Fast and Live Model Auto Scaling with $O(1)$ Host Caching

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

[1]Institute of Parallel and Distributed Systems, SEIEE, 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.

We first show that data plane can be made *fast* with no/$O(1)$ caching by loading parameters through the compute network between GPUs because: (1) its speed is comparable host cache and is underutilized; (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 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 instance with cooperative execution, thus handles cases even when the compute network is not sufficiently fast.

Our system BLITZSCALE reduces the serving tail latencies by up to 86% without caching, and we achieve comparable performance (or even better) to an optimal setup where all the parameters are cached at all the host for autoscaling.

1 Introduction

Recent years have seen rapid growth in applications powered by deep learning models [10, 45, 28, 46, 52]. These models are typically served in model-serving-as-a-service systems (MAAS) [54, 8, 20, 65, 17, 7, 24], which manage accelerators (e.g., GPU) clusters and allocate appropriate number of serving *instances* to serve models. Specifically, a serving instance is the set of GPUs used in parallel inference that hold a full copy of model parameter.

An MAAS system has two key 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 hardware demands, (5x required within 2 seconds), e.g., due to requests bursts at the seconds level [38, 67], see also Figure 1 and §2.2.

Model autoscaling is a promising technique for mitigation [65, 8, 24, 3]. When serving a model, MAAS provisions the average number of instances required over the long term, which remains relatively stable. This improve utilization. Upon bursts, the system automatically scales new instances, trying to avoid SLO violations.

Autoscaling speed is critical in minimizing SLO violations because scaling introduces non-trivial overhead to serving latency. The inference time of a model (i.e., Llama2) is typically 80 ms–900 ms, and users expect a tight response time (< 1 second) [9, 68, 27]. Achieving such rapid scaling is challenging, especially for large language models (LLMs) with 10–400 GB parameters, because the scaling is bottlenecked by the **data plane**, which loads model parameters into serving instances. While SSDs are available and utilized in current work [24], the speed provided by SSDs of GPU servers (2–10 Gbps per GPU [29, 6, 22]) is still far from ideal. For instance, loading Llama-2 7 B to a GPU takes 11.2 seconds with 10 Gbps SSD. Worse even, the data plane is **stop-the-world**: scaled instances cannot serve requests until all parameters are loaded. This exacerbates SLO violations when data plane is slow.

To this end, state-of-the-art system like ServerlessLLM adopts a multi-tiered caching system by storing models in the host(CPU) DRAM of machines [30, 34]. While caching can leverage the fast host-GPU link (e.g., 256 Gbps PCIe) for the data plane, achieving a high hit rate is unfeasible. ServerlessLLM reports a hit rate of 40%-75%, which is confirmed by our empirical study in §3. The key reason is that a MAAS system hosts thousands of models [18], so caching all models on all hosts is impractical especially when a model must scale across multiple instances on multiple hosts. The hosted models come from a model zoo like HuggingFace [23], which offers 1,170,000 available models, with 1,596 of them having over 15,000 downloads each. Storing these frequently downloaded models would demand 23 TBs of memory, clearly exceeding the capacity of host memory and even SSDs.

To achieve fast data plane even without relying on cache hit, we make the following two key contributions:

Data plane made fast with $O(1)$ or no caching with compute network multicast. First, a MAAS is backed by fast GPU-GPU/CPU compute fabrics [42], which are 100–400 Gbps RDMA and even 16 Tbps NVLink [29, 6, 22], much faster than SSD and comparable to host-GPU link. The fabric is used for data transfer during serving and we found that they are under-utilized (up to 14.4% of total bandwidth), even in network-heavy workloads like P/D disaggregation [47, 68, 31, 16, 64] (§3). This under-utilization suggests

*These authors contributed equally to this work

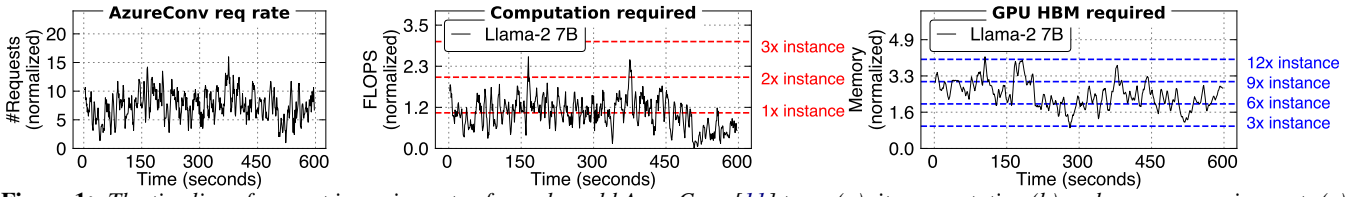

Figure 1: *The timeline of request incoming rate of a real-world AzureConv [11] trace (a), its computation (b) and memory requirements (c) when serving this workload without SLO violation.*

that we can borrow it for loading parameters.

Second, network-based data plane requires no or minimal caching to achieve fast scaling. If a model is already deployed on some instances, we can directly multicast the parameters from those instances, eliminating the need for caching. Such multicast is extremely efficient because a serial forwarding multicast [57] can load bulk data (e.g., model parameters), regardless of the number of receivers. 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 because the aggregated host memory of all machines is sufficient to cache all models of a MAAS.

Although fast networking can significantly accelerate the data plane, a stop-the-world loading remains a bottleneck in cases with tight speed requirements. Our measurements of BurstGPT on Llama2-70B shows that a tight 500 ms stop time is needed to eliminate 95% of SLO violations. Achieving so requires 560 Gbps per-GPU* parameter transfer bandwidth, which exceeds the available bandwidth of typical 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 finish, the scaled instance should be able to serve requests.

Data plane made live with fine-grained scaling abstraction and cooperative execution. Model scaling cannot be live using on-demand data loading techniques commonly found in serverless [63, 59, 32] or inference loading overlap in PipeSwitch [12] (§4), because an instance can only serve requests after all its parameters are loaded. We found that this stop is due to their coarse-grained scaling abstraction: existing systems only scale and serve at the instance level, but *models can execute in a fine-grained, layer-by-layer way with multiple instances*. With this fine-grained layer-wise scaling, we can offload part of the layer's computation from overloaded instances to scaled instances, thus improving the overall serving throughput even before the scaled instance has loaded all the parameters.

Challenges and solutions. First, utilizing network-based multicast for parameter loading is more complex than it seems. Though the mechanism of executing a multicast is simple, the challenge lies in generating an optimal multicast plan that minimizes the data loading time. This challenge arises because: (1) Creating an optimal multicast plan is NP-hard on heterogeneous networks like serving clusters [15]. In autosclaing scenarios, we must generate this plan online with diverse network topologies, as the hardware used for scaling is selected dynamically based on their usage. (2) Modern serving workloads, particularly LLM serving, already heavily use the network of some instances. Although the overall data center network is under-utilized, network-based autoscaling can interfere with the serving workloads, resulting in 1.5× longer parameter loading time and up to 32% degraded serving performance. Current multicast solutions [19, 16, 26] mainly target offline scenarios like training, with relative static network configurations. Thus they tolerate long plan generation time and do not consider the interference from serving workloads. To address the issue, we propose a model-aware multicast planner, which leverage the static data flow in model serving to quickly generate a near-optimal, interference-free multicast plan for parameter loading (§5.1).

Second, it is challenging to schedule live autoscaling due to the complexity of dynamic scheduling decisions, i.e., which instance execute which layers. Dynamic decisions are necessary because the parameters on scaled instances change dynamically. Doing so is non-trivial. A naive best-effort scaling that executes as much layers as possible cannot balance the load. It is because at the beginning of autoscaling, the new instances can only execute few layers, thus many requests are still queued. A better solution is to adjust the load holistically by considering future incoming layers, and we realize this with a zigzag scheduling method and achieves 50% tail latency reduction under extremely bursty workload. (§5.2).

Demonstration with BLITZSCALE. We built BLITZSCALE, an MAAS system with the fastest autoscaling speed with $O(1)$ caching. We adopt a global parameter pool to manage the model parameters across all the machines, and integrate a model-aware network multicast planner to do the parameter loading. For instances that may still suffer from SLO violations due to the insufficient data plane, we will lively autoscale with zigzag scheduling. We evaluated BLITZSCALE across a variety of recent models with different size and architectures, including Llama2-7B, Llama2-70B, and Llama3-8B. First, BLITZSCALE has 30–80% shorter TTFT, and has up to 50% shorter TBT than ServerlessLLM under real-world traces (i.e., BurstGPT [60], AzureCode and AzureConv [11]). Second, BLITZSCALE has a mangitude smaller tail latency loading from network than S-LLM loading from SSD, and has comparable performance to an optimal setup where all

*70 B model uses four GPUs per-instance.

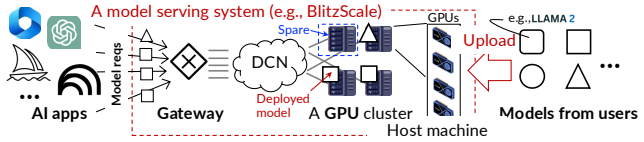

Figure 2: *An illustration of Model as a Service (MaaS) system. DCN states for data center network.*

the parameters are cached at all the host thanks to the live autoscaling.

We will open source BLITZSCALE upon publication.

2 Background on MAAS and Autoscaling

2.1 Model and model-serving-as-a-service (MAAS)

Model serving: LLM & non-LLM, P/D disaggregation. A serving workload takes a trained (or open-source) model, queries it with a request and gets the results.

All known models follow a layer-by-layer computation paradigm (see Figure 7 (a)). For each user request, the model may be queried once, as in vision tasks, or multiple times, as in large language models (LLMs) [56]. For example, in LLMs, the model is first queried with input text (prompt) to produce an initial prediction (token). This phase is termed *prefill*. The prediction is then used to generate subsequent tokens iteratively (auto-regressive) until the model predicts an end-of-sequence token. The auto-regressive phase is termed *decode*. Note that in LLM, the inference time 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).

When processing a request, the model may cache intermediate states during execution (stateful). For example, when an LLM processes a request, the key and value matrix multiplication results (termed *KVCache*) are cached in GPU for acceleration decode phase.

Finally, single request may be served among multiple model replicas, leading to data movement between instances. For example, prefill-decode disaggregation [47, 68, 31] (P/D disaggregation) is a common serving setup for LLMs. It uses two replicas of the same model to serve the prefill and decode phases of a request, respectively. The intermediate results from the prefill model are passed to the decode model.

MAAS systems and serving instances. As shown in Figure 2, a model-serving-as-a-service (MAAS) system [54, 8, 20, 65, 17, 7, 24] allows users to upload their models*. The system allocates its managed computing resources—a cluster of GPU servers—to host these models and serve workloads. Existing MAAS systems define *instance* as the set of GPUs used in parallel inference that store a complete copy of a model.

An important feature of MAAS platforms is their charging model. *They charge users based on the number of requests processed within SLO* [7, 17], not on the number of deployed instances. This gives the platform flexibility to adjust the number of instances dynamically, improving hardware utilization.

*The users can also use platform's pre-deployed models.

2.2 Unpredictable GPU demands and model autoscaling

Unpredictable and fluctuating GPU demands. Determining the right number of instance for serving a model is challenging, because the incoming request rate for a serving workload fluctuates over time and is hard to predict [24, 50, 67]. In Figure 1 (a)'s real-world LLM inference trace, incoming inference requests increase 5 × within 2 seconds, showing no clear patterns.

Moreover, the complex execution nature of model serving, such as LLMs, causes computation and memory requirements for each request to fluctuate unpredictably. For example, in serving LLMs, different batches leave from 500 MBs to 50 GBs of KVCache in GPU memory due to continuous batching [36]. The stay time of this cache is also unpredictable due to the auto-regressive nature of LLMs. This further amplifies the fluctuations in instances requirements, as seen in real-world traces (see Figure 1 (b) and (c))*.

Handling fluctuations through model autoscaling. Model autoscaling, which dynamically deploys* serving instances on spare GPUs, is a promising solution to handle fluctuated computation and memory demands [24, 65, 8].

The state-of-the-art autoscaling system, ServerlessLLM [24], focuses on accelerating parameter loading(data plane). It uses SSDs on ech GPU server to store the parameters of all hosted models and employs a parameter loading mechanism that can fully utilize SSD bandwidth to load the parameters into instances. Unfortunately, ServerlessLLM does not accout 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 Peer-network between Instances

Model autoscaling requires fast data plane. Figure 3 (a)—(d) characterizes the scaling speed requirements for both medium-sized (7 B) and large models (70 B). We conducted our experiment using our own implementation of DistServe [68] that supports the state-of-the-art autoscaling policy (described in §5.3). DistServe disaggregates the instances for prefilling and decoding, allowing us to precisely measure the SLO violations of TTFT and TBT. To characterize how scaling speed affects SLO compliance, we use a simulator that provisions models to all GPUs and applies manual delays to simulate different scaling speeds. We set TTFT and TBT SLO based on inference speed of different models. Specifically, 450ms and 150ms for Llama2-7B model, and 1250ms and

*Measured with Llama2-7B model.

*We can also stop a serving instance to scale down. Since scaling down is simpler, we omit its details for brevity.

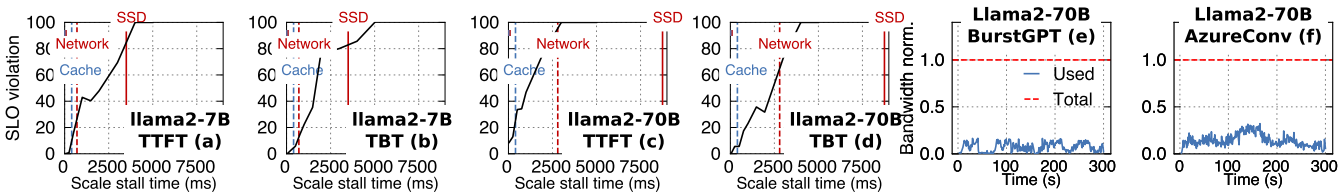

Figure 3: *A characterization of SLO attainment for different inference cases (a)–(d) with varied duration of autoscaling stops on BurstGPT [60]. (e) and (f): an analysis of compute network usage in serving workloads. The evaluation setup is in §6.*

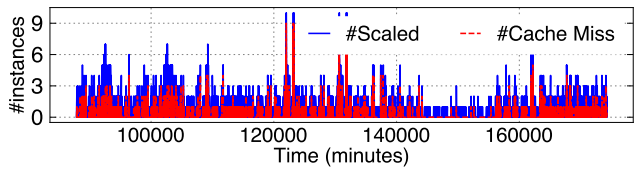

Figure 4: *An analysis of host cache miss during autoscale of ServerlessLLM [24] on the BurstGPT [60] workloads.*

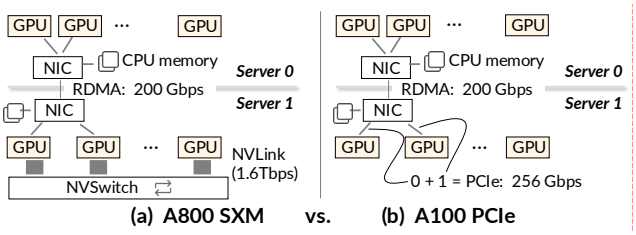

Figure 5: *A illustration of networking in* MAAS *with NVLink (a) and without it. Note that the 256 Gbps PCIe is shared between two GPUs attached to it [40].*

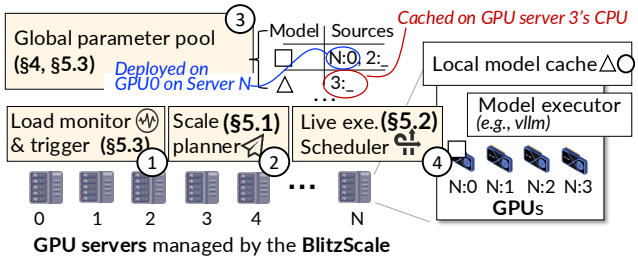

Figure 6: *The system architecture of* BLITZSCALE.

200ms for Llama2-70B model with tensor parallelism degree of 4.

Model serving requires fast scaling speed. For example, with a 70 B model, maintaining TTFT below 1100ms requires a minimum per-instance scaling speed of 256 Gbps, which is only achievable with host cache (Cache). The scaling time requirement correlates directly with inference time—our evaluated workload (BurstGPT [60]) has an average TTFT of 771ms (with queuing time), and to achieve 1250ms SLO for most of requests requires ~500ms stop time, and thus 560 Gbps per-GPU* network (measured by dividing the model size with the requirement). This speed requirement far exceeds what vendor-provided per-GPU [29, 6, 22] SSDs bandwidth can deliver (2–10 Gbps per-GPU, detailed in the appendix).

Cache miss is common. While accessing the host model cache can achieve low SLO violations, cache misses are common in real-world traces. Figure 4 analyzes the number of instances scaled and cache misses encountered in the BurstGPT workload using ServerlessLLM [24]. Following its setup, we set a 5-minute keep-alive interval for the model cached at the host. The miss rates range from 20-46%, depending on the time, which aligns with the numbers reported in their paper (25–60%). Interestingly, many misses occur when scaling multiple instances, because involving more hosts increases the probability of cache misses. This indicates that minimizing cache miss penalties is especially crucial when scaling across multiple instances.

Opportunity: compute network between instances. First, compute networks connecting GPUs (and CPUs) have comparable or even faster speeds than host-to-GPU connections. As shown in Figure 5, the inter-GPU network (RDMA) operates at 200 Gbps, which is close to the PCIe speed (256 Gbps) used when loading cached parameters from host CPU to GPU.

*70 B model uses four GPUs per-instance.

With NVLink, these speeds can be even higher. More importantly, these networks are underutilized by serving. Figure 3 (e) and (f) measures the network usage of DistServe [68], a P/D disaggregated serving system that heavily transfers KV-Cache from prefill instances to decode instances. To measure peak usage, we provisioned all the GPUs for serving, use the workload traces with the maximal request rate that our clusters can serve. We observe that only less than 15% of the network capacity is used.

4 System Overview

BLITZSCALE scales models through the network with minimal caching. We achieve this by first managing model parameters—scattered across GPUs behind serving instances (for deployed models) and CPUs (with local model cache)—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 the system architecture. We have a load monitor (①) that tracks the

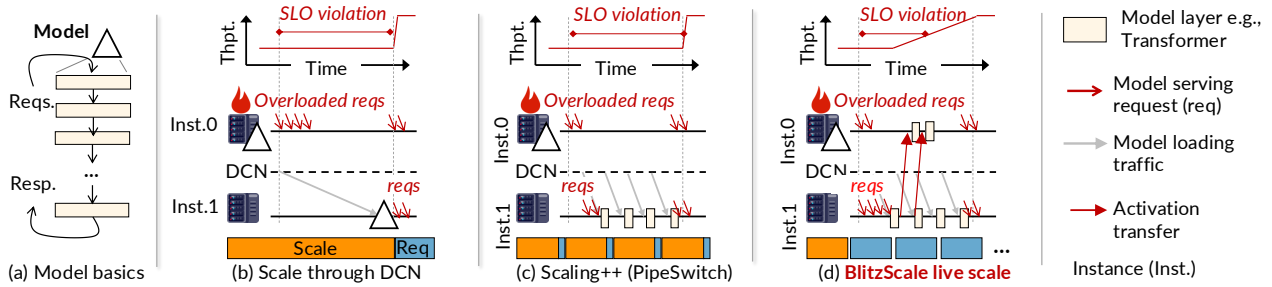

Figure 7: *(a) An overview of how model executes requests. (b) An illustration of model scale through DCN. (c) An illustration of an optimized scaling method with overlapped execution [12], but it still cannot be live. and (d) how* BLITZSCALE *scales models lively.*

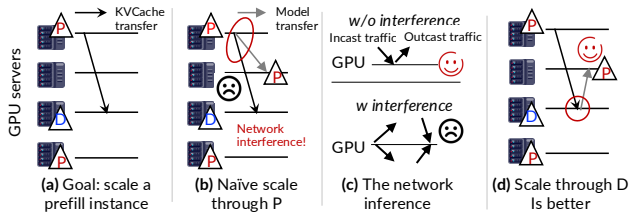

Figure 8: *(a) An illustration of scaling a prefill instance for LLM P/D disaggregated serving. (b) Naively scaling from a prefill instance imposes network interference. (c) Interference can be avoided by leveraging the bi-directional feature of modern DCN networking. (d) An improved scale plan with the bi-directional in mind.*

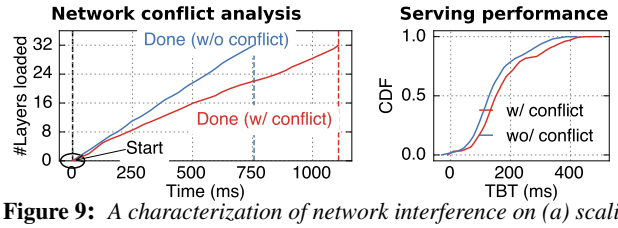

Figure 9: *A characterization of network interference on (a) scaling speed and (b) serving performance.*

serving load for each deployed model instances, and decides whether to scale and how many new instances are required (§5.3). Afterward, a scale planner (②) will derive a scaling plan that guides how to load parameters onto these instances (§5.1). The planner will consult the global parameter manager (③) for the available parameter sources, which stores a mapping between the model name as well as a list of sources. In our example, the new instance can pull the parameters of model □ from the GPU0 on host N $(N, 0)$, or from the host 2's CPU memory $(2, \_)$. During autoscaling, our live execution (exe.) scheduler (④) will redirect requests between instances to realize a live autoscaling. On each machine, we deploy our serving engine similar to vllm [36] for execution.

Challenges and approaches. Despite leveraging fast networking, making autoscaling fast and live is non-trivial:

C#1. Online interference-free scale plan generation. Generating the scale plan is similar to a *multicast* plan generation problem [15, 19, 26, 13, 14], i.e., how to quickly distribute data from some sources to targets. Two distinctions arise in model autoscaling: First, the plan must be generated online on dynamically changing network topologies, but optimal plan generation is NP-hard [15] on a heterogenous network. Second, we should eliminate interference between the scale plan and the serving workload. Figure 8 shows an example. Suppose a P/D disaggregation case. As we have introduced in §2.1, it uses at least two serving instances, a prefill and a decode. The KVCache is transferred from the prefill instances to the decode instances (a). Suppose we want to scale a prefill instance: If we naively select another prefill instance as the source (b), then the scale will compete the network with the serving workload, leading to 1.5x longer scale time as well as 32% TBT increases (Figure 9).

To this end, we design a serving-guided greedy plan generation method based on the following observations (§5.1). First, the network heterogeneity mainly comes from NVLink, whose scale time is negligible (120ms to broadcast a Llama3 8B to intra-node GPUs, see Figure 5). Thus, we can safely eliminate target instances from the plan generation when they can be grouped together with NVLink to simplify plan generation. Second, loading parameters from the network is bandwidth-intensive, so we can greedily generate a plan with serial forwarding chains [57], which typically is optimal in broadcast time. Finally, the network (RDMA) between GPU servers is bi-directional [61, 44], meaning that the network flow of incast and outcast can transfer without interference (c). Thus, we can systematically generate a plan that avoids interference with the known knowledge of how serving instances use the network, e.g., loading parameters from the decode instance to the prefill instance (Figure 8 (d)).

C#2. Efficient live autoscale. Live autoscaling is necessary especially even with the fast network, SLO violations can still happen (Figure 3). Current approaches like PipeSwitch [12] overlap parameter loading with inference, trying to realize live autoscaling. As shown in Figure 7 (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, thus hiding future layer loading with current layer execution. However, this approach cannot be live especially for LLM, because, until all the layers are loaded, inst. 1 cannot process requests, whose stop time is similar to

a naive scale if the load dominates the time (b).

To realize live autoscaling, we break the scaling granularity from instance to layer-level, so we can offload computation from overloaded instances to instances with partially loaded parameters for live execution. Figure 7 (d) illustrates this. When inst.0 becomes overloaded and inst.1 has begun 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. This improves serving throughput. To see why, considering serving a 7-layer model, so inst.0 alone will have a throughput of 1/7. With our live scaling, after loading one layer on inst.1, both instances can cooperatively handle requests, increasing the throughput 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—at only half the load time.

§5.2 describes our zigzag scheduling for coordinating overloaded and new instances during live autoscaling to achieve an optimal performance for live autoscale.

5 Detailed Design and Implementation

5.1 Online network-based scale plan generation

When the planner is notified to scale the parameters on t target instances, it will get s sources from the parameter pool, find t spare instances and generate a plan to send parameters from ss to ts.

The plan is essentially spanning trees [57], e.g., $s_0 \rightarrow t_0, s_1 \rightarrow t_1, t_1 \rightarrow t_2$ means there are two trees: s_0 will send the parameter to t_0, and s_1 will send the parameter to t_1 and t_1 will send the parameter to t_2. Besides minimizing the scale time, there are additional two requirements for the plan: (1) generating the plan in a short time and (2) minimizing the interference with serving workloads.

To quickly generate the plan online, we use a three-step greedily algorithm as shown in Algorithm 1. First, we group all targets connected via NVLink into a group (Line 1). The parameters will only load to one instance in the group, while others receive the parameter through NVLink broadcast. This reduces complexity without sacrificing speed, because the NVLink broadcast time is negligible. For example, for a Llama2-7B model, Transferring it with 200 Gbps RDMA would take 650ms, while broadcasting it with NVLink only takes 105ms in practice.

While NVLink can dramatically improve multicast speed, it is not always applicable because: (1) GPUs with NVLink are typically used to form a single instance for large models (e.g., Llama-2 70B) and (2) clusters may lack NVLink (see Figure 5 (b)).

Second, to avoid network interference, we remove links between sources and targets that could cause interference, guided by the serving workloads (Line 2). For example, in P/D disaggregation, only the prefill instance will send data to the decode instance. Since these flows are static throughout the inference, we can systematically prune them before generating the plan. Our prune is based on whether a source has outcast link: e.g., if s_i also sends outcast network, we will delete s from the sources.

Finally, we will generate the multicast plan by greedily

Algorithm 1: *Scale Plan Generation Algorithm*

Input: $\mathbf{D_{src}}$ is the set of sources (s_i) for loading parameters, organized in a priority queue by their bandwidth. $\mathbf{D_{tgt}}$ is the set of targets (t_j) sorted by their bandwidth. $\mathbf{BWO_s}$ and $\mathbf{BWI_t}$ is the outcast and incast bandwidth of a source s and a target t, respectively.

Output: The scale plan $\mathbf{Plan}$

1 $\mathbf{D_{tgt}} \leftarrow$ Grouped(D_{tgt})
2 $\mathbf{D_{src}} \leftarrow$ Pruned(D_{src})
3 $Plan \leftarrow \emptyset$ $t \leftarrow D_{\text{tgt}}$.pop(); ▷ `t will be nil if sets are empty`
4 **while** $t \neq nil$ **do**
5 s, CurrentBW$_s$ $\leftarrow \mathbf{D}_{\text{src}}$.pop()
6 CurrentBW$_t \leftarrow \min(\text{CurrentBW}, \mathbf{BWI}_t)$
7 CurrentBW$_s \leftarrow$ CurrentBW $- \min(\text{CurrentBW}, \mathbf{BWI}_t)$
8 Plan.append((s, t))
9 $\mathbf{D}_{\text{src}}$.push(s, CurrentBW$_s$) **if** CurrentBW$_s > 0$
10 $\mathbf{D}_{\text{src}}$.push(t, CurrentBW$_t$)
11 ▷ `Priority is based on insertion order for the same BW to minimize chain size`
12 $t \leftarrow \mathbf{D}_{\text{tgt}}$.pop()
13 **end**

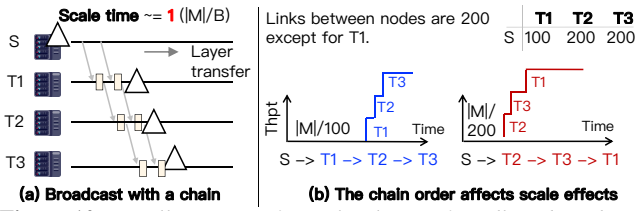

Figure 10: *An illustration of (a) why chain is friendly to broadcast with huge bandwidth requirement and (b) why we chooses a specific chain order. $|M|$ is the model size and B is the slowest network bandwidth between nodes in a chain.*

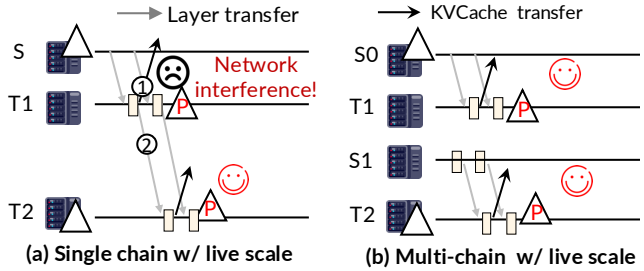

Figure 11: *An illustration of why multiple chains are better especially under live scaling.*

constructing broadcasting chains, based on the fact that a serial forward chain is better in bandwidth-dominated transfer [57]. Specifically, the overall transmission time is irrespective of the instances scaled with such a chain. Figure 10 (a) shows an example: suppose we construct a plan of $s \rightarrow T1 \rightarrow T2 \rightarrow T3$. 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.

Line 4–11 shows the algorithm. There are two important design choices in our algorithm. First, the order of target nodes in a chain is important especially when the network speed between nodes is different. We choose an order sorted by the link speed between nodes to achieve a faster increases in the serving throughput. Figure 10 (b) shows why: suppose the source (S) can send parameters to $T2$ faster than $T1$. A chain order of $S \rightarrow T2 \rightarrow T1$ is better than $S \rightarrow T1 \rightarrow T2$ because the downtime is only half. Thus, we will select candidate sources and targets based on their bandwidths. Second, we choose a multi-chain approach instead of a single chain, though a single chain has a similar multicast time. This is because multi-chain can enable more interference-free live scale. Figure 11 (a) shows this: suppose we want to scale two prefill instances and we want to do it lively. Since the scale is live, the KVCache will be transferred to the decode instances. With a single chain, only $T2$ can do the scale without interference, because at $T1$, the KVCache transfer (①) interferes with the parameter forward traffic (②). With two chains backed by two parameters sources (b), both $T1$ and $T2$ can live scale without interference.

When constructing the chains, a target instance may also cause interference in some corner cases, e.g., if the instance only use one GPU, and the GPU shares a NIC with another GPU (see Figure 5) that runs a prefill. We only control the scale bandwidth usage (e.g., only use the half bandwidth) in this case (not shown in the algorithm) for simplicity. This is because without autoscaling, interference can still happen during normal workloads.

5.2 Efficient live autoscaling with zigzag scheduling

Selecting instances for live scaling. After getting the chains from Algorithm 1, we will select instances to participate in live autoscaling. The selection criteria is twofold: (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 since live autoscaling requires a cooperation from it. 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. Specifically, for each overloaded instance, we will identify an instance in the chain following the above criteria, which is typically the tail instance as it typically 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 requests from inst.0 to it for execution. The redirection time is negligible because the request payloads are much smaller than the model. All new requests to inst.0 are also redirected to inst.1. (2) After the first layer is loaded on inst.1, 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.

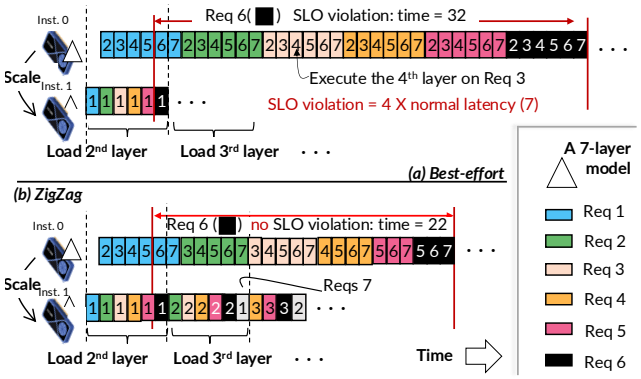

Figure 12: *An illustration the necessity of zigzag scheduling. Note that the execution starts when the first layer has been loaded to instance 1 (inst.1). Our example assumes the time of loading a layer can do 6-layer computations.*

The scheduling problem. During the above step (3), a key problem to tackle 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. One straightforward 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. This approach adapts scheduling with model loading and utilizes inst.1 even it has only loaded a few layers. However, during the initial loading phase, inst.1 has limited serving capacity, so most requests are still queued at inst.0, still causing SLO violations due to imbalanced workloads. Figure 12 (a) shows a concrete example of a 7-layer model executed with 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 batchsize of 2000 prefill tokens under 200Gbps RDMA). 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 being completed on inst.0, as their execution time has reduced a little due to the imbalanced loads.

The zigzag scheduling. To address this issue, the observation is that by delaying request scheduling on inst.0, the inst.1 will

have more layers coming, giving us opportunities to balance the workload. Figure 12 (b) shows an example. 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 request 2–5. Thus, the second layer execution of request 3–6 is overlapped with the execution of layers 3–6 for request 2. Thanks to this overlap, the overall inference time decreases from 32 to 22, not violate the SLO.

To realize the above zigzag scheduling, we formulate the problem as follows. Note that like existing works [36, 64, 66], the scheduling assumes a first-come-first-serve (FCFS) policy. For ease of presentation, we first assume non-LLM and then extend to LLM. More specifically, our scheduling method contains two parts: pipeline configuration and request scheduling.

(1) Pipeline configuration. Given N request batches with equal size, we first determine the pipeline configuration $(\mathrm{T}_i, \mathrm{S}_i)$ for each request, where T_i and S_i are the number of layers to be executed on the target and source GPU for request i, respectively. To goal is to minimize the average latency with an 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, considering the example in Figure 12 (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' execution time (queuing 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 the request 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 (and possible interference) since the network used for model activation is negligible.

Minimizing the objective requires meeting the following constraints:

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

C1 is trivial: the pipeline should be fully executed. **C2** ensures that the execution preserves the 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 when the target instance executes request 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 pipeline layer.

While solving an ILP is NP-hard, our problem remains manageable (less than 40ms to solve for Llama2-7B model) because models typically have only a few dozen layers (32 for Llama2-7B and 80 for Llama2-70B). 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. Thus, we can utilize the waiting for the target instance to load the first layer to solve the ILP.

(2) Scheduling requests. Based on the pipeline configuration, the scheduler on the new instance rehearses request execution. To achieve FIFO, we use a priority queue for all requests. A problem is that when executing a specific request, the new instance may lack its required layers. These requests are moved to a pending queue. For timely rescheduling these requests, we implement a preemption mechanism: whenever a new layer loads, we scan the pending queue and execute all requests that require that layer.

Coping with LLM. LLMs introduce two problems here. First, the execution time of equal-sized batches varies due to different sequence lengths, so our formulas and constraints described above cannot be directly applied to LLM serving. Fortunately, the prefill time of a layer has been shown to be approximately linear to the total batched token size [47, 64], so we can add a regulation parameter for each request to address this issue when scaling prefill-only serving (e.g., autoscaling a prefill instance in P/D disaggregation). A more tricky case involves handling decode, e.g., when scaling instances that combine prefill and decode, or scaling a decode instance in P/D disaggregation. The complexity arises because decode batch sizes change dynamically due to its auto-regressive nature. For live scaling of a decode instance in P/D disaggregation, we present a solution below to address these challenges. As for live scaling of a P-D colocated instance, we leave it for future work.

Second, live scaling a decode instance in P/D disaggregation is impossible due to the incast nature of both parameter load and KVCache transfer. To address this, we leverage the observation that prefill instances and decode instances share the same parameters. This allows scaling a decode instance by switching a prefill instance to a decode one. Specifically, when live scaling a decode instance, we first switch a prefill instance to a decode instance. At the same time, we scale multiple prefill instances to compensate for the loss in the

number of prefill instances.

5.3 Global parameter pool, monitor and others

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 network-based parameter multicast, our global parameter pool only needs to ensure that each hosted model has at least one parameter copy stored either in a instance or at one of the local cache of a GPU server. Realizing this is simple: during system initialization, we distribute one copy of the models' 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.

In this paper, we focus on the autoscaling mechanism, so we use a simple distribution policy that ensures exactly one copy exist for each model across the machines managed. While replicating the parameters enable more live autoscaling opportunities (e.g., see Figure 11), it comes at the cost of extra memory usage, and we leave it as future work.

Workload monitor and scale up trigger. Our paper focuses on the autoscaling mechanism, which is orthogonal to the autoscaling policies, including collecting workload metrics with workload monitoring and determining how many new instances to scale based on these metrics. Our current implementation follows priori work [53, 2] that first records the serving loads with tokens per second and KVCache usage globally. When the average 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.

Scale down. Follow previous work [48, 24], we use a timeout-based scale-down policy: when the average load falls below a lower bound in a time window, we shutdown some instances and revoke all GPUs assigned to them. Given BLITZSCALE's rapid autoscaling capabilities, we adopt an extremely short timeout of 2 seconds.

Point-to-point (P2P) connection pool. 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 [41]), 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 precreate a connection pool that supports full-mesh connections on each. While the compute network (RDMA) has potential scalability issue [35], it only occurs when transferring small payloads and can be addressed using advanced RDMA transport like DCT [62].

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 [48, 24].

	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

BLITZSCALE is a MAAS system capable of serving both traditional models and LLMs with 2,4000 lines of rust and c++ code. It fully supports widely applied LLM optimizations like P/D disaggregation and continuous batching. Two notable implementation details are: First, to efficiently coordinate layer execution with networking requests, we implement an asynchronous scheduling layer at each host using Rust, because our initial implementation on PyTorch is extremely inefficient due to Python's runtime overhead. Second, we implement a communication library with a pre-established connection pool that abstracts both NVLink and RDMA. Third, to determine resource demand and scaling timing, we implemented an accurate autoscaling trigger similar to existing systems [2, 53]. For other components, including scheduling and GPU kernels, we leverage existing systems wherever possible. For instance, all our GPU kernels for LLM come from FlashInfer [1].

Testbed. Our evaluations are conducted on two testbeds as shown in Table 1. A key distinction is that cluster A has NVLink connections between GPUs within a host (GPU-GPU intra), while cluster B does not. Therefore, cluster A can serve larger models (e.g., 70 B) with Tensor parallelism [36].

Traces and models. Because the requirement of autoscaling is closely related to the incoming request rates, we choose three real-world traces: BurstGPT [60], AzureCode and AzureConv, both from Azure [47]. The detailed trace shapes are shown in the first column in Figure 13. Similar to priori work [39, 5, 30], to fit these traces to our experiment setup, we rescale the trace according to suggestions from their paper [60]. As these traces contain the datasets (e.g., the prefill and decode length of each request), we directly use them in our evaluations, e.g., the BurstGPT workload is evaluated using the BurstGPT datasets. Note that the a higher request rate in the figure does not necessarily mean a higher load,

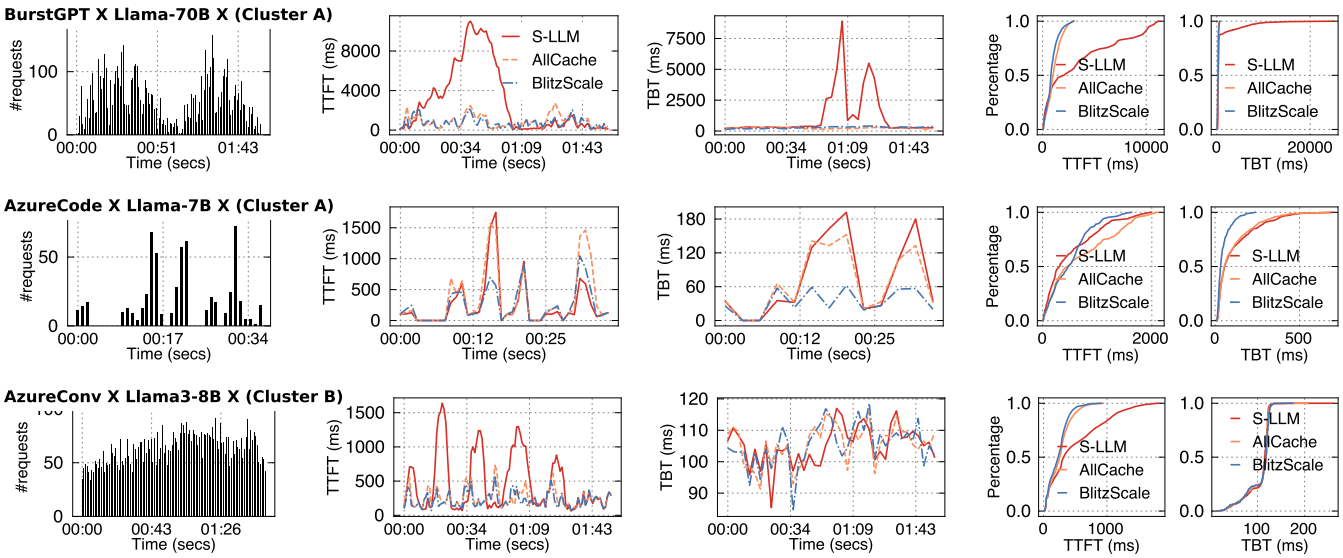

Figure 13: *End-to-end performance comparison between* BLITZSCALE *and ServerlessLLM on various workloads, models and clusters.*

because the prefill/decode length of each request is different, which could significantly impact the serving requirements.

For models, we focus on evaluating LLMs because others non-LLMs are much smaller and is trivial scale efficiently with our network-based multicast. Specifically, we choose Llama-2 7B, 70 B and Llama-3 8B, a widely used model family, with model sizes that can fit into a single GPU (7 B and 8 B) for each instance and require multiple GPUs (70 B) for a serving instance. The minimal number of GPUs used for the 70 B model is 4.

Comparing targets. We compare BLITZSCALE with the following baselines:

1. ***ServerlessLLM (S-LLM)*** *[24]* is the state-of-the-art MAAS with a focus on accelerating autoscaling speed. It utilizes host memory to cache recent loaded models with a keep-alive eviction policy. Under cache misses, it adopts a parameter loading technique that can fully utilize SSD.

2. ***AllCache*** is the optimal version of ServerlessLLM that always load the parameters from the host cache.

3. ***DistServe*** [68] is the state-of-the-art LLM serving system that incorporates P/D disaggregation. It does not support autoscaling.

6.1 Autoscaling performance under real-world traces

Due to space limitation, for each model, we choose one trace on one of the cluster to evaluate the performance. Figure 13 presents the end-to-end performance. Note that one for each trace, we scale the request rate such that the peak incoming rate can be handled by all the instances—otherwise autoscaling cannot help. For all the workload, we run with prefill and decode disaggregation, with prefill and decode instances scales independently based on their loads. These workloads are more challenging for our network-based scaling, because the original serving has already heavily used the RNIC of some instances. We focus on the comparison with S-LLM and AllCache in this section and leaves the comparison with DistServe in the next section, because DistServe cannot dynamically adjust the instances to cope with dynamic workloads.

First, BLITZSCALE has 86.1%, 55.5%, 31.0% shorter TTFT, and has 18.1%, 57.8% and 1.1% shorter TBT than S-LLM, respectively. The reduction mainly comes from the fast autoscaling provided by our network-based scaling. The penalties are obvious in 70 B model: because loading its parameters from SSD is too slow. As shown in Table 1, our cluster only supports a 10 Gbps per-GPU bandwidth, so a 4-GPU 70 B instance needs 28 seconds to load the parameters. Note that while all prefill workloads are affected by short bursts in the traces, decode is relatively resilient. This is because each decode will stay for a relative (unpredictable) long time, so the bursts are amortized (e.g., in AzureConv, decode instances seldom scale up or down). An exception is the AzureCode, where the decode also fluctuates with the prefill, because its decode is much shorter than the others (with an average generated tokens of 28, which is 211 of AzureConv).

Second, BLITZSCALE has close to AllCache performance in all workloads. This is because the network we used has a 78% bandwidth of the host-GPU link, and our live autoscaling can fill the missing gap by allowing the new instances to serve even with partial parameters loaded. An interesting phenomenon is that on AzureCode, we are even faster than AllCache, because our LLM-trick can lively mutate a prefill instance to a decode instance, avoiding network traffic and thus realizing a zero-cost autoscaling.

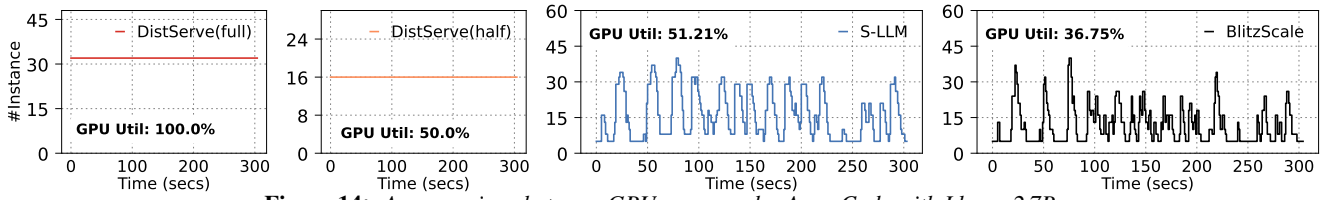

Figure 14: *A comparison between GPU usage under AzureCode with Llama-2 7B.*

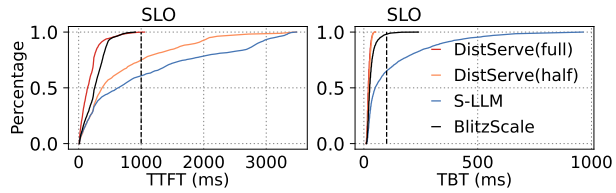

Figure 15: *A comparison of TTFT and TBT under AzureCode on Llama-2 7B.*

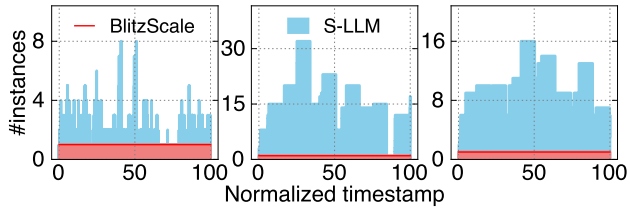

Figure 16: *A comparison of host cache usage on S-LLM and* BLITZSCALE *under all three workloads.*

6.2 Performance and resource usage

GPU utilization. Figure 14 and Figure 15 compares BLITZSCALE with S-LLM, AllCache and DistServe for GPU utilization and performance on AzureCode workloads. We omit comparison with other traces/models/clusters as they are similar. As DistServe cannot dynamically adjust the number of instances, we evaluate two setups: DistServe (full) will utilize all GPUs and DistServe (half) will only use half of the GPUs.

First, we can see that BLITZSCALE has the lowest GPU utilization, because under light loads we don't need to deploy instances on the GPU. Our utilization is 40% lower than S-LLM, though both systems can dynamically adjust the number of GPUs used. The reason behind the improvement is due to our fast auto-scaling method. When autoscaling slow and stop like S-LLM, the monitor will think the current system cannot handle the burst, so it will trigger more scales. This is unnecessary.

Compared to variants of DistServe, it is not surprising to see that BLITZSCALE significantly improves the GPU utilizations. More importantly, we achieve so with small or better performance: Compared with DistServe (half), we achieve a 35% shorter TTFT and 41% shorter TBT. This is because under load spikes, only half of the GPUs is insufficient to handle the load. Compared with DistServe (full), though we have a 2% increases in the P99 latency, we don't violate the SLO, even we set a relative tight SLO of x5 following priori work [68].

Host cache usage. Because BLITZSCALE does not rely on host caching, it can significantly reduce the host memory used for caching a model. Figure 16 shows the host memory used for caching by comparing BLITZSCALE with S-LLM. We don't compare with AllCache because it always deploy a copy on all the hosts, which is clearly huge. We also don't compare with DistServe because it doesn't require caching.

Because we have few hosts (up to four) due to hardware limits, we use logical hosts to simulate a larger evaluation setup. Specifically, for 70 B workloads, we logically group 4-GPU to a host, which allows us to simulate the performance with 16 hosts. For 7 B and 8 B workloads, we use 1-GPU for each host, allowing us to simulate a larger cluster with 32 and 16 hosts, respectively. Note single-GPU hosts are common in serving clusters [53].

We measure the cache as follows. For BLITZSCALE, we only maintain at least one copy of the model across all the hosts, so its cache usage is always 1. For S-LLM, we follows its default setting, which uses a 5-minute keep-alive time. The model is cached on the host for 5 minutes after the last request. As we can see, BLITZSCALE has orders of magnitude smaller host cache usage. Such a reduction can free more memory for other models, which can subsequently improve their performance. On BurstGPT, S-LLM has a slightly better overall cache usage, because there are many time in BurstGPT with no requests, which means that S-LLM will not cache any model on the host due to the keep-alive policy. Nevertheless, as we have extensively shown such a cache miss will cause a significant performance penalty during auto scale.

6.3 Detailed analysis of BLITZSCALE scaling

A detailed look at the scale. To better understand how BLITZSCALE scales compared to S-LLM, Figure 18 shows a throughput timeline when using BLITZSCALE and S-LLM to scale a Llama 2 7B prefill instance from one cached copy to 16 GPU instances on cluster A. The cached copy of BLITZSCALE is in one of the GPU, while S-LLM uses the host memory. Note that we use a microbenchmark to show how the system scales in the extreme case. We choose cluster A because (1) it has both NVLink that we can leverage to scale for instances within a machine and (2) it still has cross-nodes scaling with not so fast RDMA, which requires our live autoscaling technique for acceleration.

Specifically, we can first see that as soon as the instance receives the scale out requests, BLITZSCALE can instantly start

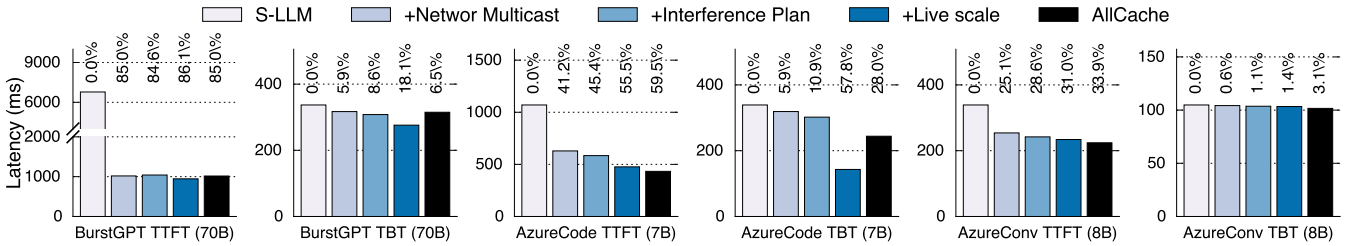

Figure 17: *An ablation study on the effectiveness of our proposed techniques. All the metrics reported are average.*

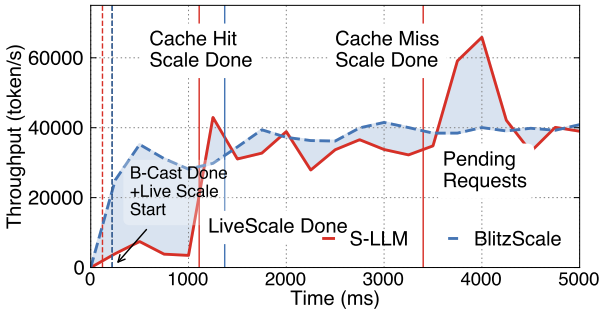

Figure 18: *A detailed look how* BLITZSCALE *and S-LLM scale a Llama 2 7B model from one cache copy to 16 GPU instances on cluster A.*

7 prefill instances through NVLink (B-Cast Done), reaching a prefill throughput of 20,000 tokens/sec. Meanwhile, it will start our live scale (+Live scale start), and gradually improving the throughput even before the scaling is done (at time 1,100). Meanwhile, in S-LLM, it can quickly start 7 instances on the machine with the cached copy, the overall scaling is is lengthy (finishes at time 3,100s). This is caused by slow SSD loading. Note that after all the scale is done, S-LLM has a throughput boost: this is because the pending requests at the queued instances are processed.

Albation study. We conclude our study of efficient autoscaling with an ablation study on the effectiveness of our proposed techniques. Figure 17 presents the results on all the three workloads. We First, we can see that using network multicast for scaling (+Network) can significantly improve the autoscaling performance, which in turn improves the average TTFT and TBT. There is a outlier, the TBT in AzureConv. This is because the decode workloads is relative stable on it so it nearly meets no scaling during our evaluation. The improvement is most obvious in BurstGPT where we evaluated with a 70 B model, because SSD is far from achieving an efficient autoscaling (discussed in details in §6.1). Second, an interference scale plan reduces the TTFT and TBT across various workloads by 0.4–5%. The improvement is marginal because even without the plan, we still have a chance to scale the instance without interference. Finally, live autoscaling further reduces the latencies by up to 40% on AzureCode TBT. This makes BLITZSCALE even faster than AllCache. This is because AzureCode requires scaling multiple decode instances while its prefill instances are relatively spare. Thus, we can directly mutate a prefill instance to a decode instance while lively scaling another prefill instance for remedy.

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 [68, 64, 37, 38, 66, 47, 25]. For example, Orca [66] proposes iterative-scheduling and selective batching to better utilize GPU resources when serving LLMs. AlphaServe [38] employs pipeline parallelism to better handle load spikes but it cannot adjust pipeline instances dynamically. DistServe [68] and Splitwise [47] disaggregates the GPUs into prefill and decode pools to better handle the unique characteristics of these two phases. These systems assume running on a fixed pool of GPUs, and we have shown the necessity of dynamically adjust pool size and how to achieve so efficiently with BLITZSCALE. Meanwhile, BLITZSCALE is orthogonal to single-instance model serving systems: we built upon them for fast model serving on a single instance, and additionally provide ultra-fast scaling when the number of serving instances fluctuates with workload demands.

Dynamic scaling serving instances. Dynamically scaling serving instances is challenging, mainly because the size of serving instance is huge due to gradually increasing model parameter size. We follow the trend of model autoscaling [12, 34, 39, 53]. For example, both PipeSwitch [12] and DeepPlan [34] leverages the layer-by-layer character of model, overlapping inference and parameter loading. But they focus on host-to-device loading. Furthermore, they cannot generate response exceeding bandwidth limit of physical link. SpotServe [39] and Llumnix [53] speeds up dynamic serving instances scaling by minimizing the cost of migrating tasks between instances. But they cannot utilize newly allocated GPU resources before all parameters are loaded, and donot optimize parameter loading between multiple instances. BLITZSCALE provides a new mechanism to scale serving instance lively, resulting in throughput increasing during parameter loading, which reduces tail latency of burst requests and thus improves serving quality.

Serverless computing. Accelerating model scaling is closely

related to coldstart acceleration in serverless computing [43, 4, 51, 21, 55, 49, 58]. Scaling model serving instances builds upon similar general-purpose acceleration techniques like container fast startup via checkpoint and restore [33]. Differently, it introduces a new bottleneck: data loading that cannot be eliminated by existing serverless methods as we have discussed in §4. BLITZSCALE overcomes this challenge by leveraging domain-specific knowledge of how models operate and our method applies to serving all known models.

8 Conclusion

Autoscaling is critical to achieve high goodput and hardware utilization in model as a service systems. Unfortunately, existing systems' slow and stop-the-world autoscaling significantly limits applicability of autoscaling. 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 demonstrate that the data plane can be made live through model-aware remote execution by breaking the scaling abstraction from instance-level to more fine-grained layer-level. Our system BLITZSCALE demonstrates orders of magnitude fewer tail latency than current solutions, showing a promising future for elastic model as a service systems.

Acknowledgment

We sincerely thank Qinwei Yang, Xinhao Luo and Mingcong Han for their expertise in GPU communication library, kernel, driver and runtime. We also thank Xiating Xie for helping refining the early draft of this paper. We thank Alibaba Tongyi lab for providing the testbed during the early stage of our research. This work was supported by a research funding from Huawei Cloud. Corresponding author: Xingda Wei (wxdwfc@sjtu.edu.cn).

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] 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.

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

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

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

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

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

[24] 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).

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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).

[32] 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.

[33] 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).

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

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

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] WANG, Y., CHEN, Y., LI, Z., KANG, X., TANG, Z., HE, X., GUO, R., WANG, X., WANG, Q., ZHOU, A. C., ET AL. Burstgpt: A real-world workload dataset to optimize llm serving systems.

[61] 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.

[62] 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.

[63] 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.

[64] 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).

[65] 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.

[66] 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.

[67] 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.

[68] 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.

A Appendix

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

Instance type	Accelerators	Local SSD BW/GPU	Remote SSD BW/GPU	Network BW/GPU	Has NVLink	Price
a2-ultragpu-8g [29]	8 x A100(80 GB)	2.58 Gbps	0.29 Gbps	12.5 Gbps	✓	40.44 USD/h
p4d.24xlarge[6]	8 x A100(40 GB)	2.31 Gbps	-	100 Gbps	✓	45.039 USD/h
ml.hpcpni2.28xlarge[22]	8 x A100(80 GB)	4 Gbps	-	100 Gbps	✗	48.23 USD/h
p4de.24xlarge[6]	8 x A100(80 GB)	2.31 Gbps	-	100 Gbps	✓	56.328 USD/h
a3-highgpu-8g[29]	8 x H100	6.09 Gbps	0.97 Gbps	100 Gbps	✓	88.25 USD/h
a3-megagpu-8g[29]	8 x H100	6.09 Gbps	0.97 Gbps	200 Gbps	✓	Unavailable
p5.48xlarge [6]	8 x H100	9.8 Gbps	-	400 Gbps	✓	Unavailable