Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Enable importance-aware model cacheability for inference serving

Hao Mo [a], Didier El Baz [b], Ligu Zhu [a], Suping Wang [a], Songfu Tan [a], Hongning Zhao [a], Lei Shi [a,*]

[a] *State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing, 100024, China*
[b] *LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, 31031, France*

ARTICLE INFO

Keywords:
Deep learning
Inference serving
Replacement algorithm
Memory cache

ABSTRACT

Inference serving systems are leveraged to deploy deep learning (DL) models as services. Accelerators such as Graphics Processing Units (GPUs) have been extensively used in these systems to reduce model execution time. As accelerators become more powerful and expensive, GPU sharing among DL models across different inference requests is a common practice. However, GPU memory capacity becomes a bottleneck when the number of collocated models increases, making this approach unsustainable. At the same time, collocated models may vary in popularity levels — some are accessed frequently and others are not, leading to low resource efficiency and system performance. While some existing inference serving systems offer the capability to dynamically load and cache model in memory, they are typically locality-aware and exhibit poor performance for DL inference serving.

To this end, we present mCache, a novel inference serving-oriented caching system to dynamically manage a set of collocated models with diverse popularity for efficient use of memory. mCache treats each DL model as a cacheable object, and loads model on demand and unloads models when not in use. Rather than using recency or frequency, we manage models in GPU memory based on rank-based importance scores, which jointly consider cache access patterns and model-specific factors, serving as a unified metric to compare different models. During model serving, the importance scores of cached models are dynamically updated, and the least important model is evicted to make room for a newly targeted model when the cache is full. Evaluation with representative DL models shows that mCache reduces memory footprint by nearly a half with a modest inference latency increase. Compared to existing serving system using Least Frequently Used (LFU) caching algorithm, mCache improves throughput by up to 1.5× and 2.39× given the 40% and 80% GPU memory capacity.

1. Introduction

Model as a Service (MaaS) is an increasingly popular deep learning (DL) inference serving paradigm. Many cloud providers such as Google Cloud (Anon, 2025e) and Amazon (Anon, 2025a) offer MaaS as an interface to usage-driven back-end services. MaaS provides an intuitive interface for developers to deploy DL model prediction services. In contrast to traditional cloud interfaces, in MaaS, developers do not explicitly provision or configure virtual machines (VMs) or containers. Instead, developers simply upload their DL model applications to the cloud; models get executed when applications are “triggered” or “invoked” by end-users, for example, the receipt of a message (e.g., an HTTP request). The cloud provider is then responsible for providing the required resources (e.g., container instances) for executing each model.

Obviously, cloud providers seek to deliver contractual-compliant DL inference performance at the lowest possible resource cost. To achieve this, some works have explored cluster-level techniques, such as autoscaling (Romero et al., 2021; Tang et al., 2019; Zhang et al., 2019; Mo et al., 2023) and scheduling (Mendoza et al., 2021; Tan et al., 2021; Wang et al., 2021; Wu et al., 2020) for DL inference jobs. These techniques are useful because DL inference service is compute-intensive and typically requires multiple Graphics Processing Unit (GPU) accelerated instances to serve in parallel. Another commonly used approach is to collocate inference jobs on the same GPU so their models can share the compute resource, which is our concern in this work. Most existing studies on collocation optimization have focused on techniques such as operator scheduling (Ding et al., 2021; Yu et al., 2021), service router (Choi and Rhu, 2020; Mendoza et al., 2021; Wu et al., 2020; Soifer et al., 2019), resource partitioning (Anon, 2025h; Dhakal et al., 2020; Ghodrati et al., 2020) to avoid job interference. However, two challenges have received less attention: (1) memory limitation. GPU

* Corresponding author.
E-mail addresses: haroldmua@cuc.edu.cn (H. Mo), elbaz@laas.fr (D. El Baz), zhuligu@cuc.edu.cn (L. Zhu), supingwang@cuc.edu.cn (S. Wang), tsf@cuc.edu.cn (S. Tan), zhaohongning@cuc.edu.cn (H. Zhao), leiky_shi@cuc.edu.cn (L. Shi).

https://doi.org/10.1016/j.engappai.2025.111908
Received 11 April 2024; Received in revised form 23 May 2025; Accepted 28 July 2025
Available online 9 August 2025

memory capacity is limited and can only accommodate a small number of models, hindering higher hardware utilization and cost efficiency. (2) resource usage disparity. While GPU memory remains filled with models during inference serving, computational utilization stays low due to some models being infrequently accessed. Ideally, we want to host as many models as possible on a single GPU, even if the total memory demand of models exceeds the memory capacity, while giving the illusion that all models are always warm (loaded in memory), yet spend resources as if they are always cold (not loaded in memory).

We thus propose model-level in-memory caching, or model caching, a novel approach to address these issues in collocation inference and, more generally, improve the performance of cloud-based DL inference serving. Model caching handles DL models as cacheable objects. Instead of loading all of the targeted models in GPU memory at once, it dynamically loads and unloads models into and out of memory based on their usage patterns. Upon the arrival of an invocation request, the serving system efficiently decides which models to retain in the cache and which ones to remove based on request information and in-memory model items, particularly when there is insufficient memory space. For instance, if the requested model is already resident in memory, this constitutes a cache hit. In that case, it is served faster since the serving system does not need to reload it again. Otherwise, the invocation represents a cache miss and experiences a longer response time owed to model load delay.

A key challenge of model caching is to decide which model to evict when the cache is full. A common practice is to determine the eviction candidates using traditional caching algorithms like Least Recently Used (LRU) or Least Frequently Used (LFU), based on the recency or frequency of their use. For example, Amazon Web Services (AWS) SageMaker (Anon, 2025b) uses LFU algorithm for cache replacement when hosting multiple models in one container. However, these traditional caching algorithms do not work effectively with model management because they primarily target higher cache hit ratio, without considering a key feature of DL inference — loading models completely into memory prior to being executed. As a result, a new cache replacement algorithm is needed for model caching.

For this purpose, we design and implement mCache, a caching system specifically designed for DL inference serving. mCache enables model cacheability by introducing a rank-based importance score that captures the priority of each model and designing an effective caching algorithm based on the score. The importance score is calculated as a uniform metric to compare DL models that are heterogeneous both in terms of model size and loading time. The model with the smallest importance score is the most suitable candidate for eviction. mCache then uses these importance variations to make caching decisions during DL inference serving. Based on these techniques, our caching solution keeps the most important models in the cache and avoids random evictions.

In summary, this paper offers the following key contributions:

- We introduce a novel importance calculation approach to identify the relative importance of in-memory model items for DL inference serving.
- We design a cache replacement algorithm based on model importance to dynamically manage models in GPU memory.
- We present the design of mCache, a new caching system integrated with the proposed algorithm to enable model cacheability for inference serving system.
- We implement mCache using Tensorflow Serving as the underlying framework. We evaluate it under four different workload characteristics and compare mCache against other caching methods. Our results show that mCache saves nearly a half memory usage with a modest increase in inference latency in correlated workload.

2. Background and related work

2.1. Deep learning inference serving

Deep learning inference serving is the process of deploying trained deep learning models into a production environment to make predictions on new, unseen data in real-time or batch mode. System design must balance competing demands such as latency, throughput, and resource efficiency, depending on the use case. For instance, low-latency inference (e.g., < 100 ms) is critical for interactive applications like virtual assistants, while high-throughput batch processing suits offline tasks like video analysis. The conventional approach to deploying DL prediction services is to provision a container and, within the container, host the DL model on a serving framework. The serving framework is analogous to a webserver and exposes services with interfaces via a REST API. For example, DL serving frameworks like TensorFlow Serving (Olston et al., 2017; Anon, 2025j), TorchServe (Anon, 2025k) and NVIDIA Triton (Anon, 2025i) host the model in a Docker (Anon, 2025d) container to enable process isolation and expose services via HTTP or RPC protocols.

Since DL models mainly need to support user-oriented applications, DL inference serving generally has certain latency constraints, which require queries to be served within a given latency. A practical approach to accelerate model inference is to adopt DL-dedicated hardware, such as GPU and TPU (Jouppi et al., 2017). This approach works because DL models contain many complex computational operations, such as matrix multiplication, which can be efficiently parallelized using accelerators, significantly reducing execution time. Meanwhile, given model prediction services' scale and elastic scaling requirements, serving systems are typically deployed in the cloud. Cloud providers can then allocate a set of GPU-accelerated container instances and use a resource manager such as Kubernetes (Luksa, 2017) to deploy and manage the service.

2.2. Multi-model co-location inference

Multi-model co-location inference is a multi-tenant single-device computing paradigm in which multiple DL models co-run on a single high-performance hardware. The introduction of collocation inference optimization is mainly due to the mismatch between the huge computational power of recent GPUs (e.g., NVIDIA A100 with 312 TFLOPS) and the inference requirements of general DL models (e.g., ResNet50 model with 4 GFLOPs). Executing such a single DL model on a modern GPU may lead to severe resource underutilization. Collocation inference thus offers a cost-efficient approach to accommodate more model deployments by improving hardware utilization (Yu et al., 2022).

However, as the number of hosted models increases, collocation inference encounters obstacles in further improving GPU resource usage. One critical reason is the constrained capacity of GPU memory, which restricts the number of models that can reside in memory simultaneously. Unlike CPU memory, which is expandable, GPU memory capacity is fixed and dependent on the specific GPU hardware type. An inference server equipped with a fixed-size memory GPU can only accommodate a limited number of models. Expanding GPU memory by using GPUs with larger capacity or increasing the number of servers may be challenging to sustain given the significant monetary cost associated with these options. As such, efficiently managing GPU memory to concurrently accommodate a diverse set of models is critical for DL inference serving.

2.3. Memory management for deep learning inference

Deploying multiple models on a single GPU requires dynamically swapping models in and out of GPU memory based on demand. When a request arrives for a model that is not currently loaded, the system

Table 1
A comparison of mCache and existing works on model caching.

Solutions	Caching policy	GPU-enabled	Cloud	Locality-aware	DL inference-oriented
Ogden et al. (2021)	BeladyMIN variant	×	×	×	×
Anon (2025b)	LFU	✓	✓	✓	×
Gujarati et al. (2020)	LFU/LRU	✓	✓	✓	×
Cox et al. (2020)	LRU	✓	✓	✓	×
Zhao et al. (2023)	LRU	✓	✓	✓	×
Dakkak et al. (2019)	LRU	✓	✓	✓	×
mCache	IAM	✓	✓	×	✓

must load the required model into memory, perform the inference, and then potentially unload it afterward. The main challenge in this process arises when a model that is not resident in GPU memory is needed for an inference task. Given the large size of these models, the time required to load them into the GPU introduces a significant performance bottleneck.

A primary strategy for addressing this issue is to keep model parameters close to the GPU to reduce loading latency. An effective approach, as proposed in Zou et al. (2023), is to cache model parameters in system RAM. By storing model parameters in host memory, the data transfer time to the GPU during inference is substantially reduced, thereby enabling quicker loading and unloading of models from the GPU, particularly when hosting multiple models on the same computing instance.

Caching models in host memory can significantly reduce loading times, often to just a few milliseconds. However, transferring data from host memory to GPU memory remains more time-consuming than the actual inference process, leading to GPU idle periods during data transfer. To address this inefficiency, the approach presented in Gujarati et al. (2020) conceptualizes GPU memory as a cache, enabling frequently or recently accessed models to remain resident in GPU memory and thereby avoid costly loading delays. The ModelMesh framework, as detailed in Cox et al. (2020), preloads model parameters and leverages GPU memory as a cache to prioritize frequently used models. It employs an LRU caching strategy to facilitate efficient model serving. Similarly, the approach in Zhao et al. (2023) implements a GPU memory caching mechanism that leverages the GPU's LRU list to determine which models to evict. Trims (Dakkak et al., 2019) introduces GPU caching via a daemon process that provisions GPU memory by intercepting CUDA requests and applying basic caching strategies, such as LRU, to reduce latency. Ogden et al. (2021) propose a lightweight cache eviction policy based on a BeladyMIN variant, designed to optimize model caching in system memory for edge-based inference scenarios. In the commercial domain, AWS SageMaker (Anon, 2025b) adopts an LFU-based algorithm to dynamically load and cache models based on access frequency.

The aforementioned solutions treat GPU memory as a cache, which aligns with the focus of our work. However, they typically rely on traditional cache replacement policies (e.g., LRU), which are suboptimal for model caching. The effectiveness of such algorithms depends heavily on factors like the inter-arrival time distribution of requests and the relative popularity of cached objects, aiming primarily to improve conventional metrics such as cache hit ratio. In contrast, mCache is purpose-built for deep learning inference and prioritizes user-centric metrics, particularly inference latency. It explicitly considers model-specific characteristics, such as model size and the latency overhead caused by cache misses. A comparison between mCache and related work is presented in Table 1.

3. Motivation

Efficient management of GPU memory requires a comprehensive understanding of the characteristics of the DL serving workload. We begin this section by analyzing a representative serving workload, with a particular focus on characteristics related to the invocation pattern and memory usage. Additionally, we compare the model execution time to the loading time.

3.1. Characterization of serving workload

For workload characterization, we use Microsoft's publicly released Azure Functions trace (Shahrad et al., 2020). We choose this trace because it represents the real-world production FaaS workloads from a major cloud provider, and recent studies (Lv et al., 2025; Strati et al., 2024; Zhao et al., 2023; Li et al., 2023; Zhang et al., 2023) have recognized its representativeness for DL serving workloads. Our use of Azure traces as a surrogate for DL serving workloads is justified by two key reasons: First, both FaaS and MaaS typically offer services for user-oriented applications, which often leads to a comparable invocation pattern. Furthermore, the Azure trace includes a significant number of components involving requests for DL models (Fuerst and Sharma, 2021; Ishakian et al., 2018). This suggests that insights gained from analyzing the FaaS workload's traffic distribution are relevant to the DL serving workload.

Invocation Patterns. We first parse the Azure trace to explore the distribution of requests rate. Fig. 1 shows the distribution of request rate by dividing the average requests per second into 100 buckets on a logarithmic scale, where the X-axis represents the average request rate of the functions within 24 h, and the Y-axis indicates the number of functions corresponding to the average request rate. Note that an application may have one or more functions, and only functions triggered by HTTP requests are counted. As shown in Fig. 1, the average requests per second range from a minimum of 1.15e−05 to a maximum of 9.67e+02. While some functions have a request rate close to 1000, only 1.17% of functions are invoked more than once per second, showing a severe long-tailed distribution. The observation indicates that within the serving workload, there is a significant variation in average request rates among different applications. Only a subset of highly popular applications experiences elevated average request rates, whereas others exhibit markedly lower rates. Additionally, there exists a pronounced disparity in the frequency of popular versus unpopular applications, with the latter outnumbering the former by a considerable margin. Such invocation patterns highlight the necessity for dynamic loading and caching of models in accordance with the serving workload.

Memory Usage. To further investigate the impact of such workload characteristics on resource usage, we show potential memory usage in Fig. 2 based on some reasonable assumptions. We process the Azure trace in three steps: (1) We categorize functions into buckets according to the quantiles of their popularity, repeating this process multiple

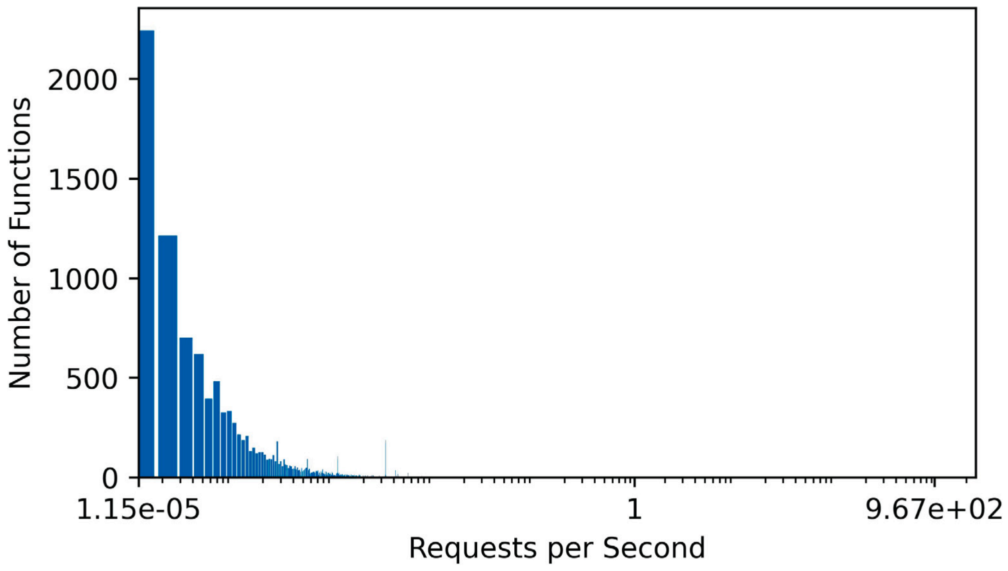

Fig. 1. Traffic distribution of Azure trace.

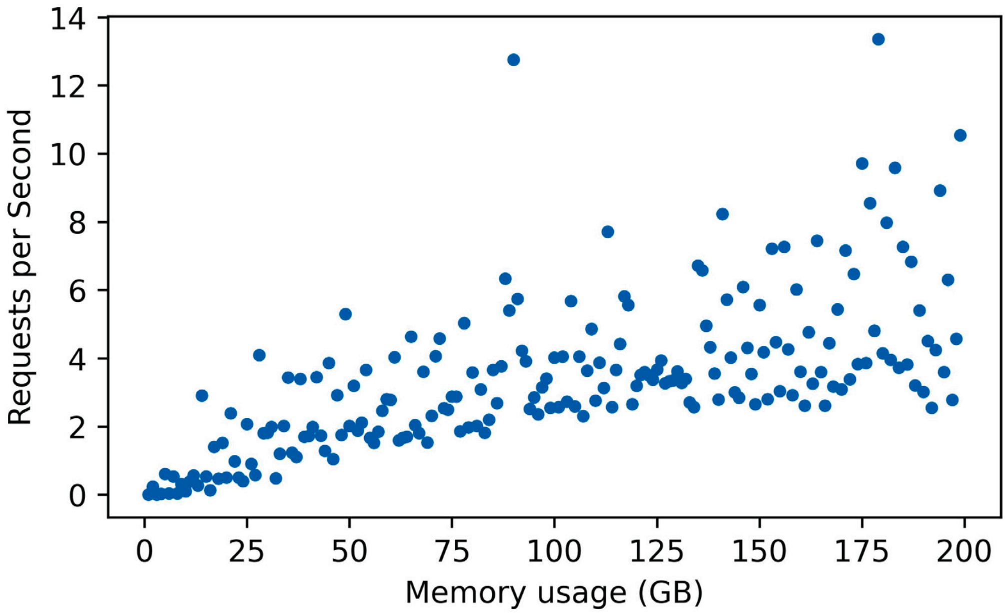

Fig. 2. Memory requirement. The amount of memory required increases faster than the number of requests per second.

times with the number of buckets varying from 1 to 200. For example, when the number of buckets is set to 10, functions are divided into ten groups based on their popularity distribution. (2) From each bucket, we randomly select one function for each grouping and then sum the number of requests per second. (3) We repeat this process 100 times and calculate the average requests per second to ensure data consistency. For this analysis, we assume that each function requires 1 GB of memory. Subsequently, we plot the scatter plot in Fig. 2, with the required memory (i.e., the number of functions) as the X-axis data and the average requests per second of functions as the Y-axis data. As illustrated in Fig. 2, even with memory usage increasing to 200 GB, the typical requests per second remain below 15. This indicates that in order to align with the request volume targeted by the MLPerf inference benchmark (Reddi et al., 2020), it would be necessary to host and maintain 200 individual models concurrently in memory. Hosting such a substantial number of models will hinder any single model from achieving computational saturation.

3.2. Execution and loading time comparison

When a model cache miss occurs, the targeted model needs to be completely loaded into memory before it can be executed, akin to a cold start. We thus use the cold start latency to simulate inference latency with a model cache miss. For latency measurement, we benchmark 9 Convolutional Neural Network (CNN) models available within Keras (Anon, 2025g), with model sizes ranging from a few MBs to hundreds of MBs. We directly count the time from load and execution actions for each model. For hosting each model, we employ a containerized TensorFlow Serving (Olston et al., 2017) instance running on an NVIDIA V100 GPU. To minimize execution time and highlight the potential dominance of loading time in overall latency, we submit inference requests with a batch size of 1 from simulated clients to TensorFlow Serving. We execute 20 load-and-unload cycles for each model, with 10 executions per cycle, and then calculate the average loading time and execution time, respectively.

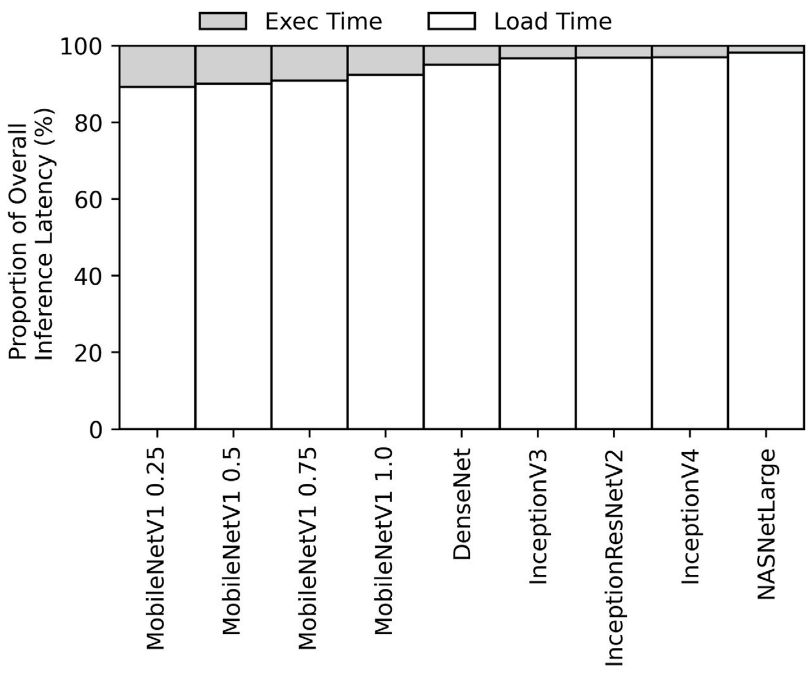

Fig. 3. The overall inference latency component proportions.

Table 2
Inference latency statistics of representative CNN models. Inference latency breakdown comprises execution time and loading time.

Model	Execution time (ms)	Loading time (ms)	Size (MB)
MobileNetV1 0.25	16.0	131.8	1.9
MobileNetV1 0.5	16.3	147.5	5.2
MobileNetV1 0.75	17.4	172.7	10.5
MobileNetV1 1.0	17.5	209.6	17.1
DenseNet	30.8	586.3	43.9
InceptionV3	34.7	1010.6	95.7
InceptionResNetV2	47.4	1476.3	121.6
InceptionV4	51.5	1642.3	171.2
NASNetLarge	70.0	3702.3	356.6

We profile the overall model inference latency and present its component proportions in Fig. 3. This figure demonstrates that the loading time is significantly greater than the execution time, thereby predominating the overall inference latency for all models. The loading time can vary from hundreds of milliseconds to several seconds, as detailed in Table 2. This observation is consistent with expectations, as loading time is affected by the speed of transferring model parameters from persistent storage, such as a disk, to RAM, and then to GPU memory. The speed of this transfer primarily depends on the inherent connection (e.g., PCIe) between the CPU and GPU, leaving limited room for optimization. Therefore, it is advisable to minimize model reloading and instead, retain frequently used models in GPU memory.

4. mCache: Model-level in-memory caching

Our study in the previous section highlights the potential for enabling fundamental model locality in DL inference serving. In this section, we discuss the challenges and design of mCache, followed by its caching algorithm.

4.1. Challenges

Our goal is to achieve a model-level in-memory caching system that treats each DL model as a cacheable object. Similar to traditional caching systems such as Content Delivery Network (CDN), model caching also employs a multi-level cache hierarchy. For instance, models can be stored in GPU memory or persistent storage such as local disks or remote model repositories. However, model caching poses several challenges to effectively cache model items compared to these traditional caching systems.

First, cache miss penalty. DL model cache misses need to be handled differently. In the case of model caching, when a cache miss occurs, the requested model must be entirely loaded into the top-level cache (i.e., GPU memory) before execution. This loading process can take from hundreds of milliseconds to several seconds. If the top-level cache is already full, a model eviction decision needs to be made based on relevant factors. This differs from other caching systems like CDNs, where the requested object can be read from any cache level and does not necessitate being fully loaded into the higher-level cache before serving the request (Berger et al., 2017).

Second, variable model size. Caching fixed-size objects in traditional caching systems simplifies resource management since they are aware of memory space requirements in advance and can allocate equivalent memory sizes for objects in all cache levels. However, the memory space requirements of DL models typically vary widely based on their sizes.

Third, dynamic access patterns. The serving workload is not static but dynamic. In the cloud environment, for example, due to the unpredictable nature of inference request arrivals (Zhang et al., 2019), perfectly preloading the targeted model is often not feasible. This poses significant challenges in maintaining a high model cache hit ratio and achieving the desired serving throughput.

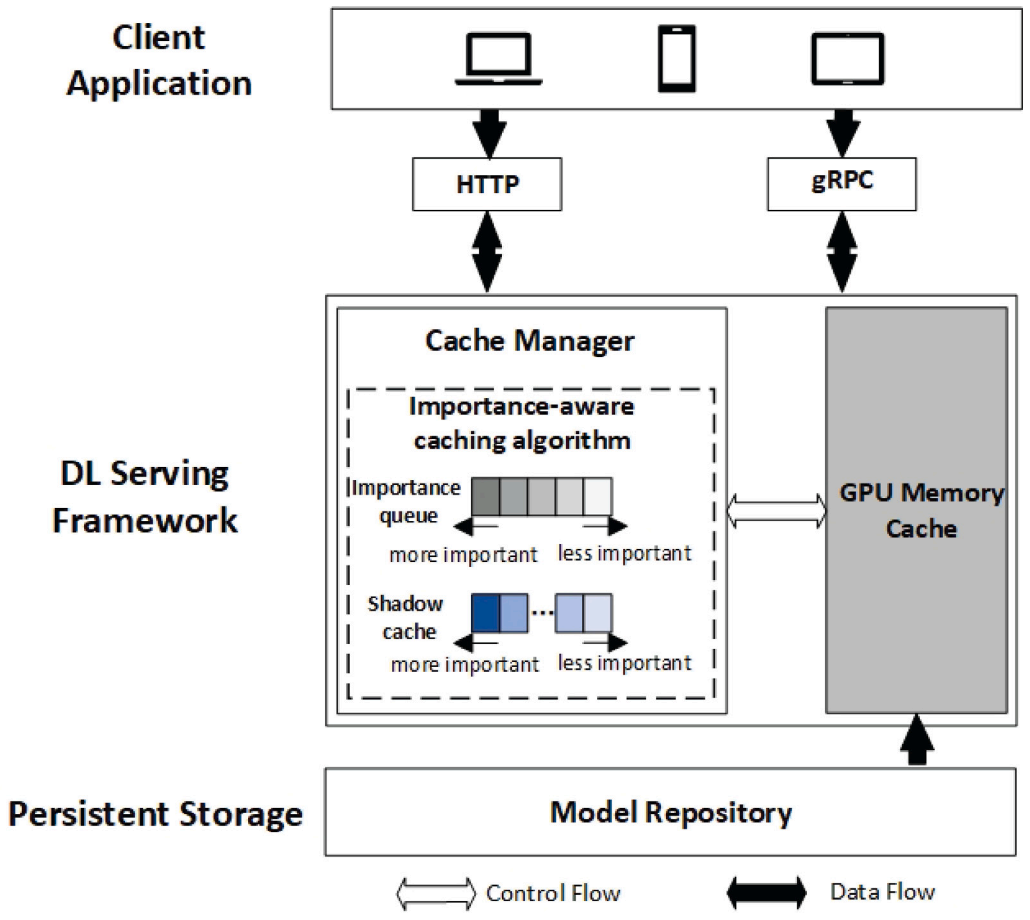

Fig. 4. The logical architecture overview of mCache.

In summary, these challenges can be summarized as the high cache miss penalty and variable memory space requirements during DL inference serving, particularly in the presence of dynamic access patterns. These challenges guide the design of our importance-aware model caching algorithm.

4.2. *System overview*

mCache is a novel caching system built upon DL serving framework to enable model cacheability for inference service. Within mCache, the models are cached in GPU memory and executed with given inputs to provide consistent response times for invocation requests from end-users. For the initial inference, the cache manager fetches the models from persistent storage and populates the cache with the models to be accessed first. Upon receiving the required model list, the cache manager checks whether each model resides in the cache. Assuming the cache is full and the requested model is not found in the cache (cache miss), our importance-aware model caching algorithm will evict the least important model previously cached to make space for the newly requested model. Specifically, the algorithm uses a min heap-based importance queue and shadow cache to track the models along with their associated importance score. The heap objects are key–value pairs, where the key represents the importance score, and the value is a reference to the model item in the cache. These objects in the heap are sorted based on their importance score, with the topmost object being referred to as the top-node. When the cache is full, the least important model at the top of the heap is evicted to make room for the incoming model. By keeping the most important models in the GPU memory cache, mCache ensures a good cache hit ratio. Fig. 4 illustrates the single-node logical architecture of mCache along with its components.

4.3. *Caching algorithm*

4.3.1. *Problem definition*

We next describe the definition of model caching problem, with an emphasis on determining which in-memory model(s) to evict to serve incoming inference requests. Our objective is to minimize cache miss penalty — the performance overhead incurred when processing an inference request that requires loading a non-resident model into memory.

At a given time, assume there is a need for a set of models I to be hosted in a serving system. Let C and E denote the set of cached models in GPU memory and the set of models awaiting eviction, respectively. Let S denotes the model size in memory. For example, S_i represents the memory usage of model $i \in I$, S_c represents the memory usage of model set C. Let T be the total capacity size of the cache (i.e., GPU memory). Only a proper subset $C \subset I$ of models can be stored in the cache simultaneously. For each incoming invocation request, the objective is to form an eviction set E such that $T - (S_C - S_E) \geqslant S_i$, i.e., to evict enough models to accommodate a new incoming model i. Note that the eviction set E is a subset of C, and E may consist of either no models or multiple models.

To identify candidates for eviction set E, a specialized metric is needed to characterize in-memory model items. Unlike traditional caching algorithms (e.g., LRU and LFU), where eviction candidates are selected based on recency or frequency pattern, in model caching, additional factors such as model size must be considered. We therefore propose model utility, a novel evaluation metric designed to assess and prioritize models for optimal cache allocation.

4.3.2. Model utility

Model caching requires targeted objects to be preloaded in the memory before execution, which means that the impact of cache misses on inference performance cannot be ignored. To this end, model utility is designed to take into account both cache access patterns and cache miss penalty. Its definition can be described as follows:

$$Utility(i) = \frac{P_i}{S_i} \cdot \frac{1}{T_i} \tag{1}$$

where P_i denotes the cache miss penalty of model i, which is equivalent to the cold start-related latency overhead. S_i denotes the memory usage of model i, and T_i denotes the time to the next incoming invocation request for model i. Here, P_i and Si could be obtained using offline performance testing and profiling. For each model i, we predict its next request arrival time T_i using its invocation history. Specifically, we estimate the request rate Δi via a sliding-window or exponentially weighted moving average, ensuring the calculation reflects recent traffic trends. Similar modeling approaches have been adopted in prior works (Sriraman and Wenisch, 2018; Zhang et al., 2020). The serving interval is then approximated as $T_i \approx T' = \frac{1}{\Delta i}$. This approximation is reasonable, as user requests typically occur independently and unpredictably, particularly when initiated by a large number of users operating asynchronously. Such behavior results in random request arrivals over time, with inter-arrival times that inherently follow an exponential distribution. These are defining characteristics of a Poisson process, which is commonly adopted as a workload model in related studies (Strati et al., 2024; Zhong et al., 2024; Chen et al., 2025).

The design of model utility is based on a key insight: when a model is evicted from the cache, it will be reloaded in the future. Essentially, the utility of a model is conceptualized as the opportunity cost occurred as if it were not evicted by choosing to remove another in-memory models with a lower utility. By accounting for both the amortized cache miss penalty and the potential cache residency duration, we develop a standardized metric for cross-model comparison. As the composition of the utility definition shows, it balances the need to keep a high-penalty but small models in the cache through Pi/Si, and reduces the number of reloading models simultaneously through T_i. The longer it takes for a model to be re-requested, the more time we can spread out the penalty of a cache miss, resulting in a lower utility. This intuitive relationship arises from the constant penalty and size, which are inherent to hardware configuration and model parameters. As a result, the $utility(i)$ decreases monotonically with the increasing time until the next request for model i.

4.3.3. Rank-based importance score

While per-model utility provides a straightforward metric for quantifying individual model importance, it fails to capture each model's contribution to overall system throughput. The relative rankings enable mCache to prioritize the highest-importance models currently in memory. To quantify each model's contribution more precisely, we develop a log-based ranking method, as formalized in Eq. (2):

$$rank_i = log(\sum_{j=1,j\neq i}^{C} F(u_i > u_j) + b_0) \tag{2}$$

The rank-based importance score for the ith model in the GPU memory cache containing C models is denoted by $rank_i$. u_i and u_j denote the utility for the ith and jth model, respectively. The term b_0 serves as a bias to adjust the range of ranks on a logarithmic scale. F is an indicator function that returns a value of 1 when the condition $u_i > u_j$ is true, and 0 otherwise. For each j item in cache, this condition assists in placing each model in the proper rank. Models with a smaller utility are assigned a lower rank and are considered more suitable candidates for eviction.

A challenge in applying importance scores lies in their dynamic nature — these scores continuously evolve with the inference workload, resulting in fluctuating cache hit ratios. To tackle this challenge, mCache implements periodic updates of the importance scores. To avoid the overhead of constructing a heap-based queue, we opt not to update the importance queue (IQ) in place. Instead, a shadow cache with the same structure as the IQ is maintained. After importance updates, the IQ becomes read-only and is utilized only for eviction, while changes are recorded in the shadow cache. Once the shadow cache is fully rebuilt, it becomes a new IQ and the original one is released. This approach allows for asynchronous update of importance scores in the heap-based queue and does not affect the model swapping process.

4.3.4. Importance-aware model caching

When the requested model items are not present in cache, the cache manager needs to fetch them from persistent storage. Furthermore, it must decide whether to evict the model item when the memory cache reaches its capacity. One challenge is that commonly used LRU-like cache replacement algorithms do not work effectively in model management for DL inference serving, as they fail to consider model priority. Hence, there is a need for a cache replacement algorithm that takes model importance into consideration, thereby improving the cache hit ratio during model serving.

The high-level algorithm for the Importance-Aware Model (IAM) cache management is outlined in Algorithm 1. Specifically, in case of a cache miss and the cache is not full, the model item read from persistent storage is directly inserted into cache. IAM then creates a corresponding heap object for the model item and adds it to the heap. When insufficient cache space is available for an incoming model, IAM iteratively evicts the model item at the top of the heap-based importance queue (IQ) that possesses the smallest importance score. This eviction process continues until enough space is freed up within the cache. Following this, IAM then generates a new heap object corresponding to the incoming model and inserts it into the heap.

Algorithm 1 Importance-Aware Model (IAM) Caching Algorithm

```
Require: $IQ$: Importance queue for currently-cached models.
 1: while request model $i$ for serving do
 2:     // Calculate rank-based importance score based on Equation (1) and Equation (2)
 3:     $r = rank(i)$
 4:     if $cache\_hit$ then
 5:         $cache.get(i)$
 6:     else if $cache\_miss$ and $cache\_not\_full$ then
 7:         // Insert this model item into the cache and IQ
 8:         $cache.insert(i)$
 9:         $IQ.set(r, i)$
10:     else
11:         while cache_free_size < size(i) do
12:             // Find the model item with the minimal score in the cache
13:             $min\_rank, min\_item = IQ.min()$
14:             // Evict the least important model item from cache
15:             $cache.evict(IQ.pop(min\_item))$
16:         end while
17:         $cache.insert(i)$
18:         $IQ.set(r, i)$
19:     end if
20: end while
```

Theoretically, the importance-aware model caching algorithm is expected to demonstrate superior performance in comparison to traditional LRU-like algorithms exploiting temporal locality. Let us take Fig. 5 as an example, with a cache capacity of three and three consecutively cached model items (#1, #2, #3), when item #4 is accessed, the LRU-like replacement algorithm will evict item #1 since it is the least recently used, as shown in Fig. 5(a). However, assuming item #2 has the lowest importance score and is the top-node in the heap, there is a higher likelihood of accessing #1 than #2 in future references.

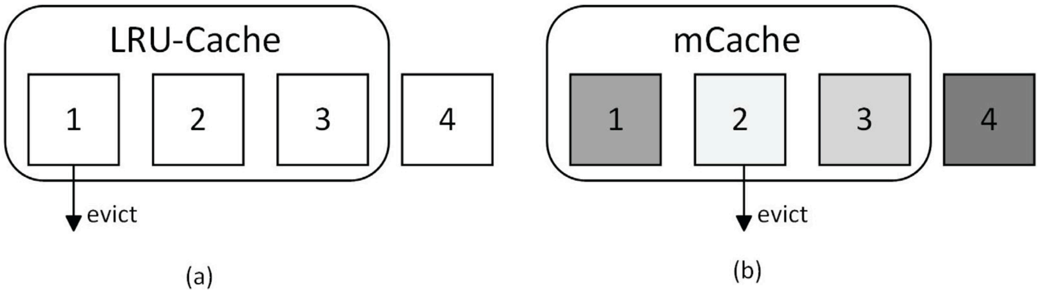

Fig. 5. The comparison of LRU and mCache. The squares in darker color represent models with higher importance score.

Therefore, evicting model #2 would lead to a higher cache hit ratio, as shown in Fig. 5(b).

5. Implementation

mCache is developed upon TensorFlow Serving and comprises around 2500 lines of Go code. It intercepts both HTTP as well as gRPC inference calls from client applications, requiring no modifications to the underlying TensorFlow Serving framework (Olston et al., 2017). When a model is requested, mCache proceeds to identify a TensorFlow Serving service that will serve the model. If the model is loaded and ready, the request will be forwarded directly to the TensorFlow Serving for execution. Otherwise, mCache will fetch it from the persistent storage and load it into TensorFlow Serving while unloading models based on the proposed IAM caching algorithm, before it forwards the request to TensorFlow Serving.

mCache implements the cache manager as a wrapper that encapsulates the IAM caching algorithm. Cache manager uses a key–value store to manage models in cache. The key denotes model ID and the value stores the data item of a model. In addition to supporting standard one-sided operations such as lookup, get, and put, the cache manager also provides an interface for periodically updating the model's importance. mCache is designed for easy deployment—it runs as a Docker container and can be managed efficiently using container orchestration systems like Kubernetes (Luksa, 2017).

6. Evaluation

This section evaluates our mCache system. We first describe our experiment setup, then analyze the performance of mCache and compare its IAM caching algorithm with other caching algorithms.

6.1. Experiment setup

Testbed. Our testbed is deployed on an inference server with two Intel Xeon Silver 4210R × 10 Core Processors, 64 GB DRAM, one NVIDIA V100 GPU (16 GB memory). The GPU is powered by Nvidia driver 470.103, CUDA 11.4, and cuDNN 8. NVIDIA Docker is leveraged to provide containers. We serve the DL models using containerized Tensorflow Serving (Olston et al., 2017) on the server and place the model files in the remote storage provided by an HDD-based NFS server.

Inference models. We conduct our experiments on several representative computer vision (CV) and natural language processing (NLP) DL models, namely, ResNet50 (He et al., 2016), SSD (Liu et al., 2016), BERT (Devlin et al., 2018), RoBERTa (Liu et al., 2021), XLNet (Yang et al., 2019), GPT-2 (Radford et al., 2019), T5 (Raffel et al., 2020). Requests for CV model services use a picture that is 600 × 600 pixels in size, and NLP model services use a short query of 20 words. These models, which vary in size from hundreds of MB to several GB, are wildly utilized across multiple tasks, including image classification, object detection, translation, text generation and question answering. All these models are obtained from Hugging Face Model Hub (Anon, 2025f).

Inference workload. There does not exist an open-source production DL inference trace to the best of our knowledge. Therefore, following the practice of prior studies (Lv et al., 2025; Strati et al., 2024; Zhao et al., 2023; Li et al., 2023; Zhang et al., 2023), we derive the inference workload from the Azure function trace (Shahrad et al., 2020). The trace consists of service statistics and invocation counts for each minute of the 24 h period. We present a three-step methodology for generating a request-level workload from the trace as follows: (1) Generate a list of service invocation events to indicate when the service is invoked. We count the number of invocations to each service in the trace and distribute them in a random order across the specified time (e.g., 10 min) based on their invocation frequency. In order to match the characteristics of Azure traces with DL workloads, we only count services triggered by HTTP requests since they are likely to be invoked by end-users. In addition, we remove services whose popularity quantile is higher than the 90th. We do this because trendy models typically remain in GPU memory and benefit less from model caching optimization. (2) Associate inference models with service invocation events. We pair models with service invocations to introduce the correlation between the two. The existing characteristics of the service invocations in the trace, such as allocated memory and execution duration, are inaccurate to match DL models. Thus, we choose one of the 7 DL models for each service in four ways: First, Random. This approach randomly selects one of the available models as the inference model for the service. Second, Round-robin. This approach first sorts the models in order of decreasing loading time and then selects the model in a round-robin manner. Third, Quantile. This approach selects the model based on the popularity quantile of the service and hence introduces a positive correlation between the popularity of a service and the loading time penalty; that is, models with high loading time are selected for services with high popularity. Fourth, Quantile-r. Contrary to the quantile approach, this approach reverses the correlation; that is, models with low loading time are selected for services with high popularity. (3) Down sample the service invocation events to an appropriate level. One hour of requests is extracted from the Azure trace, which is down sampled by 15%, including around 2389 requests. These requests are distributed across 7 models with 4 different correlations, allowing us to test a wide range of workload variations.

6.2. Analysis of mCache's performance

6.2.1. Average inference latency

We first present the average inference latency for the seven models serving with different workload characteristics. In this context, the inference latency is measured from the moment a request is sent to the server until the corresponding response is received. We have two observations from Fig. 6.

First, different workloads have significantly different results in average inference latency. As shown in the figure, the random and quantiles workloads consistently represent the uppermost and lowermost points

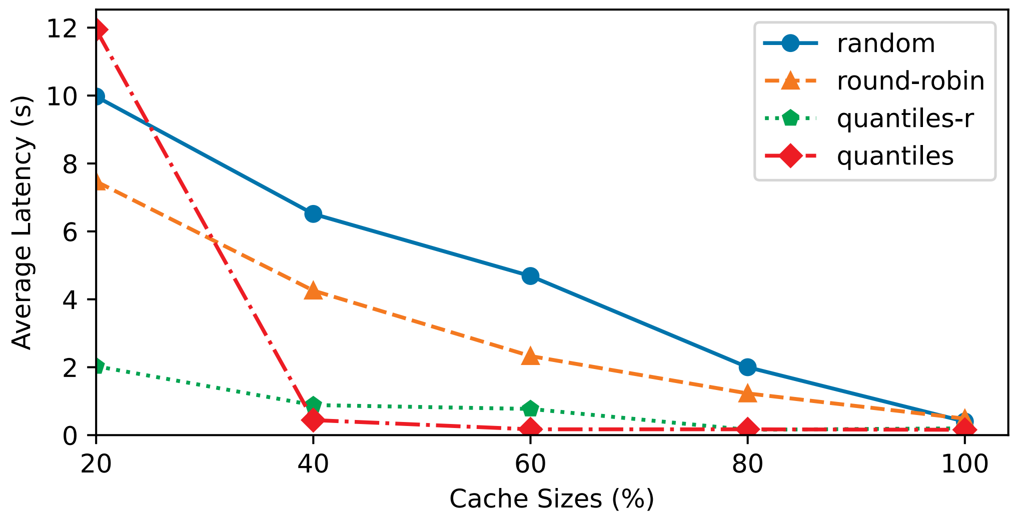

Fig. 6. Comparison of average inference latency under different workload characteristics. Percentages denote the proportion of GPU memory used as cache.

in most cases, respectively. Their difference comes from the correlation between model load penalty and popularity. For random workload, it is relatively more difficult to determine the best candidate for model eviction due to its inherent unpredictability, leading to a larger increase in inference latency as cache size decrease. Meanwhile, the quantiles workload consists of high-penalty and high-popularity models and hence contributes to reducing the cache miss ratio for frequently used high-penalty models. Note that the quantiles workload encounters exceptionally high latency at 20% of cache size. This is because the cache space is too small to accommodate multiple large-sized models. As a result, frequent model swapping occurs, leading to huge model load delays and prolonging average inference latency.

Second, different workloads exhibit distinct sensitivities to changes in the budgeted cache space. An encouraging finding is that our mCache reacts smoothly to the reduction in cache space, particularly for quantiles and quantiles-r workloads exhibiting correlation between model load penalties and access popularity. Specifically, nearly half memory savings are attained in these workloads with only modest average inference latency increase. However, for other workloads, mCache responds gracefully only to the initial reduction of cache space. For cache sizes lower than 60%, the workloads have a stronger influence on latency performance, leading to significant performance degradation.

6.2.2. *Inference latency breakdown*

In our next experiment, we break down overall inference latency into two components: model loading time and the other (model execution time and client–server communication latency). We then present the proportion of the dominating component — model loading time.

Fig. 7 shows how the percentages of the loading time changes over cache size for four different workloads. Similar to the finding in previous experimental study, we observe that the loading delay of models dominates overall inference latency in most cache sizes for each workload. As the cache sizes decrease, the proportion of loading time to the overall latency increases across all workloads. Specially, in the quantiles workload, the loading time percentage experiences a significant surge, increasing from 8% to 98%, as the cache size decreases from 80% to 20%. This phenomenon corresponds to the sharp incline observed in the quantiles workload's curve in Fig. 6. The rise in loading time in percentage indicates the considerable challenge of accurately caching high-penalty targeted models within a smaller cache space. For a cache size of 100%, note that we do not report model loading time, since all models can be simultaneously loaded into the GPU memory.

6.2.3. *Loading time CDF*

When a cache miss occurs, the model load delay is introduced, prolonging the inference latency. Thus, we further evaluate the cumulative distribution function (CDF) of model loading time, which helps demonstrate the extent of inference latency degradation due to model caching. Fig. 8 illustrates the model loading time CDF for four different workloads. As shown in figure, for up to 90% of inference requests, the model loading time remains below 10 s and 3 s for both 40% and 80% cache sizes across all workloads, respectively. This result implies that adopting model caching techniques can be beneficial in reducing memory footprint when users are willing to trade off some inference latency performance to cut costs. Additionally, we notice that the CDF displays a long tail in both Fig. 8(a) and Fig. 8(b), indicating that certain inference requests encounter prolonged response times. The reason for the long tail in the CDF is attributed to newly targeted models with high penalties experiencing cache misses, resulting in significant model load delays. Thus, improving the cache hit ratio for these models presents a promising direction for future research. Based on the results above, we argue that model caching is indeed feasible in practical DL inference serving, especially when their cache miss penalty is moderate.

6.3. *Comparison with other algorithms*

Previously we demonstrated how well mCache can perform under four difference workload characteristics. We next compare the ability of mCache's IAM to manage cached model replacement in GPU memory against several generic cache replacement algorithms used as baselines. These include LRU, a temporal locality-aware algorithm employed in prior studies (Gujarati et al., 2020; Cox et al., 2020; Zhao et al., 2023; Dakkak et al., 2019), and LFU, a frequency locality-aware algorithm used in other works (Anon, 2025b). Additionally, we also implement two advanced hybrid locality-aware algorithms for further comparison, namely ARC (Megiddo and Modha, 2003) and SRRIP (Jaleel et al., 2010). For fair comparison, we evaluate the model cache hit ratio with random workload characteristic, since this workload does not favor any specific caching algorithm and is common in a real-world production environment. In summary, the compared baselines are as follows:

- Least Recently Used (LRU), which removes the least recently used item when the cache is full, assumes that items accessed recently are more likely to be used again in the near future.
- Least Frequently Used (LFU), which removes the least frequently used item when the cache is full, assumes that items with lower access frequencies are less likely to be accessed again.

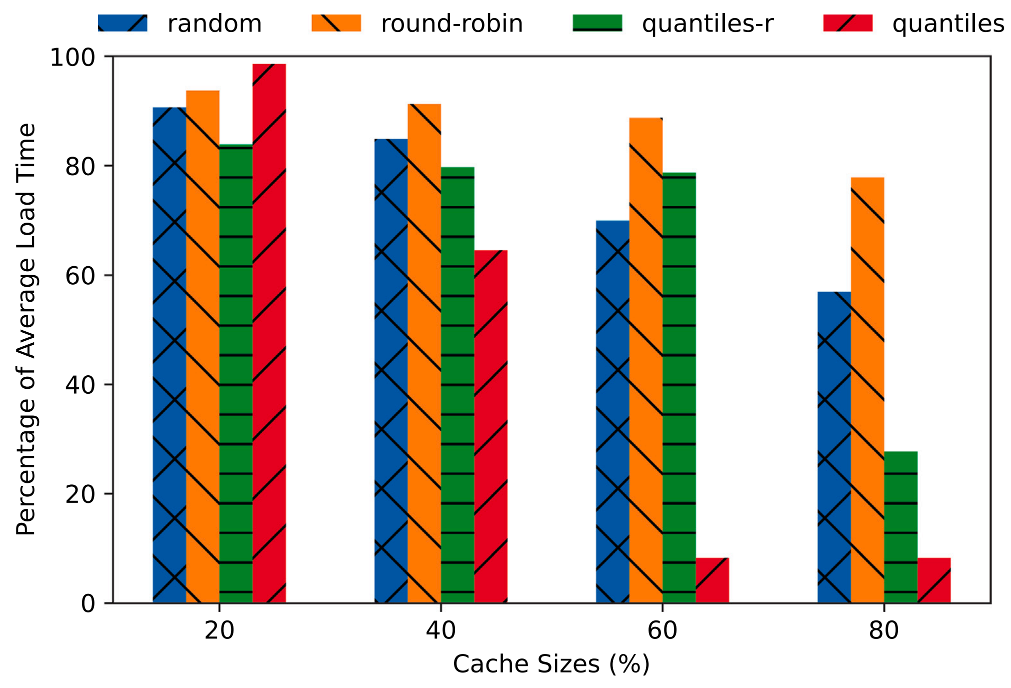

Fig. 7. The proportion of average loading time to the overall latency under different workload characteristics. Percentages denote the proportion of GPU memory used as cache.

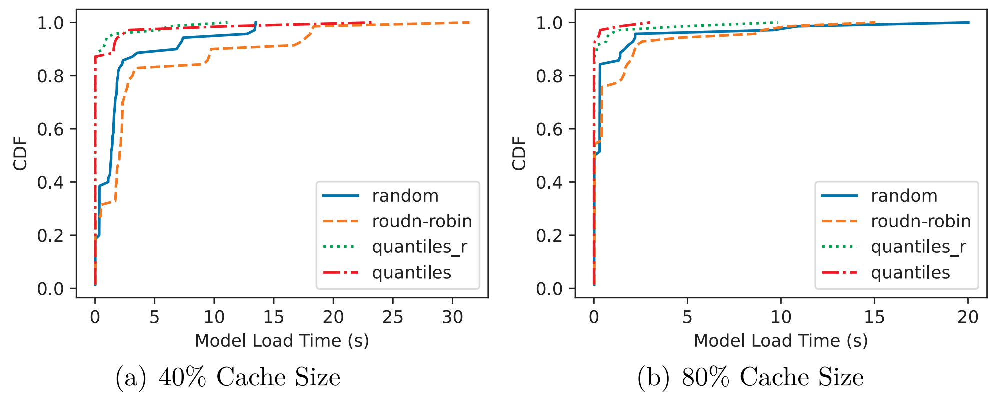

Fig. 8. Model loading time CDF at 40% and 80% of cache sizes under different workload characteristics.

- Adaptive Replacement Cache (ARC), which keeps track of both frequently used and recently used items plus a recent eviction history for both, aims to strike a balance between LRU and LFU algorithms and efficiently adapt to changing access patterns, resulting in improved cache hit rates and overall system performance.
- Static Re-Reference Interval Prediction (SRRIP), which provides good scan resistance while also allowing older cache items that have not been reused to be evicted. By assigning a Re-Reference Prediction Value (RRPV) to each cache item, SRRIP tracks the likelihood of reuse, incrementing RRPVs for cache misses and evicting the item with the highest RRPV value, ensuring efficient cache utilization and eviction of less-referenced item.

6.3.1. *Cache hit ratio*

Fig. 9 presents the average hit ratios while serving seven DL models at different cache sizes under a random workload characteristic. Notably, all caching algorithms exhibit a considerably low hit ratio at a cache size of 20%. This low hit ratio serves as an indicator of thrashing, wherein models are frequently being unloaded and loaded due to insufficient cache space to effectively accommodate the working set of models. However, as the capacity size of the cache increases, the hit ratio shows a gradual improvement. We further compare IAM with other caching algorithms at each cache size. We find that IAM indeed improves the cache hit ratio, e.g., by about a half compared to other caching algorithm at the cache size of 40%. The reason for the improved hit rate is that IAM has foreknowledge of the upcoming requests, and hence it can more accurately select which models to evict. However, as the cache size increases, the outperformance of IAM over the other caching algorithms shrinks because there is more cache space to accommodate newly targeted models. In conclusion, by ensuring a higher hit ratio with limited available cache space, mCache holds effectively more model in the limited cache space, therefore achieving higher GPU resource efficiency for DL inference serving.

6.3.2. *Throughput*

In our next experiment, we access the throughput of mCache under random workload, which serves to demonstrate the superiority of mCache in efficiently processing inference requests with a limited cache space. Fig. 10(a) shows that IAM has around 1.5 times better throughput compared to baseline LFU algorithm at 40% of cache size. In the experiment with 80% cache size, as shown in Fig. 10(b), IAM demonstrates a significantly higher throughput, approximately 2.39 times better than the baseline LFU algorithm. Similar observations

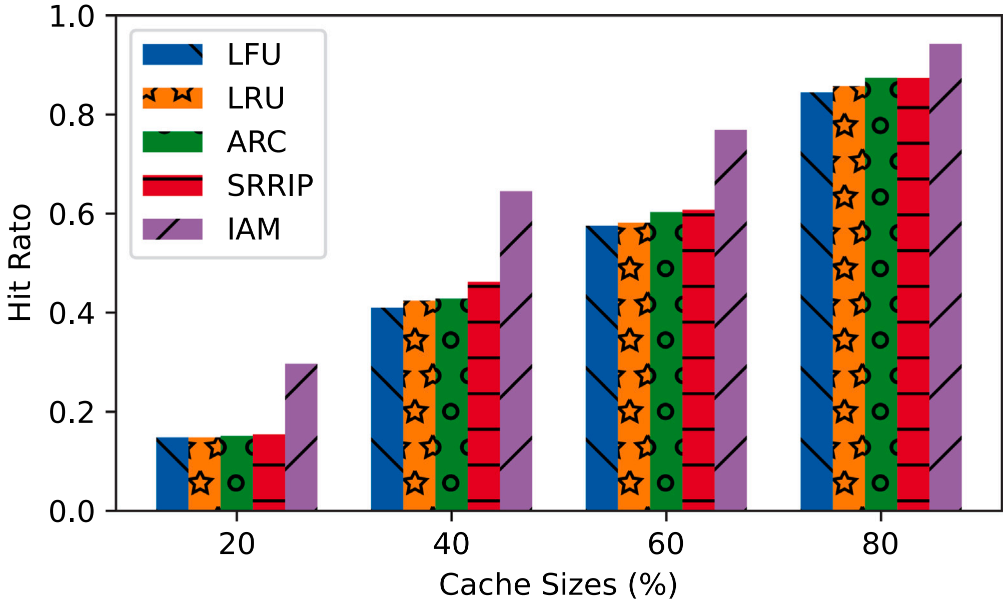

Fig. 9. Comparison of model cache hit ratio of various caching algorithms and cache sizes under random workload. IAM denotes the caching algorithm used in our proposed mCache system. Percentages denote the proportion of GPU memory used as cache.

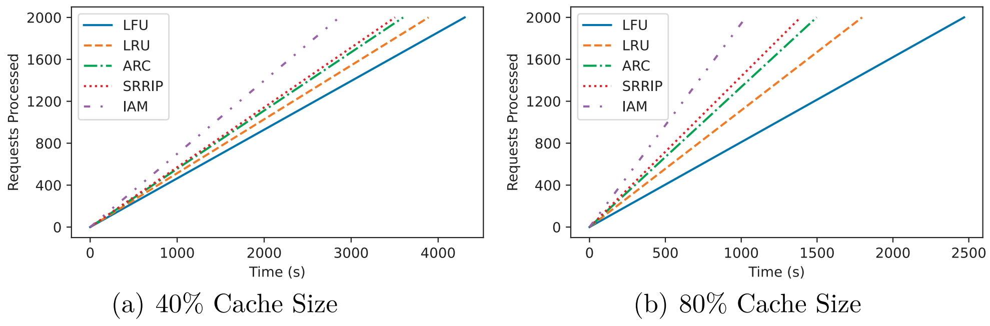

Fig. 10. Comparison of throughput of different caching algorithms at 40% and 80% of cache sizes using a random workload.

are made for other cache sizes, showcasing the consistent throughput improvement of IAM. Remarkably, IAM maintains its superior performance even when compared to the superior SRRIP algorithm. The improvement in the ability to process more requests is because the IAM algorithm is optimized for throughput by targeting the hosting of high-importance models. Consequently, IAM experiences reduced model swapping, especially for high-penalty models, which in turn frees up more time for model execution, enabling faster processing of requests compared to other algorithms.

6.3.3. *Model loading time*

A core goal of the caching algorithm is to minimize the delay of memory access, which is referred to as model loading time in this context. In this set of experiments, we evaluate the performance gain achieved in model loading time.

Table 3 presents the comparison of different caching algorithms in terms of their ability to reduce the average model loading time under random workload characteristic. As we can see in the table, IAM achieves the lowest mean model loading time, demonstrating 5.96, 4.21, and 1.68 s across cache space of 40%, 60%, and 80%, respectively, and is equivalent to 27%, 43% and 62% improvements against LFU baseline, respectively. This indicates that the larger the cache size, the greater the advantage of IAM over LFU in optimizing model caching and achieving faster loading time. We also note a big gap between LFU and other algorithms in terms of model caching gains in reducing loading time, while their cache hit ratio exhibits relatively minor difference, as shown in Fig. 8. This phenomenon is attributed to the LFU algorithm's tendency to evict model items with the lowest access frequency. Moreover, in scenarios characterized by random workload, there is typically a lower rate of access to high-penalty and large-sized models, rendering them less frequently used and thus more susceptible to eviction by the LFU algorithm. Consequently, when less frequently accessed high-penalty models are evicted, the process of reloading them incurs a non-negligible time delay, often ranging from several times to ten times longer than that of other models. In contrast, IAM is aware of the penalty of reloading models, prioritizing the retention of high-penalty models in memory. As a result, IAM outperforms LFU and other caching algorithms primarily due to its awareness of model penalty and size, leading to more effective caching decisions.

7. Discussion

Heterogeneity of GPUs. Our solution inherently supports the use of heterogeneous GPUs for model caching. It only requires running the same profiling procedure for each unique GPU type, and applying the profiled parameters P_i and S_i in the proposed caching algorithm.

Usability. Our solution is built upon the commonly used deep learning serving frameworks, and users do not need to make any changes to their code. To enable automatic model loading and unloading, our solution introduces only minor, user-transparent modifications to TensorFlow Serving's model configuration file. Additionally, our proposed

Table 3
Model loading time statistics with random workload.

Cache	Algorithm	Avg. (s)	Gain
40%	LFU	8.20	–
	LRU	6.93	15%
	ARC	6.36	22%
	SRRIP	6.32	23%
	IAM	**5.96**	**27%**
60%	LFU	7.35	–
	LRU	5.48	25%
	ARC	4.89	33%
	SRRIP	4.74	36%
	IAM	**4.21**	**43%**
80%	LFU	4.38	–
	LRU	2.57	41%
	ARC	2.23	49%
	SRRIP	2.01	54%
	IAM	**1.68**	**62%**

importance-aware caching technique for multi-model collocation inference and the Cache Manager implementation can be adopted by other serving frameworks (e.g., TorchServe (Anon, 2025k), Bentoml (Anon, 2025c)) to enhance their GPU-based inference performance.

Deployment Scenario Limitations. Multi-model collocation inference is ideal for hosting a large number of models that use the same ML framework on a shared serving container. When workloads involve mixed access patterns (i.e., frequent requests to popular models and sporadic requests to less popular ones), our model-cacheable inference solution can efficiently serve this traffic with fewer resources and higher cost savings, particularly when the models are fairly similar in size and invocation latency. However, DL applications should be tolerant of cold-start latency penalties when invoking infrequently used models. For applications with substantially higher transactions per second or latency requirements, dedicated serving containers remain the preferable option.

8. Conclusion

The GPU memory capacity is a major bottleneck in DL inference serving when multiple models collocate in a single GPU. To address this bottleneck, we present mCache, a novel caching system specifically designed for DL inference serving that aims to reduce memory footprint for collocation inference. mCache realizes an importance-aware caching algorithm that leverages importance scores to dynamically manage models in GPU memory, thereby enabling fundamental cacheability for inference serving systems. Experiments show that mCache reduces memory footprint with a modest increase in inference latency, and improves throughput by up to 1.5× and 2.39× given the 40% and 80% GPU memory capacity compared to similar serving system using LFU caching algorithm. As MaaS become more and more popular, we hope mCache will inspire the next generation of memory cache system designed for DL inference serving.

CRediT authorship contribution statement

Hao Mo: Writing – review & editing, Writing – original draft, Software, Data curation, Conceptualization. **Didier El Baz:** Writing – review & editing, Supervision. **Ligu Zhu:** Writing – review & editing, Supervision, Conceptualization. **Suping Wang:** Software, Data curation. **Songfu Tan:** Visualization, Software. **Hongning Zhao:** Visualization, Data curation. **Lei Shi:** Validation, Supervision, Resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is supported by the Fundamental Research Funds for the Central Universities (No. CUC25SG013) and the Foundation of Key Laboratory of Education Informatization for Nationalities (Yunnan Normal University), Ministry of Education (No. EIN2024C006).

Data availability

The authors do not have permission to share data.

References

Anon, 2025a. Amazon machine learning. https://aws.amazon.com/machine-learning.
Anon, 2025b. AWS SageMaker multi-model-endpoints. https://docs.aws.amazon.com/sagemaker/latest/dg/multi-model-endpoints.html.
Anon, 2025c. Bentoml documentation. https://docs.bentoml.com.
Anon, 2025d. Docker. https://www.docker.com.
Anon, 2025e. Google cloud prediction API documentation. https://cloud.google.com/ai-platform/prediction/docs.
Anon, 2025f. Huggingface model hub. https://huggingface.co/models.
Anon, 2025g. Keras, . https://keras.io/api/applications.
Anon, 2025h. NVIDIA multi process service (MPS). https://docs.nvidia.com/deploy/pdf/CUDA-Multi-Process-Service-Overview.pdf.
Anon, 2025i. Nvidia triton inference server. https://developer.nvidia.com/nvidia-triton-inference-server.
Anon, 2025j. Tensorflow serving documentation. https://www.tensorflow.org/tfx/guide/serving.
Anon, 2025k. Torchserve documentation. https://docs.pytorch.org/serve.
Berger, D.S., Sitaraman, R.K., Harchol-Balter, M., 2017. Adaptsize: Orchestrating the hot object memory cache in a content delivery network.. In: NSDI, vol. 17, pp. 483–498.
Chen, W., Lu, C., Xu, H., Ye, K., Xu, C., 2025. Multiplexing dynamic deep learning workloads with SLO-awareness in GPU clusters. In: Proceedings of the Twentieth European Conference on Computer Systems. pp. 589–604.
Choi, Y., Rhu, M., 2020. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In: 2020 IEEE International Symposium on High Performance Computer Architecture. HPCA, IEEE, pp. 220–233.
Cox, C., Sun, D., Tarn, E., Singh, A., Kelkar, R., Goodwin, D., 2020. Serverless inferencing on kubernetes. arXiv preprint arXiv:2007.07366.
Dakkak, A., Li, C., De Gonzalo, S.G., Xiong, J., Hwu, W.-m., 2019. Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service. In: 2019 IEEE 12th International Conference on Cloud Computing. CLOUD, IEEE, pp. 372–382.
Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
Dhakal, A., Kulkarni, S.G., Ramakrishnan, K., 2020. Gslice: controlled spatial sharing of GPUs for a scalable inference platform. In: Proceedings of the 11th ACM Symposium on Cloud Computing. pp. 492–506.
Ding, Y., Zhu, L., Jia, Z., Pekhimenko, G., Han, S., 2021. Ios: Inter-operator scheduler for CNN acceleration. Proc. Mach. Learn. Syst. 3, 167–180.
Fuerst, A., Sharma, P., 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 386–400.
Ghodrati, S., Ahn, B.H., Kim, J.K., Kinzer, S., Yatham, B.R., Alla, N., Sharma, H., Alian, M., Ebrahimi, E., Kim, N.S., et al., 2020. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO, IEEE, pp. 681–697.
Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., Mace, J., 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In: 14th USENIX Symposium on Operating Systems Design and Implementation. OSDI 20, pp. 443–462.
He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 770–778.
Ishakian, V., Muthusamy, V., Slominski, A., 2018. Serving deep learning models in a serverless platform. In: 2018 IEEE International Conference on Cloud Engineering. IC2E, IEEE, pp. 257–262.
Jaleel, A., Theobald, K.B., Steely Jr., S.C., Emer, J., 2010. High performance cache replacement using re-reference interval prediction (RRIP). ACM Sigarch Comput. Archit. News 38 (3), 60–71.
Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al., 2017. In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. pp. 1–12.

Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J.E., et al., 2023. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In: 17th USENIX Symposium on Operating Systems Design and Implementation. OSDI 23, pp. 663–679.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., Berg, A.C., 2016. Ssd: Single shot multibox detector. In: Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, the Netherlands, October 11–14, 2016, Proceedings, Part I 14. Springer, pp. 21–37.

Liu, Z., Lin, W., Shi, Y., Zhao, J., 2021. A robustly optimized BERT pre-training approach with post-training. In: China National Conference on Chinese Computational Linguistics. Springer, pp. 471–484.

Luksa, M., 2017. Kubernetes in Action. Manning Publications.

Lv, C., Shi, X., Lei, Z., Huang, J., Tan, W., Zheng, X., Zhao, X., 2025. Dilu: Enabling GPU resourcing-on-demand for serverless DL serving via introspective elasticity. In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. pp. 311–325.

Megiddo, N., Modha, D.S., 2003. ARC: A Self-Tuning, low overhead replacement cache. In: 2nd USENIX Conference on File and Storage Technologies. FAST 03.

Mendoza, D., Romero, F., Li, Q., Yadwadkar, N.J., Kozyrakis, C., 2021. Interference-aware scheduling for inference serving. In: Proceedings of the 1st Workshop on Machine Learning and Systems. pp. 80–88.

Mo, H., Zhu, L., Shi, L., Tan, S., Wang, S., 2023. HetSev: Exploiting heterogeneity-aware autoscaling and resource-efficient scheduling for cost-effective machine-learning model serving. Electronics 12 (1), 240.

Ogden, S.S., Gilman, G.R., Walls, R.J., Guo, T., 2021. Many models at the edge: Scaling deep inference via model-level caching. In: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems. ACSOS, IEEE, pp. 51–60.

Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., Soyke, J., 2017. Tensorflow-serving: Flexible, high-performance ml serving. arXiv preprint arXiv:1712.06139.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al., 2019. Language models are unsupervised multitask learners. OpenAI Blog 1 (8), 9.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J., 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. 21 (1), 5485–5551.

Reddi, V.J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., et al., 2020. Mlperf inference benchmark. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture. ISCA, IEEE, pp. 446–459.

Romero, F., Li, Q., Yadwadkar, N.J., Kozyrakis, C., 2021. Infaas: Automated model-less inference serving.. In: USENIX Annual Technical Conference. pp. 397–411.

Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., Bianchini, R., 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. arXiv preprint arXiv:2003.03423.

Soifer, J., Li, J., Li, M., Zhu, J., Li, Y., He, Y., Zheng, E., Oltean, A., Mosyak, M., Barnes, C., et al., 2019. Deep learning inference service at microsoft. In: 2019 USENIX Conference on Operational Machine Learning. OpML 19, pp. 15–17.

Sriraman, A., Wenisch, T.F., 2018. μTune: Auto-Tuned threading for OLDI microservices. In: 13th USENIX Symposium on Operating Systems Design and Implementation. OSDI 18, pp. 177–194.

Strati, F., Ma, X., Klimovic, A., 2024. Orion: Interference-aware, fine-grained GPU sharing for ml applications. In: Proceedings of the Nineteenth European Conference on Computer Systems. pp. 1075–1092.

Tan, C., Li, Z., Zhang, J., Cao, Y., Qi, S., Liu, Z., Zhu, Y., Guo, C., 2021. Serving DNN models with multi-instance GPU: A case of the reconfigurable machine scheduling problem. arXiv preprint arXiv:2109.11067.

Tang, X., Wang, P., Liu, Q., Wang, W., Han, J., 2019. Nanily: A qos-aware scheduling for DNN inference workload in clouds. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems. HPCC/SmartCity/DSS, IEEE, pp. 2395–2402.

Wang, L., Yang, L., Yu, Y., Wang, W., Li, B., Sun, X., He, J., Zhang, L., 2021. Morphling: fast, near-optimal auto-configuration for cloud-native model serving. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 639–653.

Wu, X., Xu, H., Wang, Y., 2020. Irina: Accelerating DNN inference with efficient online scheduling. In: 4th Asia-Pacific Workshop on Networking. pp. 36–43.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R.R., Le, Q.V., 2019. Xlnet: Generalized autoregressive pretraining for language understanding. Adv. Neural Inf. Process. Syst. 32.

Yu, F., Bray, S., Wang, D., Shangguan, L., Tang, X., Liu, C., Chen, X., 2021. Automated runtime-aware scheduling for multi-tenant DNN inference on GPU. In: 2021 IEEE/ACM International Conference on Computer Aided Design. ICCAD, IEEE, pp. 1–9.

Yu, F., Wang, D., Shangguan, L., Zhang, M., Liu, C., Chen, X., 2022. A survey of multi-tenant deep learning inference on GPU. arXiv preprint arXiv:2203.09040.

Zhang, J., Elnikety, S., Zarar, S., Gupta, A., Garg, S., 2020. Model-Switching: Dealing with fluctuating workloads in Machine-Learning-as-a-Service systems. In: 12th USENIX Workshop on Hot Topics in Cloud Computing. HotCloud 20.

Zhang, H., Tang, Y., Khandelwal, A., Stoica, I., 2023. SHEPHERD: Serving DNNs in the wild. In: 20th USENIX Symposium on Networked Systems Design and Implementation. NSDI 23, pp. 787–808.

Zhang, C., Yu, M., Wang, W., Yan, F., 2019. Mark: Exploiting cloud services for cost-effective slo-aware machine learning inference serving. In: 2019 USENIX Annual Technical Conference. USENIX ATC 19, pp. 1049–1062.

Zhao, M., Jha, K., Hong, S., 2023. GPU-enabled function-as-a-service for machine learning inference. In: 2023 IEEE International Parallel and Distributed Processing Symposium. IPDPS, IEEE, pp. 918–928.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., Zhang, H., 2024. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In: 18th USENIX Symposium on Operating Systems Design and Implementation. OSDI 24, pp. 193–210.

Zou, D., Jin, X., Yu, X., Zhang, H., Demmel, J., 2023. Computron: Serving distributed deep learning models with model parallel swapping. arXiv preprint arXiv:2306.13835.