USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

BlitzScale: Fast and Live Large Model Autoscaling with *O*(1) Host Caching

Dingyan Zhang, Haotian Wang, Yang Liu, and Xingda Wei, *Shanghai Jiao Tong University;* Yizhou Shan, *Huawei Cloud;* Rong Chen and Haibo Chen, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/osdi25/presentation/zhang-dingyan

This paper is included in the Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation.

July 7–9, 2025 • Boston, MA, USA

ISBN 978-1-939133-47-2

Open access to the Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

BLITZSCALE: Fast and Live Large Model Autoscaling with $O(1)$ Host Caching

Dingyan Zhang, Haotian Wang†, Yang Liu†, Xingda Wei✉[1], Yizhou Shan[2], and Rong Chen, Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University [2]Huawei Cloud

Abstract

Model autoscaling is the key mechanism to achieve serverless model-as-a-service, but it faces a fundamental trade-off between scaling speed and storage/memory usage to cache parameters, and cannot meet frequent scaling requirements across multiple hosts. The key problem is that data plane performance is slow, and scaled instances remain stopped while parameters are loading.

In this paper, we first show that the data plane can be made *fast* with no or $O(1)$ caching by loading parameters through the compute network between GPUs because: (1) its speed is comparable to host cache and is underutilized, and (2) scaling multiple instances requires no or $O(1)$ caching with network-optimized multicast. Second, autoscaling can be made *live* by breaking the scaling abstraction for inference from a coarse-grained instance-level to a fine-grained layer-level. This allows us to offload the layer computation from the overloaded serving instances to the scaled ones without waiting for the parameters to be fully loaded.

Under real-world workloads, our system BLITZSCALE achieves up to 94 % lower tail latency reductions compared to state-of-the-art autoscaling system (ServerlessLLM), and it reduces the GPU time used for serving by 49 % when compared with serving systems that do not support autoscaling like DistServe and vLLM with the same service-level-agreement.

1 Introduction

Recent years have seen rapid growth in applications powered by deep learning models like large language models (LLMs) [13, 53, 34, 54, 61]. Due to the huge computation requirements, these models are typically served in model-serving-as-a-service systems (MAAS) [63, 11, 23, 76, 20, 10, 29], which manage a cluster of accelerators (e.g., GPUs) and provision an appropriate number of serving *instances* containing GPUs to each model deployed.

An MAAS system has two design objectives: *maximizing goodput*—the number of requests that meet the service level objective (SLO), and *minimizing instances provisioned* to each model to improve hardware utilization. Achieving both is challenging due to the unpredictable short-term fluctuations in a model's instance demands, (5 × required within 2 seconds), because the request arrival rate bursts at seconds-level [45, 80], where the memory usage of each request is also unpredictable due to the auto-regressive nature of LLMs [45, 80, 29] (see also Figure 1 and §2.2).

Model autoscaling is a promising solution [76, 11, 29, 3]. With autoscaling, a MAAS system only provisions the average number of instances required over the long term for each served model, which remains relatively stable [71]. This improves utilization. Upon bursts, the system automatically scales new instances, avoiding SLO violations due to request queueing caused by insufficient instances provisioned.

Autoscaling speed is critical in minimizing SLO violations because the queued requests are not served until the instances are scaled. For instance, the inference time of a Llama3-8B is 80-900 ms on commodity GPU (A800), while users expect a tight response time (< 1 second) for scenarios like chatbot [12, 81, 33]. Meeting such tight SLO requires less than 500 ms scaling time, but achieving this is challenging especially for LLMs with 10-400 GB parameters. The key reason is the **slow data plane** of autoscaling—the process of loading the model parameters to instances' GPUs. While high bandwidth SSDs are utilized in current work [29], the speed provided by SSDs of GPU servers (2-10 Gbps per GPU [35, 9, 27]) is still far from ideal. For instance, loading Llama3-8 B to a GPU takes 12.8 seconds with 10 Gbps SSD. Another factor obstructing fast scaling is that existing scaling methods are **stop-the-world**: the scaled instances cannot serve requests until all parameters are loaded. This implies that autoscaling is directly bottlenecked by the data plane.

To mitigate the above issues, state-of-the-art systems like ServerlessLLM further adopt a multi-tiered caching system by caching model parameters in the host (CPU) DRAM to accelerate the data plane [36, 41]. Under cache hit, they can leverage the fast CPU-GPU link (e.g., 256 Gbps PCIe) to load parameters. However, achieving a high hit rate is unfeasible: ServerlessLLM reports a hit rate of 40–75 %, which is confirmed by us (see §3). The root cause is that a MAAS typically hosts many models, thus achieving a 100 % cache hit requires caching all these models on the DRAM of each host, clearly impractical. Vendors typically host many models because there are hundreds of popular open-source model families designed for different purposes [28]. Meanwhile, each model family has different scales for balancing the serving cost and accuracy [4]. Finally, developers can upload their customized fine-tuned models based on open-source models [10].

To achieve fast model scaling without relying on cache hit, we make the following two key contributions:

1. Data plane made fast with $O(1)$ or no caching with

†These authors contributed equally to this work.

✉Xingda Wei is the corresponding author (wxdwfc@sjtu.edu.cn).

compute network multicast. First, a MAAS is backed by fast GPU-GPU/CPU compute fabrics [50], which are 100–400 Gbps RDMA and even 16 Tbps NVLink [35, 9, 27]—much faster than SSD and comparable (or even faster than) CPU-GPU PCIe. The compute fabric is used for data transfer during serving and we found it largely under-utilized, i.e., up to 7.4 % of total bandwidth even in network-heavy workloads like serving LLMs with prefill and decode disaggregation [55, 81, 38, 19, 75] (§3). Thus, we can borrow such fast links for accelerating the data plane of autoscaling.

Second, network-based data plane requires no or minimal caching to achieve fast scaling. Specifically, if a model is already deployed on some instances, we can directly multicast the parameters from deployed instances through the network, eliminating the need for caching. Such multicast is extremely efficient because a serial forwarding multicast [66] can load bulk data (e.g., model parameters), regardless of the number of receivers. Even if no instance is deployed, multicast can be done with $O(1)$ host caching by simply broadcasting the parameters from the host with the cached model. This $O(1)$ caching per-model allows us to avoid all cache misses since the aggregated host memory of all machines is sufficient to cache all models served by a MAAS.

Although fast networking can significantly accelerate the data plane with minimal caching, a stop-the-world loading remains a bottleneck in cases when the networking is not fast enough. For example, to achieve at most 40 % SLO violations when serving a BurstGPT workload with Qwen2.5-72B, the system needs to achieve a tight 500 ms stop time. Achieving so requires 576 Gbps per-GPU[1] parameter transfer bandwidth, far exceeding the available bandwidth of typical compute network setups (e.g., 200 Gbps per-GPU) and even when caching at the host (256 Gbps PCIe). Thus, we argue that an ideal parameter loading should be *live*: before the data loading finishes, the scaled instance should be able to serve requests.

2. Data plane made live with fine-grained scaling abstraction and cooperative execution. Model scaling cannot be live using traditional on-demand data loading techniques commonly found in serverless computing [74, 69, 39] or inference loading overlap in PipeSwitch [15] (§4), because an instance can only emit results once all the parameters are loaded. This stop is rooted in the coarse-grained scaling abstraction of existing systems: they can only scale and serve at the instance level. To realize live scaling, our key insight is that *models can be served in a fine-grained, layer-by-layer manner*. With this fine-grained layer-wise scaling, we can offload part of the layer's computation from overloaded instances to scaled instances with cooperative execution, thus improving the overall serving throughput even before the scaled instance has loaded all the parameters.

Challenges and solutions. First, utilizing network-based multicast is non-trivial in our setup. Though the mechanism of multicast is simple, i.e., simply forwarding parameters between instances with the network, the challenges lie in generating the multicast plan, i.e., determining how the data flows between instances. First, we need to quickly generate an efficient plan online on diverse network topologies since our sources and destinations are dynamically determined, but generating an optimal plan is NP-hard on heterogeneous networks in serving clusters [18]. Second, we need to avoid network interference between the scaling and serving, otherwise, we observed a 1.5 × longer scale time and 50% degraded tail TBT (§4). Current solutions [22, 19, 32] mainly target offline scenarios like training, so they can tolerate long plan generation time and don't need to consider interference from serving workloads. To address the issue, we propose a model-aware multicast planner, which leverages the key features of compute network and the static data flow in model serving to quickly generate a near-optimal, interference-free multicast plan for scaling (§5.1).

Second, it is challenging to schedule how requests are executed between deployed and live scaling instances, i.e., which instance executes which layers. The challenge lies in the fact that the serving capability of the scaling instances is limited—it can only execute layers with parameters loaded, and this capability is dynamically changing. A naive best-effort scaling that executes as many layers as possible cannot balance the load because at the beginning of autoscaling, the new instances can only execute few layers, with requests still queued at the overloaded instances. A better solution is to adjust the load holistically by considering future incoming layers, and we realize this with a ZigZag pipeline scheduling and achieve 50% tail latency reduction under bursty workloads (§5.2).

Demonstration with BLITZSCALE. We built BLITZSCALE, an MAAS system with the fastest autoscaling speed with $O(1)$ caching. We adopted a global parameter pool to cache the model parameters across all the machines, and integrated the aforementioned interference-free multicast plan for scaling and efficient ZigZag scheduling-based live scheduling. To show the effectiveness of BLITZSCALE, we evaluated BLITZSCALE by running real-world traces (i.e., BurstGPT [71], AzureCode and AzureConv [14]) across a variety of recent models with different sizes and architectures, including Llama3-8B, Mistral-24B, and Qwen2.5-72B. First, BLITZSCALE has 47–75 % shorter time-to-first token, and has up to 94 % shorter time-between-tokens than the state-of-the-art work (ServerlessLLM [29]). Second, compared to serving systems without autoscaling support, i.e., vLLM [44] and DistServe [81], BLITZSCALE reduces the GPU used for serving a single model by 49 % with no SLO violations compared to an over-provisioning setup that provisions the GPUs based on the maximum request rate. BLITZSCALE is open-sourced at https://github.com/blitz-serving/blitz-scale.

[1] 72 B model requires at least four GPUs per-instance for serving.

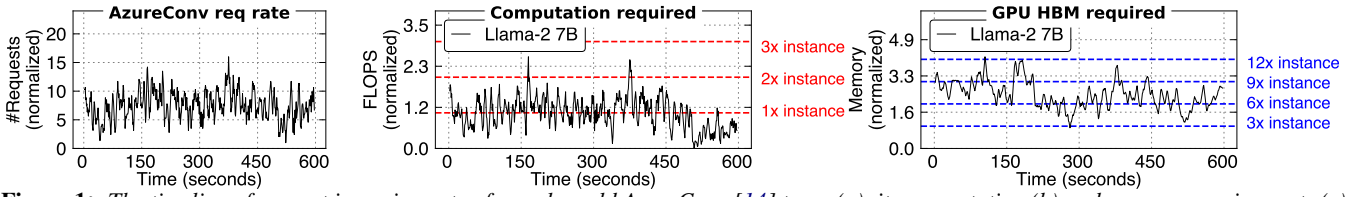

Figure 1: *The timeline of request incoming rate of a real-world AzureConv [14] trace (a), its computation (b) and memory requirements (c) when serving this workload without SLO violation.*

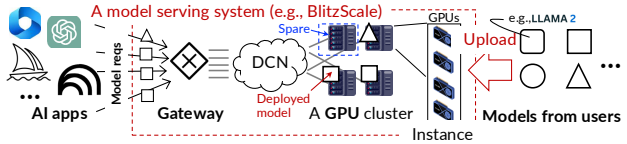

Figure 2: *An illustration of Model as a Service (*MAAS*) system. DCN states for data center network.*

2 Background: MAAS and Autoscaling

2.1 System setup: model-serving-as-a-service (MAAS)

MAAS system setup and the serving instance. BLITZSCALE targets a MAAS scenario [63, 11, 23, 76, 20, 10, 29]: the cloud allows the users to deploy their model serving on its managed hardware, and only charges users based on the number of requests processed within SLO [10, 20]. The model can be popular open-source models like Qwen [4] or customized models uploaded by the users. Thanks to the above pricing strategy, the cloud can dynamically adjust hardware resources to a specific model to maximize its hardware utilization. Figure 2 shows a typical deployment of a MAAS system. For each user-deployed model, the system dynamically allocates the required hardware resources (GPUs) to serve the inference requests of the model. Note that due to the diversity of AI applications, there would be hundreds or even thousands of models served simultaneously by a MAAS system [21].

In this paper, we use *instance* to denote a set of GPUs storing a complete copy of a model parameter for serving this model. An instance can have a single GPU or multiple GPUs when the parameter is large and sharded across them, e.g., with tensor parallelism [47]. Because each instance has a maximal serving throughput, the cloud can deploy multiply instances of the same model by provisioning multiple sets of GPUs, where the number of instances is dynamically scaled based on the incoming request rate. BLITZSCALE supports autoscaling with all existing model serving methods at the instance level.

Serving within an instance: non-LLM & LLM. Each serving instance processes requests in the following workflow: it queries the model in a layer-by-layer computation paradigm (see Figure 9 (a)) and gets the final results. For simple models like vision models, the model is queried once with the input data (image). On the other hand, for large language models (LLMs) [65], the request queries the model multiple times: the model is first queried with input text (prompt) to produce a result (token). This first query is typically termed *prefill*. The token is then used to generate subsequent tokens iteratively until the model returns an end-of-sequence token. The auto-regressive phase is termed *decode*.

We note two important features of LLMs. First, the performance for prefill and decode is measured separately. Prefill is evaluated with the time-to-first-token (TTFT) while decode is evaluated using time-between-tokens (TBT). Second, the LLM query is stateful: the intermediate results—usually termed as *KVCache*—are cached in GPU memory during the auto-regressive phase of a request for acceleration.

Serving across instances: prefill and decode (PD) disaggregated LLM serving. Observing the different computing paradigms of prefill and decode, recent works propose separating the instances for prefill and decode (PD disaggregation) when processing serving requests [55, 81, 38]. Specifically, for each request, one instance processes the prefill phase (prefill instance) and another instance (decode instance) processes the decode phase. This paradigm requires excessive data movement between the two instances because the prefill instance needs to transfer the KVCache to the decode instance. BLITZSCALE works for both PD disaggregated and non-disaggregated LLM serving.

2.2 Dynamic hardware demands when serving a model

Determining the hardware requirements, i.e., the right number of instances for serving a model is challenging because the hardware demands are unpredictable and fluctuating. First, the incoming request rate for a serving workload fluctuates over time and is hard to predict [29, 59, 80]. Figure 1 (a) presents the number of requests sent to a single model service over time from a real-world trace—BurstGPT [71]: the incoming inference requests increase 5 × within 2 seconds with no predictable trend. Since the FLOPS of an instance is fixed, the unpredictable rate causes the computation requirement—FLOPS required to finish the pending requests within SLO—unpredictable. Figure 1 (b) confirms this by measuring the requirement of the prefill instances when serving the BurstGPT with Llama2-7B.

Second, serving modern models like LLM has non-trivial and unpredictable memory requirements. As shown in Figure 1 (c), the KVCache usage of the decode instances is multiple times larger than the memory capacity of a single instance and fluctuates over time (3–12 ×) when serving the BurstGPT workload with Llama2-7B. The root cause is that the KVCache of requests must be stationary in GPU memory during the decode phase. The KVCache of requests are large,

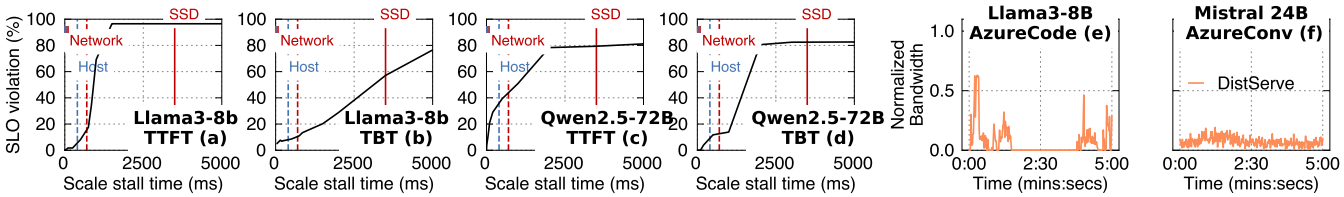

Figure 3: *A characterization of SLO attainment for different inference cases (a)–(d) with varied duration of autoscaling stops on BurstGPT [71]. (e) and (f): an analysis of compute network usage in serving workloads. The evaluation setup is in §6.*

e.g., 190–760 GB for Llama2-7B to serve BurstGPT, and the stay time is unpredictable due to the auto-regressive nature of LLMs. To avoid performance losses when out of memory, a MaaS system must provision sufficient instances to hold KVCache from ongoing requests, so the number of instances required by a model also unpredictably fluctuates.

2.3 Model autoscaling for handling dynamic demands

Model autoscaling, which dynamically deploys[2] serving instances on spare GPUs to scale up the serving capability, is a promising solution to handle fluctuated and unpredictable computation and memory demands [29, 76, 11]. The rationale is that though a single model's workload is unpredictable, the aggregated workloads of all models served by a platform are relatively stable [80]. Thus, when the load of a specific model service increases, we are able to find spare GPUs from other models to scale up the serving capability of this model.

Autoscaling an instance requires two basic steps: (1) initialize a proper execution context, e.g., create CUDA contexts (control plane) and (2) load the model parameters to the GPUs' memory (data plane). We focus on (2) because (1) can be minimized with recent advances in GPU startup methods like checkpoint and restore [40, 79] and our Rust/C++-based serving platform (see also §6.3). For (2), the state-of-the-art system—ServerlessLLM [29] optimizes the data plane with SSD-optimized parameter loading. Unfortunately, it does not account for the scaling speed required by models. Our measurements in the next section show that SSD-based scaling significantly lags behind applications' requirements.

3 Characterizing Scaling Requirements and Compute Network between Instances

Model autoscaling requires fast data plane. If the data plane speed is not fast enough, during burst period, the requests still violate the SLO due to increased queueing time. Specifically, SLO defines the tolerable end-to-end latency measured from the time a request is sent to the system to the time the inference response is returned. Thus, the latency includes the queueing delays waiting for the scaled instance to be ready for inference.

To characterize how different scaling speeds affect SLO attainments, we implemented a simulator on DistServe [81] that provisions models to all GPUs and applies manual delays based on the simulated speed for modeling different scaling speeds. We set TTFT and TBT SLO based on inference speed of different models following prior works [81]. Specifically, we use 450 ms and 150 ms for Llama3-8B model, and 1250 ms and 200 ms for Qwen2.5-72B model with tensor parallelism degree of 4. §6 describes the detailed evaluation setup.

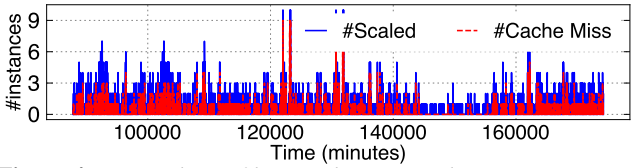

Figure 4: *An analysis of host cache misses when running ServerlessLLM [29] on BurstGPT [71].*

Figure 3 (a)–(d) shows the results: We can see that for a 72 B model, maintaining SLO violations below 60 % requires a minimum per-instance scaling speed of 220 Gbps per-GPU[3], which is only achievable when the model parameters are loaded from the host memory. The scaling time requirement correlates directly with inference time—our evaluated workload (BurstGPT [71]) has an average TTFT of 771 ms (with queueing time). Thus, to achieve 1250 ms SLO for all requests, the scale time must be below 500 ms, so a 576 Gbps per-GPU network speed is required (measured by dividing the parameter size by the scale time). This far exceeds what vendor-provided per-GPU SSDs bandwidth can deliver (2–10 Gbps per-GPU [35, 9, 27], detailed in §A.2).

Loading model parameters from host memory is not effective due to misses. While caching the parameters on the host CPU memory can meet the scaling speed requirement for some setups (e.g., 8 B, 24 B) with the fast host-GPU interconnects (256 Gbps PCIe 4.0), cache misses are common in real-world traces, because the scarce host memory cannot support the caching all models deployed on the MaaS. Figure 4 presents the number of instances scaled and cache misses encountered in the BurstGPT workload using ServerlessLLM [29]. Following its setup, we set a 5-minute keep-alive interval for caching models at the host. The miss rates range from 20–46%, depending on the time, which aligns with the numbers reported in ServerlessLLM's paper (25–60%). Interestingly, many misses occur when scaling multiple instances, because involving more hosts increases the

[2] It also stops a serving instance to scale down. Since scaling down is simpler, we omit its details for brevity.

[3] 72 B model uses four GPUs per-instance.

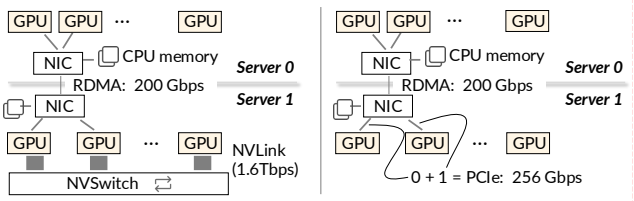

Figure 5: *A illustration of networking in* MAAS *with NVLink (a) and without it. Note that the 256 Gbps PCIe is shared between two GPUs attached to it [48].*

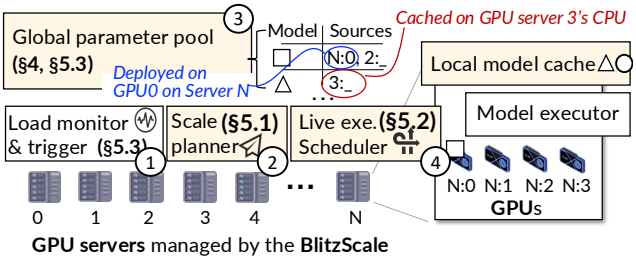

Figure 6: *System architecture of* BLITZSCALE. *New modules introduced by* BLITZSCALE *are marked with* □.

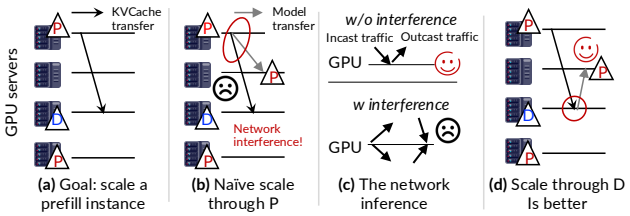

Figure 7: *(a) An illustration of scaling a prefill instance for LLM PD disaggregated serving. (b) Naively scaling from a prefill instance imposes network interference. (c) Interference can be avoided by leveraging the bi-directional feature of modern DCN networking. (d) An improved scale plan with the bi-directional in mind.*

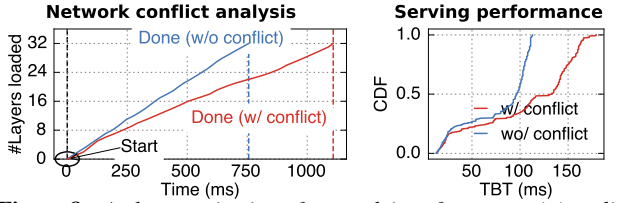

Figure 8: *A characterization of network interference on (a) scaling speed and (b) serving performance.*

probability of scaling a model on a host without the cached parameters. Therefore, we still need accelerating scaling speed when the model parameters are not cached at the host.

Opportunity: fast and underutilized compute network. First, compute networks between GPUs (and CPUs) have comparable or even faster speeds than host-to-GPU link. As shown in Figure 5, the inter-GPU network (RDMA) operates at 200 Gbps, which is close to the host-to-GPU PCIe speed (256 Gbps). With NVLink, the speed is much faster. More importantly, these networks are underutilized during serving. Figure 3 (e) and (f) measure the peak network usage of DistServe [81], a PD disaggregated serving system that heavily utilizes the network due to KVCache transfers. To measure peak usage, we provisioned all the GPUs for serving, and evaluate a workload with the maximal request rate that our clusters can serve. Even under peak load, more than 40% of the network capacity is free, opening up the opportunity to use the compute network for the scaling data plane.

4 System Overview of BLITZSCALE

BLITZSCALE scales models through the compute network to accelerate scaling even under cache misses on the host. We achieve this by first managing model parameters—scattered across GPUs behind serving instances (for deployed models) and CPUs (cached at the host)—through a global parameter manager. The manager maintains a mapping between models and their sources. With the manager, we can quickly read parameters from these sources with the fast RDMA or NVLink. Besides, we also offload computation from overloaded instances to instances with partially loaded parameters to achieve live scaling.

System architecture and workflow. Figure 6 shows our system architecture. Like prior work [56, 29, 62], we have a load monitor (①) tracking the serving load for each model service, and deciding whether to scale and how many new instances are required (§5.3). On each machine, we further adopt off-the-shelf GPU kernels FlashInfer [1] to query the model efficiently. The key differences are twofold. First, our scale planner (②) will derive a scaling plan that guides how to load parameters onto the scaled instances (§5.1) with compute network efficiently. The planner consults the global parameter manager (③) to identify the sources of model parameters. In our example, the new instance can load the parameters of model □ from the GPU0 on host N $(N, 0)$, or from host 2's CPU memory $(2, \_)$. Second, during scaling, our live execution (exe.) scheduler (④) will redirect requests between instances to fully utilize the scaled instances even before the parameters are fully loaded (§5.2).

Challenges and approaches. Despite leveraging fast networking, making autoscaling fast and live needs to address the following challenges.

C#1. Online interference-free scale plan generation. Generating the scale plan is similar to generating a *multicast* plan [18, 22, 32, 16, 17], i.e., how to quickly distribute data (parameters) from some sources to targets. There are two additional requirements for autoscaling. First, the plan must be generated online on dynamically changing sources and targets, but optimal plan generation is NP-hard [18] on a heterogeneous network. Second, the plan needs to eliminate interference between loading and the serving workload. Figure 7 shows an example when a model is served via PD disaggregation. In PD disaggregation serving workload KVCache is migrated from prefill to decode instances (a), and this migration overhead can be hidden [55]. However, sup-

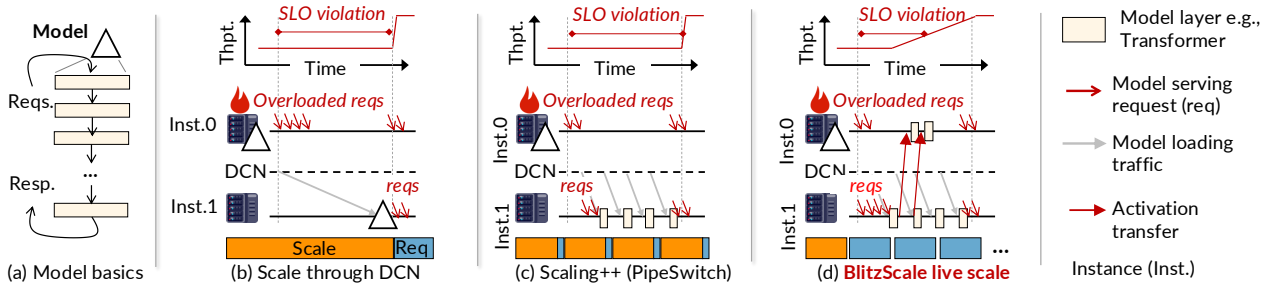

Figure 9: *(a) An overview of how model executes requests. (b) An illustration of naive model scale through DCN. (c) An illustration of an optimized scaling method with overlapped execution [15], but it still cannot be live. (d) How* BLITZSCALE *scales models lively.*

pose we want to scale a prefill instance: if we naively select a prefill instance as the source (b), the scaling will compete the network bandwidth with the serving workload, leading to 1.5 × longer scale time as well as 50% tail TBT increase due to the amplified KVCache migration overhead (Figure 8 (b)).

To this end, we design a serving-guided greedy plan generation method based on three observations (§5.1). First, the network heterogeneity mainly comes from NVLink, whose speed is extremely fast, i.e., it can broadcast a Llama3 8B to 8 GPUs within 120 ms. Thus, we can abstract instances linked with NVLink as a logical instance group to eliminate NVLink from the network topology. Second, loading parameters from the network is bandwidth-intensive, so we can greedily construct serial forwarding chains [66] for multicast, which is optimal in the common case. Finally, the network (RDMA) between GPU servers is bi-directional [72, 52], meaning that the network flows of incast and outcast don't interfere (c). Thus, we can leverage this feature to avoid interference by removing flows in the same direction on the same network link during plan generation. For example, we can load the parameters from the decode instance to the prefill instance (see Figure 7 (d)).

C#2. Realizing live autoscale. Live autoscaling—allowing the scaled instance to increase system throughput before all parameters are fully loaded—is necessary because SLO violations can still happen (Figure 3 (c)) even with fast networking. It is challenging to achieve this in existing systems. For example, PipeSwitch [15] and DeepPlan [42] leverage the layer-by-layer execution nature of models to perform inference: As shown in Figure 9 (c), once the first layer is loaded on inst.1, they redirect the overloaded requests to it for execution. Meanwhile, inst.1 will load subsequent layers concurrently. However, such overlapping is not live because, until all the layers are loaded, inst.1 still cannot finish requests to increase the system throughput.

To this end, we propose a novel cooperative execution scheme for live autoscaling. The key observation is that though the scaled instance alone cannot finish the requests until all layers are loaded, it can alleviate the load of the overloaded instance with its loaded layers, thus improving the serving throughput. Figure 9 (d) illustrates this. When inst.0 becomes overloaded and inst.1 is under scaling, after inst.1 has loaded the first layer, we redirect all requests from inst.0 to inst.1 for execution. Once inst.1 completes the first layer's execution, it forwards the activation back to inst.0 to process the remaining layers, and the system throughput increases with reduced queued latencies, as queued requests are processed faster. To see why the throughput increases, consider serving a 7-layer model. inst.0 alone will have a throughput of 1/7. With our live scaling, after loading one layer on inst.1, inst.0 only needs to execute 6 layers, so its throughput increases to 1/6. The throughput continues to improve as more layers are loaded, reaching the peak (doubled) after half of the layers have been loaded—half of the scaling time. §5.2 describes our ZigZag scheduling for coordinating overloaded and new instances during live autoscaling to achieve optimal performance for live autoscaling.

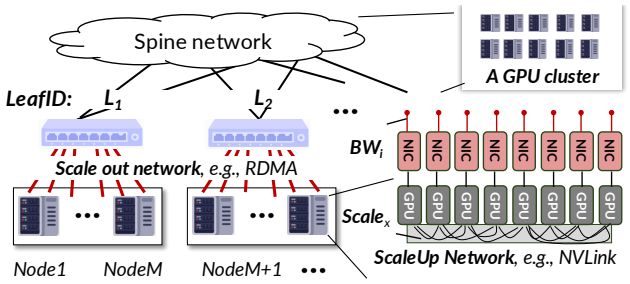

Figure 10: *An illustration of how* BLITZSCALE *models the network.*

5 Detailed Design and Implementation

5.1 Online network-based scale plan generation

When the planner is notified to scale the parameters onto n new GPUs, it will get s sources from the parameter pool, find t spare GPUs as target and generate a plan on how to send parameters from s sources to a subset of n GPUs in t targets. There are three metrics to minimize for the generation: (1) the scale time, (2) the plan generation time, and (3) the interference with serving workloads.

Modeling the network between GPUs. Effectively generating a plan requires a model of the network between sources and targets, which is non-trivial due to the complexity of the network topology in serving clusters. Our model assumes a

Input: D_{src}, source GPUs; D_{tgt} target GPUs; n: number of GPUs to scale;
BW_i: return the scale out network bandwidth of GPU i;
L_i: return the leaf switch ID of GPU i;
$Scale_i$: the scaleup domain ID of GPU i.
Output: **Plan**, a graph indicating how the parameters are loaded.

```
1  D_src = D_src.prune().group_by(L_i).sorted_by(sum([BW_i])).flatten()
2  D_tgt = D_tgt.group_by(Scale_i).sorted_by(sum([BW_i]))
                ▸ sort D_tgt according to the order of Leaf ID in D_src
          .groupby(L_i).sortby(D_src.index(L_i).min()).flatten()
3  m = 0; Plan = ∅
4  While not D_tgt.empty() and m < n:
   |  ▸ G_tgt: targets group connected by the scale up network
   |    L_tgt: the leaf switch of these targets.
5  |  G_tgt, L_tgt = D_tgt.pop_front()
6  |  If D_src.filter(= L_tgt).sum([BW_i]) ≥ G_tgt.sum([BW_i])
   |  |  ▸ sources within the leaf have sufficient bandwidth for loading
7  |  |  D_tmp = D_src.truncate(= L_tgt); D_src ++ D_tmp
8  |  G_src = D_src.truncate(sum([BW_i]) ≥ G_tgt.sum([BW_i]))
9  |  Plan = Plan ++ (G_src, G_tgt); m = m + |G_tgt|
10 |  D_src = G_tgt ++ D_src
11 return Plan
```

Notation:

- **++** ▸ join 2 data collections, e.g., [x, y] ++ [z, w] = [x, y, z, w]
- **[Func]$_i$** ▸ apply **Func** to all elements of a data collection, e.g., G.sum([BW$_i$]) is the summation of bandwidth of all elements in G
- **Iter. index(value)** ▸ locations where a unique value first occurs in **Iter**
- **Iter.truncate(predicate)** ▸ pop front until the first remaining element satisfies predicate
- **Iter.truncate(flatMap, predicate)** ▸ pop front and apply flatMap onto the popped elements until the result satisfies predicate

Figure 11: *The pseudocode of the plan generation algorithm.*

simplified scale-up and scale-out network hybrid networking, widely adopted for GPU clusters [25, 70, 50]. Figure 10 (a) illustrates the modelling.

First, we model the GPUs connected via fast scale-up networking like NVLink as groups of GPUs. Such GPUs have ultra-high interconnect bandwidth (1,600-3,600 Gbps) so scaling within a group has negligible overhead. On the other hand, GPUs connected via slower scale-out networking like RDMA are more difficult to model due to a hierarchical structure. To this end, we adopted a simple leaf-to-spine model that covers widely deployed topologies including Clos and Rail-optimized [70] with different subscription ratios: each GPU (i) has a BW_i bandwidth connected to a leaf switch (LeafID), where GPUs within the same leaf switch have a full-mesh connection, i.e., the bandwidth between GPUs (i) and (j) is $\min(\mathrm{BW}_i, \mathrm{BW}_j)$ with full bandwidth. Second, leaf switches are connected to spine switches, with inter-leaf bandwidth equal to or smaller than the intra-leaf bandwidth. For simplicity, we don't model the spine network and rely on upper-tier protocols like Virtual Link Trunking (VLT [30]) and Equal-Cost Multi-Path (ECMP [37]). to support efficient inter-leaf networking.

Multicast-chain-based greedily plan generation. To quickly generate the plan online, we use a three-step greedy

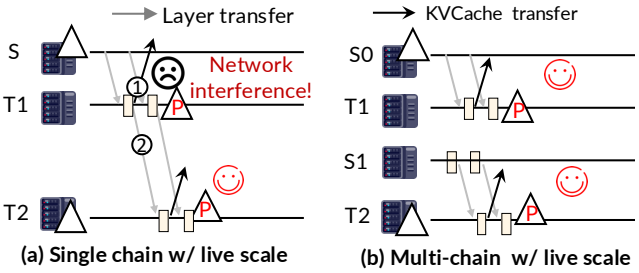

Figure 12: *An illustration of why multiple chains are better especially under live scaling.*

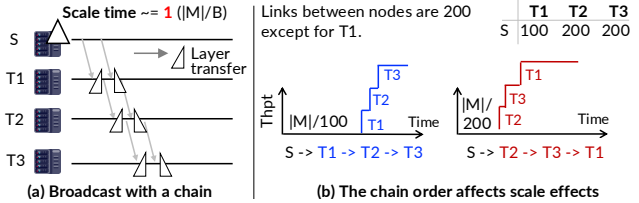

Figure 13: *An illustration of (a) why chain is friendly to broadcast with huge bandwidth requirement and (b) why we chooses a specific chain order. $|M|$ is the model size and B is the slowest network bandwidth between nodes in a chain.*

algorithm as shown in Algorithm 11. First, we prune the sources to avoid any interference with serving workloads (Line 1). Second, we group targets connected with scale-up networking like NVLink as a group (Line 2) such that we can leverage the NVLink broadcast to efficiently realize parallel sharded parameter transfer, see Figure 14 and described below. Finally, we form multiple serial broadcast chains (Line 3–10) to generate the plan.

Specifically, a serial broadcast chain is formed by a set of source and target nodes, i.e., $S \rightarrow T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$. Note that a node in a chain may have multiple GPUs. Such a chain has a nice property that it is optimal in bandwidth-intensive transfer like model scaling, because the overall transmission time is irrespective of the instances scaled with such a chain. As shown in Figure 13 (a), when T1 receives the first layer, T1 immediately forwards it to T2. Meanwhile, S will continue to send the second layer to T1, so the time of sending the first and second layer is overlapped.

While a serial chain is sufficient for efficient parameter broadcasting for nodes that are connected with the same bandwidth links, multiple chains are necessary in a leaf-spine network where inter-leaf bandwidth may vary and in our live scale setup. This is because (1) multi-chain avoids relatively slow inter-leaf communications if each leaf switch has sources and targets (Line 6–7), and (2) it enables more interference-free live scaling especially in PD disaggregation setting. Figure 12 illustrates the latter: suppose we want to scale up two prefill instances in a live manner, the KVCache will be transferred to the decode instances once prefill is done. With a single chain, only $T2$ can join the live scale without network interference, because at $T1$, the KVCache transfer (①) interferes with the parameter forward traffic (②). With

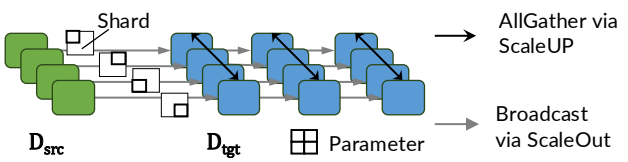

Figure 14: *An illustration shareded parameter transfer with scale up network.*

two chains backed by two parameter sources (b), both $T1$ and $T2$ can live scale without interference.

Note that the order of nodes in the chain is important: we chose a decreasing order with respect to the aggregated link speed between nodes (Line 2, 5). This is because sending to nodes with higher bandwidth achieves a faster increase in the serving throughput. As shown in Figure 13 (b): suppose the source (S) sends parameters to $T2$ twice as fast as $T1$. A chain order of $S \rightarrow T2 \rightarrow T1$ is better than $S \rightarrow T1 \rightarrow T2$ because the downtime of $T2$ is only half. Note that the source can be a group of GPUs because GPUs typically have dedicated network cards in our setup.

Optimization: parallel sharded parameter transfer from multiple sources. For a broadcast chain where the source and target contain GPUs with duplicated parameters, we further leverage the scale-up network to parallelize a transfer link. Figure 14 shows a concrete example. Suppose the source and target nodes have four GPUs each. For such a transfer, each source GPU only needs to forward 1/4 of the sharded parameters to each target GPU, where the target GPUs can use NVLink-based AllGather to get the full parameters. This reduces the scaling time to 1/4 as the NVLink AllGather time is negligible.

5.2 Efficient live autoscaling with ZigZag scheduling

Selecting instances for live scaling. After getting the chains from Algorithm 11, we select instances to participate in live autoscaling based on the following criteria: (1) the necessity of live autoscaling, i.e., when a stop-the-world scaling will cause SLO violation and (2) the presence of overloaded instance that can cooperate. Both are readily available: (1) we can profile the relationship between load speed and SLO violation in Figure 3 for the judgement and (2) autoscaling is typically triggered when the system is overloaded. Thus, for each overloaded instance, we will identify an instance in the chain that satisfies (1), typically the tail instances in the chains as it has the slowest link.

Live autoscale protocol with paired instances. Suppose we have selected a new instance (inst.1) to offload computations from an overloaded instance (inst.0). To begin live autoscaling, we use a three-step transition protocol: (1) Once inst.1 starts loading parameters, we redirect all queued and new requests from inst.0 to it for execution. The redirection time is negligible because the request payloads are much smaller than the model. (2) After the first layer is loaded on inst.1,

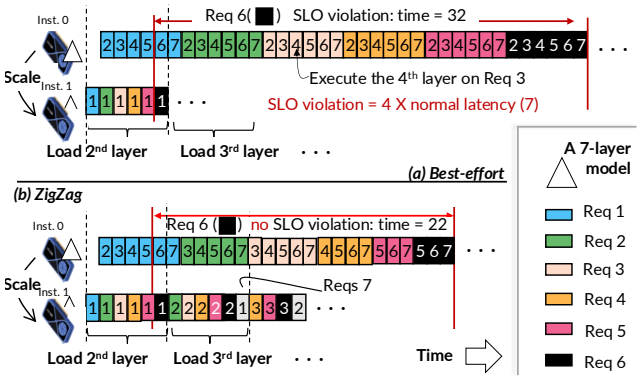

Figure 15: *An illustration of the necessity of ZigZag scheduling. Note that the execution starts when the first layer has been loaded to instance 1 (inst.1). Our example assumes the time of loading a layer can perform 6-layer computations.*

it begins executing the first layer of all requests. Note that during the loading of the first layer, inst.0 remains active by processing its pending requests. Finally, when the model has completed loading on inst.1 (3), requests will be re-distributed evenly between both instances.

The scheduling problem. A key issue to address in the above (3) is how to best utilize inst.1 to maximize the goodput during live autoscaling. Specifically, we should decide a pipeline configuration for each request batch, i.e., how many layers to execute on inst.1 and inst.0, respectively. One naive policy is best-effort: for each batch, we execute as many layers as possible on inst.1 (not exceeding half) and execute the rest on inst.0. While it adapts configurations with model loading, we found it is suboptimal because inst.1 has limited serving capacity during initial loading, so most requests are still queued at inst.0, causing SLO violations. Figure 15 (a) shows a concrete example of a 7-layer model executed with the best-effort scheduling. The load time of one layer can do 6-layer computations, a common setup (e.g., Llama2-7B model with a moderate batch size of 2000 prefill tokens under 200 Gbps RDMA network). Thus, before the second layer has been loaded to inst.1, the current request batches (req 1–6) can only use a $(1, 6)$ pipeline configuration. However, request 6 will suffer from SLO violation due to waiting for requests 1–5 to be completed on inst.0, as their execution time has reduced a little due to the imbalanced loads.

The ZigZag scheduling. To address this issue, our observation is that by delaying request scheduling on inst.0, inst.1 will have more layers coming, opening opportunities to balance the workload between instances. Figure 15 (b) shows this: After requests 2–5 have been executed on inst.1, we delay their execution on inst.0 and wait for the second layer to come. This allows us to adopt a more aggressive pipeline configuration $(2, 5)$ for them. Note that the delay won't waste GPU because we can schedule pending requests (e.g., 6). After the second layer has been loaded, inst.1 can come back (thus, in a ZigZag way) to execute the second layer of re-

quests 2–5. Thus, the second layer execution of requests 3–6 is overlapped with the execution of layers 3–6 for request 2. Thanks to this overlap, the overall inference time of request 7 decreases from 32 to 22, now within the SLO.

The above ZigZag scheduling can be formulated as follows. Assuming a first-come-first-serve (FCFS) scheduling policy, the de facto for serving [44, 75, 77]. For ease of presentation, we first assume non-LLM and then extend to LLM in §5.4. The scheduling has two parts:

(1) Pipeline configuration. Given N request batches with equal execution time, we first determine the pipeline configuration $(\mathrm{T}_i, \mathrm{S}_i)$ for them, where T_i and S_i are the number of layers to be executed on the target and source GPU for request i, respectively. The goal is to minimize the average latency, which can be formulated with the following Integer Linear Program (ILP):

$$\text{Latency}_{\text{avg}} = (\sum_{req=1}^{N}\sum_{i=1}^{req}\mathrm{S}_i)/N$$

To see why such a formula holds, consider the example in Figure 15 (b). Each request's latency is the time the source instance finishes its part of the computation, which includes its own execution time and the sum of its previous requests' time (queueing time). We only need to consider previous requests because they are executed in a FIFO order. In our example, request 3's latency is 17 (12 for requests 1 and 2's execution and 5 for its own). For non-LLM, the execution time of each layer is the same if the batch size is the same. Note that we omit the activation transfer latency since it is negligible.

The problem has the following constraints:

$$\begin{aligned}
\min\quad & \text{Latency}_{\text{avg}} \\
\text{s.t.}\quad & \mathrm{S}_i + \mathrm{T}_i = L, \quad \forall i \quad \text{Pipeline limit } \textbf{(C1)}, \\
& \sum_{j=1}^{i}\mathrm{T}_j \le \sum_{j=1}^{i-1}\mathrm{S}_j, \forall i > 1 \quad \text{Pipeline dependency } \textbf{(C2)}, \\
& \text{Time}_l * \mathrm{T}_i \le \sum_{j=1}^{i-1}\mathrm{T}_j + (N-i+1)\times(\mathrm{T}_i - 1), \\
& \forall i > 1 \quad \text{Load limit } \textbf{(C3)}
\end{aligned} \tag{1}$$

C1 ensures that the pipeline should be fully executed. **C2** states pipeline dependency: when the source instance executes request i's S_i layers, the target instance must finish the execution of T_i. The start execution time of request i on the source is $\sum_j^{i-1}\mathrm{S}_j$. The finish time of i on the target instance is $\sum_j^{i-1}\mathrm{T}_j + \mathrm{T}_i$, which simplifies to $\sum_j^{i}\mathrm{T}_j$. Finally, **C3** ensures that once the target instance request's i's T_i layers, all these layers must be loaded, where Time_l is the time to load one layer normalized to the execution time of one layer in pipeline. The term $(N-i+1)\times(\mathrm{T}_i-1)$ indicates that the load time can be overlapped with executing of the succeeding requests of i.

Input: **Q**: an atomic distributed priority queue that stores the requests to be scheduled. The priority is defined as follows: for **req i** and **req j**, the **P(i) > P(j)** if and only if **i < j** and **i** have loaded layers unexecuted. The subscription (e.g., **i**) indicates the request's arrival time.

At New Instance (target instance)

```
0  spawn(update_Q_when_layers_come) ▸ Run in the background
1  while execute under live:
2  |   q = Q.get_front()
3  |   forward_one_layer(q)
```

At Old Instance (source instance)

```
4  while execute under live:
   |   ▸ Pull the pending requests with
5  |   If q, activation = pull_the_earlist_request(Q):
6  |   |   forward_all(q, activation)
```

Figure 16: *The pseudocode of the ILP-free ZigZag scheduling.*

While solving this ILP is NP-hard, it remains manageable (less than 40 ms to solve for Llama3-8B) because models typically have only a few dozen layers. Additionally, we only need to configure the pipeline for the batches of requests executed during parameter loading, which is a dozen of so in practice. Nevertheless, to further eliminate the solving time for models with more layers (e.g., 80 layers for Qwen-72B), we also derived an ILP-free method that we described below.

(2) Scheduling requests in an ILP-free ZigZag way. Specifically, we found that by delaying sending the requests on the source instance and letting the target instance execute the requests once it is free, we can achieve ZigZag scheduling without solving the ILP. Figure 16 shows the pseudocode of how we schedule the requests on both instances. The new instance maintains a priority queue (that can be pulled by the source instance via RPC) for all requests, where the priority is defined by (1) the FCFS order and (2) requests with next-to-execute layer loaded coming first. Once the new instance has executed one layer (Line 3), we keep the executed request in the queue so it can be scheduled back once more layers are loaded. The requests are scheduled on the source instance only if it is not overloaded, i.e., has no pending requests (Line 5). Thus, if the source instance is busy, the request will still be executed on the target instance.

5.3 Global parameter pool and scaling policy

Global parameter pool and local memory cache. Our global parameter pool tracks the locations of the model parameters across deployed GPUs and host CPUs with local memory cache. To ensure at least one copy of the model parameters is available in the memory of GPU or host at the cluster scale, during system initialization, we distribute one copy of the model's parameters evenly to the CPU hosts and track their locations at a centralized manager. When a model is deployed to or reclaimed from a GPU, we further update the locations in the manager, and reclaim/reload cached copies on the host cache.

Scaling policy. Our paper focuses on the autoscaling mechanism, which is orthogonal to the autoscaling policies, includ-

ing collecting workload metrics with workload monitoring and determining how many new instances to scale based on these metrics. Our current implementation follows prior works [62, 2] that first records the serving loads with tokens per second and KVCache usage globally.

For scaling up, when the average monitored load surpasses a pre-defined upper bound, we allocate sufficient instances to meet that demand. The upper bound can be derived by profiling the average serving load per-instance offline. We leave a more detailed explanation in another paper. For scaling down, we follow previous works'[56, 29] timeout-based policy: when the average monitored load falls below a lower bound in a time window, we shut down some instances and revoke all GPUs assigned to them. Given BLITZSCALE's rapid autoscaling capabilities, we adopt an extremely short sub-second level timeout.

5.4 Specializations and optimizations for LLM

While most techniques described above work for all models following a layer-by-layer architecture, the unique characteristics of LLMs especially for LLMs served with PD disaggregation require several specializations and optimizations.

Retrofitted live pipeline scheduling formulation. Our formulas and constraints described in §5.2 cannot be directly applied to LLMs because the prefill and decode time of a layer is approximately linear to the total batched token size [55, 75]. For prefill-only live scheduling, e.g., autoscaling a prefill instance in PD disaggregation, we fix the formulation by adding a regulation parameter for each request batch by profiling its execution time with the counts, similar to a priori work [75]. A more tricky case involves handling decoding, e.g., when scaling instances that combine prefill and decode, or scaling a decode instance in PD disaggregation. The complexity arises because decode batch size changes dynamically due to its auto-regressive nature. Fortunately, our ILP-free scheduling method can also work for decoding.

Supporting PD colocation. We seamlessly support PD colocation since a PD-colocated instance is a normal model instance. Meanwhile, our ILP-free ZigZag scheduling also applies to pipelined execution under PD colocation [5].

Live scaling decode instances in PD disaggregation. Live scaling a decode instance in PD disaggregation without interference is impossible due to the incast bandwidth contention of both parameter loading and KVCache transfer. Thus, we leverage the fact that the prefill and decode instances share the same model parameters, so we can live scale a decode instance by first mutating some prefill instances to decode instances, while concurrently live scaling the prefill instances to compensate for the prefill throughput.

Optimized scaling policy for PD disaggregation. Prescaling instances can hide the cost of scaling, but a too-early scaling wastes GPU resources. In PD disaggregation, we found we can pre-scale decode instance at zero cost, because the need for scaling decode instances can be evidenced by the requirement for scaling prefill instances. Specifically, once we found a significant requirement for scaling prefill instances, we will simultaneously scale decode instances. This effectively hides the scaling cost of decode instances, and is even effective for other systems like ServerlessLLM [29], see §6.1.

	Cluster A (m x g)	**Cluster B** (m x g)
GPU	A800 80 GB (4x8)	A100 80 GB (2x8)
GPU-GPU (intra)	1.6 Tbps NVLink	256 Gbps PCIe
GPU-GPU (inter)	100 Gbps RDMA	100 Gbps RDMA
Host-GPU	128 Gbps PCIe	128 Gbps PCIe
SSD-GPU	10 Gbps	10 Gbps

Table 1: *Evaluation clusters. m is the number of hosts and g is the number of GPUs per host.*

6 Evaluation

System implementation. BLITZSCALE is a MAAS system capable of serving both traditional models and LLMs with 24,000 lines of Rust and C++ code. It builds upon widely applied LLM optimizations like PD disaggregation and continuous batching. We leverage existing highly-optimized serving system components (with no autoscaling support) wherever possible. For instance, all our GPU kernels for LLM come from FlashInfer [1]. We choose a native-language-based framework implementations because we found it is challenging to implement fine-grained scheduling in Python. §A provides more implementation details.

Testbed. Our evaluations are conducted on two testbeds listed in Table 1. Cluster A can serve larger models (e.g., 72 B) with tensor parallelism [44] thanks to the NVLink while cluster B is more suitable for serving single-GPU models.

Evaluated traces and models. Because the scaling requirements are closely related to the incoming request rates, we chose three typical real-world traces: BurstGPT [71], AzureCode and AzureConv, both from Azure [55]. The detailed trace shapes are shown in the first column in Figure 17. Since the traces are collected from clusters with different serving capabilities, we follow the standard approach [46, 8, 36] to scale the traces to fit our clusters. Specifically, we scale the trace with temporal pattern preserved using TraceUpscaler [57], and the scaled average request rate is half of the maximum serving capacity of our cluster.

For models, we focus on evaluating LLMs because other non-LLMs are much smaller and trivially scale efficiently with BLITZSCALE. Specifically, we choose Llama3-8B, Mistral-24B and Qwen2.5-72B, all are popular LLM models with high accuracy. Since BLITZSCALE is only sensitive to the model size, we may omit the detailed model family name and only uses their sizes in the following description for simplicity. For small model (8B), it only needs one GPU

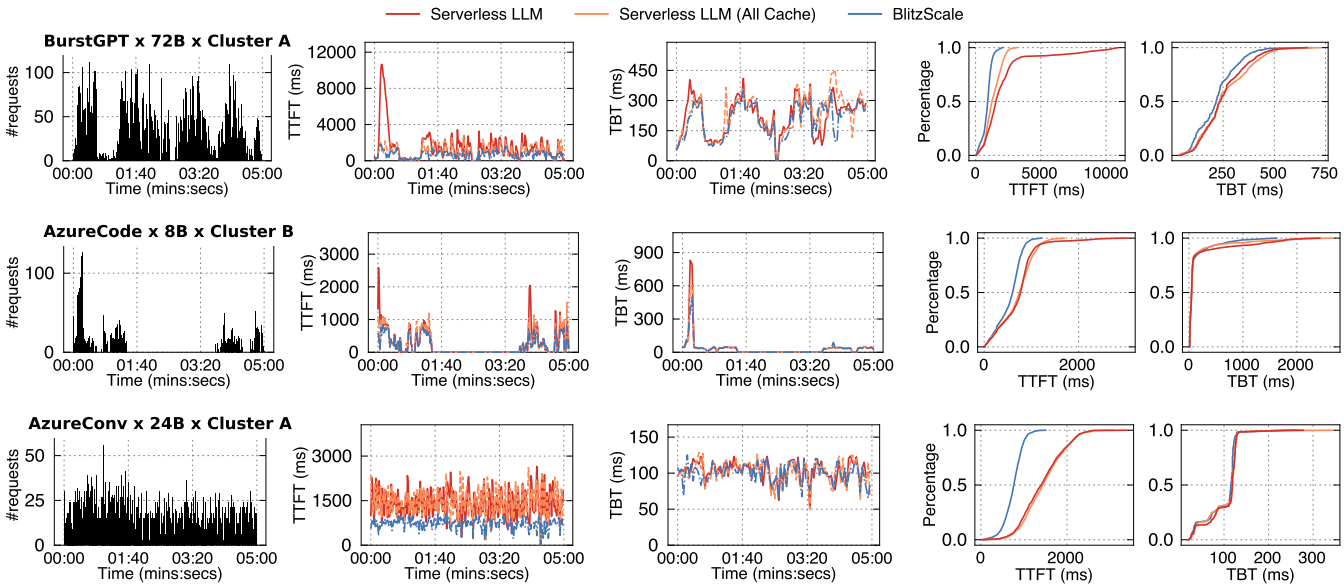

Figure 17: *End-to-end performance comparison between* BLITZSCALE *and ServerlessLLM on various workloads, models and clusters.*

per instance while for 72 B models the minimal number of GPUs used by one instance is 4.

Comparing targets. Without explicit mention, we compare BLITZSCALE with the following baselines:

1. ***ServerlessLLM (S-LLM)*** ***[29]*** is the state-of-the-art MAAS with a focus on accelerating autoscaling speed. It utilizes host memory to cache recently loaded models with a time-to-live eviction policy. Under cache misses, it loads parameters from SSD with SSD bandwidth fully utilized.
2. ***ServerlessLLM optimal (AllCache)*** is the autoscaling speed optimal version of ServerlessLLM that always loads the parameters from the host cache.
3. ***DistServe*** [81] is the state-of-the-art LLM serving system without autoscaling support. It leverages PD disaggregation. We chose it because autoscaling is more challenging in PD disaggregation due to the complexity of multiple instances scaling (prefill and decode) and the need to avoid scaling interference. We compare with other common PD colocation systems like vLLM [44] in §6.4.

For a fair comparison, we adopted the same scaling policy for both BLITZSCALE and variants of S-LLM.

6.1 Autoscaling performance under real-world traces

Due to space limitations, for each model, we choose one trace on one of the clusters to evaluate the performance. Figure 17 presents the end-to-end performance when serving with a prefill and decode disaggregation setup where the instances for different phases are scaled independently. The first column shows the request rate of the trace, the second and third columns show the mean TTFT and TBT, respectively, where each point is the average latency measured during a small time window (1s), and the final two columns present the cumulative distribution function (CDF) of the TTFT and TBT during the evaluation period, respectively. We focus on comparing with S-LLM and AllCache in this section and leave the comparison with DistServe in the next section, as it does not support autoscaling.

Overall performance. First, we can see that BLITZSCALE has the lowest TTFT and TBT in all workloads thanks to the fast autoscaling speed. Specifically, on BurstGPT, the TTFT is 75.5 % and 21.1 % shorter than S-LLM and AllCache, respectively, and the TBT is 7.4 % and 5.1 % shorter, respectively. Nevertheless, the degrees of improvement are different across metrics due to the unique characteristics of the prefill and decode phases. Meanwhile, the behaviors of systems, especially S-LLM, are different across workloads due to the different request arrival patterns (see the first column). We elaborate on the differences in the following.

TTFT vs. TBT. BLITZSCALE is more effective in reducing the TTFT than TBT on all workloads. This is due to two reasons. First, the decode instance can be pre-scaled thanks to our optimized policy (§5.4), which we apply to all baselines. Specifically, when the prefill throughput increases, BLITZSCALE (and its baselines) will simultaneously scale the decode instances, yet no more decode instances are needed at the scale time. Thus, the scaling time is overlapped with the prefill time, which hides some scaling overheads. Second, decode scales less than prefill because as long as there is sufficient memory on the decode instances, all systems can handle decoding with a slightly increased TBT due to no queueing. Since all models adopt modern LLM optimization group query attention [6] with low memory footprint, decoding instances are more sufficient than prefill instances. Nevertheless, BLITZSCALE still achieves a 5.1–7.4 %, 88.3–

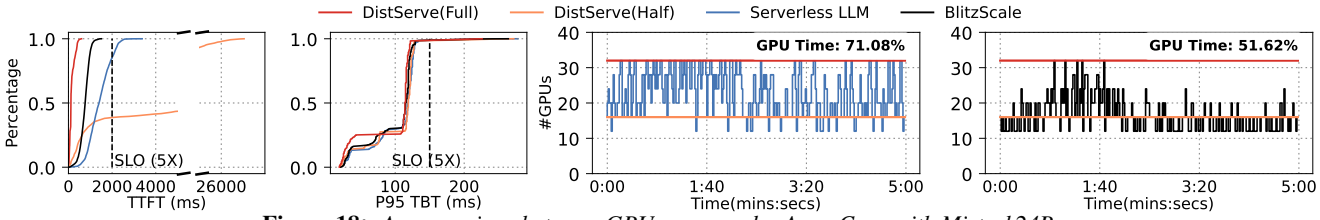

Figure 18: *A comparison between GPU usage under AzureConv with Mistral 24B.*

94.1 % and 0.7–1.8 % shorter TBT than S-LLM and AllCache on three workloads, respectively.

Comparisons between different workloads results. BLITZSCALE always outperforms AllCache thanks to fast network-based autoscaling as well as live scaling, but S-LLM has different behaviors compared to AllCache in these workloads. On BurstGPT, S-LLM first has a sharp TTFT spike at the first burst (time 0:05), while it is close to AllCache in future bursts, because future bursts can benefit from host cache. In comparison, on AzureCode, S-LLM has spikes under both bursts (time 0:05 and time 03:25), because the gap between two bursts makes the host cache evicted due to a time-to-live policy. Finally, on AzureConv, since the bursts continuously arrive, S-LLM always hits the host cache, so the performance—see the CDF graphs—is similar to AllCache.

6.2 Performance and resource usage

Comparison with non-autoscaling systems. We first compare BLITZSCALE with DistServe. Since DistServe does not support autoscaling, its performance is highly dependent on the number of provisioned instances. Therefore, we evaluate two setups: DistServe (full) uses all GPUs in our cluster and represents an optimal performance at the cost of GPU waste. On the other hand, DistServe (half) uses GPUs with the average number of instances required to handle all the workloads within the evaluation period. For simplicity, we only present the results on AzureConv on 24B models, the overall trends are similar. We have carefully calibrated DistServe's performance, such that when autoscaling is disabled in BLITZSCALE, DistServe has the same performance as BLITZSCALE in all setups.

The first two columns of Figure 18 present the latency results. First, it can be observed that DistServe (full) has the best performance, because the GPU is over-provisioned so it doesn't suffer from queueing or scaling overhead. Nevertheless, BLITZSCALE still achieves the same service level objective (SLO) as DistServe (full) while S-LLM incurs 18.7 % SLO violations. We follow the traditional 5 × SLO [81] since all our workloads (chat and code generation) are latency-sensitive. Specifically, if a request's end-to-end (TTFT or TBT) latency is exceeds 5 × the average latency, it violates the SLO. Finally, DistServe (half) has the poorest performance: on average, BLITZSCALE has a 95.8 % and 1 % shorter TTFT and TBT than DistServe (half). BLITZSCALE achieves this by using the same GPU time for serving this model as DistServe

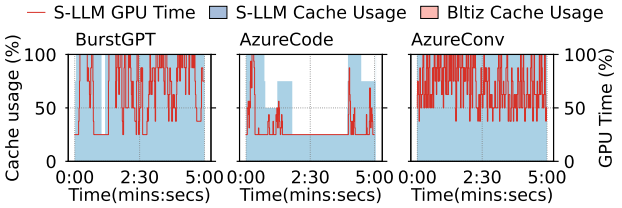

Figure 19: *A comparison of host cache usage on S-LLM and* BLITZSCALE *under the evaluated workloads.*

(half), and this time is 50 % smaller than DistServe (full), which we elaborate next.

GPU time used. The last two columns of Figure 18 show the GPU time used by S-LLM and BLITZSCALE, respectively. For S-LLM and BLITZSCALE, we collected the aggregated GPU usage at each time point for both prefill and decode instances, and the overall time is calculated by integrating the area under the curve. For variants of DistServe, their GPU time is constant across the evaluation period. We can see that BLITZSCALE has 19.46 % lower GPU time than S-LLM thanks to the fast autoscaling capability: with low scaling speed, there would be more queued requests, so the system would trigger more scaling operations that use more GPU time. This is unnecessary with BLITZSCALE. Even with less GPU time used, BLITZSCALE has a 48.1 % and 1.8 % shorter TTFT and TBT than S-LLM, respectively.

Host cache usage. Compared to ServerlessLLM, BLITZSCALE also consumes less host memory for parameter caching. Figure 19 reports the host memory usage for different systems. We omit AllCache and DistServe, as AllCache always fully replicates parameters to all hosts while DistServe does not need caching. We normalize the host cache usage as different workloads use different clusters. The results deliver two messages. First, BLITZSCALE only needs minimal host caching (less than one) to achieve fast autoscaling: this is as expected by our design because we prefer to load parameters from GPUs of instances that serve the model, and even when no serving instance is available, we only need one host copy due to the network-based multicast. Second, the memory usage of ServerlessLLM is proportional to the number of hosts involved in the serving, so a model can quickly "pollute" the host cache. This is non-optimal for an MAAS system because it can simultaneously serve many models while the host cache is limited.

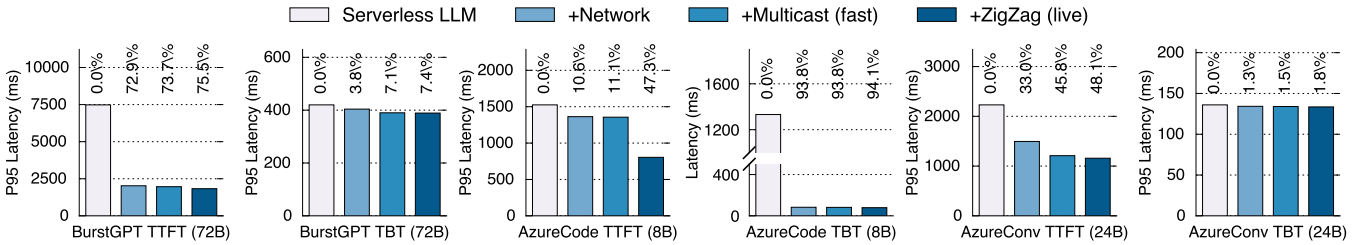

Figure 20: *An ablation study on the effectiveness of our proposed techniques.*

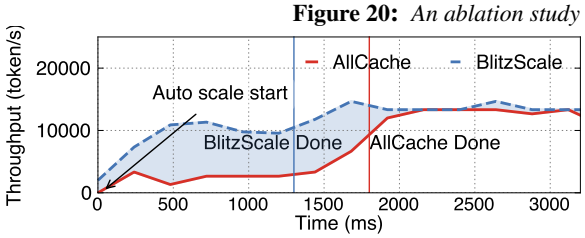

Figure 21: *A detailed look at how* BLITZSCALE *and AllCache scale a 24 B model to 6 prefill instances on cluster A.*

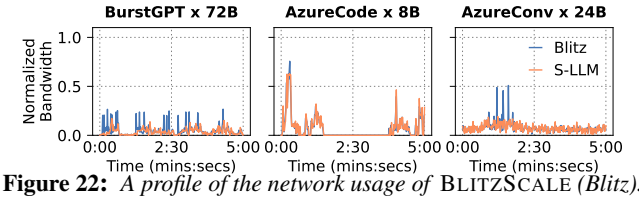

Figure 22: *A profile of the network usage of* BLITZSCALE *(Blitz).*

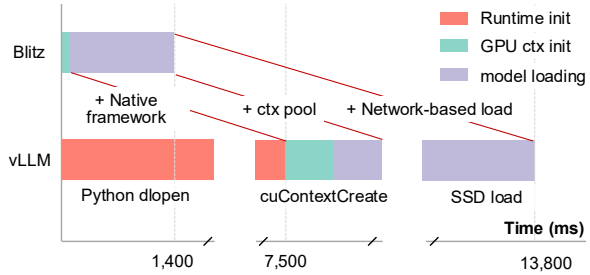

Figure 23: *A comparison of init time of* BLITZSCALE *and vLLM*

6.3 Detailed performance analysis

A detailed look at the live scale. Figure 21 shows a throughput timeline when using BLITZSCALE and AllCache to scale six 24 B prefill instances. BLITZSCALE utilizes two broadcast chains (each involving 3 instances), while the end instances involve a live autoscale. The start instances are the decode instances. For AllCache, it directly loads the parameters from the host memory of the scaled instances. We can see that first, even with a few loaded layers (e.g., at time 500 ms), BLITZSCALE can gradually emit tokens as a result of live execution. Second, BLITZSCALE can scale faster even compared with AllCache, thanks to our NVLink-based fused link transmission protocol: it can finish scaling in 1,200 ms while AllCache takes about 2,000 ms.

Ablation study. Figure 20 conducts an ablation study on the effectiveness of our proposed techniques. We measured the effectiveness by incrementally enabling different techniques and reporting the results on the three workloads: "+Network" leverages fast computing network instead of SSD for autoscaling, "+Multicast (fast)" further applies our optimized parameter broadcast protocol described in §5.1, while "+ZigZag (live)" enables live autoscaling of §5.2.

First, we can see that all techniques are effective in improving the end-to-end serving performance, but the degrees differ across workloads. First, "+Network" improves the scaling performance in all workloads thanks to the higher bandwidth for the autoscaling data plane. Second, "+Multicast (fast)" is effective in AzureCode and AzureConv, but it is less effective in BurstGPT due to the limitations of our cluster (up to 8 instances can be scaled on 72 B model), so there are no cases to simultaneously scale multiple instances, which is the targeted case for this technique. Live autoscaling is mostly effective in AzureCode because it is evaluated on a cluster with slow networking (Cluster B). Finally, our techniques are not such effective on decoding because decode instances are sufficient in most cases, which we have discussed in §6.1. One exception is AzureCode: in this workload, the prefill throughput increases slower than others (see the first column of Figure 17), so the decode instances are triggered later. As a result, the slow scale of ServerlessLLM's SSD cannot be hidden, making a faster scaling more beneficial.

Network usage. Figure 22 shows the network usage of BLITZSCALE and S-LLM: we can see that though BLITZSCALE leverages compute network for autoscaling and the scale frequency is high (see the last column of Figure 18), the additional network usage is negligible.

Control plane vs. data plane of model autoscaling. Figure 23 compares the control plane and data plane overhead during model autoscaling with vLLM. We can see that with proper optimizations, the control plane overhead is negligible.

6.4 Performance under LLM PD colocation

Finally, Figure 24 compares the performance of BLITZSCALE and vLLM where the serving is conducted in PD colocation on BurstGPT workloads with Llama2-7B model. The general trend is similar to PD disaggregation: BLITZSCALE has comparable performance with over-provisioned vLLM, while compared with an average provisioning, BLITZSCALE has a 0.24× shorter P99 TTFT. Interestingly, we found BLITZSCALE has even shorter tail TTFT compared with over-provisioned vLLM, because our scheduling framework is optimized for cluster serving.

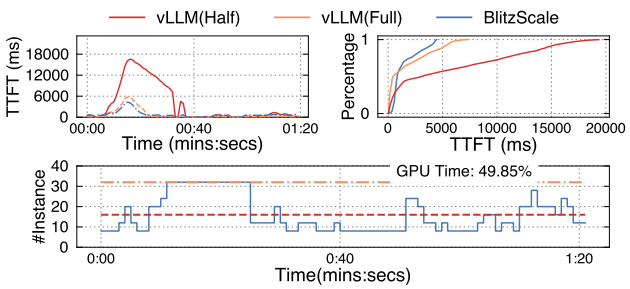

Figure 24: *A comparison of* BLITZSCALE *and vLLM on the BurstGPT workload.*

7 Related work

Optimizing model serving without autoscaling. Serving models at scale is non-trivial. A significant body of research focuses on how to efficiently utilize GPUs to accelerate model serving [81, 75, 44, 45, 77, 55, 31, 68], e.g., Orca [77] proposes iterative-scheduling and selective batching. AlphaServe [45] employs pipeline parallelism to better handle load spikes, but it cannot adjust pipeline instances dynamically. These systems assume running on a fixed pool of GPUs, and we have shown the necessity of dynamically adjusting pool size and how to achieve so efficiently with BLITZSCALE. BLITZSCALE complements these single-instance model serving systems with fast autoscaling mechanisms: we build upon them for fast model serving on a single instance, and additionally provide ultra-fast scaling when the system needs to change the number of serving instances.

Dynamic scaling serving instances. Dynamically scaling serving instances is challenging, mainly because the size of model weights is huge and still increasing, so loading them to the accelerators (data plane) is time-consuming. Some existing works accelerate the loading [15, 41, 46, 62]: For example, both PipeSwitch [15] and DeepPlan [41] leverage the layer-by-layer character of models to overlap the inference and parameter loading to hide the loading cost. They only focus on host-to-device loading and such overlap is not live especially when the models are large. SpotServe [46] and Llumnix [62] realize live migration but migration cannot fully unleash the computing capabilities of both instances. BLITZSCALE provides a new mechanism to scale serving instances lively during parameter loading, resulting in throughput increase even with unfinished loading, which we have shown critical in reducing latencies under bursty workloads.

A concurrent work λScale [78] also focuses on using network to accelerate model autoscaling. The key difference is that λScale scatters the parameters scaling on multiple instances to reduce the time for scaling at the cost of decreased serving throughput, while BLITZSCALE seamlessly scales full parameters on all instances with a similar speed, yet does not sacrifice the throughput thanks to our multicast-chain-based scaling. Moreover, during scaling BLITZSCALE has a gradually increasing throughput thanks to our live scaling while λScale is still a stop-the-world approach.

Accelerating coldstart in serverless computing. Accelerating model scaling builds upon coldstart acceleration in serverless computing [51, 7, 60, 26, 64, 58, 67], which focuses on starting general-purpose computing instances like containers. We built upon these works, e.g., for accelerating container startup time, yet designed efficient network-based live autoscale tailored for model scaling with the domain-specific knowledge of model serving.

8 Conclusion and Future Work

Autoscaling is the key to achieving both high goodput and hardware utilization in model as a service systems, but the performance overhead introduced by current slow and stop-the-world autoscaling significantly limits its effectiveness. In this paper, we first show that the data plane of model autoscaling can be made fast with less than $O(1)$ caching by leveraging network-based model-aware multicast. We next show that the data plane can be made live through model-aware remote execution. Equipped with these two techniques, our system BLITZSCALE has at most 94.1 % better performance and 19.46 % better resource utilization compared to state-of-the-art serving systems with and without autoscaling, respectively. We believe our work demonstrates the potential and practicability of autoscaling-empowered model as a service systems.

While fast and live autoscaling of BLITZSCALE takes a key step toward modern elastic serving systems, several challenges remain. First, during our investigations, we found scaling policies—determining when and how to scale—may also impact system efficiency. The policy depends heavily on workload characteristics, which we leave as future work. Second, BLITZSCALE currently focuses on instance-level scaling, whereas modern models can scale by changing the parallel configuration within an instance, e.g., scaling experts in mixture-of-experts (MoE) models. While BLITZSCALE in principle works for such a setup, we leave the detailed exploration in the future work.

Acknowledgment

We would like to thank OSDI reviewers and our shepherd for their insightful feedback. We sincerely thank Qinwei Yang, Xinhao Luo, and Mingcong Han for giving us feedback on GPU communication library, kernel, driver and runtime. We also thank Xiating Xie, Chenhan Wang and Sundi Guan for refining the presentation of this paper. We thank Alibaba Tongyi Lab for providing the testbed during the early stage of this work. This work was supported in part by the National Key Research & Development Program of China (No. 2022YFB4500700), the Fundamental Research Funds for the Central Universities, the National Natural Science Foundation of China (No. 62202291, 62272291), as well as a research grant from Huawei Cloud.

References

[1] FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer, 2024.

[2] Kubernetes horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale, 2024.

[3] Standardized serverless ml inference platform on kubernetes. https://github.com/kserve/kserve, 2024.

[4] Qwen series. https://github.com/QwenLM/Qwen, 2025.

[5] AGRAWAL, A., KEDIA, N., PANWAR, A., MOHAN, J., KWATRA, N., GULAVANI, B., TUMANOV, A., AND RAMJEE, R. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 117–134.

[6] AINSLIE, J., LEE-THORP, J., DE JONG, M., ZEMLYANSKIY, Y., LEBRÓN, F., AND SANGHAI, S. GQA: training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023* (2023), H. Bouamor, J. Pino, and K. Bali, Eds., Association for Computational Linguistics, pp. 4895–4901.

[7] AKKUS, I. E., CHEN, R., RIMAC, I., STEIN, M., SATZKE, K., BECK, A., ADITYA, P., AND HILT, V. SAND: towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 923–935.

[8] ALI, A., PINCIROLI, R., YAN, F., AND SMIRNI, E. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow. 15*, 10 (2022), 2071–2084.

[9] AMAZON. Amazon ec2 accelerated computing instances. https://docs.aws.amazon.com/ec2/latest/instancetypes/ac.html.

[10] AMAZON. Amazon bedrock. https://aws.amazon.com/bedrock/, 2024.

[11] ANYSCALE. Ray serve: Scalable and programmable serving. https://docs.ray.io/en/latest/serve/index.html, 2024.

[12] ARAPAKIS, I., BAI, X., AND CAMBAZOGLU, B. B. Impact of response latency on user behavior in web search. In *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014* (2014), S. Geva, A. Trotman, P. Bruza, C. L. A. Clarke, and K. Järvelin, Eds., ACM, pp. 103–112.

[13] AWS. Amazon rekognition. https://aws.amazon.com/en/rekognition/, 2024.

[14] AZURE. Azure llm inference traces. https://github.com/Azure/AzurePublicDataset, 2024.

[15] BAI, Z., ZHANG, Z., ZHU, Y., AND JIN, X. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 499–514.

[16] BANIKAZEMI, M., MOORTHY, V., AND PANDA, D. K. Efficient collective communication on heterogeneous networks of workstations. In *1998 International Conference on Parallel Processing (ICPP '98), 10-14 August 1998, Minneapolis, Minnesota, USA, Proceedings* (1998), IEEE Computer Society, pp. 460–467.

[17] BEHRENS, J., JHA, S., BIRMAN, K., AND TREMEL, E. RDMC: A reliable RDMA multicast for large objects. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018* (2018), IEEE Computer Society, pp. 71–82.

[18] BHAT, P. B., RAGHAVENDRA, C. S., AND PRASANNA, V. K. Efficient collective communication in distributed heterogeneous systems. *J. Parallel Distributed Comput. 63*, 3 (2003), 251–263.

[19] CAI, Z., LIU, Z., MALEKI, S., MUSUVATHI, M., MYTKOWICZ, T., NELSON, J., AND SAARIKIVI, O. Synthesizing optimal collective algorithms. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021* (2021), J. Lee and E. Petrank, Eds., ACM, pp. 62–75.

[20] CLOUD, A. Build with generative ai on alibaba cloud. https://www.alibabacloud.com/zh/solutions/generative-ai/build, 2024.

[21] CLOUD, A. The model list. https://help.aliyun.com/zh/model-studio/getting-started/models?spm=a2c4g.11186623.help-menu-2400256.d_0_2.4bb8b0a8ClhJuI&scm=20140722.H_2840914._.OR_help-T_cn%23DAS%23zh-V_1, 2024.

[22] COWAN, M., MALEKI, S., MUSUVATHI, M., SAARIKIVI, O., AND XIONG, Y. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023* (2023), T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds., ACM, pp. 502–514.

[23] DEEPINFRA. Custom llms. https://deepinfra.com/docs/advanced/custom_llms, 2024.

[24] DEEPSEEK. Deepep. https://github.com/deepseek-ai/DeepEP, 2025.

[25] DELL. Ai fabrics. https://infohub.delltechnologies.com/en-us/l/dell-technologies-ai-fabrics-overview-1/ai-fabrics-3/3/. 2025.

[26] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 467–481.

[27] ENGINE, V. Volcano engine accelerated computing instances. https://www.volcengine.com/docs/6459/72363.

[28] FACE, H. The ai community building the future. https://huggingface.co, 2024.

[29] FU, Y., XUE, L., HUANG, Y., BRABETE, A., USTIUGOV, D., PATEL, Y., AND MAI, L. Serverlessllm: Locality-enhanced serverless inference for large language models. *CoRR abs/2401.14351* (2024).

[30] G., C. N., ET AL. Virtual link trunking for network configuration and resource optimization. U.S. Patent Application US20160301608A1, oct 2016. https://patents.google.com/patent/US20160301608A1/en.

[31] GAO, B., HE, Z., SHARMA, P., KANG, Q., JEVDJIC, D., DENG, J., YANG, X., YU, Z., AND ZUO, P. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 111–126.

[32] GHAZIMIRSAEED, S. M., ZHOU, Q., RUHELA, A., AND BAYATPOUR, M. A hierarchical and load-aware design for large message neighborhood collectives. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020* (2020), C. Cuicchi, I. Qualters, and W. T. Kramer, Eds., IEEE/ACM, p. 34.

[33] GIGASPACES. Amazon found every 100ms of latency cost them 1% in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales, 2024.

[34] GITHUB. Accelerate your development speed with copilot. https://copilot.github.com, 2024.

[35] GOOGLE. Google accelerator-optimized machine family. https://cloud.google.com/compute/docs/accelerator-optimized-machines.

[36] GUJARATI, A., KARIMI, R., ALZAYAT, S., HAO, W., KAUFMANN, A., VIGFUSSON, Y., AND MACE, J. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 443–462.

[37] HOPPS, C. Analysis of an equal-cost multi-path algorithm. Request for Comments 2992, Internet Engineering Task Force (IETF), nov 2000. Accessed: 2025-05-16.

[38] HU, C., HUANG, H., XU, L., CHEN, X., XU, J., CHEN, S., FENG, H., WANG, C., WANG, S., BAO, Y., SUN, N., AND SHAN, Y. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. *CoRR abs/2401.11181* (2024).

[39] HUANG, J., ZHANG, M., MA, T., LIU, Z., LIN, S., CHEN, K., JIANG, J., LIAO, X., SHAN, Y., ZHANG, N., LU, M., MA, T., GONG, H., AND WU, Y. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024* (2024), E. Witchel, C. J. Rossbach, A. C. Arpaci-Dusseau, and K. Keeton, Eds., ACM, pp. 421–437.

[40] HUANG, Z., WEI, X., HAO, Y., CHEN, R., HAN, M., GU, J., AND CHEN, H. PARALLELGPUOS: A concurrent os-level GPU checkpoint and restore system using validated speculation. *CoRR abs/2405.12079* (2024).

[41] JEONG, J., BAEK, S., AND AHN, J. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023* (2023), G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, Eds., ACM, pp. 249–265.

[42] JEONG, J., BAEK, S., AND AHN, J. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems* (New York, NY, USA, 2023), EuroSys '23, Association for Computing Machinery, pp. 249–265.

[43] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 185–201.

[44] 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 *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 611–626.

[45] LI, Z., ZHENG, L., ZHONG, Y., LIU, V., SHENG, Y., JIN, X., HUANG, Y., CHEN, Z., ZHANG, H., GONZALEZ, J. E., AND STOICA, I. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 663–679.

[46] MIAO, X., SHI, C., DUAN, J., XI, X., LIN, D., CUI, B., AND JIA, Z. 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, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024* (2024), R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds., ACM, pp. 1112–1127.

[47] NARAYANAN, D., SHOEYBI, M., CASPER, J., LEGRESLEY, P., PATWARY, M., KORTHIKANTI, V., VAINBRAND, D., KASHINKUNTI, P., BERNAUER, J., CATANZARO, B., ET AL. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2021), pp. 1–15.

[48] NVIDIA. Doubling all2all Performance with NVIDIA Collective Communication Library 2.12. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/, 2024.

[49] NVIDIA. Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2024.

[50] NVIDIA. Nvidia dgx superpod: Next generation scalable infrastructure for ai leadership. https://docs.nvidia.com/dgx-superpod/reference-architecture/scalable-infrastructure-h200/latest/_downloads/bbd08041e98eb913619944ead1f92373/RA-11336-001-DSPH200-ReferenceArch.pdf#page=8.10, 2024.

[51] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 57–70.

[52] OLUMIDE OLUSANYA AND MUNIRA HUSSAIN. Need for Speed: Comparing FDR and EDR InfiniBand (Part 1). https://dl.dell.com/manuals/all-products/esuprt_software/esuprt_it_ops_datcentr_mgmt/high-computing-solution-resources_white-papers77_en-us.pdf, 2022.

[53] OPENAI. Chatgpt. https://chatgpt.com, 2024.

[54] OPENAI. Creating video from text. https://openai.com/index/sora/, 2024.

[55] PATEL, P., CHOUKSE, E., ZHANG, C., SHAH, A., GOIRI, Í., MALEKI, S., AND BIANCHINI, R. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024* (2024), IEEE, pp. 118–132.

[56] ROMERO, F., LI, Q., YADWADKAR, N. J., AND KOZYRAKIS, C. Infaas: Automated model-less inference serving. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 397–411.

[57] SAJAL, S. M., ZHU, T., URGAONKAR, B., AND SEN, S. Traceupscaler: Upscaling traces to evaluate systems at high load. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024* (2024), ACM, pp. 942–961.

[58] SAXENA, D., JI, T., SINGHVI, A., KHALID, J., AND AKELLA, A. Memory deduplication for serverless computing with medes. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 714–729.

[59] SHAHRAD, M., FONSECA, R., GOIRI, I., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 205–218.

[60] SHILLAKER, S., AND PIETZUCH, P. *FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing*. USENIX Association, USA, 2020.

[61] STABILITY.AI. Activating humanity's potential through generative ai. https://stability.ai, 2024.

[62] SUN, B., HUANG, Z., ZHAO, H., XIAO, W., ZHANG, X., LI, Y., AND LIN, W. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 173–191.

[63] TOGETHER.AI. Inference that's fast, simple, and scales as you grow. https://www.together.ai/products#inference, 2024.

[64] USTIUGOV, D., PETROV, P., KOGIAS, M., BUGNION, E., AND GROT, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 559–572.

[65] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (2017), I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., pp. 5998–6008.

[66] VERSTOEP, K., LANGENDOEN, K., AND BAL, H. E. Efficient reliable multicast on myrinet. In *Proceedings of the 1996 International Conference on Parallel Processing, ICCP 1996, Bloomingdale, IL, USA, August 12-16, 1996. Volume 3: Software* (1996), K. Pingali, Ed., IEEE Computer Society, pp. 156–165.

[67] WANG, A., CHANG, S., TIAN, H., WANG, H., YANG, H., LI, H., DU, R., AND CHENG, Y. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 443–457.

[68] WANG, J., HAN, J., WEI, X., SHEN, S., ZHANG, D., FANG, C., CHEN, R., YU, W., AND CHEN, H. Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)* (July 2025), USENIX Association.

[69] WANG, K. A., HO, R., AND WU, P. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019* (2019), G. Candea, R. van Renesse, and C. Fetzer, Eds., ACM, pp. 39:1–39:16.

[70] WANG, W., GHOBADI, M., SHAKERI, K., ZHANG, Y., AND HASANI, N. Rail-only: A low-cost high-performance network for training llms with trillion parameters. In *IEEE Symposium on High-Performance Interconnects, HOTI 2024, Albuquerque, NM, USA, August 21-23, 2024* (2024), IEEE, pp. 1–10.

[71] WANG, Y., CHEN, Y., LI, Z., KANG, X., TANG, Z., HE, X., GUO, R., WANG, X., WANG, Q., ZHOU, A. C., AND CHU, X. Burstgpt: A real-world workload dataset to optimize llm serving systems. https://arxiv.org/abs/2401.17644, 2024.

[72] WEI, X., CHENG, R., YANG, Y., CHEN, R., AND CHEN, H. Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 987–1004.

[73] WEI, X., LU, F., CHEN, R., AND CHEN, H. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 121–136.

[74] WEI, X., LU, F., WANG, T., GU, J., YANG, Y., CHEN, R., AND CHEN, H. No provisioned concurrency: Fast rdma-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association.

[75] WU, B., LIU, S., ZHONG, Y., SUN, P., LIU, X., AND JIN, X. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *CoRR abs/2404.09526* (2024).

[76] YANG, Y., ZHAO, L., LI, Y., ZHANG, H., LI, J., ZHAO, M., CHEN, X., AND LI, K. Infless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 768–781.

[77] YU, G., JEONG, J. S., KIM, G., KIM, S., AND CHUN, B. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), M. K. Aguilera and H. Weatherspoon, Eds., USENIX Association, pp. 521–538.

[78] YU, M., YANG, R., JIA, C., SU, Z., YAO, S., LAN, T., YANG, Y., CHENG, Y., WANG, W., WANG, A., AND CHEN, R. lambdascale: Enabling fast scaling for serverless large language model inference, 2025.

[79] ZENG, S., XIE, M., GAO, S., CHEN, Y., AND LU, Y. Medusa: Accelerating serverless LLM inference with materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025* (2025), L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds., ACM, pp. 653–668.

[80] ZHANG, H., TANG, Y., KHANDELWAL, A., AND STOICA, I. SHEPHERD: serving dnns in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023* (2023), M. Balakrishnan and M. Ghobadi, Eds., USENIX Association, pp. 787–808.

[81] 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. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 193–210.

Instance type	Accelerators	Local SSD BW/GPU	Remote SSD BW/GPU	Network BW/GPU	Has NVLink	Price
a2-ultragpu-8g [35]	8 x A100(80 GB)	2.58 Gbps	0.29 Gbps	12.5 Gbps	✓	40.44 USD/h
p4d.24xlarge[9]	8 x A100(40 GB)	2.31 Gbps	-	100 Gbps	✓	45.039 USD/h
ml.hpcpni2.28xlarge[27]	8 x A100(80 GB)	4 Gbps	-	100 Gbps	✗	48.23 USD/h
p4de.24xlarge[9]	8 x A100(80 GB)	2.31 Gbps	-	100 Gbps	✓	56.328 USD/h
a3-highgpu-8g[35]	8 x H100	6.09 Gbps	0.97 Gbps	100 Gbps	✓	88.25 USD/h
a3-megagpu-8g[35]	8 x H100	6.09 Gbps	0.97 Gbps	200 Gbps	✓	Unavailable
p5.48xlarge [9]	8 x H100	9.8 Gbps	-	400 Gbps	✓	Unavailable

Table 2: *A survey of* MAAS *hardware configurations from GPU vendors.*

A Appendix

A.1 Notable implementation details

Network library We implement a communication library that abstracts both NVLink and RDMA to holistically transfer parameters, similar to DeepEP [24]. During our implementation, we found establishing communication group between machines is slow (e.g., 100 ms) when using off-the-shelf group communicators (e.g., NCCL [49]), which significantly limit the effectiveness of network-based scaling. Fortunately, we found that our plan only requires P2P communication between each pair of nodes. Therefore, we pre-create a connection pool that supports full-mesh connections on each. While the compute network (RDMA) has potential scalability issue [43], it only occurs when transferring small payloads and can be addressed using advanced RDMA transport like DCT [73].

Native serving engine with CUDA context pool. Before execution, a CUDA context with loaded kernels (cuModule) must be created on GPU. Creating such a CUDA context takes about 500 ms, and is non-negligible in serving instance autoscaling. To mitigate such an overhead, BLITZSCALE preserves a small CUDA context pool with pre-loaded kernels and transfers parameters to GPU within one of the existing CUDA contexts, similar to an existing work [40]. Furthermore, BLITZSCALE is built using C++ and native CUDA APIs, eliminating the overhead of initializing PyTorch (e.g. `dlopen`).

Fault tolerance. When machine failures occur, we will autoscale new instances using our scaling mechanism. One problem is that cached parameters on the failed machine are lost, so we need to redistribute these parameters to other machines to maintain our global parameter pool invariant. For other components in the system like scheduler or monitor failures, we follow the same procedure in existing work for recovery [56, 29].

A.2 Hardware configurations for MAAS

Table 2 lists the hardware configurations backed by typical MAAS systems.