

We study the effect of scaling the resources allocated to an LLM type, and its effects on the capital efficiency and the SLAs of both IW and NIW workloads. As these VMs are captive and costly resources for the CSP, the goal is not to reduce resource usage but to put them to efficient use to increase the utility – preferably by processing IW and NIW workloads, or otherwise leasing them as spot instances.

The baseline approach uses a *siloed deployment*. Here, separate pools of LLM instances are maintained for IW and NIW workloads, for each LLM type in a region. Within a pool, instances may be allocated to internal workloads if there is sufficient demand, or released to spot instances otherwise. This uses a greedy approach for scaling, where we reclaim a spot instance and increase the instance set count for a model when the effective memory utilization (effective memory utilization excludes the memory used for model weights) for the instance set (which is a reliable proxy for the request load) increases above 70%, and returns an instance to spot if the utilization drops below 30%. These decisions happen on each request arrival. We always remain within the min/max instance counts. The downside of this is the fragmenting of VMs to captive pools, which can limit their effective utilization.

As an alternate, we propose a *reactive scaling heuristic* using a single *unified pool* of instances to serve both IW and NIW requests across all models in a region. Here, the NIW requests are queued and processed only when the endpoint's utilization falls below a certain threshold utilization by the IW requests (60%). One or two NIW requests are processed based on the effective utilization. If the value falls below 50%, two requests are added to the queue. This has the benefit of sharing the model instances between IW and NIW, and also allowing the pool of VMs to be used to deploy any of the LLMs, thereby improving their use for IW and/or NIW workloads rather than donate to spot.

While switching an LLM instance between spot and internal endpoint take 1*min*, switching a VM from hosting one instance to another takes 10*mins*. Note that all scaling events are triggered based on effective utilization, which is measured when a request reaches a regional endpoint. Additionally, we allow a cool down period of 15 seconds between any two scaling events.

We evaluate these siloed baseline and unified reacting scaling heuristic strategies for four LLM models: Bloom, Llama 2, Llama 3.1 and Llama 3.2, deployed in all the three US regions. There are 20 instances per model per region – all are part of a single pool when managed by the heuristic, and for the siloed approach, we assign 16 for IW and 4 for NIW. The minimum instance set count per endpoint is 2 and maximum is 3. We use a realistic simulation harness for SAGESERVE which is built on top of Splitwise [24], whose results closely match real-world behavior (see Section 7.1). Figure 2 shows the TPS distribution observed when simulating different models using SAGESERVE when using 8xA100s per instance. We replay one day of workload data from Tuesday of the week in Figure 4.

Figure 5 shows the instance count at US-Central every 15 mins by the siloed and unified approaches for each model, and the total model instance hours (area under the curve) for the day. It can seen that the unified heuristic consumes few instance hours for Bloom and Llama 2, while it is the same for Llama 3.1 and 3.2 since they received few requests and maintained the minimum instance count – siloed allocates 2 instances each for IW and NIW while unified shares the 2 instances for the IW and NIW requests. This consolidation is reflected in the higher memory utilization for the unified heuristic in Figure 6, while not sacrificing the SLA latency for IW. In both approaches, change in the 95%ile TTFT ranges from 0 to 12% (c.f., Table 1). Both approaches process all the queries in the trace (1.4M IW and 0.2M NIW), but unified is able to use fewer resources, and donate more to spot, donating 52 instance hours more than the siloed approach. Also, the memory utilization of Llama 2 is generally less than Bloom, indicating that using the unified pool to allocate across model instances (inter-model scaling) can further help adapt better to complementary demand compared to siloed that does not allow allocation of VMs across models.

Table 1: 95%ile of TTFT and end-to-end latencies for serving different models using siloed and unified approach.

Strategy	Model	TTFT (s)	E2E (s)
Siloed	Bloom-176B	14.5	55.3
Siloed	Llama2-70B	34.9	98.3
Siloed	Llama3.1-8B	1.0	10.6
Siloed	Llama3.2-3B	1.0	19.2
Unified	Bloom-176B	12.9	53.3
Unified	Llama2-70B	34.5	99.1
Unified	Llama3.1-8B	1.0	10.5
Unified	Llama3.2-3B	1.0	18.9

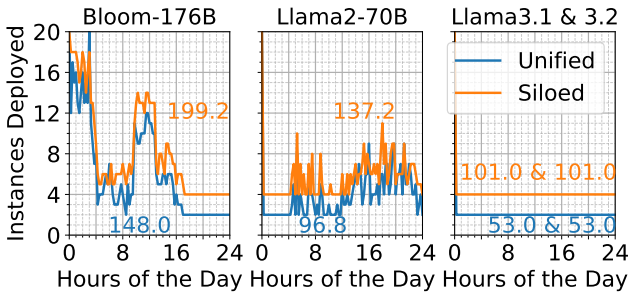

Figure 5: Model instance count across time and total instance-hours for unified vs. siloed strategies for 1-day of workload in US-Central, with peak 20 instances per model.

> Mixing IW and NIW workloads improves resource utilization and cost efficiency, opening new optimization avenues through flexible NIW processing (c.f. Table 1 and Figure 5). Moreover, reactive scaling can either harm IW SLOs due to insufficient resources or raise costs from overallocation (Figure 1). This motivates a proactive scaling approach that leverages predictable TPS (as shown in Fig. Figure 4) and workload mixing.

In the following sections, we first formulate this as an optimization problem and propose an ILP to find the optimal allocation of GPUs (subsection 5.1), describe the overall architecture of our resource management systems (section 6) and finally, we show the effectiveness of our system(section 7).

5 Optimization Problem

We solve the problem of request routing and capacity allocation to serve fast and slow LLM inferencing workloads within the required SLA while maximizing utilization.

Definition. *Given a captive set of VMs of specific types in multiple regions,*

- *we need to continuously ensure that the right number of model instances of different model types are provisioned as*

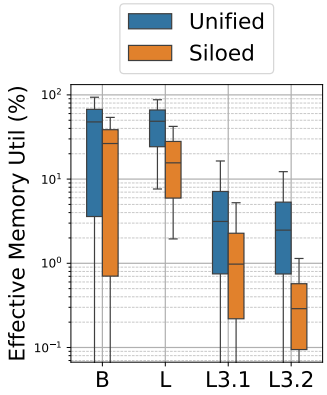

Figure 6: Memory usage by strategies, for Monday workload at US-West. Both the methods answered 1.4M IW and 0.4M NIW requests during the day. The unified approach used 52 instance hour less than the siloed approach.

endpoints at the regions, and

- *route the incoming workload across these endpoints, such that*
- *we maximize the utility of the workload requests completed within their SLA, and*
- *maximize the capacity utilization of the VMs for the internal workloads.*

We have two parts to solving this high level problem.

First, we need to **optimally provision instances** for model endpoints in different regions to handle this workload, within the available VM capacities. Since there is an overhead for (re)provisioning an LLM instance set onto VMs, we need to consider the inefficiency due to this process when the GPU VMs are not serving. Given this, such reprovisioning decisions are viable at a coarse granularity, e.g., every 15 mins – while some reprovisioning such as reclaiming spot instances can be fast, e.g., done each 1 min.

Second, we need to **route the requests** to the model instances in different regions while meeting the SLA. These routing decisions can use real-time information on the load and responsiveness of the region endpoints. As part of reprovisioning resources and routing, we need to use relevant tools [24] to estimate the expected latency for serving requests at a certain request rate to ensure we meet the SLA. Any spare capacity of model instances that are not being used for workloads can be released to external spot instances.

Next, we formulate these as an optimization problem.

5.1 Problem Formulation

Table 2 has the notations used in the problem definition.

Constraints. Say, the current number of VMs of GPU type k assigned to a model i at region j at a given time is $n_{i,j,k}$. Let $\delta_{i,j,k}$ be the optimal number of changes to be made to this VM count to service all IW requests in the next hour.

Table 2: Variables used in optimization problem

Symbol	Type	Description
l	`int`	Model types
r	`int`	Number of regions
g	`int`	GPU types
$n_{i,j,k}$	$[\texttt{int}]_{l\times r\times g}$	Instances of model i at region j on GPU type k
$\rho_{i,j,w}$	$[\texttt{int}]_{l\times r}$	TPS requested for model i from clients at region j for future time window w
$\theta_{i,k}$	$[\texttt{float}]_{l\times g}$	TPS provided by model i on GPU k
α_k	$[\texttt{float}]_g$	Cost of acquiring VM of GPU type k
$\sigma_{i,k}$	$[\texttt{float}]_{l\times g}$	Cost of starting a model i on VM k
$\delta_{i,j,k}$	$[\texttt{int}]_{l\times r\times g}$	ILP output: optimal number of changes in VM allocation

Servicing Interactive Workload within each Region. Say the forecasted rate of tokens to be served per second for an IW workload for model i at region j during each time window w over the next decision making window of, say, 1 hour, is $\rho_{i,j,w}$. Each model type in a region should be able to serve at least a fraction $0 < \varepsilon \leq 1$ of it's peak future request load received from clients within its region in real-time. Excess load $(1-\varepsilon)$ can be rerouted to other regions to reduce the number of model-instance changes needed.

$$\sum_k (n_{i,j,k} + \delta_{i,j,k}) \times \theta_{i,k} \geq \max_w \ \varepsilon \times \rho_{i,j,w} \ \forall i, j$$

Servicing Interactive Workloads across all Region. Next, we ensure that all IW requests for each model i received from across all regions j can cumulatively be processed using its model instances present across all region, with rerouting.

$$\sum_j \sum_k (n_{i,j,k} + \delta_{i,j,k}) \times \theta_{i,k} \geq \max_w \sum_j \rho_{i,j,w} \ \forall i$$

Non-negative number of instances. We should never deallocate more models than are allocated, $\delta_{i,j} \geq -n_{i,j}$.

Objective. We minimize the wasted resource overheads when provisioning VMs and instances required to support IW workloads while maintaining their SLAs. We incur two overheads when provisioning a new model instance: (i) VM start up cost to instantiate a new VM (γ), and (ii) the deployment cost (μ) when loading the model and its weights on a VM, when we have acquired and are paying for the VM but are unable to use it yet to serve requests. This makes acquiring new VMs less attractive than spot instances.

$$\gamma = \sum_k \left(\alpha_k \times \sum_{i,j} \delta_{i,j,k} \right)$$

$$\mu = \sum_k \sum_i \sum_j \left(\sigma_{i,k} \times \max(0, \delta_{i,j,k}) \right)$$

The objective is $\arg\min(\gamma + \mu)$

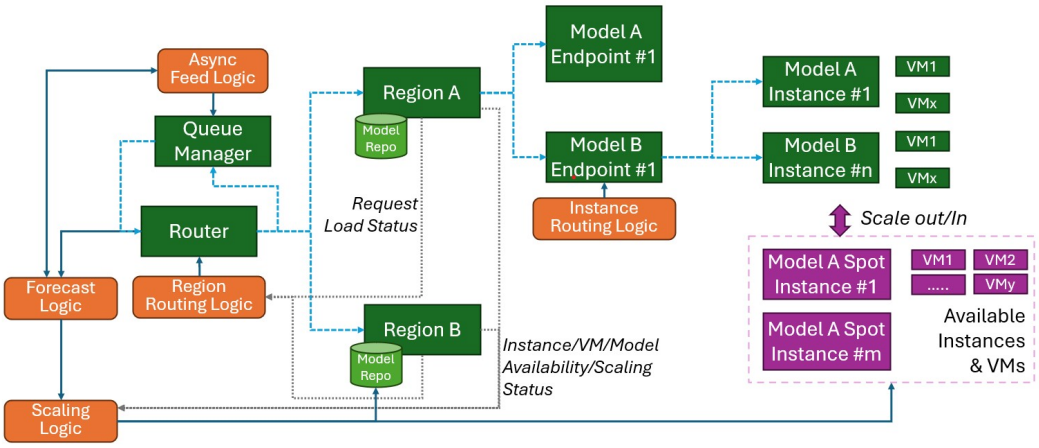

Figure 7: Architecture of SAGESERVE framework. Client requests are received at the global router. IW requests are routed to and served by instances at various regions. NIW requests are held by the queue manager and introduced into the router selectively. The forecasting module runs the optimization to scale instances.

6 Architecture of SAGESERVE

We next describe the approach and design of our proposed SAGESERVE framework solves the proposed problem.

6.1 Overview

Figure 7 shows the SAGESERVE LLM serving architecture operating across regions. At the frontend, SAGESERVE provides serving APIs similar to other LLM serving platforms. For IW requests, it uses a real-time streaming API that returns outputs once each token is generated [18, 23]. For NIW requests, it adopts an interface similar to Batch APIs [19, 22] which takes requests and returns responses asynchronously. The global router receives a request and routes it to the relevant region hosting the LLM instances which can service the request (§ 6.2). Other components orchestrate the forecasting and optimization logic (§ 6.4), scaling logic (§ 6.5 and NIW queue manager (§ 6.3). These are discussed next.

6.2 Routing Logic

The routing module includes the routing to regions by the global router, routing to endpoints within a region, and routing locally among the instances of an endpoint.

Global routing for IW requests. When IW requests are received at the global router, the region routing logic checks the effective memory utilization of all the available regions hosting the model and routes the request to the region with memory utilization less than the desired threshold (70%). If multiple regions match, we can specify an order of preference, e.g., based on network proximity. The effective memory utilization is calculated as the ratio of the sum of the effective memory utilized to the effective memory available across all instances for a model in a region. The effective available memory for a model instance is obtained by subtracting model weights from the total VM memory. If none of the preferred regions has memory utilization less than the threshold, then the region with the least memory utilization among the choices is selected.

Global routing for NIW requests. NIW requests are sent by the global router to a Queue manager that holds these requests (Figure 7). Each model endpoint at the regions periodically send a signal to the Queue manager when their effective utilization falls below a specific threshold (60%, in our experiments). Then, the NIW requests waiting in the queue for that region and model are incrementally removed by the Queue Manager and routed to that available endpoint, as described in Section 6.3.

Routing logic at region endpoint. For both IW and NIW requests, we route requests arriving at a region to the least loaded deployment endpoint for that model, based on the effective memory utilized. Requests are sent to the instance with the minimum remaining tokens to proces, based the Join The Shortest Queue Logic [9].

Scheduler at the model instance. A local queue is maintained at the instance as well. IW requests arriving at an instance are assigned priority 0 and available immediately for inference. The scheduler at the model instance selects requests and batches them for inference in a first-come, first-served manner. NIW requests may also arrive with a priority 0, as set by the Queue Manager, and are considered on par with IW requests. NIW requests with the default priority 1 are selected for inference if there are no priority 0 requests waiting at the instance queue. NIW requests are incrementally added to the queue.

6.3 Queue Manager for NIW

The Queue Manager take care of asynchrnously routing NIW requests to a specific region and endpoint. Upon receiving a signal from an model endpoint on its capacity availability, an asynchronous feedback logic initiates several steps. It pulls the requests that are queued at the Queue Manager for that model type and routes them to the region from which the signal was received. All NIW requests whose age is less than 10 hours are given a priority of 1, while requests that have an age greater than 10 hours are given a priority 0, similar to IW requests. If the signal received indicates that the effective memory at the regional endpoint is less than 60%, one request is removed from the Queue Manager added sent to the endpoint; if it is less than 50%, two requests are sent.

6.4 Forecast Logic and Optimization Module

For IW requests, we forecast the rate of token generation requests that will be received per region per model type using an ARIMA model [32], which we find to be accurate enough to predict the diurnal load for each model from a region. From the forecasting module, we take the maximum TPS expected in the next hour to estimate the TPS capacity requred for an IW model. We add a buffer of β to this forecasted TPS to handle transient bursty workloads and to offer capacity to serve NIW requests. We calculate the buffer as $\beta = 10\%$ of the NIW load we received in the past hour. The output of the forecast module is passed to the optimization module, uses an Integer Linear Programming (ILP) solver to solve the optimization problem in Section 5.1. This returns $\delta_{i,j,k}$, which is the change in the number of instances assigned to model i at region j running on GPU VM of type k. The forecast and optimization module is invoked every hour.

6.5 Scaling Logic

The long-term scaling logic uses the hourly output from the forecasting module, which solves the optimization problem using ILP. If the recommendation, $\delta_{i,j,k}$, for a specific region, model type and GPU VM type is positive, we need to scale out the instances; and if the prediction is negative, we need to scale in. For scale out, we first reclaim spot instances of the identical model type which are faster to acquire, and if none are available, we reclaim spot instances from other model types which can be slower to provision. Similarly, for scaling down, we donate the instances to the spot instances of the same model type.

6.5.1 When to initiate scaling?

Immediate (LT-I). One naive approach is to immediately scale instantly to the instance count recommended by the long-term scaler every hour. We term this as Immediate (LT-I). However, the recommendation is based on the peak load that will occur in the next 1 hour. So scaling out immediately can cause transient over-provisioning, well before the peak load actually arrives.

We introduce two additional systematic scaling strategies to pace the rate at which the deployment state catches up with the forecasted load. These approaches help improve the utilization of VMs and utility serving requests, making spare capacity available to other models that may require it in the same region.

Instance Utilization (LT-U). If the scaler suggests scaling out the instances, we do so only when the effective memory utilization actually starts increasing and goes over the threshold of 70% used in out experiments. We keep increasing the instance count as long as this threshold is breached and until we achieve the instances count suggested by the optimization model. Similarly, if the scaling logic suggests downscaling, we do so when the utilization goes below the 30% threshold, again until we have reduced to the suggested number of instances. The region endpoint provides the effective memory usage every time it receives a new request.

Instance utilization and ARIMA gap (LT-UA): Our optimization model can make erroneous recommendations if the ARIMA predictions are inaccurate, which is bound happen if bursty requests occur often. As before for LT-U, we defer the scale out or in based on the memory usage thresholds actually being breached in either direction. However, we do not strictly stop the scale out or in if the instance count reaches the target count. Instead, during the last 20 minutes of the hour, if we have reached the scale out instance count and the observed TPS load is $\geq 5\times$ of what ARIMA predicted, then we continue to scale up the instance count. On the other hand, if the TPS load received is $\leq 0.5\times$ of what the ARIMA prediction, we continue scaling down. Here, we switch from a memory-based strategy to a traffic-based strategy during the last 20 minutes of the decision window if the memory-based strategy has reached its goal.

7 Evaluation

The evaluation aims to answer the following questions:

1. How effectively can the proposed approach (SAGESERVE) utilize GPU resources while maintaining latency targets for both interactive and non-interactive workloads? (§ 7.3.1)
2. Can the proposed approach (SAGESERVE) respond quickly and maintain latency targets for both interactive and non-interactive workloads in the face of unexpected load bursts and prediction errors? (§7.3.4)
3. How does the proposed framework (SAGESERVE) scale when tested using a one-week trace, taking into account diurnal patterns and differences between weekday and weekend workloads? (§ 7.3.5)

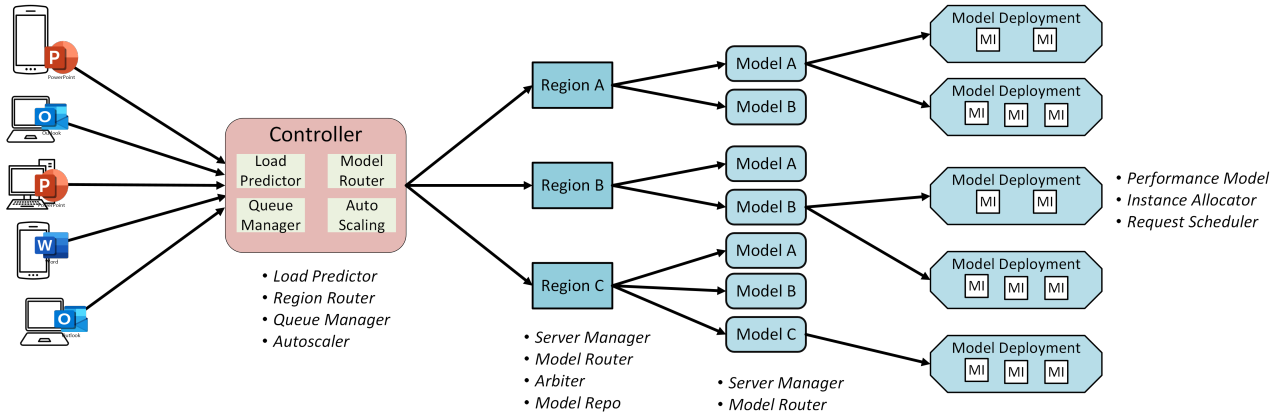

Figure 8: Overview of the SAGESERVE simulator

7.1 Implementation and Setup

To evaluate our proposal, we implemented a simulator that replicates the LLM serving stack on top of vLLM [13] and Splitwise [24]. The overall design of the simulator is shown in Figure 8. A single instance of Splitwise is equivalent to one model deployment in a specific region.

We first profile Bloom-176B, Llama3.1-8B, Llama3.2-3B, and Llama2-70B on a VM with A100-80GB with various input/output sizes as in [24]. The simulator provides TTFT, Time Between Tokens (TBT), E2E per request, and machine level performance using the performance models trained on characterization profiles. The accuracy of the performance model is validated using mean absolute percentage error (MAPE) on a 80:20 train:test dataset split. The error is less than 3%, as shown in Figure 3.

The ARIMA-based load predictor, region router, queue manager, and autoscaler are embedded within the controller module of the simulator. The controller uses these components to coordinate with different regions. Each region has a server manager, arbiter, model repository, and model router. The model repository stores all the model weights for ease of deploying a model. Furthermore, each model endpoint in a region has a server manager and model router. The request scheduler, instance allocation, and performance models to capture request-level metrics and machine-level utilization are present at each model deployment. We run our experiment harness for four models and three regions. The minimum instance count within a deployment is assumed to be two, and the maximum instance count in a model deployment is assumed to be three.

GPUs and LLM Models. We have three regions, US East, US West and US Central, and four standard LLM model types, Bloom, Llama-2, Llama-3.1 and Llama-3.2, used by IW and NIW with their default weights. All the model types are assigned 20 instances per region. We assume homogeneous hardware in the simulation and also the number of GPU cards needed by each models as identical. The *performance (TPS)* for each LLM type instance on each GPU type is shown in Figure 3

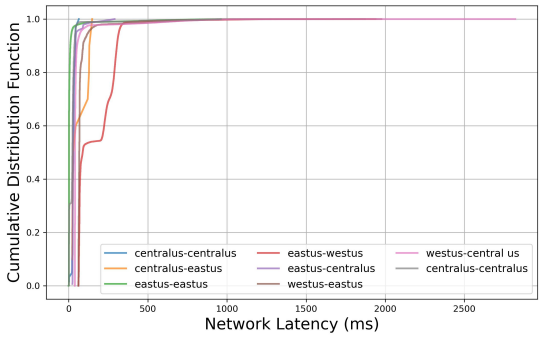

Figure 9: CDF of network latency between source and sink regions.

Deployment Characteristics. There are network overheads between different regions that affect the latency for client requests from one region being routed to instances in different regions. The latency distributions from source to sink regions are shown in Figure 9 based on realistic traces. For around 90% of the regions, the network latency is within 500ms with less than 2% of cases having a network latency of 2.5s.

There is a latency for deploying a new LLM model onto a set of GPU VMs. These times depend on the LLM model size. We assume the redeployment time for a model with weights available in the same region as around 10 minutes for all the models regardless of their parameter size. Though the time varies slightly based on the time it takes to transfer weights, the redeployment time of a model for which weights are absent in the local region is $\approx$ 2 hours.

In case existing spot instances are being reclaimed, the time to do so is a median of 1 minute and a maximum of 5 minutes. We assume that the time for routing the client request

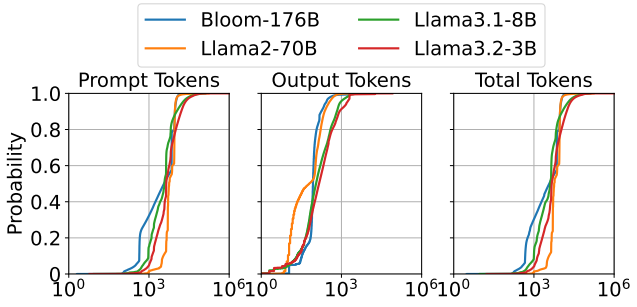

Figure 10: CDF of Prompt, Output and Total Token Counts in log scale. Majority of requests in real world traces have greater than 1000 prompt tokens and fewer than 1000 decode tokens. Token distribution varies per model.

to the target region's endpoint is negligible. The simulator also captures the unavailability of the VMs when being re-provisioned.

Workload Characteristics. We consider 3 client regions (US-E(all), US-C(all), US-W(all)) and focus on a one week period in November, 2024 for workloads generated for the 4 major model types. This forms the core dataset for our IW and NIW workloads as described earlier. For each of these LLM types, Figure 10 provides a distribution of the input and output token counts. In general, the majority of requests have an input token count greater than 1k, while the majority of the output token count distribution is before 1k and varies with model type.

7.2 Baselines

In the baselines, we consider the unified reactive heuristic from § 4 that represents the status quo. Whenever a request arrives at the regional endpoint of a model type, the effective utilization is measured, and if the value is greater than 70% (< 30%), the instance is scaled out (in). This approach is followed every time a request reaches an endpoint, with a cooldown period of 15s between scaling events. Furthermore, scaling events always happen using spot instances – surplus instances are donated to spot, and instances are acquired from spot when needed. Inter-model scaling is also allowed using spot instances, i.e., redeployment of a new model to replace the spot instance hosting another model type. For all the experiments, we discuss four strategies: unified reactive heuristic (reactive), LT-I, LT-U, and LT-UA.

7.3 Results

7.3.1 Effectiveness of Proactive Strategies in Reducing GPU-hours

We first evaluate the effectiveness of SAGESERVE for a single day trace. Figure 11 shows the trends of instance hour usage

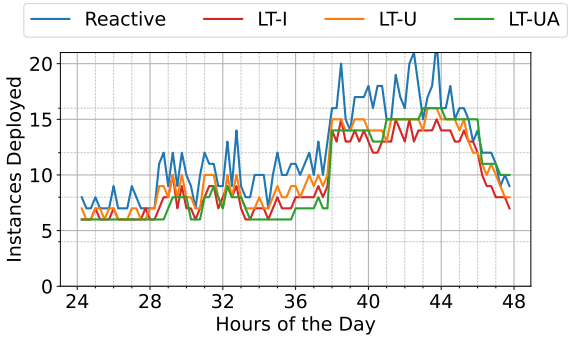

Figure 11: Aggregated sum of instances deployed across regions for Llama-2 on a peak traffic day. Area under curve for Reactive, LT-I, LT-U, and LT-UA are 302.25, 227.25, 247.5, 233.25 instance-hours, respectively. This translates to savings of roughly $0.5M when deploying the system on Azure over a week ($98.32 per hour for H100 clusters at the time of writing)!

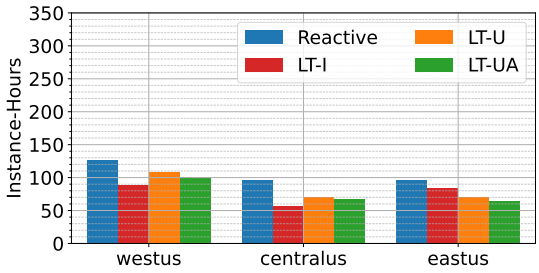

Figure 12: Llama-2, Instance-Hours for different strategies. SAGESERVE is able to reduce the instance hours across different regions with different workload patterns.

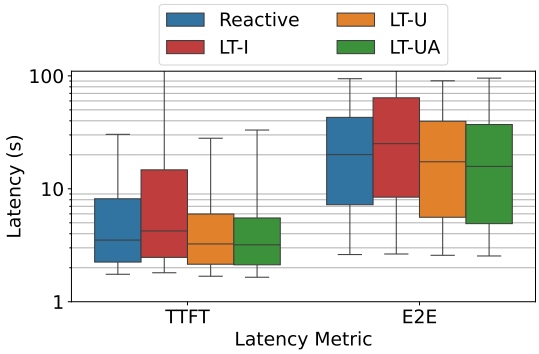

Figure 13: Latency Metrics across regions for Llama-2 on a peak traffic day. TTFT and E2E latency of serving requests is not harmed by proactive scaling approaches with real world workloads containing bursts.

by hour, aggregated across all the three regions for Llama-2 70B model. Our forecast aware strategies consistently use less instances as compared to the reactive strategy, with LT-I,

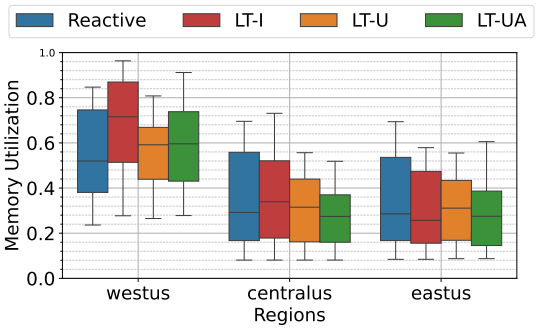

Figure 14: Llama-2, memory utilization for different strategies. Memory utilization in LT-I increases as immediate deallocation of instances can lead to higher pressure on the model instances. This problem is resolved in LT-U and LT-UA with heuristic based short term scaling.

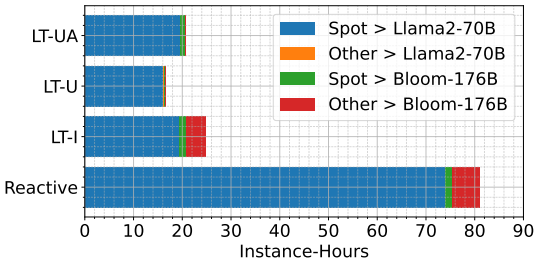

Figure 15: Overhead of scaling introduced by different strategies. Spot > indicates the event of acquiring spot by Llama. Other > Bloom indicate the event of acquiring spot instance of other model and redeploying Bloom. SAGESERVE wastes less GPU cycles by reducing frequent scaling up of model instances.

LT-U and LT-UA using 24.8%, 18.2% and 22.8% less instance hours respectively. This is intuitive as LT approaches do not react to momentary bursts in traffic and scale according to the forecasts. These results are also evident in Figure 12, which shows LT strategies are better throughout all regions.

7.3.2 GPU Cost Reduction without SLO Violations

LT-I utilizes less GPU hours but it slightly harms the TTFT and E2E latency of requests, as evident from Figure 13. This is because while immediately scaling up does not significantly benefit requests when traffic is lower, immediate scale down slows down requests, potentially harming their SLOs. These issues are fixed in LT-U and LT-UA, where instances are scaled on demand. These strategies helps us downscale only when we can do so while serving low latency requests, while making sure we do not up (or down) scale too much.

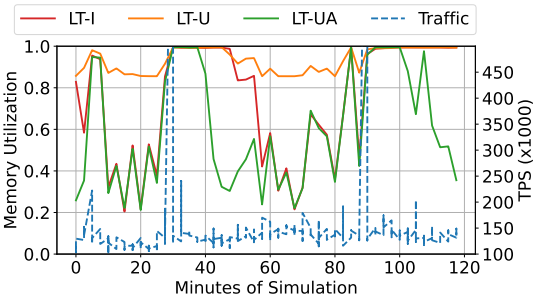

Figure 16: The performance of LT-UA in case of sudden bursts shows that the effective memory utilization of Llama-2 in US-East remains low for LT-UA, while the value spikes for other strategies. We introduced two bursts in the incoming traffic, which lead to an initial spike in the memory utilization of all three strategies. LT-UA was able to recover quickly from this by allocating instances above the threshold set up the forecasting logic, while LT-I and LT-U's utilization remained high.

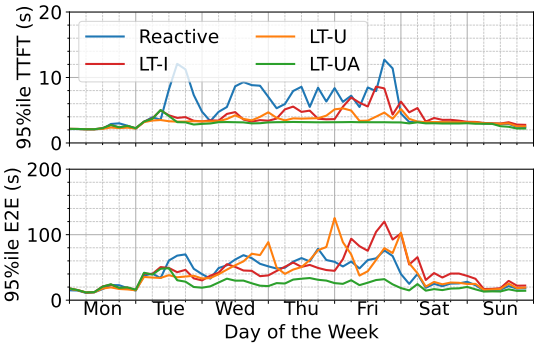

Figure 17: 95%ile latency metrics binned by 3 hours for Llama2-70B. SAGESERVE is able to reduce the 95% TTFT and E2E latency of request as compared to reactive scaling as it allocates instances proactively when incoming traffic is increasing.

7.3.3 Scaling Costs

We see in Figure 15 that SAGESERVE is able to reduce the GPU cycles wasted during instance deployment by about 70%. Due to fluctuations in traffic, reactive scaling approaches generally waste GPU cycles in constant scaling up operations. With the forecast knowledge, SAGESERVE's methods are able to reduce the number of times we upscale as well, resulting in much better utilization of acquired hardware.

7.3.4 Burst Management using SAGESERVE

As shown in the blue curve of Figure 16, we randomly increase the incoming load to 8x in order to simulate sudden bursts in traffic. While LT-U and LT-I are able to maintain their latency and memory utilization when small bursts in

traffic happen, they do not scale above the threshold set by the ILP and the ARIMA forecast even when large bursts occur. This is evident in the peaking latency metrics during this period shown in Figure 16, where we see that the green curve of LT-UA is able to reduce back it's memory utilization faster than LT-I and LT-U. Therefore, in such scenarios, LT-UA is able to cope with the uncertainity much better. As discussed in subsubsection 6.5.1, we set the threshold to scale up at 5x predicted traffic.

7.3.5 Validation on Week Long Trace

Figure 17 displays the 95%ile of TTFT and E2E latency over the course of one week. The insights gained from a one-day trace also apply in this case. The reactive strategy shows inferior performance, while other strategies achieve better performance metrics. LT-U and LT-UA behave similarly during weekdays, with a slight change in performance at the start of the weekend. This indicates the effect of the LT-UA strategy across longer time scales, which accounts for errors in ARIMA forecasts when the trend in TPS differs during the weekend. Overall, SAGESERVE scales well with longer trace and different request arrival trends.

8 Conclusions

We present SAGESERVE, a holistic system for serving LLM inference requests with a wide range of SLAs, which maintains better GPU utilization, reduces resource fragmentation that occurs in silos, and increases utility by donating surplus instances to Spot instances. SAGESERVE achieves this through its unique elements, namely, a holistic deployment stack for requests of varying SLAs, its async feed module, and long-term aware proactive scaler logics that capitalize on the underutilized instances of another model in the same region by inter-model redeployment.

Future work includes extending SAGESERVE to accomodate workloads with a continuum of SLAs and conducting extensive studies on the benefits of the proposed approach with deployments across heterogeneous hardware types. We plan to open-source our trace data and simulator.

References

[1] Chatgpt. http://chat.openai.com.

[2] Copilot. http://copilot.microsoft.com.

[3] Gemini. http://gemini.google.com.

[4] 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.

[5] BigScience. Introducing the world's largest open multilingual language model: Bloom [online]. In https://bigscience.huggingface.co/blog/bloom.

[6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. Advances in Neural Information Processing Systems, 35:16344–16359, 2022.

[7] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 135–153, 2024.

[8] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. M\'elange: Cost efficient large language model serving by exploiting gpu heterogeneity. arXiv preprint arXiv:2404.14527, 2024.

[9] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. Analysis of join-the-shortest-queue routing for web server farms. Performance Evaluation, 64(9-12):1062–1081, 2007.

[10] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean:{VM} allocation service at scale. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 845–861, 2020.

[11] Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri, Rujia Wang, Chetan Bansal, Victor Rühle, et al. Intelligent router for llm workloads: Improving performance through workload-aware scheduling. arXiv preprint arXiv:2408.13510, 2024.

[12] Ferdi Kossmann, Bruce Fontaine, Daya Khudia, Michael Cafarella, and Samuel Madden. Is the gpu half-empty or half-full? practical scheduling techniques for llms. arXiv preprint arXiv:2410.17840, 2024.

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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, pages 611–626, 2023.

[14] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 663–679, 2023.

[15] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. Andes: Defining and enhancing quality-of-experience in llm-based text streaming services. arXiv preprint arXiv:2404.16283, 2024.

[16] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Distributed serving of large language models via max-flow on heterogeneous gpus. arXiv preprint arXiv:2406.01566, 2024.

[17] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pages 1112–1127, 2024.

[18] Microsoft. Online endpoint deployment for real-time inferencing, 2024. https://learn.microsoft.com/en-us/azure/machine-learning/concept-endpoints-online.

[19] Microsoft. Run azure openai models in batch endpoints to compute embeddings, 2024. https://learn.microsoft.com/en-us/azure/machine-learning/how-to-use-batch-model-openai-embeddings.

[20] Chengyi Nie, Rodrigo Fonseca, and Zhenhua Liu. Aladdin: Joint placement and scaling for slo-aware llm serving. arXiv preprint arXiv:2405.06856, 2024.

[21] OpenAI. Announcing the stargate project.

[22] OpenAI. Batch api, 2024.

[23] OpenAI. Streaming api, 2024. https://platform.openai.com/docs/api-reference/streaming.

[24] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. arXiv preprint arXiv:2311.18677, 2023.

[25] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), pages 118–132. IEEE, 2024.

[26] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vattention: Dynamic memory management for serving llms without pagedattention. arXiv preprint arXiv:2405.04437, 2024.

[27] Yifan Qiao, Shu Anzai, Shan Yu, Haoran Ma, Yang Wang, Miryung Kim, and Harry Xu. Conserve: Harvesting gpus for low-latency and high-throughput large language model serving. arXiv preprint arXiv:2410.01228, 2024.

[28] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In 2011 IEEE 4th International Conference on Cloud Computing, pages 500–507. IEEE, 2011.

[29] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In Proceedings of the Fifteenth European Conference on Computer Systems, pages 1–16, 2020.

[30] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. {ServiceRouter}: Hyperscale and minimal cost service mesh at meta. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 969–985, 2023.

[31] P. Schmid, O. Sansevieroa, P. Cuenca, and L. Tunstall. Llama 2 is here - get it on hugging face [online]. In Available: https://huggingface.co/blog/llama2.

[32] Robert H Shumway, David S Stoffer, Robert H Shumway, and David S Stoffer. Arima models. Time series analysis and its applications: with R examples, pages 75–163, 2017.

[33] Prateeksha Varshney and Yogesh Simmhan. Autobot: Resilient and cost-effective scheduling of a bag of tasks on spot vms. IEEE Transactions on Parallel and Distributed Systems, 30(7):1512–1527, 2018.

[34] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. arXiv preprint arXiv:2305.05920, 2023.

[35] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. Can't be late: Optimizing spot instance savings under deadlines. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 185–203, 2024.

[36] 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, July 2022. USENIX Association.