

λ Scale: Enabling Fast Scaling for Serverless Large Language Model Inference

Minchen Yu^{†*}, Rui Yang^{‡*}, Chaobo Jia[†], Zhaoyuan Su[‡], Sheng Yao[§], Tingfeng Lan[‡], Yuchen Yang[§],
Yue Cheng[‡], Wei Wang[§], Ao Wang[◊], and Ruichuan Chen[△]

[†]The Chinese University of Hong Kong, Shenzhen [‡]University of Virginia

[§]Hong Kong University of Science and Technology [◊]Alibaba Group [△]Nokia Bell Labs

ABSTRACT

Serverless computing has emerged as a compelling solution for cloud-based model inference. However, as modern large language models (LLMs) continue to grow in size, existing serverless platforms often face **substantial model startup overhead**. This poses a significant challenge in efficiently **scaling model instances to accommodate dynamic, bursty workloads** commonly observed in real-world inference services. In this paper, we introduce λ Scale, an **efficient serverless inference system** to achieve **fast model scaling**. The key idea behind λ Scale is to leverage high-speed **RDMA networks** between GPU nodes for **fast model multicast**, while enabling distributed inference execution during model transmission—referred to as **“execute-while-load”**. λ Scale proposes an efficient model scaling scheme, **λ Pipe**, which supports adaptive model multicast and dynamically constructs execution pipelines across receiving nodes for collaborative, distributed inference. Additionally, λ Scale supports efficient **model management** across GPU and host memory, allowing fast scaling for models **across different storage tiers**. Evaluation results show that λ Scale enables fast model scaling and effectively handles load spikes, achieving up to 5 \times tail-latency improvement and 31.3% cost reduction compared to state-of-the-art solutions on real-world LLM inference traces.

1 INTRODUCTION

Recent advancements in machine learning (ML) have fueled a surging demand for cloud-based ML inference services [24, 29, 44, 51, 53]. Serverless computing offers a compelling cloud model for inference serving that can effectively handle dynamic request patterns [1, 20, 48, 49]. In this approach, users simply publish ML models, expose service endpoints, and delegate resource provisioning and scaling responsibilities to the cloud platform. Serverless computing is also economically appealing as users pay only for actual resource usage, eliminating the resource idling cost.

However, current serverless inference platforms suffer from significant **cold-start problems**, especially as modern models grow increasingly large and resource-intensive (e.g.,

large language models, or LLMs). The cold-start process, involving remote model loading and initialization, can result in **long startup delays up to several minutes** [27]. Such high latencies prevent platforms from **rapidly scaling out to handle highly dynamic workloads**, making it impractical to support online inference services with stringent latency requirements on the order of milliseconds or seconds [18, 27, 51, 54].

To mitigate cold starts, existing platforms typically **cache inference models in local GPUs or host memory during idle periods** and **resume execution upon request arrivals** [1, 22, 27, 48, 50]. However, these approaches have **limited scalability and incur high resource costs**. In multi-tenant serverless environments, workers typically need to handle multiple large models simultaneously, but storing all these models locally is infeasible and/or cost prohibitive due to limited GPU and host memory capacity. While recent works explore the use of local SSDs to expand storage capacity [27], the **data transfer between SSDs and GPUs introduces significant delays under dynamic workloads**. As a result, platforms either suffer from **unacceptable startup costs** or resort to **overprovisioning GPU workers** to maintain active model replicas at all times, leading to **high resource costs**.

An efficient serverless inference platform **should not settle for the fundamental tradeoff between startup latency and resource overprovisioning**. Ideally, it should **scale out rapidly in response to load spikes without incurring additional resource costs**. *Fast serverless model scaling is achievable through two key insights*. First, modern GPU clusters employ **high-speed network interconnects** (e.g., 400Gbps with RDMA capability) [10, 32, 35], providing opportunities for **efficient model multicast and fast scaling**. Second, model inference can begin before a node receives all model parameters, enabling collaborative, distributed inference execution across multiple nodes during model loading. This **“execute-while-load”** approach, combined with **low-latency model loading**, substantially improves system scalability under spikes.

Following these insights, we propose λ Scale, a **scalable serverless inference platform** that delivers fast and resource-efficient model scaling. λ Scale enables distributed inference execution during cross-node model loading (i.e., “execute-while-load”), leveraging high-speed RDMA networks and the

* Co-first authors.

GPUDirect RDMA (GDR) [16]. However, realizing this approach poses both algorithmic and system-level challenges.

A key challenge for λ Scale is to develop efficient solutions for cross-node model loading and distributed execution. λ Scale requires to accommodate varying model scaling demands and enable collaborative, distributed model execution in a dynamic environment where the model is being loaded. Existing distributed inference solutions do not meet such requirements, which typically lack support for dynamic, on-demand model scaling and rely on static resource allocations for distributed model execution [39, 48, 53]. To address this challenge, λ Scale introduces an abstraction of an *execution pipeline* and a novel scheme, *Dynamic Pipelined Execution with Multicast* (λ Pipe). This scheme partitions the model into fine-grained blocks for efficient multicasting while dynamically constructing execution pipelines — complete model instances spanning multiple receiving nodes for collaborative, distributed inference execution.

λ Pipe comprises three key designs. First, λ Pipe designs an *adaptive model-multicast mechanism* based on the binomial pipeline [28] algorithm. This mechanism optimizes the granularity and transfer order of the model blocks to enable the rapid construction of execution pipelines, thus improving overall inference performance (referred to as *Adaptive Model Multicast*). Second, λ Pipe judiciously groups nodes into execution pipelines at runtime to improve overall resource efficiency while minimizing data transfers during distributed inference. This strategy effectively improves system throughput and reduces request queueing under load spikes (referred to as *Pipelined Inference Execution*). Furthermore, once model loading is complete, λ Pipe allows workers to seamlessly switch to local execution mode without incurring cross-node overheads (referred to as *Mode Switching*).

In addition to λ Pipe, λ Scale must efficiently manage models cached across various media (e.g., GPU memory and host memory) to facilitate rapid model scaling from different storage tiers. To address this challenge, λ Scale proposes two key system designs. First, λ Scale employs a *holistic model startup mechanism* that adapts to varying model locality. It also enables model instances stored in both GPU and memory to collectively improve scaling performance (referred to as *Locality-driven Model Startup*). Second, λ Scale implements a *memory management system* that seamlessly consolidates data within model blocks for efficient model transmission and supports GPU memory pre-allocation for improved efficiency (referred to as *Efficient Memory Management*).

We have implemented λ Scale and evaluated it against state-of-the-art solutions, including *ServerlessLLM* [27], *FaaS-Net* [46], and *NCCL* [15]. λ Scale efficiently handles load

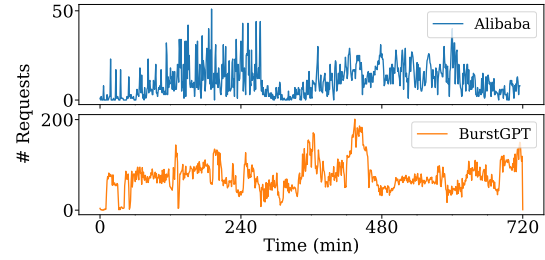


Figure 1: Normalized request rates of two representative serverless inference services. Trace 1 (top): a 12-hour serverless inference trace collected from Alibaba Cloud. Trace 2 (bottom): a 12-hour trace from a real-world LLM workload [47].

spikes, leading to a 2.4 \times to 5 \times improvement in 90th tail latency while reducing resource costs by 17.8% to 31.3% compared to the baselines on real-world LLM inference traces [47]. Microbenchmark results further show that λ Scale delivers fast model transmission and completes the scaling of Llama-13B across 8 nodes in less than 1 second, outperforming NCCL by up to 1.5 \times .

2 BACKGROUND AND MOTIVATION

2.1 LLM Inference

LLM inference autoregressively generates text output token by token from a user input (prompt) until it reaches the end-of-sentence (EoS) token or the maximum token limit. Each token depends on prior context, causing response times to vary. To speed up token generation, the model typically uses a KV cache [36] to cache the context from previous computations, reducing redundant work. Tokens are streamed to the user as they are produced, enabling real-time interaction. Performance is typically measured by *time-to-first-token latency* or TTFT (time to generate the first token) and *tokens-per-second* or TPS (token generation throughput).

2.2 Serverless Inference and Requirements

In recent years, serverless computing has emerged as a compelling option for hosting ML inference services—often termed “serverless inference” [20, 21, 27, 31, 43, 46, 48, 49, 51]. This approach allows users to simply publish models with inference code as functions, while cloud providers automatically manage resource provisioning, auto-scaling, scheduling, and fault tolerance. Serverless inference platforms offer fine-grained, pay-per-use billing [2, 4, 6], enabling substantial cost savings given the dynamic nature of inference workloads [24, 25, 29, 30, 34, 38, 43, 44, 53]. These advantages have led to a growing trend of serverless-based LLM inference services [11, 17, 27], where end-users simply query a backend LLM and receive its output tokens as responses.

Key to serverless inference is to enable rapid scaling of model serving instances to effectively handle real-world dynamic workloads. Fig. 1 shows 12-hour normalized request rates of two representative LLM inference workloads from two production serverless platforms: a serverless inference service at Alibaba Cloud [2] and a regional Azure OpenAI GPT service [47]. These traces reveal highly bursty request arrival patterns, with request rates surging by more than one order of magnitude within just a few minutes. Such bursty requests can overwhelm existing serving instances, leading to violations of the latency requirements defined by the Service-Level Objectives (SLOs) of inference services. Therefore, serverless inference platforms must rapidly scale out model serving instances to accommodate load spikes, ensuring low inference latency and SLO attainment. We detail the request serving results under these traces in §2.3.

2.3 Inefficiency of Existing Solutions

We next discuss existing serverless inference platforms and their issues in achieving fast model scaling.

Solution#1: Loading remote models. In serverless inference platforms, a typical scaling process involves retrieving models from remote model registries (e.g., Hugging Face) or storage services (e.g., S3) to GPU nodes, which then launch the model serving instances. As models continue to grow in size—particularly LLMs with tens or hundreds of billions of parameters—fetching them across the network becomes increasingly time-consuming. For instance, it takes over 18 minutes to load a Llama-70B model (140 GB) over a 1 Gbps network; and the overhead is further exacerbated by parallel model loading due to network bandwidth contention and registry throttling [46]. Worse, initializing and loading large models within GPU nodes can take anywhere from tens of seconds to minutes to complete [27, 50]. These significant startup overheads are incompatible with the requirements of real-time inference services, making the remote loading approach inefficient.

Solution#2: Overprovisioning GPUs. To mitigate the cold-start problem, existing serverless inference solutions opt to maintain a sufficient number of active function instances with reserved GPUs, even when these instances are idling [3, 5, 48]. This approach, however, leads to significant resource waste due to the dynamic nature of inference workloads. In particular, eliminating cold starts requires the platform to provision excessive GPU resources to accommodate peak loads (e.g., spikes in Fig. 1), which in turn results in substantial GPU idling during periods of low demand. Consequently, the extremely low GPU utilization directly contradicts the pay-per-use model fundamental to serverless computing [50].

Solution#3: Caching models in host memory and SSDs. Recent studies propose to cache models in host memory and then load them into GPU upon request arrivals [22, 27, 33, 50].

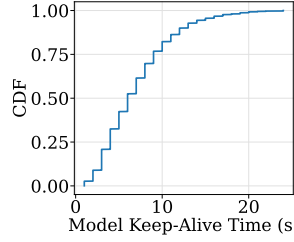


Figure 2: Distribution of models' keep-alive time in memory.

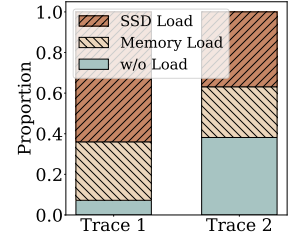


Figure 3: Proportion of the 3 types of model loading.

While this approach reduces the reservation cost compared to GPU overprovisioning, it still fails to achieve fast scaling in large-scale, multi-tenant serverless GPU clusters due to three key factors. 1) The dynamic and bursty nature of inference workloads often requires concurrent executions of a model on many nodes, sometimes involving the entire cluster. 2) Large-scale platforms typically host thousands of large models, resulting in massive host memory consumption. 3) Each node has limited host memory (e.g., up to 100s of GBs), which is often insufficient to accommodate even a few large models. Consequently, excessive host memory model caching becomes impractical, hindering the ability to achieve efficient model scaling.

Compared to host memory, SSDs offer larger capacities and can store more models [27]. However, loading models from SSDs to GPUs remains time-consuming due to the limited SSD bandwidth. Our measurements (see testbed in §7.1) show that loading a Llama-70B model from an SSD to a GPU takes over 30 seconds even with optimized implementations — an order of magnitude slower than loading from host memory. This delayed loading hampers the ability of serverless inference platforms to scale models quickly on demand.

To illustrate the inefficiency of this approach, we conduct two simulations based on real-world workload traces.

Short model keep-alive time in memory. We first examine how long a model instance remains in memory before being evicted in a multi-tenant inference platform. Each node's memory is configured to hold up to 3 models, while 12 models are stored in SSDs. This configuration is based on the size of Llama-70B and the hardware specifications of our testbed (see §7.1). We set each model's per-node request rate to 1 per minute, which reflects a typical request pattern in production serverless inference platforms [50]. We use the LRU eviction policy and depict the distribution of model's keep-alive times in Fig. 2. As we can see, models are frequently reloaded and evicted from memory, with over 95% of them staying in memory for fewer than 15 seconds before being evicted.

High cache miss ratio. Next, we measure the cache miss ratio for models cached in memory by replaying the two traces shown in Fig. 1. Based on the findings of Fig. 2, we set models'

keep-alive time to 15 seconds—the tail of the distribution. Fig. 3 shows the proportion of three loading cases across the two traces: model load from memory, model load from SSD, and a hot start (w/o load). We observe that **SSD loads (i.e., cache misses) account for 64% and 36% across the two traces**, respectively. This indicates that relying on memory caching alone is inadequate, resulting in a significant portion of slow model loads from SSDs or even remote storage, which severely impacts scalability and user experience.

In summary, existing solutions either suffer from long startup times or incur large GPU and memory costs in order to keep models active, forcing a rigid tradeoff between scaling efficiency and resource costs. This inherent limitation prevents the platform from achieving fast model scaling, a critical requirement for serverless inference.

2.4 Key Insights and Challenges

Key insights. To achieve fast model scaling, we present three key observations. **First**, **modern GPU clusters increasingly adopt high-speed interconnects between GPU nodes** (e.g., 100-400Gbps with RDMA capability) [10, 32, 35] and support advanced data transfer techniques such as GPUDirect RDMA or GDR [16]. **Second**, **multicast-based collective communication techniques are inherently well-suited for cross-node model scaling**, enabling rapid distribution of model instances across receiving nodes. **Third**, **models can begin computation before being fully loaded**, enabling an “execute-while-load” approach. This allows inference tasks to be distributed and executed across multiple nodes in parallel during model loading.

Following these observations, we advocate that a scalable serverless inference platform should synergize the “execute-while-load” approach with efficient cross-node model loading. Specifically, models should be rapidly transferred and replicated across worker nodes, while new worker nodes perform collaborative, distributed inference execution. This approach allows the platform to handle load spikes on demand, maintaining low request queueing delays without the need to keep models long-lived in GPUs or host memory.

Challenges. Realizing the idea above presents three key challenges. **C1: Low-Latency Model Loading.** The platform must handle varying scaling demands while consistently achieving high scaling performance. Hence achieving efficient, low-latency model loading across nodes remains a challenge. **C2: Dynamic Model Execution.** Unlike existing distributed inference solutions designed for static environments [39, 53], the “execute-while-load” approach requires collaborative, distributed inference among nodes when the model is being loaded. This introduces a challenge for the platform to dynamically configure and optimize inference execution plans at runtime. **C3: Efficient Model Management across Storage Tiers.** Maintaining models in host memory is

a prevalent practice to accelerate model loading [22, 27, 50]. Therefore, the platform should efficiently manage models across various storage tiers, including GPU memory and host memory, while delivering fast model scaling.

In the following sections, we present λ Scale, a scalable serverless inference platform to address the aforementioned challenges. We provide an overview of λ Scale in §3 and delve into its detailed designs in §4 and §5.

3 λ SCALE OVERVIEW

Design choices. The key to achieving fast model scaling is efficient cross-node communication. While collective communication libraries such as *NCCL* [15] are widely adopted in GPU clusters, they are typically optimized for long-running, static environments such as large-scale model training. For example, we find that *NCCL* lacks the flexibility to handle frequent reconfigurations of communication groups when scaling model-serving instances under dynamic workloads [9], which introduces additional overhead that delays end-to-end model loading (see §7.2).

Therefore, we opt to implement a lightweight, yet efficient, multicast framework in λ Scale based on the binomial pipeline [23, 28] algorithm to achieve scalable model inference. This approach is well-suited to address the challenges outlined in §2.4. Compared to other multicast solutions such as binary tree [46], the binomial pipeline generally delivers superior multicast performance (see §7). By leveraging fast data transfer techniques like RDMA and GDR, λ Scale can efficiently implement and optimize the binomial-pipeline-based multicast, achieving low-latency model loading (**C1**). Additionally, this approach provides high flexibility to orchestrate fine-grained execution tasks at runtime and manage models across storage tiers, which facilitates dynamic model execution (**C2**) and efficient model management (**C3**).

Overview. Fig. 4 illustrates an architecture overview of λ Scale. λ Scale runs a cluster manager to dispatch end-user queries to worker nodes, manage global resources, and coordinate model scaling and pipeline execution. λ Scale implements an efficient model scaling scheme— λ Pipe—on the model scaling and pipeline execution controllers. Specifically, the model scaling controller coordinates fine-grained model distribution across participating nodes, in which a model is partitioned into blocks and transmitted using a binomial-pipeline-based approach. In this approach, the nodes are organized into a hypercube communication topology¹, where each node transmits model blocks to its adjacent nodes. The number of time steps required to complete block transmission is proven to be optimal (see details in [28]). Additionally, the pipeline execution controller is responsible for distributing inference execution across nodes during model scaling. It

¹This is achievable in real-world GPU clusters, which often use optimized network topologies such as fat trees [19] to ensure efficient communication.

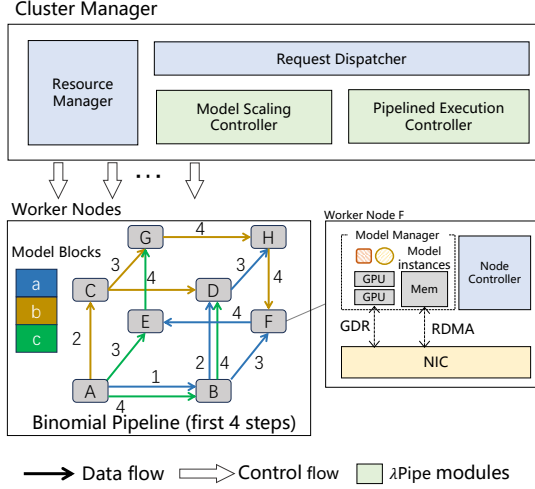


Figure 4: λ Scale architecture overview. In this example, Node A initiates a binomial pipeline multicast for a model partitioned into three model blocks. Each participating worker node transmits a model block in a sequential step (indicated by numbered labels along the data flow arrows). A receiver node forwards the blocks it has received to its neighbors (e.g., Node B forwards block *a* at steps 2 and 3). The color-coded model blocks correspond to the data flow paths.

follows optimized block transfer orders to judiciously distribute block-level inference tasks among participating nodes to improve overall performance. (see details in §4).

Each worker node deploys user-provided models as model-serving instances and operates a node controller that synchronizes with the cluster manager, reports local status, and coordinates node-level tasks. Additionally, λ Scale runs a model manager at each node to track local resources such as GPUs and host memory and manage model instances. The model manager is responsible for model execution and transmission tasks according to the instructions of the node controller. It leverages GDR to efficiently exchange data across GPUs on different nodes, bypassing the data movement through the host to GPUs. It also supports direct access to models stored in remote memory via RDMA. These designs effectively improve network resource efficiency and distributed inference performance.

We next describe how λ Pipe achieves efficient model loading and dynamic model execution (**C1 and C2**) in §4, and then discuss λ Scale’s model management (**C3**) in §5.

4 λ PIPE DESIGN

4.1 Design Rationale and Overview

λ Pipe aims to efficiently support collaborative, distributed inference execution before nodes receive the entire model. The key objective is to maximize the overall system throughput —measured in tokens generated per second for LLM

inference—which in turn reduces request queueing under load spikes. We therefore design λ Pipe based on *two key principles*. 1) Minimizing delays in assembling complete model instances across nodes enables distributed inference to begin as early as possible. 2) Reducing data movement (e.g., KV caches in LLMs) across nodes enhances overall efficiency.

λ Pipe introduces the abstraction of an *execution pipeline* to facilitate distributed inference. An execution pipeline serves as a model-serving instance spanning a group of nodes that collectively maintain a complete model and jointly perform pipeline parallelism (Fig. 5). Inference requests are assigned to specific execution pipelines, and the designated pipeline iteratively computes all output tokens (Fig. 6 (a)). This minimizes the amount of intermediate results exchanged between nodes and eliminates the need to transfer the KV cache, ensuring efficient distributed execution.

λ Pipe provides efficient supports for execution pipelines through three key designs. First, λ Pipe extends the binomial pipeline to enable adaptive model multicast, supporting fast model distribution under various scaling scenarios and minimizing the time required to generate execution pipelines. Second, λ Pipe adopts an efficient strategy to generate execution pipelines and fully leverage the advantages of adaptive model multicast, which improves overall inference performance. Finally, λ Pipe allows participating nodes to switch to local execution mode once they have received the full model replica. In addition to cross-node scaling, execution pipelines can also be applied in memory-based model loading, which we describe in §5.

4.2 Adaptive Model Multicast

Multicast modeling and sub-groups. We begin by modeling the multicast process in λ Pipe. Consider a cluster of N nodes and a scaling scenario where the source node holds a model instance and needs to distribute it to the remaining $N - 1$ destination nodes. We denote this scaling process as $1 \rightarrow N$. Assume the size of model is M and it is partitioned into b blocks for multicast. Following the binomial pipeline (Fig. 4), $1 \rightarrow N$ takes $b + \lceil \log N \rceil - 1$ steps to complete, where all nodes can have at least one block after the first $\lceil \log N \rceil$ steps and take remaining $b - 1$ steps to receive other blocks. Let t be the time required for each step and T be the total end-to-end multicast time. We have $t \propto M/b$, i.e., in proportion to the amount of data exchanged between each pair of nodes in one step, and $T \propto M(1 + \log N/b)$.

More generally, we consider a scaling operation $k \rightarrow N$, i.e., k nodes distributing the model to the remaining $N - k$ nodes (where $1 \leq k < N$)². λ Pipe evenly divides the N nodes into k sub-groups. Each sub-group consists of L nodes, where L is either $\lfloor N/k \rfloor$ or $N\%k$, and each sub-group

² $k \geq 1$ can be easily met in practice, such as by maintaining at least one model replica in host memory across the cluster.

Algorithm 1 k -Way Transmission Strategy

Input:

- k sub-groups $\{G_0, \dots, G_{k-1}\}$
- b ordered model blocks $\{M_0, \dots, M_{b-1}\}$

Output:

- Block transfer orders for k sub-groups $\{O_0, \dots, O_{k-1}\}$

```

1:  $l \leftarrow \lceil b/k \rceil$  ▷ Size of block chunks
2:  $\mathcal{S} \leftarrow \{\{M_j \mid j \in [l \cdot i, \min(l \cdot (i+1), b) - 1]\} \mid i \in [0, k-1]\}$ 
   ▷ Partition blocks into  $k$  chunks
3: for  $i \in [0, k-1]$  do ▷ Generate  $O_i$  via circular shift
4:    $O_i \leftarrow \bigoplus_{j=0}^{k-1} \mathcal{S}_{(i+j) \bmod k}$ 

```

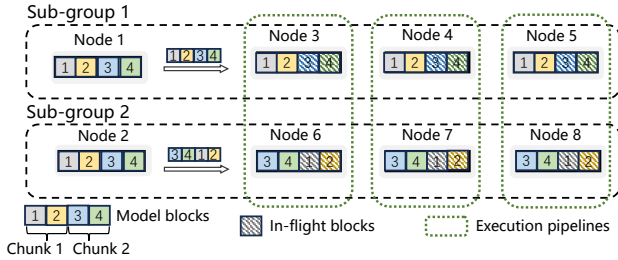


Figure 5: Example of $2 \rightarrow 8$ scaling. Each sub-group transfers model chunks in circularly shifted order in parallel, to construct three pipeline parallel inference execution flows (Node 3 and 6, Node 4 and 7, and Node 5 and 8). This strategy allows multiple execution pipelines to be started as soon as enough model blocks are distributed while enabling non-blocking binomial pipeline multicast.

performs a $1 \rightarrow L$ scaling. This process supports $k \rightarrow N$ scaling for arbitrary values of k and N .

Selective block sizes. Determining the block size b is critical to the performance of model multicast. λ Pipe selectively configures this parameter to balance the transmission performance and execution efficiency. We note that coarse-grained model partitioning (i.e., a small b) often results in longer end-to-end transmission times (i.e., a large T), while increasing b generates more model blocks and leads to additional communication overhead for intermediate results during pipeline execution. According to our modeling, T exhibits an "elbow point" with respect to b : increasing the block size initially enhances transmission performance, but the benefits diminish beyond a certain threshold (detailed in Fig. 18). Therefore, λ Pipe configures this point as b , achieving good transmission performance while minimizing the additional overhead in pipeline execution. Notably, configuring b requires only offline profiling and has no impact on runtime performance.

Optimized transfer order. We propose a k -way transmission strategy to optimize the order of model block transfers across k sub-groups, which enables λ Pipe to minimize the time required for assembling the complete model. Algorithm 1 outlines the k -way transmission strategy. It first

partitions the model blocks into k equal-sized chunks (lines 1-2) and then generates the block transfer orders for each sub-group by circularly shifting these chunks (lines 3-4). This design ensures that the sub-groups work in tandem, with the first complete model instance becoming available after just b/k time steps. Fig. 5 illustrates a $2 \rightarrow 8$, 2-way transmission scaling scenario, where two source nodes (Node 1 and Node 2) distributed model blocks to six destination nodes across two sub-groups. The model is partitioned into four blocks, which are grouped into two equal-sized chunks: Blocks 1-2 and Blocks 3-4. To improve the overall throughput, each sub-group follows a **circular shifting strategy** when transferring model blocks. Specifically, Sub-group 1 (Node 1 as the source) starts by sending model blocks in the order of Blocks 1-2 and Blocks 3-4 to its destination nodes (Node 3-5). Meanwhile, Sub-group 2 (Node 2 as the source) follows a complementary pattern, transferring model blocks in the **circular shifted order** (Blocks 3-4 followed by Blocks 1-2) to its destination nodes (Node 6-8). This circular shifting of block chunks ensures that different groups of destination nodes receive different parts of the model concurrently, allowing them to collaboratively construct three model inference execution pipelines using pipeline parallelism. As the multicast progresses, the destination nodes incrementally collect and assemble complete model instances, enabling efficient and scalable model execution.

Algorithm 2 Execution Pipeline Generation Strategy

Input: – k sub-groups $\{G_0, \dots, G_{k-1}\}$

– L_i : the number of nodes in G_i

– n_i^j : the j^{th} node in G_i

Output: – All generated execution pipelines \mathcal{P}

```

1:  $\mathcal{G} \leftarrow$  sub-groups with unassigned nodes
2: while  $|\mathcal{G}| > 0$  do
3:   if  $|\mathcal{G}| = 1$  then ▷ A pipeline within a single sub-group
4:      $P \leftarrow$  ordered nodes in  $\mathcal{G}_0$ 
5:      $\mathcal{P} \leftarrow \text{append}(\mathcal{P}, P)$ 
6:   else ▷ Generate pipelines across sub-groups
7:      $a \leftarrow \min_{G_i \in \mathcal{G}} L_i$ 
8:     for  $t \in [0, a-1]$  do
9:        $P \leftarrow []$ 
10:      for  $G_i \in \mathcal{G}$  do
11:         $P \leftarrow \text{append}(P, n_i^t)$ 
12:       $\mathcal{P} \leftarrow \text{append}(\mathcal{P}, P)$ 
13:   Update  $\mathcal{G}$ 

```

4.3 Pipelined Inference Execution

Generating execution pipelines. Building on the optimized block transfer order, we propose an efficient execution pipeline generation strategy outlined in Algorithm 2. The key idea is to construct execution pipelines from as many sub-groups as possible, maximizing the benefits of k -way

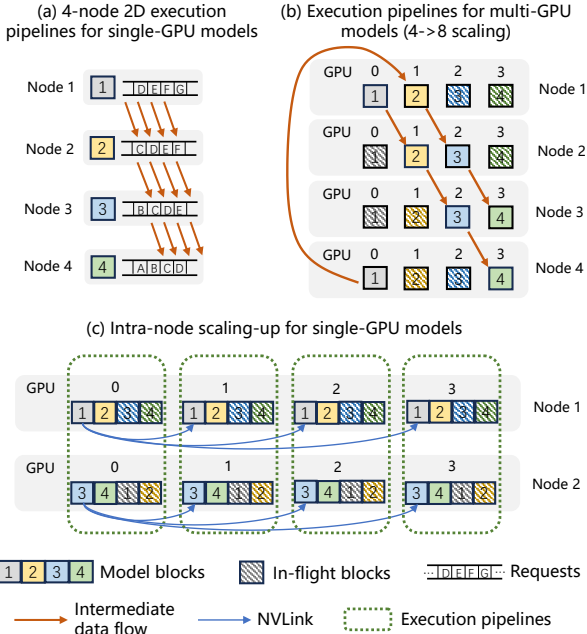


Figure 6: Example of execution pipelines. (a) A 4-node execution pipeline where each node executes its associated model block for multiple in-flight requests in parallel. (b) Execution pipelines across multi-GPU nodes where each pipeline contains GPUs from different nodes. (c) Intra-node model replication to scale up the inference performance, where each local model block replica forms a separate cross-node execution pipeline.

transmission. Specifically, when the remaining unassigned nodes belong to only one sub-group, λ Pipe directly forms an execution pipeline using these nodes (lines 3-5). Otherwise, it selects one node from each available sub-group to construct the pipeline (lines 6-12). λ Scale prioritizes pipeline configurations where each sub-group has an equal number of nodes (i.e., $N \bmod k = 0$).

2D execution pipelines. Fig. 6 (a) illustrates how a 4-node execution pipeline processes multiple batches (one or multiple requests) of inference requests in parallel using a 2-dimensional pipelining strategy. Along the first dimension, each node is assigned a specific model block and computes a different batch of requests. Once a node completes processing a batch for its assigned block, it passes the intermediate result to the next node and starts computing the next batch along the second dimension. This 2D pipeline strategy efficiently utilizes resources distributed along the multicast route, enabling rapid scaling and processing of accumulated requests during spikes. λ Scale schedules requests across multiple pipelines based on their available resources to improve overall resource efficiency.

Models on multiple GPUs. When a model spans multiple GPUs, λ Pipe generates execution pipelines across GPUs

within the same node and/or from nodes to minimize performance loss during model scaling. Fig. 6 illustrates the multi-GPU model scaling scenario, where each model instance is distributed across multiple GPUs. During scaling, as model blocks partially arrive, λ Pipe dynamically selects one of the three execution strategies based on model size and resource availability (i.e., whether a node has multiple GPUs).

Case 1: Cross-node execution pipeline for single-GPU models (models fitting in a single GPU): This is the default execution strategy already described in §4.2. **Case 2: Cross-node execution pipeline for multi-GPU models** (models that do not fit in a single GPU): GPUs that have received complete blocks can immediately begin forming execution pipelines across nodes to support distributed, pipelined inference for large models, without waiting for the full model to load (see Fig. 6 (b)). **Case 3: Intra-node scaling-up for single-GPU models:** λ Pipe can opportunistically leverage multiple local GPUs on the same node to accelerate scaling if there are available local GPU resources on the same node. As soon as the first GPU receives model blocks, it can quickly replicate them to other local GPUs using high-speed node-local communication mediums like NVLink, which offers bandwidth up to an order of magnitude higher than RDMA networks (see Fig. 6 (c)). This fast intra-node replication enables rapid local model scaling-up. Replicated model blocks can then form cross-node execution pipelines (Case 1). This hybrid approach maximizes resource utilization and enhances inference performance by opportunistically reducing data movement overhead.

4.4 Mode Switching

After model scaling is complete, each node maintains a full model instance and directly serves inference requests locally³. A key challenge is to ensure that each node keeps corresponding runtime states (i.e., KV caches) for the active requests it serves. λ Pipe addresses this challenge through *KV cache recomputation*. It evenly distributes incomplete requests of an execution pipeline among all participating nodes, and then each node recomputes its assigned requests using available tokens generated by distributed inference during the multicast phase. This approach generally incurs lower overhead compared to transmitting existing KV caches, which requires costly all-to-all communication across participating nodes. This mechanism ensures λ Pipe to seamlessly switch from pipelined execution mode to local execution mode while maintaining low inference latency.

³Current λ Scale supports models that fit within the GPU capacity of a single multi-GPU node (e.g., up to 640 GB of collective GPU memory for an 8-A100 node). We leave the support for larger models spanning multiple nodes for future work.

5 EFFICIENT MODEL MANAGEMENT

λ Scale supports efficient model management in both host memory and GPUs through two key system designs.

Locality-driven model startup. λ Scale introduces a multi-level, locality-driven model startup scheme to efficiently support model instances stored in various storage tiers. λ Scale optimizes locality using several startup strategies: **(1) GPU (hot start)**: The model is fully loaded into GPU memory, enabling fast, local execution. **(2) Memory (warm start)**: The model is cached in host memory. λ Scale directly loads host-memory-cached models into GPUs for inference execution, and before models are fully loaded, constructs an execution pipeline across multiple nodes of this kind for enhanced inference performance (λ Pipe in §4). **(3) Null (cold start)**: The model is neither cached in GPU or host memory, requiring λ Scale to perform cold-start scaling by directly retrieving model blocks from remote GPU and/or Memory nodes.

Efficient memory management. λ Scale employs two strategies to manage GPU and host memory for model blocks. **(1) Tensor packing**: λ Scale maps each model block to a contiguous memory region for enhanced transmission efficiency. By consolidating all tensor data associated with a single model block into a contiguous memory chunk, λ Scale enables bulk transfer of entire blocks, improving bandwidth efficiency. Notably, the tensor memory layout optimization has no impact on inference execution. **(2) GPU memory pre-allocation**: λ Scale pre-allocates memory chunks for model blocks and intermediate results, as their sizes remain consistent across requests during pipeline execution. Runtime states with dynamic memory requirements (e.g., KV caches) are internally managed by inference engines (e.g., vLLM [37]). This design ensures memory efficiency while minimizing memory allocation overhead at runtime.

6 λ SCALE IMPLEMENTATION

We have implemented λ Scale in 10K lines of Python and 4K lines of C++, structured into two core components: the cluster manager and worker nodes. λ Scale’s source code will be released upon the acceptance of the paper.

The **Cluster Manager** is implemented in Python (see Fig. 4). Each worker node contains a **Model Manager**, which consists of two key modules: (1) an *inference module*, responsible for executing both local inference (within a GPU) and distributed inference (across nodes) and (2) a *transfer module*, which implements GDR/RDMA-based model block transfer. For the inference module, we extend the codebase of Meta’s Llama inference framework [14] to support both local and distributed inference using Python. For the transfer module, we implement a standalone RDMA P2P transfer library in C++ that supports both GPU and host memory using libibverbs [12], leveraging the RDMA read operation with a pull-based pattern. The binomial pipeline algorithm

Table 1: Testbed Configurations. Each node is equipped with 1TB RAM and 4TB of NVMe SSD local storage.

Testbed	GPU	NIC	Memory	SSD	#Nodes
Testbed1	1×H800	1×400Gb/s IB	64GB/s	5GB/s	12
Testbed2	4×H800	1×400Gb/s IB	64GB/s	5GB/s	6

is implemented in **Cluster Manager** and its GDR/RDMA semantics are implemented in **Model Manager**’s transfer module. All the key RDMA P2P transfer APIs (e.g., RDMA queue pair establishment and RDMA read operation) are exposed to Python via Pybind11 [13].

7 EVALUATION

We evaluate λ Scale by addressing three key questions: **(1)** How fast can λ Scale distribute model blocks across a GPU cluster (§7.2)? **(2)** How does λ Scale scale LLM inference performance compared to state-of-the-art baselines under concurrent, stress-test workloads (§7.3, §7.4)? **(3)** How elastic and cost-effective is λ Scale under a bursty real-world LLM workload (§7.5)? **(4)** How do different design choices and configurations impact λ Scale’s performance? (§7.6)

7.1 Experiment Setup

Testbeds. All experiments are conducted on a shared HPC cluster, with an exclusive allocation of up to 24 NVIDIA H800 GPUs and 12 nodes per user. We configure two testbeds to evaluate λ Scale under different scalability scenarios (Table 1). Testbed1 is used when a single GPU is sufficient to host the entire model (e.g., Llama-2 7B), allowing to stress the scalability test via inter-node communication. Testbed2 is used when a single GPU is not large enough to host a model (e.g., Llama-2 70B), requiring multiple GPUs per node to validate scalability while involving inter-node communication (e.g. model parallel inference).

Models and configurations. We consider two primary experimental parameters: model size and k (refer to k -way transmission in §4.2 for details). We test λ Scale with the Llama-2 [45] series LLMs with 7B, 13B, and 70B parameters and a k value ranging from {1, 2, 4}. Experiments for Llama-2 7B and 13B are conducted on Testbed1, while Llama-2 70B tests run on Testbed2. Unless explicitly stated, all configurations follow these defaults.

Measurement metrics. Our evaluation focuses on key metrics including throughput, latency, and cost-effectiveness. (1) Throughput (tokens per second) measures λ Scale’s ability to sustain high-load inference requests. (2) Latency (time-to-first-token) reflects λ Scale’s efficiency in generating the first token quickly, a critical metric for low-latency LLM serving. (3) Cost-effectiveness (GPU time) evaluates how elastically λ Scale provisions and releases GPU resources.

Baselines. We compare λ Scale against three baselines, covering both industry-standard and state-of-the-art research systems as discussed in §2.3.

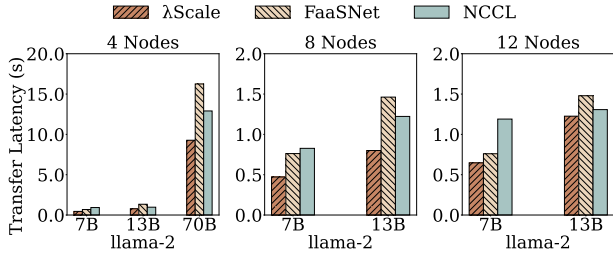


Figure 7: End-to-end model multicast latency. The 4-node test involves 16 GPUs, with each node containing 4 GPUs. The 8-node and 12-node tests use 8 GPUs and 12 GPUs, respectively. All tests have $k = 1$ (a single source).

ServerlessLLM [27]: The state-of-the-art serverless LLM inference system designed for dynamic scaling. We implement *ServerlessLLM* and remove Ray Serve’s [41] cluster management overhead to isolate its inference performance.

FaaSNet [46]: An industry-adopted serverless function container provisioning system that optimizes P2P transfer topology for auto-scaling. We use its default binary tree topology and extend it to support GDR-based model loading. *NCCL* [15]: An industry-standard communication library for multi-GPU training and inference developed by NVIDIA, optimizing collective communication primitives such as all-reduce and broadcast using GDR. Since *NCCL* lacks native multicast support, we adapt its broadcast primitive by dynamically forming process groups and transmitting model blocks to designated groups, effectively enabling multicast.

7.2 Multicast Performance

We first evaluate λ Scale’s raw block transmission latency achieved under binomial pipeline multicast.

Model transfer latency. Fig. 7 shows the end-to-end multicast latency with different GPU cluster settings. Overall, λ Scale achieves up to a 1.82 \times and 1.53 \times speedups over *FaaSNet* and *NCCL*, respectively. We observe that λ Scale’s multicast performance advantage increases with both model size and cluster scale. For smaller models on fewer nodes (e.g., 7B on 4 nodes), λ Scale is only modestly faster than the other systems; however, with larger models and more nodes (e.g., 70B on 12 nodes), λ Scale’s performance benefit expands considerably. This improvement comes from λ Scale’s binomial pipeline, which splits a model into blocks and utilizes the entire cluster bandwidth resource to transmit blocks, maximizing link utilization and GPU-level parallelism. We explain the performance differences next when investigating fine-grained block arrival latencies.

Block transfer latency. To zoom in, we measure the fine-grained block arrival latency for the Llama-2 13B model (Fig. 8). We observe that λ Scale receives the first and last model blocks almost at the same time across all cluster sizes. Interestingly, while λ Scale and *NCCL* perform comparably

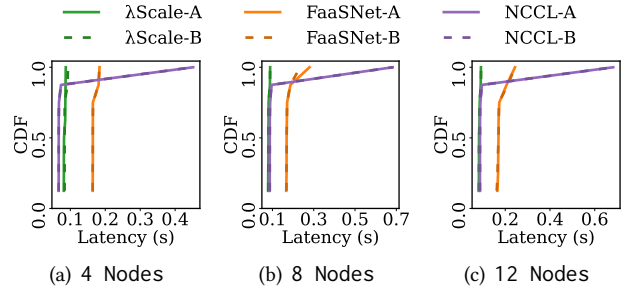


Figure 8: Model block transfer latency CDF. We randomly select two nodes (A and B) from the cluster and report the arrival latency for each block. Other nodes show similar pattern.

for most blocks, *NCCL* experiences a significantly high tail latency for the *first* block, due to high group initialization overhead [9]: *NCCL* requires creating a group for target processes before performing the broadcast. However, establishing a new group incurs high cost of up to hundreds of milliseconds as we observe in our tests. Another trend is that *FaaSNet*’s tail latency grows as the cluster size increases, while λ Scale maintains consistently low latency for all blocks. This is because *FaaSNet*’s tree-based multicast algorithm (binary tree) limits parallelism to a fixed number of children nodes at the bottom of the topology [46], while λ Scale’s binomial pipeline effectively improves the network I/O parallelism (# of active senders) at each step.

7.3 Throughput Performance

We stress-test λ Scale’s application-level throughput.

Scaling via GDR. We measure λ Scale’s throughput scaling ability under high-stress loads by varying k and compare it against *ServerlessLLM*, *FaaSNet*, and *NCCL*. λ Scale (green) consistently outperforms the baselines by achieving peak throughput significantly faster across various k levels. One key observation is that, while *FaaSNet*, *NCCL*, and *ServerlessLLM* steadily scale their throughput on a similar timeline (e.g., 0.6s for *FaaSNet* for Llama-2 7B), λ Scale effectively halves its ramp-up time when its k increases. For the Llama-2 13B example, λ Scale begins scaling at around 1.2s when $k = 1$, whereas with $k = 4$, scaling starts significantly earlier, at around 0.25s. This is because λ Pipe’s block transfer and opportunistic execution pipeline allows GPUs to collaboratively load and serve requests as soon as sufficient model blocks are loaded into GPU memory, rather than waiting for the full model to be transferred (see §4.2). Meanwhile, *ServerlessLLM*-SSD suffers from a dramatically longer ramp-up period for throughput scaling, primarily because: (1) it lacks GDR/RDMA multicast support, relying solely on local SSD, which slows down model loading; (2) it waits until the entire model is loaded into a GPU before serving requests;

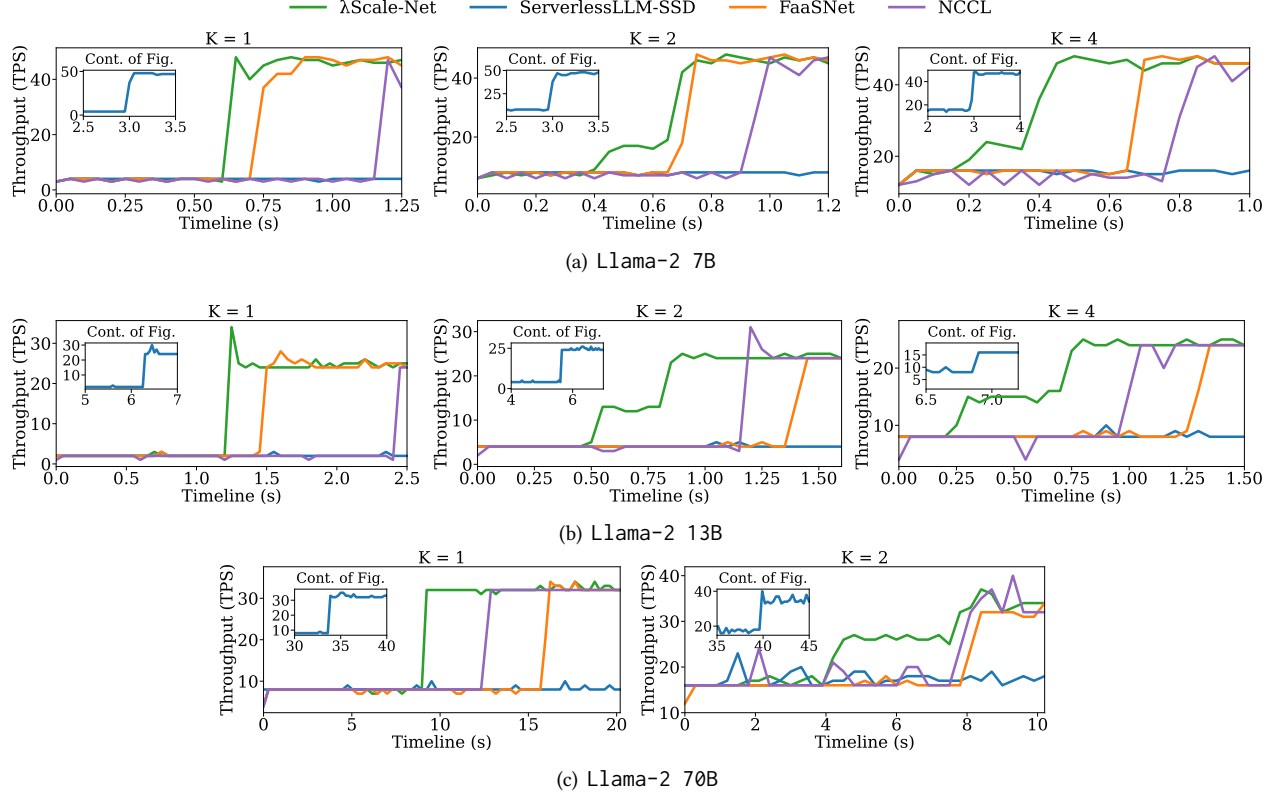


Figure 9: Throughput scaling via GDR with varying model sizes. *ServerlessLLM* relies on local SSDs during scaling, while all other systems use GDR for inter-node communication. The k configurations apply only to λ Scale. The mini-plots show the extended timeline for *ServerlessLLM*.

and (3) even for local SSD-based loading, it does not incorporate LLM-specific optimizations like λ Scale’s k -way transmission and optimized transfer order. *FaaSNet* and *NCCL* leverage GDR in multicast, making them more efficient than *ServerlessLLM*. However, they lack support for collaborative distributed execution and have lower transmission parallelism, which limits their scaling efficiency. Another notable observation is that, when $k \geq 2$, λ Scale exhibits a staircase-shaped performance plateau (e.g., between 0.25s and 0.75s for $k = 4$ with Llama-2 13B). This behavior is likely due to implementation overhead, which may not be fully optimized. Addressing this performance plateau is part of our future work.

Scaling via local cache. A key design of *ServerlessLLM* is its use of host memory caching to improve performance by loading models from local host memory. For fair comparison, λ Scale is configured to utilize local memory caching similarly to *ServerlessLLM*. Specifically, both systems are configured such that R out of N nodes have models loaded into their GPUs, while the remaining k nodes in the cluster load the model from the local host memory cache ($N = R + k$). For this evaluation, k is fixed at 8 for Llama-2 7B and 13B, and

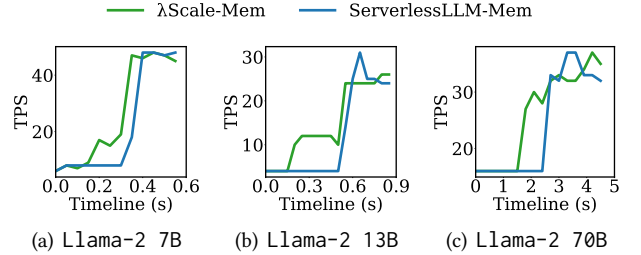


Figure 10: Throughput scaling via local cache with varying model sizes.

at 2 for Llama-2 70B. Fig. 10 shows that λ Scale scales 2 \times to over 4 \times faster than *ServerlessLLM* in many scenarios. A key difference lies in that λ Scale’s λ Pipe enables collaborative inference during the loading phase, allowing requests to be served across k nodes as soon as the first model block becomes available in GPU. In contrast, *ServerlessLLM* waits until the model is fully loaded before processing requests.

Cold-start comparison. We evaluate throughput performance under a cold-start scenario, where no model instances are preloaded into GPUs across the cluster. Instead, one node

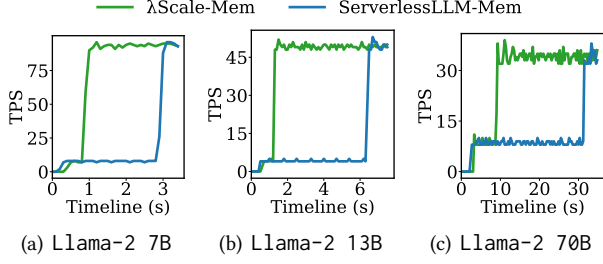


Figure 11: Cold-start throughput (scaling via GDR without models preloaded to GPUs, with $k = 1$).

stores the model in its host memory cache. In *ServerlessLLM*, the remaining nodes load the model from SSD, while in λ Scale, the cached model is loaded into local GPU(s) and multicasted across the cluster via GDR. To ensure fairness, both systems use $k = 1$. As shown in Fig. 11, λ Scale significantly outperforms *ServerlessLLM* by $3.75\times$ to $11.4\times$ across the three model sizes. This improvement is due to λ Scale’s λ Pipe, which allows immediate request serving as soon as the first block is loaded into GPU, eliminating unnecessary delays.

7.4 Latency Performance

Next, we measure the TTFT latency across varying loads. **Scaling via GDR.** Fig. 12 (a)-(c) plots the TTFT latency and Fig. 12 (d)-(f) shows the zoomed-in CDF given a specific RPS (requests per second) level. We observe that λ Scale starts serving all 50 requests in 1.1s, which is $2\times$, $1.4\times$, and $8\times$ faster than *FaaSNet*, *NCCL*, *ServerlessLLM*, respectively. The 7B and 70B models exhibit a similar trend. *ServerlessLLM*-SSD suffers from a long-tail TTFT latency, caused by: (1) slow SSD I/Os during on-demand loading, and (2) delayed inference execution due to waiting for the entire model to be fully loaded into GPUs.

Scaling via local cache. Similarly, Fig. 13 (a)-(c) shows the TTFT latency of λ Scale and *ServerlessLLM* when scaling with local memory cache and Fig. 13 (d)-(f) shows the zoomed-in CDF, under the same setup used in Fig. 10 (§7.3). As shown in Fig. 13(b), λ Scale is $1.63\times$ faster than *ServerlessLLM* at the 90th percentile of requests. Even in the best-case scenario for *ServerlessLLM*, where local memory caching is used, λ Scale still outperforms *ServerlessLLM* in TTFT. This is, once again, due to *ServerlessLLM*’s lack of transmission and execution pipeline.

7.5 Real-World LLM Workload

In this section, we evaluate λ Scale’s performance using BurstGPT [47], a real-world LLM workload trace collected from regional Azure OpenAI GPT services. The original workload is highly bursty and we select a 30-minute trace snippet from the workload for evaluation. We make the following assumptions: *NCCL* and *FaaSNet* prioritize loading

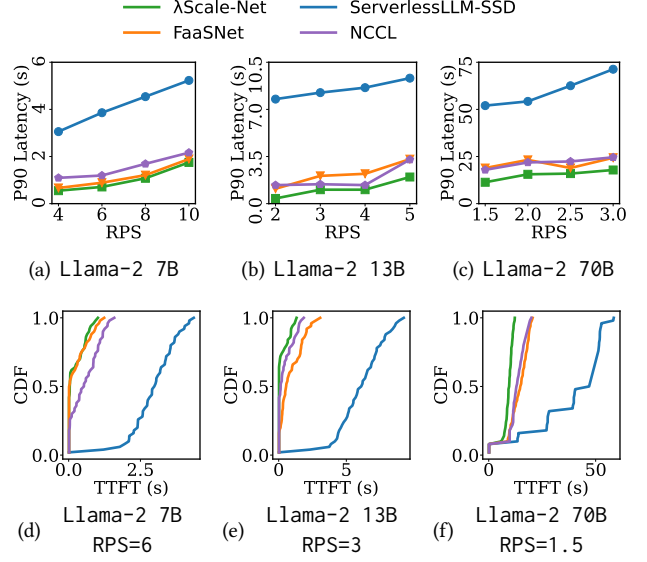


Figure 12: Latency scaling via GDR.

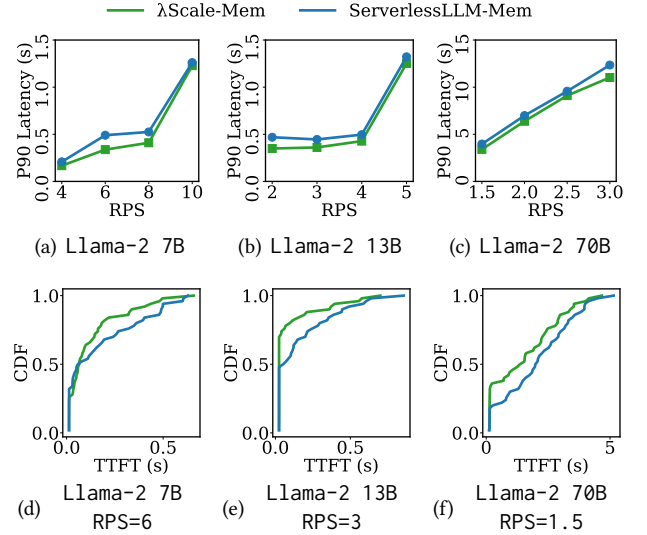


Figure 13: Latency scaling via local cache.

models from remote GPUs using GDR and only fall back to local SSD load if none of the GPUs in the cluster have an available model instance. *ServerlessLLM* relies solely on local-cache-based loading—it loads models from host memory on a cache hit and from SSD on a cache miss. As *ServerlessLLM*, λ Scale supports best-effort local host memory caching but falls back to λ Pipe multicast if the requested model is not in host memory.

Scaling behaviors. Fig. 14 shows the dynamic GPU allocation timeline in response to fluctuating RPS. *Ideal Scaling* assumes zero model-loading overhead, where a model can be instantly loaded into and swapped out of GPU(s) without delay. This is not practically achievable due to real-world constraints such as resource limitation and data transfer cost.

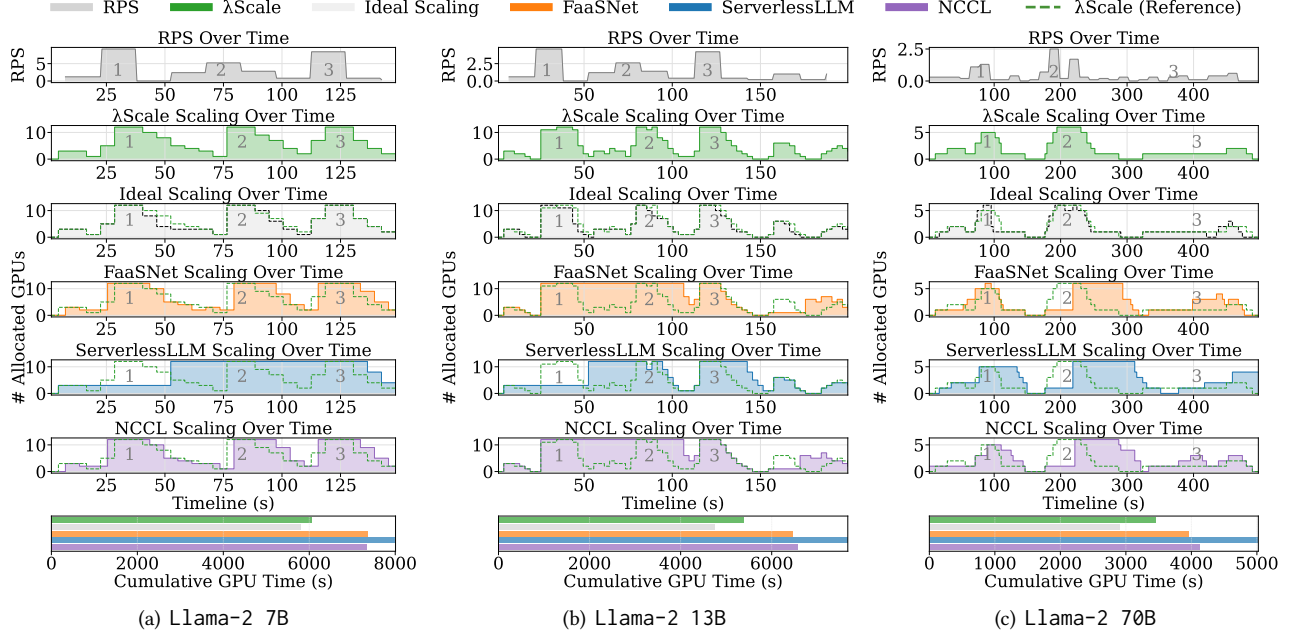


Figure 14: GPU allocation over time under a 30-minute BurstGPT workload. Top: RPS over time. Middle (Row 2-5): System-specific results sharing the same timeline. Bottom: Cumulative GPU consumption of all systems. The RPS timeline include labeled spikes (Top). The green dotted line shadows λ Scale’s scaling behavior, allowing direct comparison against other systems.

λ Scale scales out and in significantly faster than other systems across three model sizes. All three baselines experience delayed scaling out and delayed scaling in when responding to workload spikes. To quantify the cost effectiveness of GPU resources, we measure the cumulative GPU time for each system. λ Scale consumes up to 17.8%, 18.1%, and 31.3% less GPU resource than *FaaSNet*, *NCCL*, and *ServerlessLLM*, respectively. While *Ideal Scaling* achieves the lowest cumulative GPU time, λ Scale maintains the closest GPU consumption to this ideal case, with a small gap ranging from 4.3% and 18.6%, demonstrating λ Scale’s superior auto-scaling capability.

CDF curves of *NCCL* and *FaaSNet* shift rightward, due to frequent model loading from SSD, leading to longer tail latency. Second, the CDF curve of *ServerlessLLM* shifts leftward, likely due to a high cache hit rate when loading models into GPUs.

7.6 Sensitivity Analysis and Ablation Study

We conduct further experiments to evaluate the impact of the number of transfer blocks, k -way transmission, and system optimizations on λ Scale’s performance. We fix $k = 1$ to isolate the effects of the number of blocks and system optimizations on transfer latency. For k -way transmission analysis, we vary k to quantify its impact on LLM request serving.

Impact of k -way transmission on throughput. Fig. 16 compares λ Scale’s throughput performance under different k -way transmission levels. λ Scale-*Net* ($k=4$) achieves the fastest scaling, demonstrating that a higher K -way transmission levels significantly enhance performance. In contrast, λ Scale-*Half-Reorder* ($k=2$), scales more slowly, showing a moderate performance drop. Finally, λ Scale-*Non-Reorder* ($k=1$) performs the worst, characterized by the slowest scaling and the lowest overall throughput.

Effect of system optimizations on transfer latency. Fig. 17 shows the impact of various optimizations on transfer latency in λ Scale. The “None” configuration, without any optimizations, shows the highest transfer latency, exceeding

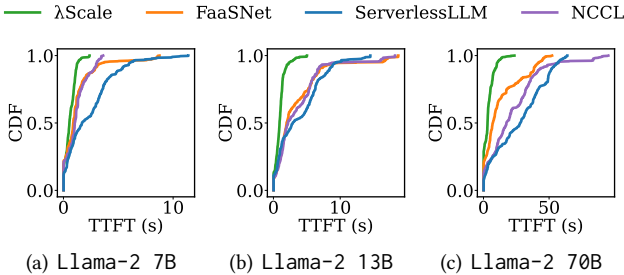


Figure 15: CDF of TTFT under the BurstGPT workload.

TTFT comparison. Fig. 15 shows the TTFT distribution, λ Scale outperforms all baselines across the three models. Compared to previous TTFT results reported under static workloads (§7.4), two key observations emerge. First, the

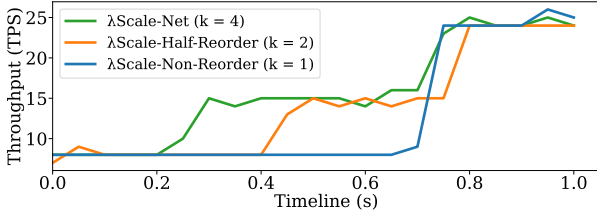


Figure 16: Impacts of k -way transmission on throughput performance.

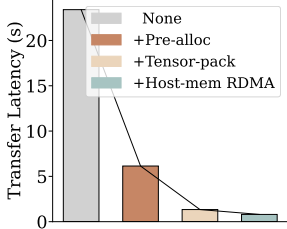


Figure 17: Performance breakdown of transfer latency.

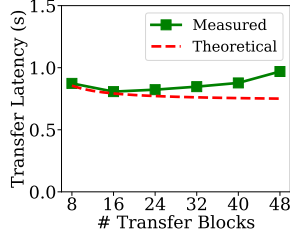


Figure 18: Latency with varying number of transfer blocks

20 ms. Adding pre-allocation (“+Pre-alloc”) significantly reduces the latency, demonstrating its importance in minimizing allocation overhead. Further improvement is observed with tensor packing (“+Tensor-pack”), which reduces latency by optimizing the data layout for efficient transfers. The lowest latency is achieved with the addition of (“+Host-mem RDMA”). This progression underscores the cumulative benefits of these optimizations, with each step contributing to a substantial reduction in transfer latency.

Impact of the number of model blocks on transfer latency. Fig. 18 shows that 16 blocks achieve the lowest transfer latency, likely due to a balance between RDMA request processing overhead and efficient data transfer. Fewer blocks (e.g., 8) result in higher latency because larger block sizes increase RDMA request processing overhead. More blocks (e.g., 24–48) also increase latency, likely due to the additional overhead of managing and transferring a larger number of smaller blocks. Latency decreases up to 16 blocks but rises beyond this point, highlighting diminishing returns and added complexity.

8 RELATED WORK AND DISCUSSION

Pipelined inference execution. Prior works on pipelined inference execution typically rely on static resource configurations [22, 25, 26, 39, 44], whereas λ Scale focuses on dynamic pipeline construction and execution using the “execute-while-load” approach. BlitzScale [52] is a concurrent work to λ Scale, utilizing chain-based model scaling and specifically designed for Prefill/Decoding (P/D) disaggregated LLM inference settings [54]. In contrast, λ Scale proposes an efficient,

binomial-pipeline-based scaling solution without assuming P/D disaggregation.

Support for tensor parallelism. In addition to Pipeline Parallelism (PP), λ Scale can inherently support Tensor Parallelism (TP) or Hybrid Parallelism [40, 42] by extending its model partitioning strategies. By tracking block-level execution dependencies as a DAG, λ Scale can effectively coordinate inference tasks under TP-based partitioning strategies, which we leave for future work.

Support for other models. Current λ Scale supports a variety of mainstream models, including LLMs, image classification models, and vision transformers [7, 8, 14]. Enabling fast scaling for smaller models is much easier; therefore, they are excluded from our evaluation. Additionally, λ Scale can be easily extended to support new models, provided that their structures are available for model partitioning.

9 CONCLUSION

This paper presents λ Scale, a serverless inference system that achieves fast model scaling. λ Scale leverages high-speed RDMA networks for fast model multicast across GPU nodes and enables cross-node, collaborative inference execution during model transfer. λ Scale realizes this “execute-while-load” approach through λ Pipe, a novel model scaling scheme that supports low-latency model loading and dynamic, pipelined inference execution. Combined with efficient model management, λ Scale delivers superior scaling performance, effectively sustaining load spikes and achieving up to 5 \times tail-latency speedup over state-of-the-art systems.

REFERENCES

- [1] Alibaba Cloud Function Compute. <https://www.alibabacloud.com/product/function-compute>.
- [2] Alibaba cloud: Serverless gpu overview. <https://www.alibabacloud.com/tech-news/a/serverless/402cc4hux4q-serverless-gpu-overview>.
- [3] Aliyun Function Compute Billing Scheme. <https://www.alibabacloud.com/help/en/function-compute/latest/billing-billing>.
- [4] AWS Lambda. <https://aws.amazon.com/lambda/>.
- [5] AWS Lambda Provisioned Concurrency. <https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html>.
- [6] Azure Functions. <https://azure.microsoft.com/en-us/services/functions/>.
- [7] clip-vit-large-patch14. <https://huggingface.co/openai/clip-vit-large-patch14>.
- [8] dpt-beit-large-512. <https://huggingface.co/Intel/dpt-beit-large-512>.
- [9] GitHub issue: slow ncclCommInitRank. <https://github.com/NVIDIA/nccl/issues/534>.
- [10] How microsoft’s bet on azure unlocked an ai revolution. <https://news.microsoft.com/source/features/ai/how-microsofts-bet-on-azure-unlocked-an-ai-revolution/>.
- [11] HuggingFace Serverless Inference. <https://huggingface.co/docs/api-inference/en/index>.
- [12] Libibverbs. <https://github.com/linux-rdma/rdma-core/blob/master/Documentation/libibverbs.md>.
- [13] Libibverbs. <https://pybind11.readthedocs.io/en/stable/>.
- [14] Meta Inference Engine. <https://github.com/meta-llama/llama-models>.
- [15] NCCL. <https://developer.nvidia.com/nccl>.
- [16] Nvidia gpubdirect rdma documentation. <https://docs.nvidia.com/cuda/gpubdirect-rdma/index.html>.
- [17] Together AI Serverless Inference. <https://www.together.ai/products#inference>.
- [18] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.
- [19] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.
- [20] Ahsan Ali, Riccardo Pincirol, Feng Yan, and Evgenia Smirni. BATCH: Machine learning inference serving on serverless platforms with adaptive batching. In *Proc. ACM/IEEE Supercomputing*, 2020.
- [21] Ahsan Ali, Riccardo Pincirol, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10), 2022.
- [22] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proc. USENIX OSDI*, 2020.
- [23] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. Rdmc: A reliable rdma multicast for large objects. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 71–82, 2018.
- [24] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing. In *Proc. USENIX ATC*, 2022.
- [25] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proc. USENIX NSDI*, 2017.
- [26] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *Proc. ACM SoCC*, 2020.
- [27] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Locality-enhanced serverless inference for large language models. In *Proc. USENIX OSDI*, 2024.
- [28] P. Ganesan and M. Seshadri. On cooperative content distribution and the price of barter. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS’05)*, pages 81–90, 2005.
- [29] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proc. USENIX OSDI*, 2020.
- [30] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proc. USENIX OSDI*, 2022.
- [31] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. Optimus: Warming serverless ml inference via inter-function model transformation. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1039–1053, 2024.
- [32] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, Santa Clara, CA, April 2024. USENIX Association.
- [33] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proc. ACM EuroSys*, 2023.
- [34] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proc. ACM SOSP*, 2019.
- [35] Joyjit Kundu, Wenzhe Guo, Ali BanaGozar, Udari De Alwis, Sourav Sengupta, Puneet Gupta, and Arindam Mallik. Performance modeling and workload analysis of distributed large language model training and inference, 2024.
- [36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP ’23*, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.
- [37] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- [38] Yunseong Lee, Alberto Scolar, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *Proc. USENIX OSDI*, 2018.
- [39] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.
- [40] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.
- [41] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: a distributed framework for

- emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.
- [42] Deepak Narayanan et al. Efficient large-scale language model training on gpu clusters. In *Proceedings of the International Conference for High Performance Computing*, 2021.
 - [43] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *Proc. USENIX ATC*, 2021.
 - [44] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proc. ACM SOSP*, 2019.
 - [45] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.
 - [46] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proc. USENIX ATC*, 2021.
 - [47] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2024.
 - [48] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *Proc. ACM ASPLOS*, 2022.
 - [49] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proc. IEEE ICDCS*, 2021.
 - [50] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. Faaswap: SLO-aware, gpu-efficient serverless inference via model swapping. *arXiv preprint arXiv:2306.03622*, 2024.
 - [51] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MARK: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.
 - [52] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Fast and live model auto scaling with $o(1)$ host caching, 2024.
 - [53] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *Proc. USENIX NSDI*, 2023.
 - [54] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.