Llumnix: Dynamic Scheduling for Large Language Model Serving

Biao Sun*†, Ziming Huang*†, Hanyu Zhao*, Wencong Xiao, Xinyi Zhang†, Yong Li, Wei Lin

Alibaba Group

Abstract

Inference serving for large language models (LLMs) is the key to unleashing their potential in people's daily lives. However, efficient LLM serving remains challenging today because the requests are inherently heterogeneous and unpredictable in terms of resource and latency requirements, as a result of the diverse applications and the dynamic execution nature of LLMs. Existing systems are fundamentally limited in handling these characteristics and cause problems such as severe queuing delays, poor tail latencies, and SLO violations.

We introduce Llumnix, an LLM serving system that reacts to such heterogeneous and unpredictable requests by *runtime rescheduling* across multiple model instances. Similar to context switching across CPU cores in modern operating systems, Llumnix reschedules requests to improve load balancing and isolation, mitigate resource fragmentation, and differentiate request priorities and SLOs. Llumnix implements the rescheduling with an efficient and scalable live migration mechanism for requests and their in-memory states, and exploits it in a dynamic scheduling policy that unifies the multiple rescheduling scenarios elegantly. Our evaluations show that Llumnix improves tail latencies by an order of magnitude, accelerates high-priority requests by up to 1.5×, and delivers up to 36% cost savings while achieving similar tail latencies, compared against state-of-the-art LLM serving systems. Llumnix is publicly available at https://github.com/AlibabaPAI/llumnix.

1 Introduction

Large language models (LLMs) such as the GPT series [15, 49] are bringing generative AI to an unprecedented level. Their human-level generation capabilities are being quickly adopted in a wide range of domains, inspiring many imaginations for future applications, and expected to have profound influences on how people live and work.

Inference serving of LLMs plays a key role in LLM-powered services, becoming a critical workload in datacenters. Such services are typically backed by multiple instances of the LLM deployed on a GPU cluster. The system involves a *scheduler* and an *inference engine*, where a request is first dispatched by the scheduler to a model serving instance, then gets executed by the inference engine inside. The requests are typically batched for execution on each instance to increase throughput and cost efficiency.

We observe unique characteristics of LLMs that call for new design philosophy of the serving infrastructure. The first is *workload heterogeneity*. LLMs are designed to be universal, by learning as much knowledge as possible from whatever domains. People can query the same LLM in totally different situations or even build custom applications atop LLMs for various scenarios; for all of these, a context-specific input (*i.e.*, prompt) is all you need [15]. Such universality and application diversity lead to heterogeneity of the inference requests, in terms of input lengths, output lengths, expected latencies, *etc*. For instance, the task of summarizing long text can introduce significant input lengths, where the latency of returning the first token (word) is often important to user experience [38].

The second characteristic is *execution unpredictability*. Serving an LLM request needs to run the model for multiple iterations, each producing a single output token; however, it is not known a priori how many tokens will be generated eventually. Moreover, the iterative generation also brings considerable GPU memory consumption that dynamically grows with the tokens. As such, the execution time and the resource demand of a request are both unpredictable.

These characteristics make an LLM inherently a *multi-tenant* and *dynamic* environment, serving heterogeneous and unpredictable workloads on multiple instances. This behavior is fundamentally different from traditional DNN models, where the requests are homogeneous and the execution is one-shot, stateless, and deterministic. Instead, we find LLMs more similar to modern operating systems hosting processes with dynamic working sets and different priorities on multiple cores. Managing such systems has complex goals, which goes

*Equal contribution.
†Work done during internship at Alibaba Group.

beyond what existing inference serving systems are designed for. Although there has been a series of LLM-tailored inference engines that shows superior performance, such systems concentrate on the sole goal of maximizing throughput within a single instance [34, 46, 67]. The request scheduling across instances, on the other hand, has received relatively little attention; the common practice today is still to use generic scheduling systems or policies inherited from the era of traditional DNNs [4, 28, 35, 47, 53]. Such a clear gap introduces challenges in the following aspects that are crucial in multi-tenant environments and online services.

Isolation. The system can hardly provide performance isolation to requests as their memory consumption grows unpredictably. Memory contention incurs performance interference and even preemptions of certain requests in a batch [34], leading to highly unstable latencies and service-level objective (SLO) violations, significantly sacrificing user experiences.

Fragmentation. The varying request lengths and memory demands inevitably result in memory fragmentation across instances, which introduces conflicting scheduling objectives. The running requests prefer load balancing to reduce preemptions and interference, but such load balancing fragments the free memory space across instances at the same time. The fragmentation can cause long queuing delays of new requests that instead require a large space on one instance for the input sequences. This conflict is difficult for the scheduler to reconcile with unpredictable arrivals and lengths of requests.

Priorities. Requests from different applications and scenarios naturally come with different latency objectives. Online chatbots [6, 8] are interactive applications and are therefore with tight SLO constraints. On the contrary, offline applications, such as evaluation [51], scoring [36], or data wrangling [43], are less sensitive to latency. Such different latency objectives are also a consequence of the commercial purpose of earning more profits from LLMs via diversified service classes (*e.g.*, ChatGPT Plus [2]). However, existing LLM inference systems [34, 67] often treat all requests for a model equally and cannot differentiate their priorities, which has limitations in meeting different latency objectives of requests.

We introduce Llumnix, a new scheduling system for LLM serving that addresses the challenges above via *runtime rescheduling* of requests across model instances. Analogous to context switching across CPU cores in OS process management, rescheduling enables Llumnix to *react* to the unpredictable workload dynamics at runtime, instead of having to address all the complex scheduling concerns and tradeoffs with the one-shot dispatching of requests. Llumnix reschedules requests for multiple purposes (Figure 1): load balancing for reducing preemptions and interference, de-fragmentation for mitigating queuing delays, prioritization of urgent requests by creating even higher degree of isolation, saturating or draining out instances during auto-scaling more quickly.

Llumnix reschedules requests via an efficient and scalable *live migration* mechanism of requests along with their GPU memory states across instances. Straightforward rescheduling approaches could introduce substantial downtimes to rescheduled requests, especially for long sequences. By contrast, Llumnix introduces *near-zero* downtime that is *constant* to sequence lengths, by carefully coordinating the computation and the memory transfer to hide the cost.

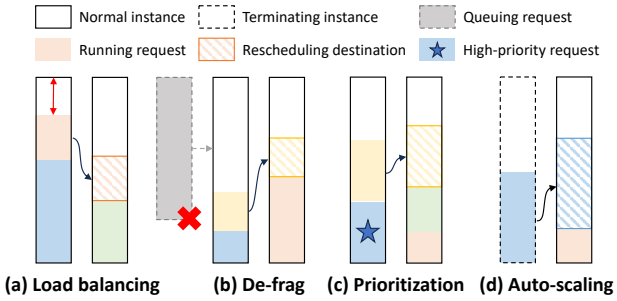

Figure 1: Example rescheduling scenarios in Llumnix.

To exploit such great scheduling flexibility of migration, Llumnix adopts a distributed scheduling architecture that enables continuous rescheduling with high scalability. Llumnix further introduces a dynamic scheduling policy under this architecture that unifies all the rescheduling scenarios with different goals elegantly. This unification is achieved via a concept called *virtual usage*: Llumnix just needs to define a set of rules for setting the virtual usages of GPU memory for requests in different scenarios, and then use a simple load-balancing policy based on the virtual usages.

We have implemented Llumnix as a scheduling layer on top of inference engines. Llumnix currently supports a representative system, vLLM [34], as the underlying engine. Evaluation on a 16-GPU cluster using realistic workloads shows that Llumnix improves P99 first-token latency by up to 15× and P99 per-token generation latency by up to 2×, compared against a state-of-the-art scheduler INFaaS [53]. Llumnix also accelerates high-priority requests by 1.5×, and achieves 36% cost saving when delivering similar tail latencies.

In summary, this paper makes the following contributions.

- We reveal the unique characteristics and scheduling challenges of LLM serving that necessitate new scheduling goals such as isolation, de-fragmentation, and priorities.
- We propose request rescheduling as a key measure to achieve these goals and realize it with an efficient migration mechanism of requests and their GPU memory states.
- We design a distributed scheduling architecture and an accompanying scheduling policy that exploit request migration to achieve the multiple goals in a unified manner.
- We implement and evaluate Llumnix to show its advantages over state-of-the-art inference serving systems.

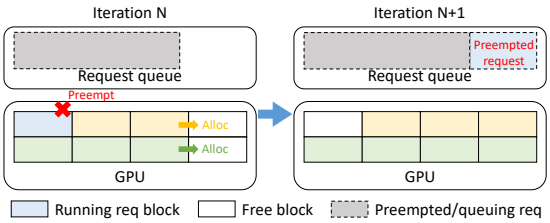

Figure 2: Request queuing and preemption using continuous batching and dynamic memory allocation.

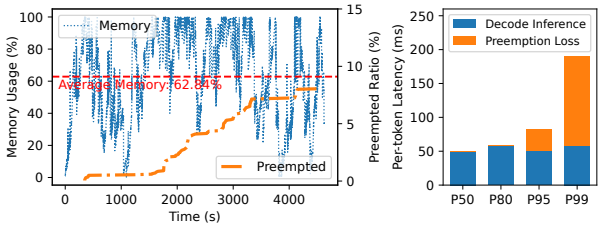

Figure 3: Request preemptions in LLaMA-7B serving.

2 Background

Application diversity of LLMs. Recent LLMs are becoming *task-agnostic*. That is, the same model can work for various tasks with context-specific inputs (*a.k.a.* the "prompts") provided. This is achieved by both increasingly larger model and dataset sizes and advanced pre-training approaches such as few-shot learning [14]. Task-agnostic models enable diverse applications, from chatbots, search engines, summarization, coding, AI assistants, to AI agents, to name a few.

The diverse applications lead to requests with different requirements for the serving. An important aspect is the *sequence lengths*. LLMs are racing to support longer sequence lengths — for example, from March to November 2023, the maximum sequence lengths of the GPT family have scaled from 32k [1] (GPT-4 [49]) to 128k (GPT-4 Turbo [50]). We expect this trend to continue as longer sequences are necessary for broader applications of LLMs. Consider an intuitive example of the tasks for summarizing and writing an article: they require sufficiently long input and output lengths, respectively. Another aspect is *expected latencies*. A real product example is that OpenAI introduces a subscription plan called ChatGPT Plus [2] to offer faster responses of common ChatGPT services. In general, different applications and situations also naturally have different levels of urgency. For example, more interactive applications like personal assistants expect shorter latencies than tasks like summarizing an article.

Autoregressive generation. The inference for state-of-the-art LLMs is *autoregressive*: the model iteratively accepts the input sequence plus all the previous output tokens to generate the next output token, until an "end-of-sequence" (EOS) token is generated. The phase for generating the first token and that for each new token afterwards are usually referred to as *prefill* and *decode*, respectively. LLM services typically return the generated tokens in a streaming manner. Therefore, the prefill and decode latencies are both user-perceivable and important to user experiences. The prefill latency determines how long it takes to start receiving the response, which can be dominated by the queuing delay. The decode latency determines the speed of receiving the following tokens subsequently.

During the autoregression, the intermediate results (key and value tensors used in the attention operation [59]) for each token are involved in the generation of all following tokens. Therefore, the inference engine typically stores these states in GPU memory for reuse, known as the *KV cache* [52].

Batching and memory management. State-of-the-art inference engines apply the *continuous batching* technique [34,67] to handle the varying sequence lengths and dynamic arrivals of requests. That is, a new/completed request can join/leave the running batch immediately, instead of waiting for all the running requests to complete. Batching also raises concern about memory management of KV cache. Since the memory demand of KV cache is not known a priori, it would clearly limit the batch size and batching benefits if the memory is reserved to the maximum length. For example, a LLaMA-2-13B [58] model supports sequence lengths up to 4k, which translates to 3.2 GB KV cache for a single request; while the memory of current GPUs remain tens of GBs, let alone the space for model weights (26 GB for LLaMA-2-13B). Therefore, recent work (vLLM [34]) proposed *dynamic memory allocation* for KV cache to increase batch size and throughput, enabled by a technique named PagedAttention: the KV cache tensors are stored in dynamically allocated blocks as the KV cache grows. Figure 2 presents an example of using continuous batching with dynamic memory allocation. The running requests are chosen based on the free memory blocks, hence there is a queuing request (the gray one) at iteration N as the memory is insufficient. At the next iteration, the system runs out of memory for the new blocks of the running requests. Therefore, the system preempts certain running requests (the blue one), which then goes back to the queue.

3 Motivation

We motivate the design of Llumnix with a series of key characteristics of LLM serving as follows.

Unpredictable memory demands and preemptions. With dynamic memory allocation, request preemptions are inevitable as a result of the unpredictable memory demands, which can significantly increase the latencies of the preempted requests. Figure 3 shows an experiment of LLaMA-7B model serving using vLLM on an A10 GPU running a trace of 2,000 requests generated from a Poisson distribution. The input and output lengths follow a power-law distribution with a mean

[1] 1k stands for 1,024 when describing sequence length in this paper.

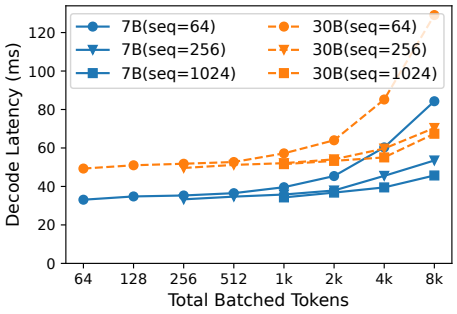

Figure 4: Latencies of one decode step of LLaMA-7B and LLaMA-30B with different sequence lengths and batch sizes.

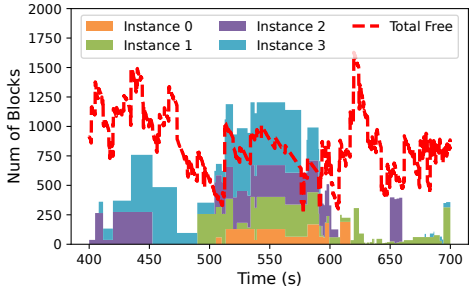

Figure 5: Total free memory vs. demands of the head-of-line queuing requests across four LLaMA-7B instances.

value of 256 tokens (details in §6). We control the request rate (0.42 req/s) to get a moderate memory load (62% on average) with some spikes due to the varying sequence lengths. Under such load, we still observe 8% of the requests being preempted. We quantify the preemption loss by measuring the latency penalty caused by preemption, including the extra queuing time and the recomputing for previous KV cache. We show different percentiles of per-token decode latency (averaged across all decode iterations of a request). We do not use the end-to-end latency because it depends on the number of iterations. We observe that the P99 per-token decode latency is much worse than the P50 (3.8×), and the preemption loss accounts for 70% for the P99 request. In particular, the P99 request experiences a total preemption loss of 50 seconds (preempted twice), showing severe service stalls and degradation of user experiences due to preemptions.

Performance interference among requests. We also observe performance interference of requests in a batch to each other, due to resource competition on GPU compute and memory bandwidth resources. Figure 4 shows the times for a decode step of LLaMA-7B (1-GPU) and LLaMA-30B (4-GPU) using different sequence lengths and batch sizes (the X-axis shows the total number of tokens in a batch for each data point). The decode speed decreases with more requests and higher interference, and the gap between the same sequence length is up to 2.6×.

Memory fragmentation. Considering the aforementioned problems, it would be better to spread requests across instances to reduce preemptions and interference. However, such spreading will make the available memory of the cluster fragmented across instances simultaneously. Here fragmentation refers to *external* fragmentation, *i.e.*, unallocated memory on an instance. Dynamic allocation techniques like PagedAttention [34] can eliminate external fragmentation during the *decode* phase, where the blocks are allocated one at a time. However, external fragmentation remains a significant problem for the *prefill* phase, which requires many blocks on an instance in one allocation to accommodate the KV cache of all tokens in the inputs. Therefore, external fragmentation can cause long queuing delays of new requests, especially those with long inputs.

Figure 5 shows an experiment of four LLaMA-7B instances, where the trace also uses the input/output length distribution with mean value 256 and a Poisson distribution with a request rate of 1.9 req/s. We implement a spreading dispatching policy that dispatches new requests to the instance with the lowest memory load for load balancing. We demonstrate the fragmentation by showing the total free memory blocks across the cluster, against the demand of the head-of-line queuing request on each instance. For most of the time span, the total free memory can accommodate the queuing requests on at least three instances (sometimes all of them). The request are queuing despite enough total memory because they exceed the free space on their own instances, which demonstrates the fragmentation and also the potential of defragmentation to reduce queuing delays.

Different emergency and priorities of requests. With requirements of products like ChatGPT Plus and the diverse application scenarios of LLMs, we foresee more applications with different latency sensitivities. However, existing systems usually treat all requests equally, where the latency-sensitive could easily be interfered by other normal ones, *e.g.*, excessive queuing delays or performance interference. This calls for a systematic approach to differentiating the request priorities for an LLM to meet their respective latency objectives.

Opportunity: request rescheduling across instances. This paper explores a new dimension that is missing in current LLM serving systems: the multiple model instances of a deployment and their interaction. A simple intuition is that when the aforementioned problems occur on a certain instance, it is possible that the whole cluster still has enough space for avoiding preempting requests, accommodating new requests, or mitigating interference. This is also a natural consequence of the varying request lengths and memory loads across instances. However, existing systems cannot exploit such free space on other instances because requests are tied on the same instance once scheduled throughout the autoregressive execution. Llumnix unifies the request scheduling component and the model inference engine to explore the potentials of fine-grained coordination among inference instances.

4 Llumnix Design

4.1 Overview

Llumnix builds upon the key idea of rescheduling LLM inference requests at runtime across model instances. Llumnix inherits continuous batching [67] and dynamic memory allocation [34] from state-of-the-art systems for high throughput. Beyond that, Llumnix exploits request rescheduling to react to the unpredictable workload dynamics in various situations with different scheduling goals, as illustrated in Figure 1.

A first goal is *load balancing* (Figure 1-a) to reduce request preemptions and interference on high-load instances. Although the dispatching can also consider load balancing of memory usage, it could be sub-optimal as the final memory usages of requests are unknown at the arrivals, due to the unpredictability of output lengths. Rescheduling complements it by reacting to the real usage growths of requests. Meanwhile, as shown before, load balancing can also lead to higher memory fragmentation and longer queuing delays of long inputs probably. Therefore, Llumnix also reschedules requests for *de-fragmentation* (1-b), *i.e.*, creating contiguous space on an instance by moving requests onto others. Although these two goals remain a tradeoff, Llumnix has a much larger space to balance them with rescheduling. Another goal is *prioritization* (1-c) of certain requests by rescheduling co-located requests away for lower load and avoiding interference. Such rescheduling provides "decicated" resources to high-priority requests dynamically, without the need for reserving machines statically. Finally, Llumnix also reschedules requests during *auto-scaling*, *e.g.*, to drain out an instance to be terminated (1-d) or saturate a new instance more quickly.

Realizing such highly dynamic rescheduling efficiently is challenging, considering the large request context states (*i.e.*, the KV cache). Naïve solutions include recomputing or copying the KV cache of the rescheduled requests, however with high computation stalls and downtime, reaching over 50× of the decoding cost (§6.2). What's more, the KV cache states increase with sequence lengths, limiting the scheduling flexibility under the trend of growing context lengths [50]. Such a high inference delay in generating next tokens greatly degrades the user experiences of LLM serving and thus prohibits request rescheduling. Llumnix addresses this challenge with a *live migration* mechanism that pipelines and coordinates the KV cache copying and the token generation computation, thereby bringing negligible downtime (§4.2).

To exploit the benefits of migration, Llumnix adopts a scalable architecture that combines global and local scheduling to decentralize the scheduling decisions and the coordinated migration actions, facilitating continuous rescheduling at scale (§4.3). Under this architecture, we further design an efficient heuristic scheduling policy that centers around the *virtual usage* concept to abstract the requirements of the different scheduling goals in a unified manner (§4.4).

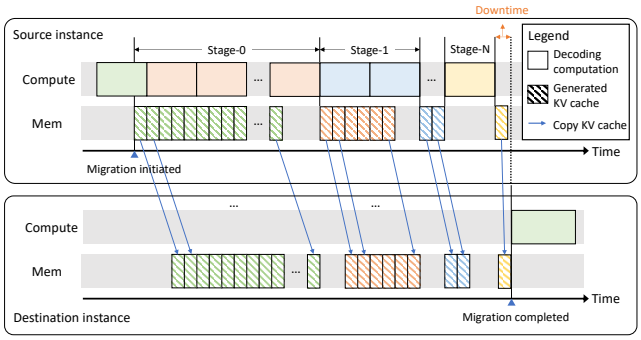

Figure 6: Llumnix adopts multi-stage migration to overlap the computation and KV cache copying for minimal downtime.

4.2 Live Migration of LLM Requests

The significant KV cache states of requests can potentially introduce great cost and serving stalls during rescheduling. Llumnix addresses this challenge by exploiting a key characteristic of LLM inference: the KV cache is *append-only*. LLM inference iteratively concatenates the output token of the current iteration with the input tokens, which is set as the input for the next iteration. In this way, inference engines also keep appending the calculated KV state of the current iteration to the KV cache parameters, leaving the parameters generated by previous iterations remain constant.

The live migration mechanism of Llumnix utilizes the inherent append-only characteristic of KV cache to pipeline the KV cache copying with the decoding computation. Because the KV cache already generated won't be modified in the following iterations, Llumnix can safely copy the KV cache of previous tokens in parallel with the computation for new tokens. In this way, Llumnix achieves *near-zero* and *constant* downtime to the rescheduled request. As shown in Figure 6, when migration is initiated, the source instance starts to copy the KV cache blocks of completed iterations, and continues the computation at the same time (stage 0). When the copying for the previous KV cache blocks is done, there will be a few more iterations (*i.e.*, blocks in Figure 6) computed in stage 0. Then, it switches to stage 1 to copy the KV cache generated by stage 0, while continuing the computation afterwards. The copying is generally much faster than the computation, thus the number of new blocks is typically small such that we can copy them in a very short period. To the end, only one iteration of computation is conducted for the KV cache migration (*i.e.*, stage-N). Therefore, Llumnix suspends the computation for the request by draining it out of the current batch and copies the remaining block, which introduces the downtime of this request. Once it is finished, the migration completes and the request resumes on the destination instance. Although the total copying duration of the whole sequence depends on the sequence length, the downtime for the request is only the period of copying the KV cache generated by one iteration,

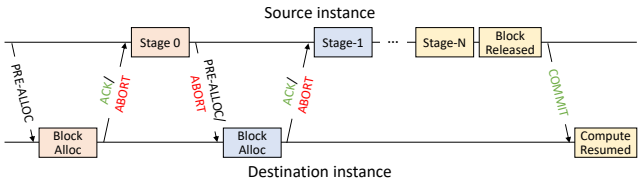

Figure 7: Handshake during migration.

which is negligible regardless of the sequence length.

The request migration approach of Llumnix borrows the key concept introduced in virtual machine (VM) live migration [17], which gradually reduces the working set to minimize the downtime. Llumnix does not require the dirty page tracing in VM migration as the working set (*i.e.*, KV cache) is append-only and does not change during migration. However, LLM serving further introduces additional challenges. Firstly, as both the source and destination instances are continually processing requests, the request might run out of memory during migration. Secondly, the request can complete in the middle of migration, due to the unpredictable execution (*i.e.*, generating EOS token) and the continuous batching [67]. To handle such exceptions and guarantee correctness during the asynchronous computation and memory copying, Llumnix introduces fine-grained coordination between the participating instances with a *handshake* process (Figure 7). Before each stage, the source instance issues a pre-allocate request with the number of blocks to migrate to make sure that the destination has enough space. The destination will try to allocate and reserve the blocks; if it succeeds or fails, the destination will notify the source to proceed or abort the migration and clean the states, respectively. Similarly, after each stage, the source instance also checks whether the request being migrated has completed or been preempted — if it has, the source will notify the destination to abort and release the reserved blocks; otherwise the source will go ahead to the next stage. The source or destination will also abort the migration if the other side fails. After the final stage finishes, the source releases its local blocks and notifies the destination to commit the migration and resume the execution of the request.

4.3 Distributed Scheduling Architecture

The live migration mechanism provides the foundation for runtime rescheduling of LLM inference requests. However, achieving fully dynamic scheduling is still non-trivial due to the higher scheduling pressure than in traditional schedulers. In particular, Llumnix would need to continuously track and reschedule every single running request throughout the cluster, rather than only dispatch incoming requests for one time or only manage running requests on one instance. This implies a higher scheduling frequency and a larger number of requests for the scheduler to track and schedule in each round.

Llumnix devises a scalable architecture that combines a cluster-level *global scheduler* and distributed instance-level

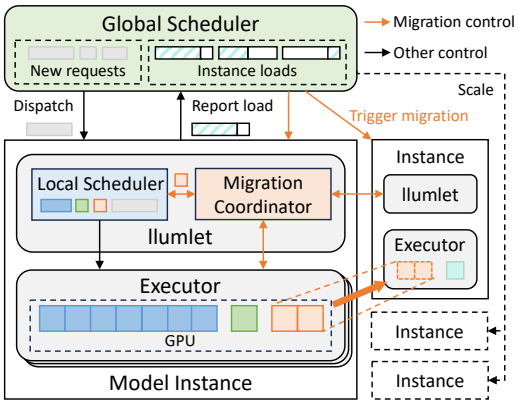

Figure 8: Llumnix architecture.

schedulers, named *llumlet*s, to enable continuous rescheduling efficiently (Figure 8). Llumnix defines a clean separation of concerns with a narrow interface between the two levels. The global scheduler does not directly track or schedule the running requests; instead, it makes all scheduling decisions oriented to the *instances*, according to the *memory loads* of them. This way, the complexity of the global scheduler remains independent from the running requests, thereby preserving similar scalability to schedulers without dynamic scheduling. The loads are reported by the llumlets periodically, based on the request status and Llumnix's scheduling policy.

The global scheduler utilizes the load information to dispatch new requests, trigger migration across instances, and control the instance auto-scaling. In particular, for migration, the decisions are not made for specific requests; the global scheduler just pairs the source and destination instances, only based on the loads, and marks them as in the corresponding states to trigger the migration. The llumlets will decide the requests to migrate and execute the migration automatically.

The llumlet of each instance consists of a local scheduler and a migration coordinator. In addition to the functionalities of similar roles in existing systems like queuing, batching, and block management, an important new task of the local scheduler is to calculate the memory load of the instance. The load is not simply the physical memory being used; instead, it is a sum of the "virtual usages" (§4.4) of the requests. The local scheduler is also responsible for deciding the requests to migrate when triggered. Given the chosen requests, the migration coordinator will coordinate with the local scheduler and the other instance, and instruct the model executor to do the memory copying, as described before.

4.4 Dynamic Scheduling Policy

4.4.1 Goals and Definitions

Llumnix's scheduling policy is designed with the following goals. The first is to improve **prefill and decode latencies**, by reducing queuing delays, preemptions, and interference.

The second goal is **load-adaptivity** to handle varying cluster load and improve cost efficiency. We notice that the benefits of rescheduling is also relevant to cluster load, which could be limited under too high/low load. Llumnix incorporates instance auto-scaling to keep appropriate cluster load for both saving costs and maximizing the benefits of rescheduling.

Besides these two goals similar to those of existing systems, Llumnix introduces a new goal of request **priorities** that comes from the new requirements of LLMs. Priorities present a systematic approach for the same LLM to serve certain requests with higher emergency, *e.g.*, from ChatGPT Plus or more interactive applications. Llumnix provides applications with an interface for specifying request priorities to meet different SLOs, in terms of *scheduling priority* and *execution priority*. Requests with higher scheduling priorities will get scheduled earlier to reduce their queuing delays. Those with higher execution priorities will be given lower instance load and hence less interference to accelerate their execution. Currently, Llumnix supports two priority classes, high and normal, to demonstrate the ability of Llumnix to prefer high-priority requests, but our design also generalizes to more priorities.

4.4.2 Virtual Usage

To achieve the multiple goals above under the distributed scheduling architecture, Llumnix needs a scheduling policy that can express these goals using simple instance-level metrics, to improve the efficiency and scalability of the global scheduler. To this end, Llumnix introduces the *virtual usage* abstraction to unify these different, sometimes conflicting goals into a simple load metric of instances. The key observation here is that the aforementioned rescheduling scenarios fall into two categories: *load balancing*, and *creating free space on one instance* (de-fragmentation, prioritization, and draining out instances). We find that they can be unified into load balancing by assuming a virtual load on the instance: to create free space on an instance, we just need to set the virtual usages of certain requests to make the instance virtually overloaded, then a load balancing policy will be triggered to migrate the requests to other instances.

上述一系列的重调度场景可被分为两类：重调度和创建空闲空间，后者可以进一步通过设置虚拟负载来实现

This observation leads us to a simple heuristic with load-balancing as the basis, combined with a set of rules for setting request virtual usages in different situations. We summarize the rules in the function `CalcVirtualUsage` in Algorithm 1 and illustrate example scenarios in Figure 9. In normal cases, the virtual usage of a request is just its physical memory usage to enable routine load balancing, as shown in Figure 9(a). We discuss the rules for other cases as follows.

Queuing requests. For the head-of-line queuing request on an instance, we assign a positive virtual usage to it to reflect its resource demand in terms of the required memory, although the physical usage is 0. Thus, queuing requests will increase the total virtual usage of the instance, then the policy will trigger migration for load balancing (which in effect is de-fragmentation for the queuing request), as shown in Figure 9(b). There could be a lot of heuristics to explore for setting the virtual usage, which controls the tradeoff between reducing queuing delays and load balancing — for example, gradually increasing the virtual usage of a queuing request until it reaches the real memory demand. Llumnix currently uses a simple rule that directly uses its real demand (line 4 in Algorithm 1), which favours reducing queuing delays. This rule is based on our observation that queuing delay can dominate the end-to-end latency and worth such preference. Our evaluation also shows that this rule preserves the benefits of load balancing, due to the high flexibility of migration.

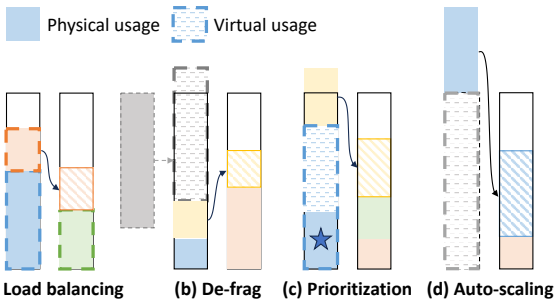

Figure 9: Llumnix combines virtual usages with a load-balancing policy to unify multiple scheduling goals.

Execution priorities. For a request with high execution priorities, Llumnix tries to prevent the instance the request is running on from exceeding a given level of real load, by reserving a memory space as headroom, as shown in Figure 9(c). This is achieved by adding such a headroom on the physical usage of a high-priority request to get the virtual usage (line 8). When there are multiple high-priority requests on an instance, this headroom is divided among them (line 10). The headroom for high-priority requests is currently defined as that required to preserve the ideal decode speed (*i.e.*, no visible interference), which is obtained through profiling. The headroom for normal requests is 0. Llumnix can also support more execution priorities by specifying the sizes for the headroom. When the headroom for a high-priority request is running up, the other normal requests will be migrated away by the load balancing policy because the instance is overloaded in terms of the total virtual usage.

Auto-scaling. When a new instance is launched, Llumnix's load balancing policy will automatically saturate it by migrating requests from other instances to it. When an instance is terminating, we artificially add a fake request with a virtual usage of infinity on it (line 7), then the remaining requests will be migrated to other instances, as shown in Figure 9(d).

4.4.3 Policies

We then describe how the specific scheduling decisions are made based on the virtual usages.

Algorithm 1: Virtual Usage and Freeness Calculation

```
Function CalcVirtualUsage(req, instance):
    if req.isQueuing then
        if req.isHeadOfLine then
            return req.demand
        return 0
    if req.isFake then
        return ∞
    return req.physicalUsage+GetHeadroom(req.priority, instance)
Function GetHeadroom(p, instance):
    return headroomForPriority[p]/instance.numRequests[p]
Function CalcFreeness(instance):
    if instance.isTerminating then
        AddFakeReq(instance.requests)
    totalVirtualUsages = 0
    for req in instance.requests do
        totalVirtualUsages+ =CalcVirtualUsage(req, instance)
    freeness = (instance.M − totalVirtualUsage)/instance.B
    return freeness
```

Dispatching. Llumnix dispatches new requests with higher scheduling priorities first. Within the same priority, it adopts a simple first-come-first-serve order. On each instance, requests are scheduled in the same order. Llumnix uses a load-balancing policy that dispatches each request to the freest instance. We introduce a metric for measuring the *freeness* of an instance defined as $F = (M - \sum V)/B$, where M is the total memory, V is the virtual usage of each request, and B is the batch size. While $(M - \sum V)$ already measures the free space, we divide it by the batch size because it determines the consumption speed, *i.e.*, the number of new tokens per iteration. Thus the metric suggests how many iterations the batch can still run for. Then Llumnix dispatches each incoming request to the instance with the highest freeness. Because the virtual usage of a request can be larger than the physical, it is possible that F is a negative value, *e.g.*, when there are queuing requests or high-priority requests. Such negative freeness values help Llumnix automatically treat such instances as overloaded and prefer dispatching requests to other instances. The freeness metric also guides the migration and auto-scaling, as shown later.

Migration. Llumnix triggers the migration policy periodically. In each round, Llumnix selects the candidate sets of source and destination instances by choosing those with freeness values smaller or greater than given thresholds, respectively. Llumnix pairs the instances from both sets by picking the two with the lowest and the highest freeness values repeatedly, and then sets them in corresponding states. The llumlet of each source instance then starts to migrate requests to the destination continuously, until it is no longer set in the source state. The llumlet prefers the requests with lower priorities and shorter sequence lengths when choosing the requests to migrate. In the next round, if an instance during migration is no longer beyond the thresholds, Llumnix will unset the migration state and the migration will stop.

Auto-scaling. Llumnix scales the instances according to the cluster load in terms of the averages freeness for the normal priority across instances. The policy maintains the average freeness within a range $[x, y]$, and adds or terminates an instance when the freeness is smaller than x or greater than y for a period, respectively. Llumnix chooses the instance with fewest running requests for termination.

5 Implementation

We implement Llumnix with 3,300 lines of Python code. Llumnix is a standalone library comprising both its own components and an interface to integrate and communicate with backend inference engines. This architecture makes Llumnix non-intrusive and extensible to different backends. Llumnix currently supports vLLM [11] as the backend, which is an open-source state-of-the-art inference engine that features continuous batching, PagedAttention, and tensor-parallel distributed inference [34, 56].

Multi-instance serving. Llumnix instantiates the multiple instances of the backend and the other components as Ray [42] actors. Ray's Python-native distributed runtime enables fine-grained coordination among these actors in a simple and efficient manner. Llumnix also launches a set of request frontend actors that exposes an OpenAI-style API endpoint [48]. Although a request can be migrated across backend instances, the generated tokens are forwarded to the frontend and then returned to end users, ensuring a steady API service.

KV cache transfer. We use the Gloo collective communication library [5] (the `Send`/`Recv` primitives) for the KV cache transfer during migration. A potential alternative is NCCL [1], which is generally faster than Gloo on GPUs but has been adopted in communication for distributed inference. However, Llumnix needs to migrate requests in parallel with the inference to minimize the downtimes, but concurrent invocations of NCCL are known to be unsafe [45]. The pipelined migration design allows us to use Gloo while maintaining negligible downtimes. Using Gloo needs to copy the KV cache between CPU and GPU memory, which is done in another CUDA stream to avoid blocking the inference computation. Note that in typical deployments, the communication-heavy tensor parallelism is limited in a single machine for high-speed transfer [44]. In such cases, migration between instances (machines) will not interfere with the tensor-parallel inference.

Block fusion. vLLM stores the KV cache in non-contiguous small blocks that are dynamically allocated. For example, the block size of a 16-bit LLaMA-7B model is 128 KB (for key or value tensors of 16 tokens in each layer), and a sequence of 1k tokens translates to 4k such blocks (32 layers). To avoid the overhead of sending these blocks using many small messages, we fuse the blocks by copying them from GPU memory to a contiguous CPU memory buffer and use Gloo to send the buffer as a whole, thereby improving the transfer efficiency.

Fault tolerance. Llumnix provides fault tolerance for each component to ensure high service availability. When the global scheduler fails, Llumnix temporarily falls back to a scheduler-bypassing mode, thus not affecting the service availability: that is, the request frontends directly dispatch requests to certain instances using simple rules, and migration is disabled. When an instance (or the co-located llumlet) fails, the requests running on it will be aborted. In particular, ongoing migration on failed instances will also be aborted (the request being migrated is not necessarily aborted, depending on if its source instance is healthy), which is handled by the handshake process. These failed actors will be automatically restarted by Ray, after which the service could go back to normal state.

6 Evaluation

We evaluate Llumnix on a 16-GPU cluster using realistic models and various workloads. Overall, our key findings include:

- Llumnix introduces near-zero downtime to requests being migrated and near-zero overhead to other running requests.
- Llumnix improves prefill latencies by up to 15×/7.7× (P99/mean) over INFaaS on 16 LLaMA-7B instances via de-fragmentation. Llumnix also improves P99 decode latency by up to 2× by reducing preemptions.
- Llumnix improves high-priority request latencies by up to 1.5× by reducing their queuing delays and accelerating their execution, while preserving similar performance of the normal requests.
- Llumnix achieves up to 36% cost saving while preserving similar P99 latencies with efficient auto-scaling.

6.1 Experimental Setup

Testbed. We use a 16-GPU cluster with 4 GPU VMs on Alibaba Cloud (type `ecs.gn7i-c32g1.32xlarge`), each with 4 NVIDIA A10 (24 GB) GPUs connected via PCI-e 4.0, 128 vCPUs, 752 GB memory, and 64 Gb/s network bandwidth.
Models. We conduct experiments using a popular model family, LLaMA [57]. We test two different specifications: LLaMA-7B, which runs on a single GPU, and LLaMA-30B, which runs on 4 GPUs of a machine using tensor parallelism. The models adopt the commonly used 16-bit precision. The version of vLLM that we based on only supports the original LLaMA with a maximum sequence length of 2k, but there have been a series of recent LLaMA variants supporting longer sequence lengths ranging from 4k to 256k [3, 7, 58, 65]. Since the model architectures and inference performance of these variants are mostly similar to those of LLaMA, we believe that our results are representative of more model types and larger sequence length ranges from a systems perspective.
Traces. Similar to prior work [34, 35, 67], we synthesize request traces to asses Llumnix's online serving performance.

Distribution			Mean	P50	P80	P95	P99
Real	ShareGPT	In	306	74	348	1484	3388
		Out	500	487	781	988	1234
	BurstGPT	In	830	582	1427	2345	3549
		Out	271	243	434	669	964
Gen	Short (S)		128	38	113	413	1464
	Medium (M)		256	32	173	1288	4208
	Long (L)		512	55	582	3113	5166

Table 1: Real and generated distributions of sequence lengths (numbers of tokens) used in our evaluation. The real distributions include those of both inputs ("In") and outputs ("Out").

We use Poisson and Gamma distributions with different request rates (requests per second) to generate request arrivals. For Gamma, we also use varying coefficients of variance (CVs) to adjust the burstiness of the requests. Each trace has 10,000 requests. We choose an appropriate range of request rates or CVs for the traces to maintain the loads within a reasonable range: nearly no queuing delays and preemptions for P50 requests, and queuing delays within a few tens of seconds for P99 requests when using Llumnix.

For the input/output lengths of requests, we use two public ChatGPT-4 conversation datasets, ShareGPT (GPT4) [10] and BurstGPT (GPT4-Conversation) [62], for an evaluation on real workloads. Considering that Llumnix targets more diversified applications, we also use generated power-law length distributions to emulate long-tail workloads that mix both frequent, short sequences (*e.g.*, for interactive applications like chatbots and personal assistants) and seldom, long sequences (*e.g.*, summarizing or writing articles). We generate multiple distributions with different long-tail degrees and mean lengths (128, 256, 512), as shown by the Short (S), Medium (M), and Long (L) distributions in Table 1. These distributions have a maximum length of 6k, thus the total sequence length of a request (input plus output) will not exceed the capacity of an A10 GPU when running LLaMA-7B (13,616 tokens). To observe the performance with different workload characteristics, we construct the traces by picking different combinations of the length distributions for inputs and outputs as follows: S-S, M-M, L-L, S-L, and L-S.
Baselines. We compare Llumnix with the following schedulers. All the baselines and Llumnix use vLLM as the underlying inference engine to focus the comparison on the request scheduling across instances.

- *Round-robin dispatching*: a simple dispatching policy to distribute requests across instances evenly, which is a typical behavior of production-grade serving systems [4, 9, 47].
- *INFaaS++*: an optimized version of INFaaS [53], a state-of-the-art scheduler for multi-instance serving. We evaluate its load-balancing dispatching and load-aware auto-scaling policies. We improve it by making it focus on the GPU

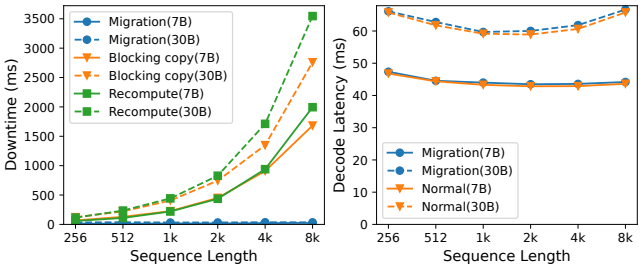

Figure 10: Downtime and overhead of migration.

memory load as it is the dominant resource in LLM serving. This load also counts in the memory required by queuing requests on each instance to reflect the queue pressure.

- *Llumnix-base*: a base version of Llumnix that is priority-agnostic (*i.e.*, treats all requests as the same priority) but enables all the other features including migration.

Key metrics. We focus on request latency, in terms of end-to-end, prefill (that of the first generated token), and decode (that since first generated token to the last, averaged over all generated tokens). We report both mean and P99 values.

6.2 Migration Efficiency

We first examine the performance of Llumnix's migration mechanism, in terms of the downtimes introduced to the migrated requests and the performance overheads for the running requests. We test both the 1-GPU LLaMA-7B and the 4-GPU LLaMA-30B models. For each model, we deploy two instances on two different machines. We use different sequence lengths, for each of which we run a batch of requests with the same total length of 8k on both instances. We migrate one of the requests from one instance to another and measure its downtime and the decode speeds of the running batches on both instances during migration.

We compare the downtime during migration with two simple approaches: recomputing, and blocking copying of the KV cache using Gloo (non-blocking for other requests). As shown in Figure 10 (left), the downtime of migration is nearly constant with increasing sequence lengths (roughly 20-30 ms), even shorter than a single decode step. In comparison, the downtimes of baselines increase with the sequence lengths, reaching up to 111× that of migration. For example, recomputing an 8k sequence for LLaMA-30B takes 3.5s, which translates to a service stall similar to 54 decode steps. We also notice that for all sequence lengths, the migration only takes two stages, which is the minimum. This is because the data copying is sufficiently fast and the number of new tokens generated during the first stage is small.

Figure 10 (right) also compares the per-step decode times during migration on the source instance with that during normal execution (results on the destination are mostly similar). We observe up to 1% performance differences for both LLaMA-7B and LLaMA-30B, showing the negligible migration overhead. Also note that such overhead exists only when there are requests being migrated (in or out) on an instance. We find that in all the serving experiments in the following sections, the average fraction of time span with ongoing migration for each instance is only roughly 10%. This implies an effective overhead that is even much smaller, which is worthwhile for the great scheduling benefits of migration.

6.3 Serving Performance

We evaluate the scheduling performance of Llumnix in online serving using 16 LLaMA-7B instances (auto-scaling is disabled except in experiments in §6.5).

Real datasets. We first compare Llumnix with round-robin and INFaaS++ using the ShareGPT and BurstGPT traces (the top two rows in Figure 11). Llumnix outperforms the baselines in end-to-end request latency by up to 2× and 2.9× for mean and P99, respectively. In particular, we observe that round-robin always performs much worse than both INFaaS++ and Llumnix: since the sequence lengths have high variance, simply distributing requests evenly can still lead to unbalanced load, impacting both prefill and decode latencies. Llumnix achieves significant gains in prefill latency over round-robin, by up to 26.6× for mean and 34.4× for P99. This is because round-robin can possibly dispatch new requests to overloaded instances, leading to long queuing delays. Llumnix also improves P99 decode latency by up to 2×, by load balancing to reduce preemptions. This margin seems smaller as the latency penalty caused by preemptions is averaged over all generated tokens. However, whenever preemption occurs, it results in a sudden service stall, which impacts user experience. Figure 11 (the rightmost column) reports the preemption loss in terms of the extra queuing and recomputing times (mean value of all requests). Llumnix reduces preemption loss by 84% on average compared to round-robin. These results highlight the importance of load balancing in LLM serving. In the following experiments using generated distributions with higher variance, round-robin showed up to two orders of magnitude worse latencies. Therefore, we omit it for the other traces for clarity of the figures and focus on the comparison between INFaaS++ and Llumnix.

Llumnix outperforms INFaaS++ in mean and P99 prefill latencies by up to 2.2× and 5.5×, and P99 decode latencies by up to 1.3×, respectively, showing the extra benefits of migration, beyond dispatch-time load balancing. Next we use more traces with different characteristics to further evaluate them for a deeper understanding of the improvements.

Generated distributions. We compare Llumnix and INFaaS++ using multiple generated distributions (bottom five rows in Figure 11). Llumnix outperforms INFaaS++ across all traces in end-to-end request latency by up to 1.5× and 1.6× for mean and P99, respectively. For prefill, the improvements are up to 7.7× for mean and 14.8× for P99. Despite dispatch-

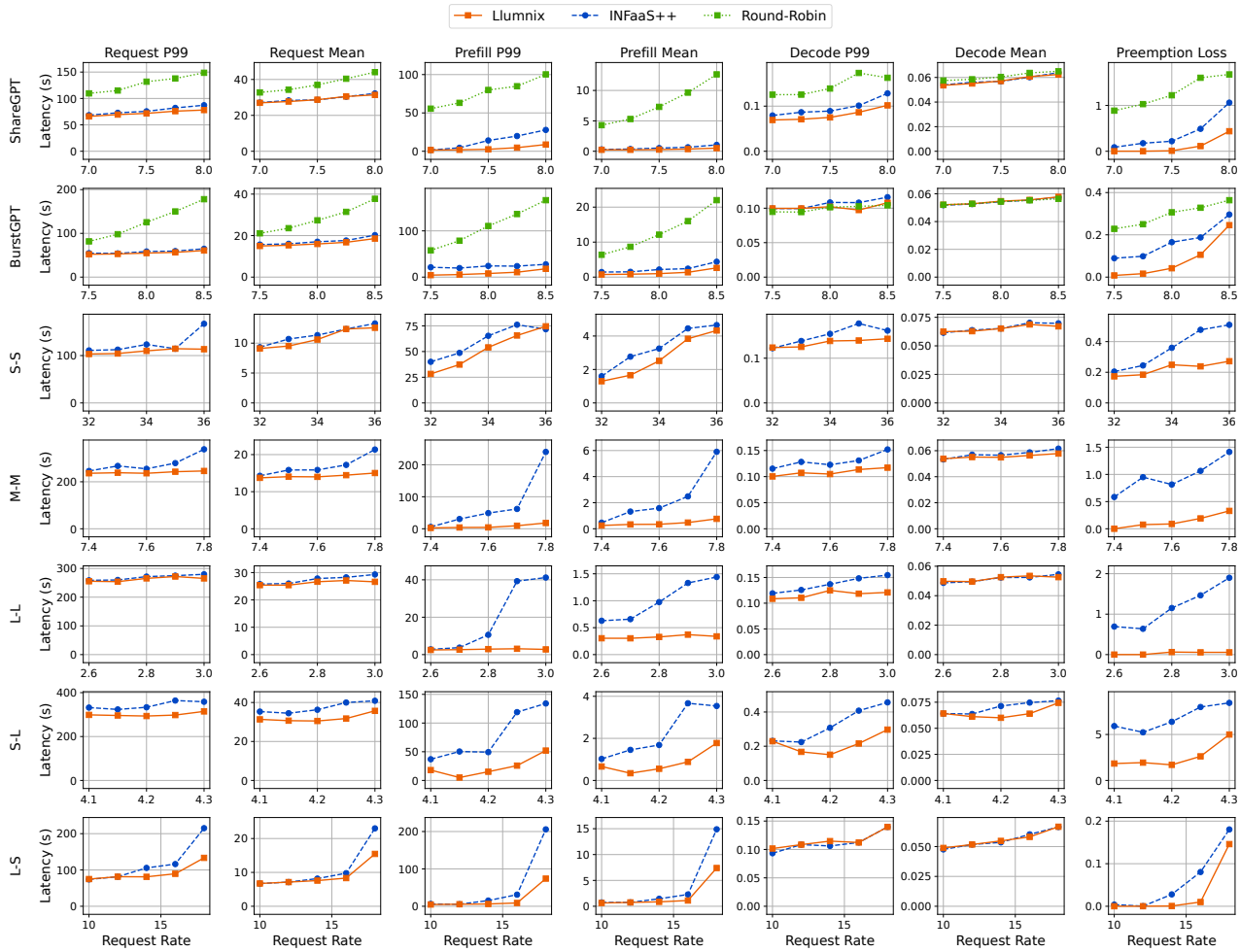

Figure 11: Request end-to-end, prefill, and decode latencies and preemption loss of serving 16 LLaMA-7B instances. Each row shows a set of experiments using a trace with a specific sequence length distribution, as annotated on the Y-axis labels.

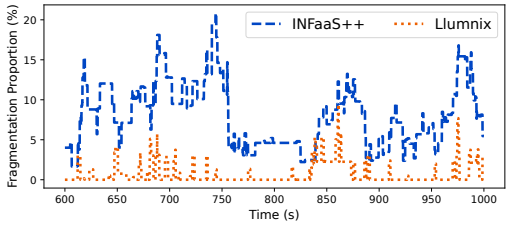

Figure 12: Memory fragmentation over time.

ing requests to instances with the lowest load, INFaaS++ can still exhibit long queuing delays due to fragmentation, especially for the long-tail requests with long inputs. Llumnix uses migration for de-fragmentation to reduce such queuing delays, showing more gains in traces with more long inputs.

To take a closer look at the memory fragmentation, we further present a case study on the experiment of the M-M trace with the request rate of 7.5. We define the fragmented memory at each moment as the portion of cluster free memory that could satisfy the demands of the head-of-line blocking requests across all instances, if no fragmentation. For example, if the total free memory is 8 GB, with three head-of-line blocking requests each requiring 3 GB, then the fragmented memory is counted as 6 GB, *i.e.*, this 6 GB memory could satisfy two queuing requests if no fragmentation. This metric suggests the memory space wasted due to fragmentation. We report the proportion of fragmented memory in the cluster total memory. In the example, if the total memory is 16 GB, then the proportion is 37.5% (6/16). Figure 12 shows the fragmentation proportion of the experiment during a busy period. We observe that INFaaS++ often shows higher than 10% fragmentation, wasting a significant amount of cluster memory. In comparison, the fragmentation is often 0 in Llumnix. The average values during this period are 0.7% and 7.9% for Llumnix and INFaaS++ respectively (92% reduction), highlighting the effect of de-fragmentation using migration.

Llumnix also improves the P99 decode latency by up to

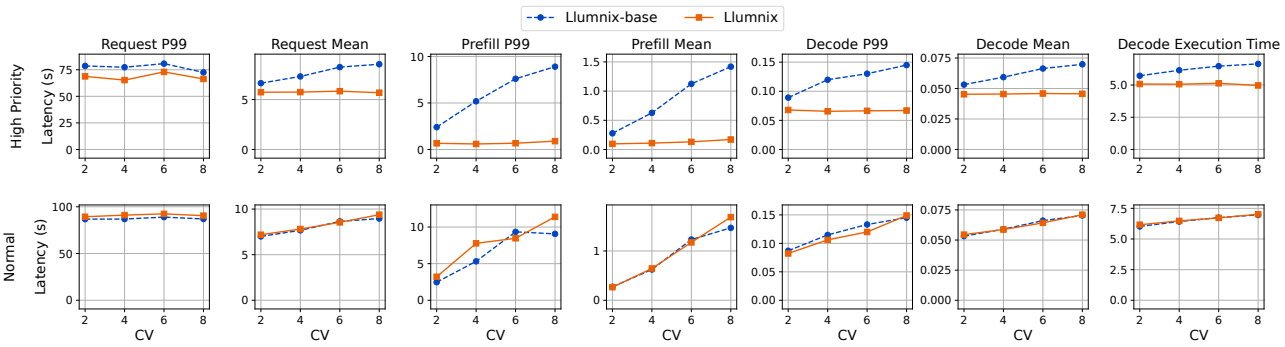

Figure 13: Performance of high-priority and normal requests, as annotated on the Y-axis labels.

2×, through migration to reduce preemptions. Although INFaaS++ already implements load balancing in dispatching to reduce preemptions, migration complements it by reacting to the real sequence lengths, which are unknown at request arrivals. As shown in Figure 11, Llumnix significantly reduces the preemption loss, in many cases down to near zero. The reduction is 70.4% on average across all experiments, which translates to an average reduction of 1.3 seconds in the end-to-end request latency.

6.4 Support for Priorities

We evaluate the support for priorities of Llumnix by randomly picking 10% of the requests and assigning high scheduling and execution priorities. We use traces with the Short-Short length distribution and Gamma arrival distribution. We vary the CV parameter to show the interference to high-priority requests due to bursty workloads and load spikes. We empirically choose a target memory load of 1,600 tokens for high-priority requests, as we observe that such load preserves near-ideal decode speed (refer to Figure 4). Llumnix translates this target load to the corresponding memory headroom for high-priority requests. We compare Llumnix with Llumnix-base, which simply treats all requests as the same priority.

As shown in the first row in Figure 13, Llumnix improves mean request latencies for the high-priority by 1.2× to 1.5× with increasing CVs. Higher CVs leads to more high-load periods, where high-priority requests can suffer more interference if not protected. Even with higher CVs, Llumnix still delivers similar latencies of high-priority requests, showing the isolation Llumnix provides to such requests. This is because Llumnix can handle changing high-priority loads by dynamically creating space for them, which is difficult in approaches like static resource reservation. For prefill latencies, Llumnix shows 2.9× to 8.6× gains for the mean, and 3.6× to 10× for the P99, respectively. This is achieved by reducing the queuing delays with high scheduling priorities. Llumnix also improves decode latencies by 1.2× to 1.5× for the mean and 1.3× to 2.2× for the P99, respectively. This improvement comes from the acceleration of the decode computation by giving lower instance loads and interference to high execution priorities, shown by the similar gains in the average decode computation time (the rightmost column). We also notice that Llumnix preserves similar performance of the normal requests (the second row in Figure 13): Llumnix increases the mean request, prefill, and decode latencies of normal requests by up to 4.5%, 13%, and 2%, respectively.

6.5 Auto-scaling

We evaluate the auto-scaling capability of Llumnix using larger ranges of request rates and Gamma CVs to show the adaptivity to load variation. By default, Llumnix uses a scaling threshold range of [10, 60], *i.e.*, Llumnix scales instances up or down when the average freeness is under 10 or above 60; recall that this metric represents the most decode steps an instance can still run for given the current batch. We let INFaaS++ use the same scaling strategy, thus both Llumnix and INFaaS++ have the same degree of aggressiveness of scaling up instances. We use a maximum instance number of 16 and the Long-Long sequence length distribution.

We first vary the request rates using Poisson distribution. As shown in the first row of Figure 14, Llumnix consistently achieves latency improvements across all request rates, *e.g.*, up to 12.2× for P99 prefill latency. We also measure the resource cost in terms of average instances used, shown in the rightmost column. Llumnix saves costs by up to 16%, because Llumnix increases the auto-scaling efficiency by saturating or draining out instances more quickly. We also test different workload burstiness with varying CVs of Gamma distribution (request rate = 2). As shown in the second row, Llumnix shows similar improvements in latencies and costs, *e.g.*, up to 11× for P99 prefill latency and 18% for the cost.

Finally, we examine the cost efficiency of Llumnix in terms of how aggressively Llumnix needs to scale out instances to preserve a certain latency objective, *e.g.*, a given P99 prefill latency. We vary the scaling up threshold t, and the scaling threshold range is determined as $[t, t+50]$. Higher values of t means that Llumnix tends to use more instances. Figure 15 shows the P99 prefill latencies and costs with different scaling

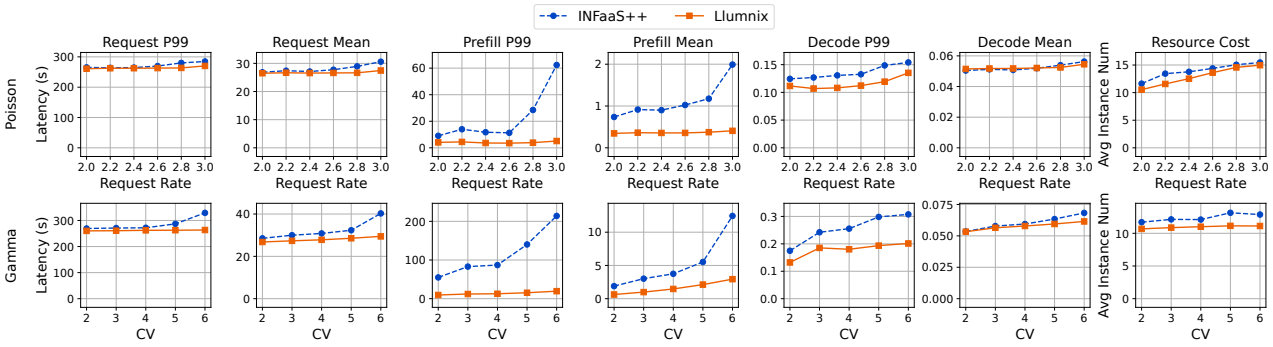

Figure 14: Auto-scaling of LLaMA-7B instances with Poisson and Gamma distributions, as annotated on the Y-axis labels.

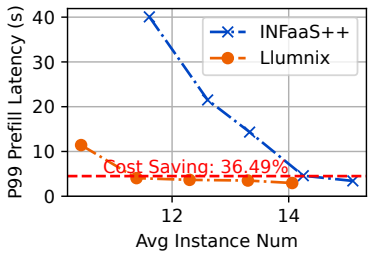

Figure 15: P99 prefill latencies vs. average numbers of instances with varying scaling thresholds.

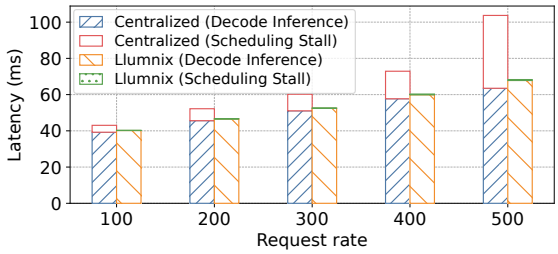

Figure 16: Per-token latencies and scheduling stalls under increasing request rates using 64 LLaMA-7B instances.

thresholds. We observe that Llumnix achieves similar P99 prefill latency (roughly 5s, the red dash line) while saving 36% of the cost compared to INFaaS++, as a result of the combination of the ability to reduce queuing delays via migration and the higher auto-scaling efficiency.

6.6 Scheduling Scalability

We conduct a scheduling stress test to examine the scalability of Llumnix with 64 LLaMA-7B instances using higher request rates. Since this cluster exceeds the size of our testbed, we replace the real GPU execution in vLLM with a simple `sleep` command, whose duration is determined by offline measurement on A10 GPUs with different sequence lengths and batch sizes. We build a simple centralized scheduler as the baseline by extending the vLLM scheduler to manage all requests across all instances. We issue requests with input and output lengths of 64 tokens with increasing request rates.

As shown in Figure 16, with increasing request rates, the baseline experiences scheduling stalls during the inference computation of up to 40ms per iteration, translating to 1.7× slowdown. Such stalls are a result of the communication between instances and the centralized scheduler synchronizing request statuses and scheduling decisions, which becomes a bottleneck under high load. By contrast, Llumnix exhibits near-zero scheduling stalls even under high request rates, showing the scalability of the distributed scheduling architecture. Llumnix offloads and distributes the intra-instance scheduling logic across llumlets so that it is done in parallel and asynchronously with the global scheduling. Moreover, llumlets only report instance-level metrics, instead of the precise status of every single request, further improving the communication efficiency.

7 Related Work

LLM inference. As transformer models show significance in model serving, recent works, such as FasterTransformer [46], TurboTransformer [25], LightSeq [61], and FlashAttention [21, 22], optimize GPU kernels to improve the inference performance. SpotServe [41] supports LLM inference using preemptible instances for improving cost efficiency. FastServe [63] optimizes request completion times using a preemptive time-slicing approach. AlpaServe [35] exploits pipeline parallelism to reduce serving latency for bursty workloads. To further increase the GPU utilization and serving throughput, Orca [67] proposes iteration-level scheduling (referred to as continuous batching in recent works and this paper) and selective batching, while vLLM [34] optimizes the memory usage with PageAttention. [55] proposes fair scheduling of requests on an LLM instance. Prior works mostly target solo-instance serving, therefore complementing to Llumnix. Llumnix explores the challenges and opportunities of deploying multi-instance LLM serving. The key

append-only characteristic of KV cache is exploited to enable migration capability of requests in the inference engine. Such a mechanism opens great policy design space to offer priority and performance isolation, improve memory efficiency, and enable instance auto-scaling. We also plan to explore the interplay between the global scheduling across instances with local scheduling techniques inside each instance (*e.g.*, preemptive [63] and fair [55] scheduling) as future works.

Request scheduling. To support deep learning model deployment, numerous systems (*e.g.*, Clipper [19], Nexus [54], DVABatch [20], and TritonServer [47]) have been proposed to optimize request scheduling for DNN inference serving. To meet the SLOs of DNN inference requests, Clockwork [29] utilizes the execution predictability of traditional DNNs, while Reef [33] and Shepherd [68] perform preemptions to serve high-priority requests. AlpaServe [35] uses a simple load-balancing dispatching policy based on queue lengths. These works mostly focus on traditional DNN model serving, where a request requires only one-time inference on the model. However, LLM inference service requires autoregressive computation on models for unpredictable numbers of iterations and introduces intermediate states (*i.e.*, KV cache), showing brand new characteristics. DeepSpeed-MII [4], albeit targeting multi-instance LLM serving, uses a simple round-robin dispatching policy that ignores LLM characteristics. Llumnix steps further to incorporate request migration and ensures high throughput and low latency, provides SLO for prioritized requests, and auto-scales instances for resource efficiency with a unified load-aware dynamic scheduling policy.

Beyond multiple model instances, INFaaS [53] further supports scheduling across multiple model types/variants, considering the performance and accuracy requirements in difference applications. This is also a typical scenario for LLMs: for example, fine-tuned models for a specific task (*e.g.*, coding [3, 13, 30]); variants with different sizes or precisions ([26,37,39]) of the base LLM. We plan to extend Llumnix to support multiple model types in future work, considering the larger tradeoff space of latency/throughput and accuracy.

Isolation vs. fragmentation. The tradeoff between isolation and fragmentation, or that between workload packing and spreading, have been a classic scheduling challenge. That is, workload packing improves resource utilization, at the expense of potential interference between co-located workloads; spreading workloads, on the contrary, provides better isolation but also increases resource fragmentation. Many research efforts have been devoted to better balancing isolation and fragmentation in datacenters for big-data jobs and virtual machines, by identifying the interference-sensitivity of workloads and optimized scheduling policies ([16,18,23,24,27,31,32,40,60,66]). This challenge was also identified in GPU clusters for deep learning workloads. Amaral et al proposed a topology-aware placement algorithm to address the tradeoff between packing and spreading deep learning training jobs on multi-GPU servers [12]. Gandiva [64] addresses the heterogeneous sensitivity to packing/spreading of different jobs with introspective job migration. This challenge becomes more complex for LLM serving due to the unpredictable autoregressive execution. Llumnix exploits request migration at runtime to react to the workload dynamics to better reconcile these two goals.

Migration. Gandiva [64] enables introspective migration for deep learning training jobs during scheduling. It utilizes the inherent iterative behavior of deep learning, and conducts checkpoint-resume approach on the minimal working set (*i.e.*, mini-batch boundary) to migrate model weights. Even though LLM inference is iterative as well, directly migrating the entire states of a request is unacceptable, because the latency SLO of an inference request is crucial. Moreover, the working set per request is linear to the sequence length, which can be considerable given the trend of longer contexts [49, 50]. The migration approach in Llumnix is inspired by virtual machine live migration [17]. By carrying out the majority of migration while LLM requests continue decoding tokens on GPUs, Llumnix minimizes the downtime of request migration, making the cost negligible regardless of the sequence lengths.

8 Conclusion

Llumnix, as implied by the name, represents our vision of serving LLMs as Unix. This vision originates in the observation that LLMs and modern operating systems have common natures such as the universality, multi-tenancy, and dynamism, and hence share similar requirements and challenges. This paper takes an important step towards this vision by drawing lessons from conventional OS wisdom including: definition of classic abstractions like isolation and priorities in the new context of LLM serving; implementation of the "context switching" as the key approach with inference request migration; and continuous, dynamic request rescheduling exploiting the migration. All these combined, Llumnix delivers better latency, cost efficiency, and support for differentiated SLOs, pointing to a new way of LLM serving.

Acknowledgement

We thank the anonymous OSDI reviewers and our shepherd for their valuable feedback, which helped improve the presentation of this paper. We thank Shiru Ren for early discussion on VM live migration techniques. This work was supported by Alibaba Group through the Alibaba Research Intern Program. This work was also supported by National Key Research and Development Program of China (Grant No. 2023YFB3001801), National Natural Science Foundation of China (Grant No. 62322201, 62072018, U23B2020 and U22A2028), Fundamental Research Funds for the Central Universities (YWF-23-L-1121), and State Key Laboratory of Software Development Environment (SKLSDE-2023ZX-05).

References

[1] Nccl, https://developer.nvidia.com/nccl/.

[2] Chatgpt plus, 2023. https://openai.com/blog/chatgpt-plus.

[3] Code llama, 2023. https://github.com/facebookresearch/codellama.

[4] Deepspeed-mii, 2023. https://github.com/microsoft/DeepSpeed-MII.

[5] Gloo collective communication library, 2023. https://github.com/facebookincubator/gloo.

[6] Google bard, 2023. https://bard.google.com/.

[7] Longllama, 2023. https://github.com/CStanKonrad/long_llama.

[8] Openai chatgpt, 2023. https://chat.openai.com/.

[9] Ray serve, 2023. https://docs.ray.io/en/latest/serve/index.html.

[10] Sharegpt_gpt4, 2023. https://huggingface.co/datasets/shibing624/sharegpt_gpt4.

[11] vllm. https://github.com/vllm-project/vllm, 2023.

[12] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 17, New York, NY, USA, 2017. ACM.

[13] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[16] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32, 2017.

[17] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005.

[18] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[19] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[20] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 183–198, Carlsbad, CA, July 2022. USENIX Association.

[21] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR*, abs/2307.08691, 2023.

[22] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[23] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 77–88, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 127–144, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: An efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 389–402, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2023.

[27] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the thirteenth EuroSys conference*, pages 1–13, 2018.

[28] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[30] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

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

[32] Jaeung Han, Seungheun Jeon, Young-ri Choi, and Jaehyuk Huh. Interference management for distributed parallel applications in consolidated clusters. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 443–456, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[34] 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.

[35] 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)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[36] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel J. Orr, Lucia Zheng, Mert Yüksekgönül,

Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. *CoRR*, abs/2211.09110, 2022.

[37] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. In *MLSys*, 2024.

[38] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *CoRR*, abs/2310.01889, 2023.

[39] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits, 2024.

[40] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible colocations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[41] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances, 2023.

[42] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[43] Avanika Narayan, Ines Chami, Laurel J. Orr, and Christopher Ré. Can foundation models wrangle your data? *Proc. VLDB Endow.*, 16(4):738–746, 2022.

[44] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[45] NVIDIA. Using multiple nccl communicators concurrently. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/communicators.html#using-multiple-nccl-communicators-concurrently.

[46] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2023.

[47] NVIDIA. Triton inference server. https://github.com/triton-inference-server/server, 2023.

[48] OpenAI. Openai api, 2020. https://openai.com/blog/openai-api.

[49] OpenAI. Gpt-4 technical report, 2023.

[50] OpenAI. Gpt-4 turbo. https://help.openai.com/en/articles/8555510-gpt-4-turbo, 2023.

[51] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022.

[52] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022.

[53] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[54] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models, 2023.

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[57] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[58] 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.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

[60] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18, New York, NY, USA, 2015. ACM.

[61] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. Lightseq: A high performance inference library for transformers. In Young-bum Kim, Yunyao Li, and Owen Rambow, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 113–120. Association for Computational Linguistics, 2021.

[62] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Towards efficient and reliable llm serving: A real-world workload study, 2024.

[63] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.

[64] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[65] Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, Madian Khabsa, Han Fang, Yashar Mehdad, Sharan Narang, Kshitiz Malik, Angela Fan, Shruti Bhosale, Sergey Edunov, Mike Lewis, Sinong Wang, and Hao Ma. Effective long-context scaling of foundation models, 2023.

[66] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 607–618, New York, NY, USA, 2013. Association for Computing Machinery.

[67] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[68] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

A Artifact Appendix

Abstract

This artifact includes the source code and scripts to run the experiments and reproduce the evaluation results of this paper.

Scope

The artifact can be used to reproduce the results of the following experiments.

- Migration efficiency: Figure 10.
- Serving performance: Figure 11.
- Support for priorities: Figure 13.
- Auto-scaling: Figure 14 and Figure 15.

Contents

This artifact includes the following contents.

- Source code of a prototype implementation of Llumnix.
- Scripts to prepare the environment, run the experiments, plot the figures, and validate the claims in this paper automatically.
- A `README` file including detailed instructions on how to use this artifact.

Hosting

The artifact is publicly available at `https://github.com/AlibabaPAI/llumnix` (the `osdi24ae` branch). Note that this is not the same branch as the official release of Llumnix (the `main` branch). We will describe their difference later.

Requirements

The artifact runs on GPU machines, with software dependencies mostly the same as those of vLLM. To reproduce our results, you would need 4 GPU machines each with 4 A10 GPUs (24 GB). We recommend that you use the same VM type as in our experiments (`ecs.gn7i-c32g1.32xlarge` on Alibaba Cloud).

Difference from the Official Release

This artifact is a research prototype and was used during the experiments of this paper. After the paper submission, we refactored it into a new implementation that is more production-ready, *i.e.*, the official release, as described in §5. Major differences between the two versions include:

- The artifact is directly based on the vLLM code base, whereas the official release is a standalone Python library, making it more extensible and non-intrusive to backend inference engines.
- The artifact is not fault-tolerant, whereas the official release provides fault tolerance for each component.
- The official release is still being actively developed, and has supported or will support a series of new features, such as scalable API servicing via distributed request frontends, support for newer versions of vLLM and more models, further improvements of the scheduling policies, *etc*.

The artifact is sufficient to reproduce the experiment results in this paper. However, if you want to use Llumnix in production or conduct further research, we do recommend the official release.