Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation

Zicong Hong[¶] Jian Lin [†] Song Guo[ζ] Sifu Luo[‡]
Wuhui Chen[‡$] Roger Wattenhofer[§] Yue Yu[$]
[¶]Hong Kong Polytechnic University [†]Shantou University [ζ]The Hong Kong University of Science and Technology
[‡]Sun Yat-sen University [§]ETH Zurich [$]Peng Cheng Laboratory
zicong.hong@connect.polyu.hk,20jlin3@alumni.stu.edu.cn,songguo@cse.ust.hk,luosf@mail2.sysu.edu.cn,
chenwuh@mail.sysu.edu.cn,wattenhofer@ethz.ch,yuy@pcl.ac.cn

Abstract

Serverless ML inference is an emerging cloud computing paradigm for low-cost, easy-to-manage inference services. In serverless ML inference, each call is executed in a container; however, the cold start of containers results in long inference delays. Unfortunately, most existing works do not work well because they still need to load models into containers from scratch, which is the bottleneck based on our observations. Therefore, this paper proposes a low-latency serverless ML inference system called Optimus via a new container management scheme. Our key insight is that the loading of a new model can be significantly accelerated when using an existing model with a similar structure in a warm but idle container. We thus develop a novel idea of *inter-function model transformation* for serverless ML inference, which delves into models within containers at a finer granularity of operations, designs a set of in-container meta-operators for both CNN and transformer model transformation, and develops an efficient scheduling algorithm with linear complexity for a low-cost transformation strategy. Our evaluations on thousands of models show that Optimus reduces inference latency by 24.00% ∼ 47.56% in both simulated and real-world workloads compared to state-of-the-art work.

CCS Concepts: • **Computer systems organization** → **Cloud Computing**.

Keywords: Serverless computing, ML inference, cold start

[*]Zicong Hong and Jian Lin contributed equally to this research; Wuhui Chen is the corresponding author.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629567

ACM Reference Format:
Zicong Hong[¶] Jian Lin [†] Song Guo[ζ] Sifu Luo[‡] and Wuhui Chen[‡$] Roger Wattenhofer[§] Yue Yu[$]. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627703.3629567

1 Introduction

Serverless computing is a paradigm shift for cloud computing, offering scalability, flexibility, and cost-effectiveness [1, 5, 7, 34]. With the growing adoption of machine learning (ML) across industries, there is an increasing demand for faster and easier ways to deploy trained models and deliver ML services at scale. This new demand and the benefits of serverless computing are motivating data scientists to try deploying their trained models on the serverless computing platform for a new ML serving paradigm, namely *serverless ML inference* [3, 4, 18, 42, 43]. This approach has been adopted by major cloud providers, including Google Cloud Platform (GCP) [27], Amazon Web Services (AWS) [33], and Microsoft Azure [6]. In addition, according to a recent case study [41], serverless ML inference often outperforms traditional cloud services in terms of cost and performance.

In the workflow of serverless ML inference, the system receives inference requests from users and invokes the *functions* composed of trained models. Each function call must either create a new container (this is called *cold start*) or reuse an already running ("warm") container (*warm start*). As shown in Figure 1, a cold start might be substantially slower than a warm start due to the need to create a container and load application logic into memory [20]. Due to resource limitations, it is impossible to maintain enough running warm containers for each type of model, and a cold start is inevitable. The long cold start latency significantly hinders the promotion and application of serverless ML inference.

Mitigating the cold start overhead is a key challenge in serverless computing, but there are some existing techniques. Some works design prewarming or keep-alive strategies for containers of each function type based on resource and usage characteristics [12, 32, 35]. Another class maintains a

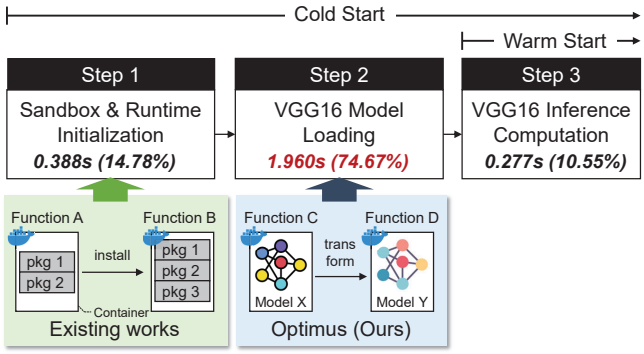


Figure 1. Timeline of serverless ML inference and the optimization direction of the existing works and ours.

shared pool of warm containers with some common packages [19, 20, 25]. It transforms a warm container in the shared pool into the destination container by importing extra packages. The latest work, Pagurus [20], allows every container to be one in the shared pool, which means a container for a function can be transformed into one for another function. For example, as shown in Figure 1, given a warm container with Package 1 and 2 for Function A, the container can additionally install Package 3 to execute Function B's request. The second approach [19, 20, 25] achieves high resource efficiency based on the idea of sharing. Our work builds on top of the second approach.

Unfortunately, we find that the existing works for serverless computing perform poorly when it comes to ML inference. We take the serverless ML inference for VGG16 as an example. As shown in Figure 1, more than 74% of the startup latency is caused by loading the ML model rather than the packages. (A larger-scale evaluation will be provided in § 3.1.) Therefore, the existing works that focus on managing packages gain little for serverless ML inference.

An intuitive solution is to develop a container management scheme specific to ML models. The scheme is expected to transform a model into another one in a container similar to that for a set of packages. However, this is challenging. As shown in Figure 1, in terms of traditional serverless computing, every package clearly describes its name and version. It is easy to compare the difference between package sets required by different functions, and there are many mature package managers to install or uninstall packages and resolve conflicts conveniently. Unlike the packages, each model is monolithic, meaning the existing technique can only delete the old model in a container and load a new one into the container to provide the inference service of the new model.

To solve this problem, we propose a new serverless ML inference system called Optimus with low cold-start overhead via a new container management scheme for ML models. It is based on several insights from the real-world workload of serverless ML inference. First, model structure loading is responsible for most of the model loading latency in serverless ML inference functions. Nonetheless, loading a model structure can reuse some structural state of the existing model in the container. For example, one model structure can be transformed into another by reshaping the weight matrix or channel number, reordering model operations, adding or removing operations, and so on, instead of loading the entire new model from scratch. This can significantly speed up the loading of a new model when using an existing model with a similar model structure already stored in a warm container. In addition, structurally similar models are common to existing ML applications, opening up opportunities for reusing this model structure between functions.

Therefore, we develop a novel idea of *inter-function model transformation*, which delves into the model required by each function at the granularity of model operations, and supports model transformation within a warm but idle container for other functions lacking warm containers. For example, as shown in Figure 1, given a container with Model X for Function C, Optimus can transform Model X into Model Y to execute Function D's requests at a low cost.

We summarize our contributions as follows.

- We observe the performance bottleneck and optimization opportunity of serverless ML inference, and we propose the idea of inter-function model transformation to mitigate the cold start problem.
- We design a set of in-container transformation meta-operators to disassemble models into the granularity of operations and transform the operations of CNN and transformer models in warm containers.
- We propose an efficient scheduling algorithm for a nearly optimal transformation strategy with a linear complexity of $O(N)$ where N is the total number of operations for the two models of each transformation.
- We implement a prototype system of Optimus. The experimental results show that Optimus reduces the serving latency by 24.00% ∼ 47.56% compared with the state-of-the-art serverless ML inference platforms.

2 Related Work

2.1 Serverless ML Inference

In serverless computing, applications are built using small, stateless functions that run on-demand without dedicated cloud server instances or infrastructure management. Cloud providers automatically scale resources based on demand and charge only for what is used, enabling faster time to market and greater agility for developers. This approach allows developers to focus on building their applications rather than managing servers. In addition to web and mobile, serverless computing has been widely adopted in data analytics [23, 26, 30] and high-performance computing [28, 29].

As the popularity of ML continues to grow, serverless ML inference is a relatively new approach to deploying and operating well-trained models in production environments [41]. Traditionally, ML models have been deployed on dedicated servers or cloud instances, requiring a significant upfront investment in infrastructure and ongoing maintenance. With serverless ML inference, ML models are deployed as serverless functions that can be executed on-demand, and data scientists can focus on building and improving their models.

In the following, we summarize the existing work on serverless ML inference. Zhang *et al.* [43] propose MArk, a hybrid ML inference system using AWS EC2 and a serverless system, where the serverless system handles arrival bursts. Ali *et al.* [3] propose BATCH, a serverless framework with adaptive batching for efficient ML serving. However, BATCH only uses a single buffer to collect requests and a serverless function to process batches, which results in high latency and cost penalties for the heterogeneous workload due to the high time and monetary cost caused by the batching overhead. To solve this problem, Ali *et al.* [4] propose MBS, a multi-buffer serverless framework for serving heterogeneous inference workloads. To improve the throughput of serverless ML inference, Yang *et al.* [42] propose INFless, which provides a unified, heterogeneous resource abstraction between CPU and accelerators and achieves high throughput using built-in batching and non-uniform scaling mechanisms.

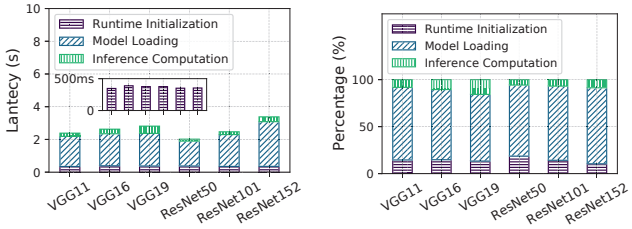
Most existing serverless ML systems focus on computational efficiency (i.e., the third step in Figure 1). Instead, our Optimus mitigates the cold start problem of serverless ML inference by improving the model loading efficiency (i.e., the second step in Figure 1). The existing work most related to us is Tetris [18]. In Tetris, if the models of two containers share an operation of the same type, size, and weight, the containers can share a copy of the operation in memory for memory efficiency. Although Tetris also speeds up model loading by transforming models that have identical operations, its performance is severely limited by the strict requirement of exactly the same operations, which will be evaluated in our experiment in § 8. In contrast, our Optimus excels in transforming heterogeneous models. Optimus, featuring its in-container meta-operators and linear complexity scheduling algorithm, proves superior in a more practical scenario with diverse models. Tens of thousands of models are proposed yearly (for example, there are 287,202 pre-trained models in HuggingFace), underscoring the need for our Optimus.

2.2 Cold start Mitigation for Serverless Computing

In serverless computing, functions run on instances, such as containers. When a function is invoked for the first time, or if a warm container is not available to run the function, the system must start a new container to encapsulate the function's runtime, initialize the software environment, load application-specific code, and execute the function. This process, known as a cold start [8, 16], can take several seconds [40], significantly increasing the latency of serverless functions. This overhead is particularly pronounced for short-running tasks (on the order of seconds), exacerbating the problem of long latency.

There are two classes of work that have attempted to mitigate the cold start problem. The first class focuses on pre-warming or keep-alive strategies for customized containers of each function based on resource and usage characteristics. For example, Shahrad *et al.* [35] characterize the serverless workloads on Azure function trace. They propose a practical resource management policy for keep-alive and pre-warming based on the observation. Inspired by the idea of caching, Fuerst *et al.* [12] propose FaasCache, a greedy dual keep-alive framework. Roy *et al.* [32] develop IceBreaker, which proposes a pre-warming and keep-alive strategy for serverless functions that fully exploits a heterogeneous mix of high-end and low-end servers. Du *et al.* [11] propose Catalyzer, which restores container images from checkpoints to accelerate the cold startup. For the second class, some works maintain a shared warm container pool consisting of containers with common packages for all types of functions. For example, Akkus *et al.* [2] propose SAND, which allows instances of the same application function to share the sandbox containing the function codes and their packages. Mohan *et al.* [22] propose PCPM, which launches a new function from a network-ready empty container for low startup latency. Oakes *et al.* [25] propose SOCK, which borrows the idea of Zygotes from Android systems for Java applications. The system identifies the set of packages most commonly used by functions based on the real-world dataset, caches some warm containers with pre-imported packages, and launches a new function from a warm container with low startup latency. Our work is complementary to the first class and can be combined with solutions of this type for further performance improvement.

Instead, we follow the second class. In the second class, the work most related to us is Pagurus [20], an efficient serverless computing system based on the idea of inter-function container sharing. Specifically, rather than booting a new container for a function from scratch, Pagurus alleviates the cold start by re-purposing a warm but idle container from another function. Repurposing is done by uninstalling the packages in the container and installing the additional packages for the new function. However, Pagurus only focuses on runtime initialization (i.e., the first step in Figure 1). This cannot solve the cold start problem of serverless ML inference since the bottleneck is the model loading instead of runtime initialization, as discussed in § 1. Our Optimus shares the same idea of inter-function container sharing but proposes a new container management system for serverless ML inference via model transformation.



(a) Request processing time **(b)** Every step proportion

Model	VGG11	VGG16	VGG19	ResNet50	ResNet101	ResNet152
Params	132.9M	138.4M	143.7M	25.6M	44.7M	60.4M
Size (MB)	506	528	549	98	171	232

(c) Number of parameters and size of varying models

Figure 2. Request processing time for varying models in serverless ML inference.

3 Background & Motivation

In this section, we analyze the serverless ML inference life cycle, find out its main bottleneck, and show the optimization opportunity of mitigating its cold start problem, which motivates the design of Optimus.

3.1 Serverless ML Inference Latency

A complete process of each serverless ML inference request includes three steps, i.e., sandbox and runtime initialization, model loading, and inference computation. Specifically, similar to serverless computing for other applications, the first step in serverless ML inference aims to create a sandbox containing both the runtime libraries and the wrapped program [11]. Then, within the sandbox, the wrapped program reads the serialized model file and deserializes the model file into a computational graph. Finally, the wrapped program executes the computational graph for the inference result of the request input. To investigate the performance bottleneck of serverless ML inference, we conduct a preliminary experiment for the two popular model families (VGG [36] and ResNet [14]) on our serverless computing testbed as described in § 8 and get the following insight.

Insight 1. *The model loading latency dominates the total latency of serverless ML inference functions.*

Figure 2 depicts the request processing time for VGG and ResNet, the percentage of different steps, and the number of parameters and model size of VGG and ResNet. First, according to the results, model loading accounts for more than half of the total time, indicating that model loading is the major bottleneck of serverless ML inference. Second, in the same model family, the model loading latency increases as the number of model layers increases. For example, the load latency of ResNet101 is slower than ResNet50 because ResNet101 has about twice the number of layers. Also, notice that the loading latency of the ResNet family is close to that of the VGG family, even though (according to Figure 2c) the number of parameters and model size of the ResNet family is far less than that of the VGG family. Therefore, the number of parameters and model size do not directly determine the model loading latency. This preliminary experiment motivates us to optimize model loading in a serverless ML inference system.

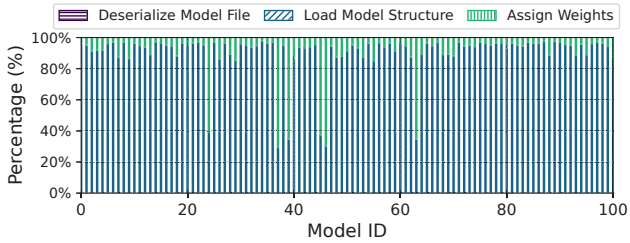


Figure 3. Latency of each step in the model loading for 100 models from Imgclsmob [37] in serverless ML inference.

3.2 Diving into Model Loading

We first dive into the process of model loading by taking TensorFlow as an example. The process can be divided into three parts: deserializing the model file, loading the model structure and assigning weights to the structure [38]. Specifically, the system first deserializes the serialized model file, such as TensorFlow SavedModel and HDF5. Next, according to the deserialization result, the system loads a model structure specifying the layers the model contains and how they are connected. Finally, the system loads a set of weights, i.e., the "state of the model", into the model structure. We evaluate the latency of these parts in the model loading for varying models on our testbed and get the following insight.

Insight 2. *The model structure loading latency dominates the model loading latency of serverless ML inference functions.*

Figure 3 depicts the percentages of different parts for model loading. We randomly choose 100 models from Imgclsmob [37], a popular model zoo that includes ResNet, VGG, DenseNet, MobileNet, etc., for various functionality (e.g., classification, segmentation, detection, and pose estimation). According to the results, the deserialization latency is negligible. The model structure loading takes up most of the model loading time (89.66% on average), while assigning weights only accounts for 10.28% of the whole loading time on average. Thus, loading the model structure is the bottleneck of the model loading, motivating us to minimize the overhead of model structure loading in serverless ML inference.

We also evaluate the model structure loading at a finer granularity. In a top-down view, each model comprises several *layers* and each layer comprises one or more *operations*, e.g., convolution (CONV), pooling and activation.

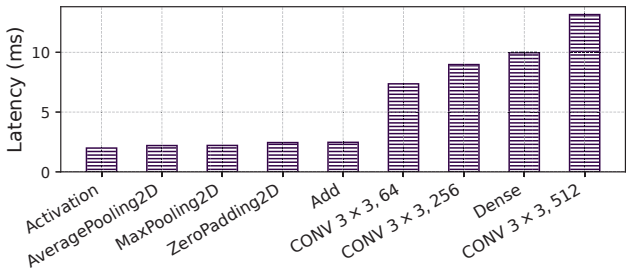


Figure 4. Loading latency for varying operations in ResNet50 in serverless ML inference.

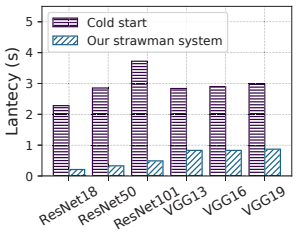
We evaluate the loading time of varying operations in ResNet50; the results are given in Figure 4. First, the loading time for different types of operations varies widely. For example, loading a CONV operation takes up to ten times longer than an activation operation. Second, the operations containing weights information (e.g., CONV, dense) have longer loading times than those without weights information (e.g., activation, pooling, and add). Third, the same type of operation needs different loading times if the operations have different shapes. For example, the loading of a CONV operation with a kernel size of 3×3 and an output size of 512 costs 78.67% more time than that of a CONV with a kernel size of 3×3 and an output size of 64.

Insight 3. *Structurally similar models are common for existing ML applications.*

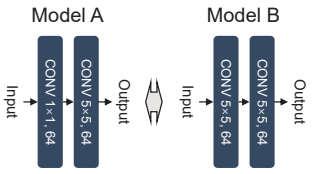
In parallel with the rapid adoption of ML and user complex and individualized needs, the number of models increases. However, structurally similar models are common in many model zoos [17], e.g., NASBench [10], Imgclsmob [37], and HuggingFace [15]. It is because most models rely on similar structure designs but with wider/deeper layers, branches, or different weights trained from different datasets. For example, CONV and attention operations are widely used in CV and NLP models, respectively.

3.3 A Strawman System: Optimization Opportunity of Model Loading

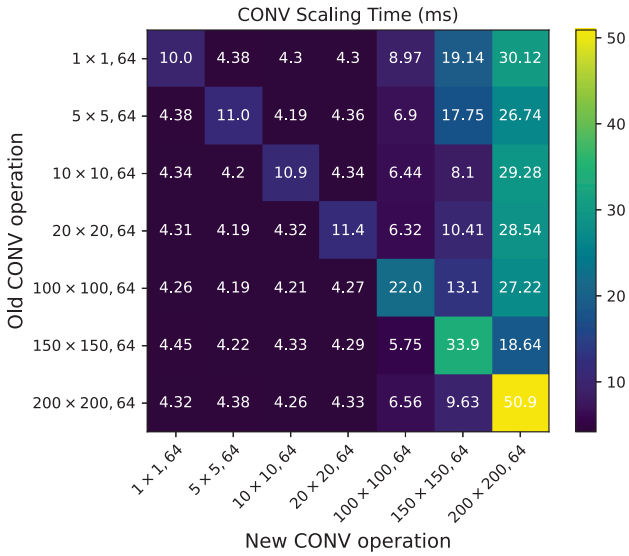
According to our three insights, we propose a strawman system to reduce the startup latency in serverless ML inference by minimizing the model structure loading. The strawman system is based on an idea of *inter-function container sharing*, which has been adopted by the existing serverless computing works [20]. In particular, the system aims to reduce the startup overhead by transforming the warm but idle containers of one function to help other functions that tend to experience cold container startup. However, the transformation of traditional serverless computing applications can be completed by uninstalling and installing packages. Thus, to extend the idea for a model transformation in serverless



(a) Transformation for varying models with the same structure.



(b) An example for the models with slightly different structure.



(c) In-container scaling of CONV operations with varying kernel sizes. The diagonal element indicates the loading time of different CONV operations, and the non-diagonal element (i, j) indicates the scaling time of the i-th CONV operation to the j-th CONV operation.

Figure 5. Evaluation results on the proposed strawman system. For a CONV operation represented by $x \times y, k$, $x \times y$ denotes kernel size and k is the number of kernels.

ML inference, our strawman system has two designs for the following cases.

Case 1: Same model structure. If the structure of the models for two functions is the same, the strawman system only replaces the old weights in the container with the new ones rather than starting a new container from scratch or loading the new model. We evaluate the case for VGG and ResNet and the results are given in Figure 5a. Despite being a simple and intuitive design, it reduces the latency of serverless ML inference by 79.83% on average.

Case 2: Different model structures. Given that two serverless ML inference functions have *slightly different* model structures. The "slightly different" means that they have the same number, type, and order of operations, but the kernel size of their CONV operations may be different, such

as the two models A and B in Figure 5b. To transform one function's container into another function's, the strawman system transforms the old model inside the container into a new model by adjusting the kernel size of the mismatching CONV operations in the computational graph. For example, in Figure 5b, to transform Model A to Model B, the strawman system can scale up the kernel size of the first CONV operation from 1×1 to 5×5. Such a transformation reduces the operation loading latency by 60.18%, as shown in Figure 5c.

We evaluate the transformation time between CONV operations with varying kernel sizes, and the results are shown in Figure 5c. First, as the filter size increases (i.e., the size of the weights matrix of the operation increases), the loading time of the CONV operation increases. Second, the value of the diagonal element is much larger than the other elements of the column. Take column 2 as an example, i.e. the CONV operations of different filter sizes are converted to the representation required for a filter size of 5×5. Directly loading the CONV operation with filter size 5×5 consumes 0.011s, while the transformation by CONV with other filter sizes takes only about 0.004s. That is, the transformation takes only one-third of the loading time. Therefore, the in-container scaling of CONV operations is better than loading the operations from scratch.

Challenges. Despite the performance improvement provided by the proposed strawman system, we find several challenges in extending our strawman system to a general case. First, the difference between model structures varies widely; thus, it is challenging to achieve transformation between a wide set of models (e.g., CNN and transformers). Second, there are many possible transformation strategies between two models in serverless ML inference; thus, it is challenging to determine an optimal strategy with the least transformation overhead.

4 Methodology & System Design

4.1 System Architecture Overview

To solve the above challenges, we propose a serverless ML inference system named Optimus and its architecture is highlighted in Figure 6. Similar to existing ML cloud platforms (e.g. AWS SageMaker and Google AI Platform), the cloud provider should take responsibility for resource management, so it is the system manager. Each client in the system can send query requests specifying the input data and model type via APIs. The arrived requests are dispatched to the corresponding containers. Different functions have different numbers of warm containers. The workload of every function may be highly dynamic and sporadic, periodic and bursty [35, 42]. Moreover, the number of warm containers is limited in each worker node to save resource consumption; thus, the system cannot provide enough warm containers for every model type. Therefore, for some requests involving the models which do not have idle containers, the traditional

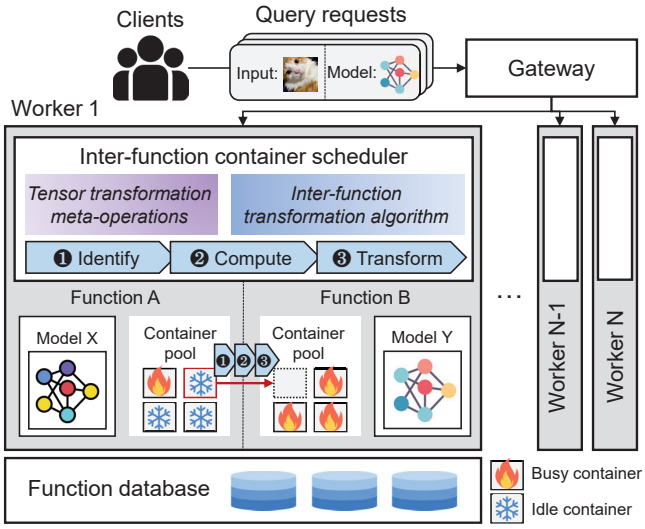


Figure 6. System architecture overview of Optimus.

serverless ML inference platforms need to start a new container from scratch. To mitigate the cold start overhead, a core component of Optimus is the *inter-function container scheduler*, which can efficiently transform the model of a warm container to another model. The scheduler is comprised of an idle container identification mechanism (see § 4.2 and Figure 6-❶), a set of in-container transformation meta-operators (see § 4.3 and Figure 6-❸), and a scheduling algorithm for inter-function model transformation (see § 4.4 and Figure 6-❷).

4.2 Idle Container Identification Mechanism

To identify the idle containers in each node, Optimus uses a timer design for each container, similar to Pagurus [20]. For a container, its timer is reset to 0 when a new request is routed to it. When its timer exceeds a predefined threshold (such as 60 seconds), the container is considered idle, and the model maintained in the container can be transformed into the models of other functions that lack warm containers.

4.3 In-container Transformation Meta-operators

To achieve a transformation between models in a container, we design five in-container transformation meta-operators. Each meta-operator acts on the *source model* and aims to transform it into a new one more similar to the *destination model*. Their functionality and examples are described below.

1) Replace: For the transformation that preserves the structure of an operation while replacing only its weights, we propose a meta-operator Replace, i.e. the new weights directly overwrite the existing weights in the container. For example, as shown in Figure 7-❶, a CONV operation with a kernel size 4×4 and the weights of a matrix of nines is replaced by the new weights of a matrix of ones.

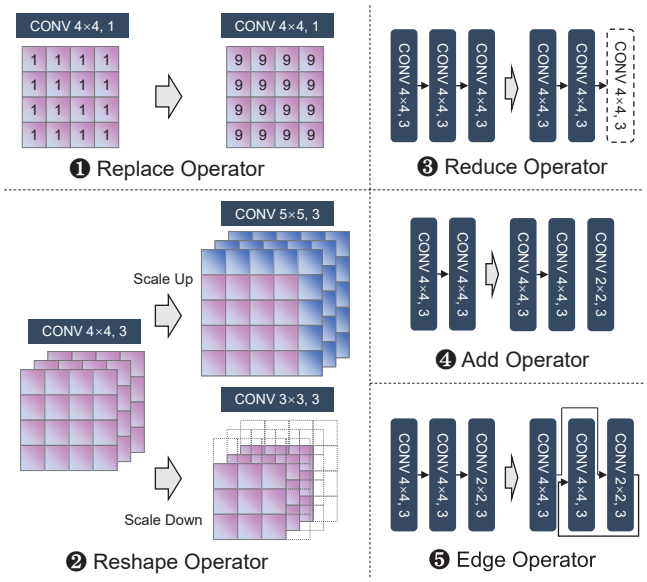


Figure 7. Illustration of in-container transformation meta-operators. For a CONV operation represented by $x \times y, k$, $x \times y$ denote kernel size, and k denotes the number of kernels.

2) Reshape: If two operations in the source model and destination model are of the same type but have different properties (e.g., kernel size, kernel number, and stride length for CONV), we propose a meta-operator Reshape that can modify its properties without regenerating a new operation. For example, as shown in Figure 7-❷, we can enlarge the kernel size of a CONV operation from 4×4 into 5×5; or reduce it from 4×4 to 3×3.

3) Reduce: For the operations in the source model that cannot be matched to any operations in the destination model, we propose a meta-operator Reduce to delete them without affecting the other operations. For example, as shown in Figure 7-❸, we delete the last CONV operation.

4) Add: If any operations of the source model cannot be transformed into the destination model via the meta-operators Replace and Reshape, we propose a meta-operator Add to add a new operation in the source model. For example, as shown in Figure 7-❹, we can create a CONV operation with kernel size 2×2 at the end layer of the model.

5) Edge: Each edge of a model denotes the data flow between two adjacent operations. To change/remove/add an edge between any two operations, we propose a meta-operator Edge. For example, as shown in Figure 7-❺, we can change the edges to switch the order of the operations in the model via the meta-operators. Besides, for the two models as shown in Figure 7-❸ and ❹, we need to remove and add the last edge of the model, respectively.

We emphasize that although we take CONV as an example in the above, the idea of in-container transformation meta-operators can be extended to other operations in CNN and transformer model [9], which we will discuss in § 5.2.

4.4 Scheduling Algorithm for Inter-function Model Transformation

Although executing meta-operators is faster than loading models from scratch, it inevitably introduces latency. To efficiently transform models between functions, we aim to find a sequence of meta-operators with low overhead. Thus, we first model each model structure as a graph in which model operations (e.g., CONV, dense) are nodes and data flows are directed edges, and the model transformation between functions can be considered as a graph transformation. We then formulate the model transformation as a new graph edit distance problem and minimize the graph edit overhead. The graph editing operators are the proposed meta-operators in § 4.3, and their cost is the execution time.

Towards the goal, Optimus includes three modules: *offline profiling for meta-operators*, *transformation planning*, and *online transformation execution*. We next describe details.

Module 1: Offline Profiling for Meta-operators. Before computing the planning solution, we first measure the execution time of different types of meta-operators ahead of time. We show the execution time of possible meta-operators for ResNet50 as an example in Figure 8, and we summarize several observations as follows. First, the execution time of meta-operators Replace depends on the size of the weights in the destination model operation. Second, the execution time of meta-operators Add depends on the type and properties of the destination model operation. For example, a meta-operator Add for CONV or dense incurs a longer latency than that for the other operations, which is also proved in Figure 4. A meta-operator Add for a CONV operation with a larger kernel needs a longer latency, as shown in Figure 5c and Figure 8, respectively. Third, the execution time of meta-operators Reshape depends on the magnitude of the destination operations' shape change. Forth, the execution time of meta-operators Reduce is constant and that of meta-operators Edge can be negligible.

Module 2: Planning for Model Transformation. Given two models g_1 and g_2, we define a sequence of meta-operators that can transform g_1 to g_2 as $\mathbf{e} = \{e_1, e_2, ...\}$. $e_i \in \mathbf{e}$ denotes one of the meta-operators described in § 4.3 and performs at step i. Its estimated cost $c(e_i)$ can be collected from the offline profiling. Thus, the cost of $\mathbf{e}$ is defined as $\sum_{e_i \in \mathbf{e}} c(e_i)$. We denote the set of all feasible meta-operator sequences from g_1 to g_2 as $\mathcal{E}(g_1, g_2)$. Note that the order of meta-operators in a sequence $\mathbf{e}$ does not change the cost. Our objective is to find an optimal meta-operator sequence $\mathbf{e}^*$ which satisfies

$$\mathbf{e}^* \in \arg \min_{\mathbf{e} \in \mathcal{E}(g_1, g_2)} \sum_{e_i \in \mathbf{e}} c(e_i). \quad (1)$$

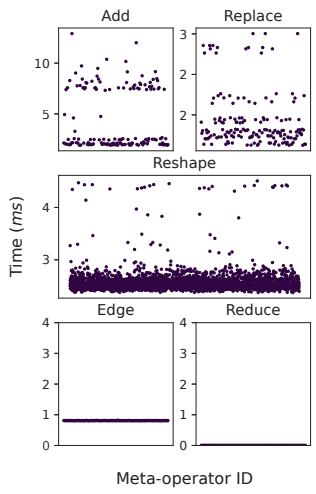


Figure 8. Execution time of varying meta-operators.

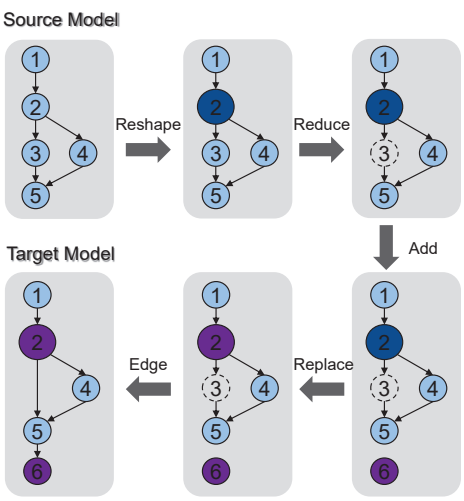


Figure 9. An example of inter-function model transformation.

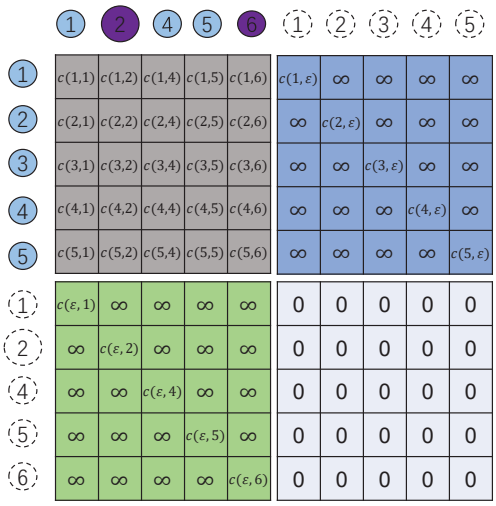


Figure 10. The transformation cost matrix of the example.

To solve the problem, we propose a basic algorithm for the inter-function model transformation as follows.

First, we define a cost matrix for the inter-function model transformation. Taking the source model and destination model as shown in Figure 9 as an example, its cost matrix is provided in Figure 10. The i-row, j-column element $c(i, j)$ in the left top corner denotes the cost of transforming Operation i in the source model to Operation j in the destination model via a meta-operator Replace or Reshape. $c(i, \epsilon)$ in the right top corner denotes the cost of deleting Operation i in the source model via a meta-operator Reduce; $c(\epsilon, j)$ in the left bottom corner denotes the cost of inserting a new Operation j into the source model via a meta-operator Add.

Then, based on the cost matrix, the optimal solution can be computed. Specifically, each operation of model g_1 either transforms to an operation of model g_2 (top left corner of cost matrix) or is deleted (top right corner of cost matrix). Conversely, each operation of model g_2 is either transformed from an operation of model g_1 (top left corner of cost matrix) or added from scratch (bottom left corner of cost matrix). Thus, a feasible solution is of the form $P = \{(1, p_1), (2, p_2), \cdots, (n+m, p_{(n+m)})\}$, where n is the number of operations in the source model, m is the number of operations in the destination model, and $p_1, p_2, \cdots, p_{n+m}$ is one of the permutations for $1, 2, \cdots, n + m$. The system can compute an optimal solution in a brute-force manner by enumerating all permutations and selecting the one that minimizes the cost with a time complexity of $O((n + m)!)$. Also, the optimal solution can be computed via Munkres algorithm with a time complexity of $O((n + m)^3)$ proposed by Riesen *et al.* [31].

Finally, if the algorithm outputs $\{(1, 1), (2, 2), (3, 8), (4, 3), (5, 4), (6, 6), (7, 7), (8, 9), (9, 10), (10, 5)\}$ for the example in Figure 9, we transform the model in the following steps. We first reshape Operation 2 in the source model to have the same shape as that in the destination model via a meta-operator Reshape. Then, we delete Operation 3 in the source model via a meta-operator Reduce and add Operation 6 via a meta-operator Add. Next, we reassign weights to Operation 2 and Operation 6 in the destination model because their weights differ from those in the source model. When all operations are ready in the destination model, we can use a meta-operator Edge to modify the data flows in the model.

Module 2⁺: Efficient Planning Algorithm. Although the basic algorithm above can find the optimal transformation solution, it introduces an unacceptable computation time for real-world models due to the time complexity of $O((n + m)!)$ or $O((n + m)^3)$. Thus, in the following, we introduce an efficient approximate algorithm based on some observations for real-world models.

We observe the following characteristics of model operations in our inter-function model transformation. First, only operations of the same type can be transformed at a low cost, while operations of different types either cannot be transformed in the implementation or always require a higher cost than loading them directly from scratch. Second, in most models, the operation form often increases from the first layer to the last layer, which can capture more complex and abstract features. For example, as one moves deeper into a VGG or ResNet model, the number of kernels in a CONV operation increases. Third, most operations in a model do not contain weights (e.g., there are 347 operations in ResNet101, of which only 101 operations have weights), and the cost of transformation between operations of the same type without weights can be considered a constant.

We then propose an efficient group-based transformation algorithm. (1) Based on the first observation, we first group all operations of the source model (or destination model)

by their type. For example, the activation operations and the CONV operations are assigned to two groups. (2) We match two operations sequentially, one by one, between two groups of the same operation type in the source and destination models. This heuristics design is motivated by the second and third observations above. Specifically, for the operations with weights, based on the second observation, the operation shape in a group shows a similar tendency. For the operations without weights, based on the third observation, the operation transformation can be performed arbitrarily. (3) The transformation between two matching operations is performed via a meta-operator Replace or Reshape. (4) After that, if there are unmatched operations in a group of the source model, we remove them via meta-operators Reduce. (5) If there are unmatched operations in a group of the destination model, we create them via meta-operators Add. (6) Finally, we can use meta-operators Edge to modify the data flows in the model.

The computational time complexity of the proposed algorithm is $O(n+m)$ because we can iterate over the operations in the source and destination models for two rounds, i.e., one for grouping and one for matching.

Module 3: Online model transformation execution. To achieve real-time transformation while the system is running, Optimus does the planning offline and executes the strategies online. The details are described below.

Planning strategy caching. When a new model registers in the global model repository of Optimus, the system measures the transformation overhead between the new model and the existing models in the repository and caches the transformation strategy based on the planning algorithm. When a request for model transformation arrives as described in § 4.2, the manager can read the cached transformation strategy, based on which the model can be transformed.

Safeguard design. According to the experiment in § 8.4, in some cases, the overhead of our inter-function model transformation is higher than that of loading models from scratch. Thus, in these cases, the system loads a new model from scratch like traditional works. In other words, the performance of Optimus can be guaranteed in the worst case.

5 Design Refinement

5.1 Model Sharing-aware Load Balancer

Similar to the existing serverless systems, the function distribution on each node has a great influence on the efficiency of Optimus. However, the inter-function model transformation of Optimus introduces a new requirement for scheduling functions to the nodes. For example, if the model structures of functions on a node vary widely, the inter-function model transformation within the node will be expensive. At the same time, if the demand dynamics of functions on a node are similar, few idle containers can assist the functions suffering from a cold start, and the transformation benefit will be limited. Thus, a new serverless load balancer considering the similarity of model structure and the complementarity of demand dynamics is needed in Optimus. Unfortunately, existing serverless systems often route user requests to nodes based on hash-based or resource usage-based methods [21], poorly suited for serverless ML inference.

To solve this problem, in Optimus, we develop a model sharing-aware load balancer that considers the similarity of model structure and the complementarity of demand dynamics between serverless ML inference functions. The proposed scheduler aims to deploy functions with similar model structures and different demand dynamics on the same node via K-medoids clustering. It takes functions as different points and measures the distance of any two functions according to their model editing distance in § 4.4 and demand dynamics difference. Specifically, given two functions for Model A and B, their model editing distance (transformation cost) $\mathcal{D}(A,B)$ can be calculated based on the planning algorithm in § 4.4. We calculate the complementarity of their demand dynamics $\mathcal{K}(A,B)$ based on the covariance of their historical record. According to their historical demand dynamics $\{l_t^A\}_{t\in T}$ and $\{l_t^B\}_{t\in T}$ for T time slots, we can get $\mathcal{K}(A,B) = corr(A,B) = \frac{\sum_{t=1}^{T}(l_t^A-\bar{l^A})(l_t^B-\bar{l^B})}{\sqrt{\sum_{t=1}^{T}(l_t^A-\bar{l^A})^2}\sqrt{\sum_{t=1}^{T}(l_t^B-\bar{l^B})^2}}$ in which $\bar{l^A}$ and $\bar{l^B}$ are the means of l_t^A and l_t^B, respectively. Next, the distance of the two functions is defined as $\gamma_i\mathcal{D}(A,B) + \gamma_j\mathcal{K}(A,B)$ in which the hyper-parameters $\gamma_i, \gamma_j \in [0,1]$. Finally, by taking the distance of any two functions (i.e., data points) as input, K-medoids can minimize the distance between data points and their cluster centre and output multiple function clusters. The load balancer tends to distribute the functions in the same cluster to the same node. Moreover, the load balancer should consider the load of nodes; thus, each function cluster is supported by a set of nodes, and the number of nodes will be adjusted along with the load change.

Additionally, there is much room for improvement of the proposed load balancer, such as hypothesis testing of the historical data, consideration of heterogeneous resource usage characteristics of models and functions, and design of online scheduling algorithms, which is left as future work.

5.2 Extension to Transformer Models

Transformer models are one of the most exciting new developments in ML. They are widely used in natural language processing [39] and computer vision [13]. Although our preliminary experiments in § 3 and the above design take CNN as an example, the idea of our inter-function model transformation can be easily applied to serverless ML inference of transformer models. In the following, we dive into transformer models and show how our idea is compatible.

In a top-down view, a transformer model consists of an embedding block and multiple (often identically parameterized) attention blocks. An attention block consists of several

layers: an attention layer, a normalization layer, and several (typically two) fully connected layers. The attention layer consists of Query (Q), Key (K), Value (V), and Output (O) operations with weights, and Logit (L) and Attend (A) operations without weights.

Despite the transformer models having their own unique operations, the in-container meta-operators proposed in § 4.3 can work on these new operations. Transformer models' transformation can be realized by adjusting the operation shape, removing redundant operations, or adding new operations similar to CNN transformation. We describe how the proposed meta-operators work with the new operations in transformer models with the following four common cases.

Case 1: Transformation between transformer models with embedding blocks of different sizes (e.g., BERT-Cased and BERT-Uncased) can be achieved by scaling up or down their weights matrix with meta-operator Reshape. Moreover, the transformation between transformer models with Q/K/V/O operations of different shapes can be achieved by scaling up or down their weights matrix with meta-operator Reshape. **Case 2:** Transformation between models with different numbers of attention blocks can be achieved by adding or removing attention blocks with meta-operators Add or Reduce. **Case 3:** The steps performed by meta-operators Replace and Edge on transformer models are similar to those of CNN. **Case 4:** The downstream task models (e.g., sequence classification and question answering) of the transformer usually add some specific layers (e.g., fully connected layers, conditional random field layers) on top of pre-trained models. The transformation between downstream task models can be achieved by transforming these specific layers like CNN.

We discuss two common examples of inter-function transformer transformations as follows. **Example 1: Transformation between BERT variants with different sizes.** There are kinds of BERT with different sizes. Take the transformation from BERT-Base to BERT-Mini as an example. BERT-Base has 12 attention blocks, 768 hidden units and 12 self-attention heads. BERT-Mini has 4 attention blocks, 256 hidden units and 4 self-attention heads. The transformation can be achieved by reshaping the Q/K/V/O operations in the reused attention blocks via meta-operators Reshape, removing redundant attention blocks via meta-operators Reduce, and using meta-operators Replace and Edge like CNN. **Example 2: Transformation between BERT variations for different downstream tasks.** Bert can be used as a base model for many downstream tasks. Take the transformation from BERT-SC for sequence classification to BERT-QA for question answering as an example. The transformation can be achieved by adding a fully connected layer via a meta-operator Add and updating the weights via a meta-operator Replace since BERT-SC and BERT-QA have one and two fully connected layers on top of BERT, respectively.

Additionally, Optimus supports CNN and transformer models so far, but there are massive new models yearly. However, we believe the idea of our in-function model transformation (i.e., the loading time of a new model can be significantly accelerated when using an existing model with similar model structures in the warm containers) can be extended to most of the new models, which is left as future work.

6 Limitations & Future Work

Fine-grained Resource Allocation. Optimus allocates the same and sufficient resources to each container. However, there are two limitations. First, such homogeneous resource allocation leads to wasted resources when considering the different model sizes in the practice. For example, if an idle container with 4GB of memory for a large model is converted to one for a small model, some memory resources are wasted because the small model may only require 2GB of memory. Second, container resources may be insufficient. Increasing the number of containers increases the benefits of resource sharing, if the containers have sufficient capacity to host different models, because the likelihood of idle containers increases. Excessive numbers of containers that exceed the capacity to host certain models will reduce these benefits.

Online Profiling. While Optimus provides offline profiling for meta-operators in § 4.4, the execution time of meta-operators may vary with the workload and resource allocation of containers; thus transformation plans generated based on outdated offline profiling may be inefficient. In future work, we will discuss online profiling, where the system periodically updates profile data while the system is running.

Privacy Issue. In Optimus, the provider keeps track of the status of containers (e.g., running models) as it should take responsibility for the resource management, and users need to provide metadata (e.g., models of functions) to the provider. However, there can be concerns about exposing key private model details to a cloud provider.

7 Implementation

We have implemented a prototype of Optimus with about 8K lines of Python code. Optimus API and communication between clients and the gateway are implemented in REST API format. Clients can invoke an inference procedure by sending an HTTP request containing the model name and input data. Optimus stores the trained models in a Docker volume that is attached directly to each container created by the system. Models are deployed to the Docker volume in HDF format. Model structure information and model-to-model transformation planing are stored with the models in JSON format. On the host machine, a gateway service runs as the container manager. It uses the Docker SDK for Python to create, run, and remove containers in the local Docker

environment. It also runs a Flask HTTP server that accepts client requests and sends them to containers.

```
from optimus import trans_plan, trans_exec
# Offline: transformation planning between models
strategy = trans_plan(source_info,
    destination_info)
# Online: transformation execution in containers
trans_exec(model_in_container, strategy)
# The execution calls meta-operator interfaces
def trans_exec(...):
  for s in strategy:
    ... # call add_oper(...) or reduce_oper(...)
    or replace_weights(...) or reshape_oper(...)
    or generate_edge(...)
```

Listing 1. Code snippet of Optimus APIs.

Each container runs a Docker image built from our modified Tensorflow. For five in-container transformation meta-operators in § 4.3, we implement the corresponding interfaces that take tf.keras.layer (e.g., tf.keras.layer.Conv2D) and operation-related data (e.g., the kernel size of CONV) as input and the destination layer as output. The interface design is compatible with ML operations in most models, including CNN, RNN, and transformer. For transformation planning, we take the source model and the transformation strategy as input, and the output is the destination model. The implementation logic is extensible to other similar ML frameworks (e.g. Pytorch). When deployed, it runs a scheduler service that accepts requests from the gateway service. The scheduler determines whether to load a new model or perform a transformation based on its content in the memory.

8 Experiment

8.1 Experimental Setup

Node setup. We evaluate Optimus using two servers: one is equipped with Intel Xeon Gold 5320 CPU with 104 cores and 128GB of memory, and the other is equipped with Intel Xeon E5-2650 v4 CPU with 48 cores, 64GB of memory and 4 NVIDIA GeForce GTX 1080 Ti. The machines are interconnected via a 10 Gbps, full-bisection bandwidth Ethernet. The servers are running Ubuntu 22.04.2 and Docker 23.0.1.

Workloads. The serverless ML inference services to evaluate Optimus are based on the following widely-used models.

- Imgclsmob [37]: a model zoo including 389 models with various functionality (e.g., classification, segmentation, detection, and pose estimation). It includes ResNet, VGG, DenseNet, MobileNet, etc.
- BERT [9]: a kind of transformer model pre-trained on a large corpus comprising books and Wikipedia. We choose 10 variations with different model sizes (i.e., BERT-Tiny, BERT-Mini, and BERT-Small), those for different inputs (i.e., BERT-Cased and BERT-Uncased), and those for various tasks, e.g., sequence classification (BERT-SC), token classification (BERT-TC), question answering (BERT-QA), next sentence prediction (BERT-NSP), and multiple choice (BERT-MC).
- NASBench [10]: a model zoo with thousands of lightweight models generated and evaluated from a fixed graph-based search space for image classification.

Moreover, the function invocation arrival pattern of the workload is generated from the following data.

- Poisson distribution: We send queries to each serverless inference service, following a Poisson distribution. To simulate frequent, middle, and infrequent workloads, λ is set as $10^{-3.5}$, 10^{-2}, and $10^{-2.5}$, respectively.
- Azure Function: To simulate production-like workload arrival patterns and characteristics, we use a two-week Microsoft Azure Function trace [44] collected from Microsoft production systems in 2021.

Comparison systems. We compare Optimus with the following serverless ML inference systems. For a fair comparison, all systems adopt a 10-minute keep-alive strategy.

- OpenWhisk: A serverless ML inference system starts a new container from scratch.
- Pagurus [20]: A serverless ML inference system starts a new container based on an idle container with common packages for ML inference.
- Tetris [18]: A serverless ML inference system starts a new container by mapping the address of runtime and operations with the same weights and structures in the running containers to the new container.

8.2 Inter-function Model Transformation

We evaluate the latency of inter-function model transformation. As a representative example, we pick ResNet, VGG, MobileNet, DenseNet, Xception, and Inception in Imgclsmob and all models in our BERT model zoo, and the results are given in Figure 11. We have the following observations.

First, compared with loading models from scratch (as shown in the last column) like the existing serverless ML inference systems, the inter-function model transformation reduces the latency by up to 99.08%. The transformation between models with more similar structures in terms of operation type and scale takes less time. In particular, the transformation between models in the same model family is commonly faster than that between models from different families. For example, as shown in Figure 11, for a model in the VGG family, transforming from models in the VGG family will be faster than from models in the other family.

Second, the latency of transformation between any two models is asymmetric. In most cases, transforming from a large model to a small one is faster than transforming from a small model to a large one. It is because the former often involves more meta-operators Reduce, meta-operators Add for small new operations, and meta-operators Reshape to

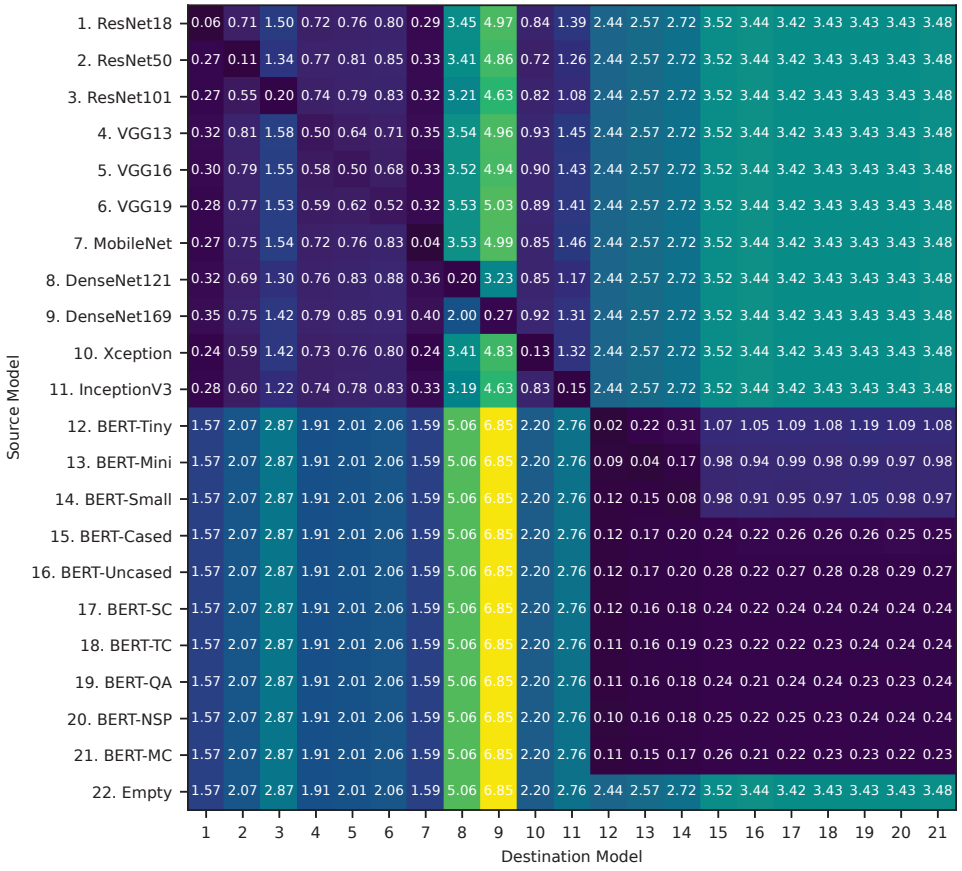
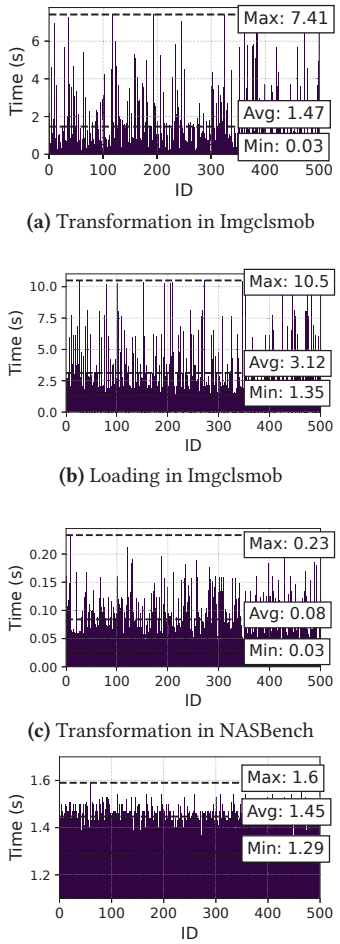


Figure 11. Inter-function model transformation latency (in seconds) between 21 representative models in Optimus. The element in the i-th row and i-th column denotes the transformation between models i with different weights. The element in the i-th row and j-th column denotes the transformation from model i to model j. The element in the 22-th row and j-th column denotes loading model j from scratch.



(a) Transformation in Imgclsmob

(b) Loading in Imgclsmob

(c) Transformation in NASBench

(d) Loading in NASBench

Figure 12. Large-scale evaluation on transformation latency between models in Optimus.

scale down the operations. They often take less time than other meta-operators, according to Figure 5c and Figure 8.

Third, the transformation between models with the same structure but different weights costs the least time. This is because it only involves meta-operators Replace, the execution of which is rapid according to Figure 8. Moreover, in some cases, the latency of the inter-function model transformation is the same as that of loading models from scratch due to the safeguard mechanism in § 4.4. Particularly, the transformation between a CNN model and a transformer model is always more costly than loading models from scratch; thus, the safeguard mechanism chooses the latter.

Fourthly, we conduct a large-scale evaluation of the latency of inter-function model transformation in Imgclsmob and NASBench. Because of the space limit, we randomly pick 500 transformation cases from Imgclsmob and NASBench, and their latency is shown in Figure 12a and Figure 12c, respectively. We also randomly pick 500 cases loading models from scratch in Imgclsmob and NASBench, and their latency is shown in Figure 12b and Figure 12d, respectively. According to the results, the inter-function model transformation can reduce the model loading latency by 52.88% and 94.48% in Imgclasmob and NASBench, respectively.

Besides, the CPU consumption of the model transformation is negligible as Optimus reads the cached transformation strategy directly (see § 4.4) without any online computation. The IO consumption of Optimus is similar to that of traditional serverless ML platforms. This is because the model parameters comprise a high percentage of the model size. Thus, parameters loading, which our Optimus does not optimize, still dominates the total IO consumption in Optimus.

8.3 Serverless ML Inference Latency

To evaluate the performance of Optimus for the workload described in § 8.1, we measure the latency of serverless ML inference requests in Optimus and the state-of-the-art works.

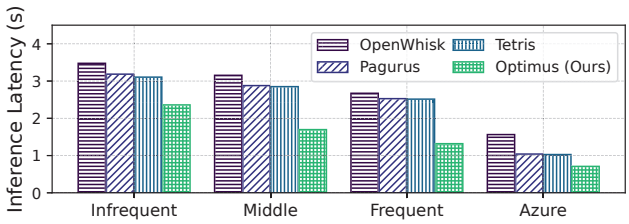


Figure 13. Average service time of serverless ML inference requests under the Poisson and Azure workloads.

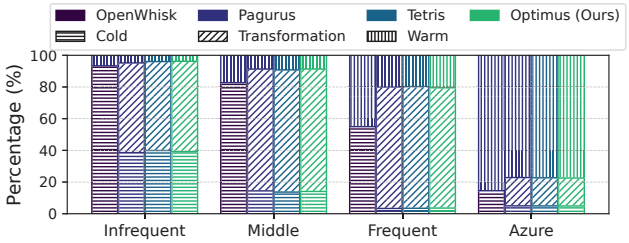


Figure 14. Percentage of cold start, model transformation, and warm start of serverless ML inference requests under the Poisson and Azure workloads.

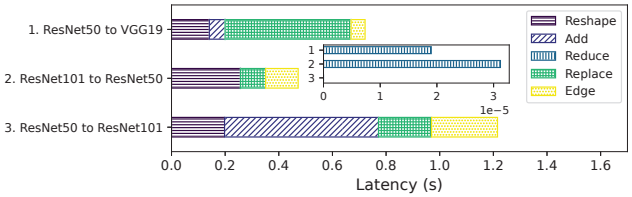


Figure 15. Latency proportion of varying meta-operators for three inter-function model transformation cases in Optimus.

The latency of an ML inference request is the sum of initialization time, computation time, and wait time. According to Figure 13, Optimus reduces the inference latency by 24.00% ∼ 47.56% compared to the existing serverless ML inference systems. Besides, the reason that the performance of Pagurus is better than that of OpenWhisk is that it can save the sandbox and runtime initialization latency. Figure 14 shows the percentages of cold startups under the Poisson distribution and Azure Function. The idea of inter-function container sharing adopted by Pagurus, Tetris, and our Optimus replaces the cold start with container transformation, thus reducing the cold start ratio.

8.4 Micro-benchmark

Figure 15 shows the proportion of varying meta-operators' latency in three inter-function model transformation cases in Optimus. According to the results, the proportion of every type of meta-operator's latency differs for different transformation cases. The transformation from ResNet50 to ResNet101 involves more meta-operators Add since there are more CONV operations in the latter. And the transformation from ResNet101 to ResNet50 reuses the existing CONV operations; thus, it removes the redundant operations via meta-operators Reduce and does not involve any meta-operators Add. Moreover, the total latency of meta-operators Replace in a transformation depends on the number of weights of the destination model as shown in Figure 2c.

Table 1. Latency of planning and execution for three inter-function model transformation cases in Optimus.

	Basic (**Module 2**)		Improved (**Module 2$^+$**)	
Transformation case	Planning	Execution	Planning	Execution
VGG16 to VGG19	12.54s	0.60s	0.6ms	0.60s
VGG16 to ResNet50	171.33s	0.70s	1.1ms	0.72s
ResNet50 to VGG19	50.95s	0.72s	1.0ms	0.72s

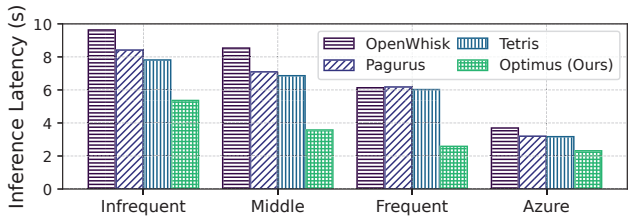


Figure 16. Average service time of serverless ML inference requests in the servers with the support of GPU.

Table 1 shows the latency of planning and execution for three inter-function model transformation cases in Optimus. "Basic" denotes the basic algorithm based on Munkres algorithm (i.e., **Module 2** in § 4.4) and "Improved" denotes the improved algorithm (i.e., **Module 2$^+$** in § 4.4). As discussed in § 4.4, the basic algorithm can achieve an optimal solution but spends a long time on planning. The improved algorithm for efficient transformation planning reduces the time cost of planning by about 99.99% with a nearly optimal solution.

The above results exclude the overhead of downloading images from remote registries since the images are pulled from local registries or caches in our implementation as described in § 7, incurring minimal costs. However, when a new node joins the system, it needs to pull an image with a size of 1.08 GB from remote registries at a time cost of about 8 seconds through a gigabit network.

8.5 GPU Support

Optimus is still effective for GPU inference, and we utilize NVIDIA Container Toolkit [24] to enable containers to access GPU. We evaluate the performance of Optimus in a GPU-enabled server regarding the latency of serverless ML inference requests. As shown in Figure 16, Optimus reduces the inference latency by 26.93% ∼ 57.08% compared to the existing systems. The reason that the inference latency in

the GPU-enabled server is longer than that in the CPU-only server is the high overhead of GPU-related runtime initialization and model loading in GPU.

9 Conclusion

We present Optimus, a new serverless ML inference system with low cold-start overhead based on a novel idea of inter-function model transformation. We implement a prototype that supports serverless ML inference queries on CNN and transformer models. The results show that Optimus reduces the average service time by 24.00% ∼ 47.56% on a Poisson simulated workload and a real workload from Azure compared to the existing serverless ML inference systems. The reason for the performance improvement is that our inter-function model transformation reduces the model loading latency in containers by up to 99.08% compared to loading models from scratch in the existing systems.

Acknowledgments

This research was supported by funding from the Key-Area Research and Development Program of Guangdong Province (No. 2021B0101400003), Hong Kong RGC Research Impact Fund (No. R5060-19, No. R5034-18), Areas of Excellence Scheme (AoE/E-601/22-R), General Research Fund (No. 15220 3/20E, 152244/21E, 152169/22E, 152228/23E), Shenzhen Science and Technology Innovation Commission (JCYJ20200109 142008673), the Pearl River Talent Recruitment Program (No. 2019QN01X130), the Major Key Project of PCL (PCL2023AS7-1). We thank all anonymous reviewers and our shepherd, who helped improve the paper.

References

[1] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S. Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00073

[4] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing Inference Serving on Serverless Platforms. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2071–2084. https://doi.org/10.14778/3547305.3547313

[5] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. 2023. Groundhog: Efficient Request Isolation in FaaS. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery.

[6] Azure. 2023. Azure Functions: Execute event-driven serverless code functions with an end-to-end development experience. Retrieved March 20, 2023 from https://azure.microsoft.com/en-us/products/functions/

[7] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery.

[8] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (nov 2019), 44–54. https://doi.org/10.1145/3368454

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[10] Xuanyi Dong and Yi Yang. 2020. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=HJxyZkBKDr

[11] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[12] Alexander Fuerst and Prateek Sharma. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[13] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, Zhaohui Yang, Yiman Zhang, and Dacheng Tao. 2023. A Survey on Vision Transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 1 (2023), 87–110. https://doi.org/10.1109/TPAMI.2022.3152247

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[15] HuggingFace. 2021. HuggingFace Model Hub. Retrieved March 20, 2023 from https://huggingface.co/models?sort=downloads

[16] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 149 (oct 2019), 26 pages. https://doi.org/10.1145/3360575

[17] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2023. ModelKeeper: Accelerating DNN Training via Automated Training Warmup. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 769–785. https://www.usenix.org/conference/nsdi23/presentation/lai-fan

[18] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/atc22/presentation/li-jie

[19] Yuepeng Li, Deze Zeng, Lin Gu, Mingwei Ou, and Quan Chen. 2023. On Efficient Zygote Container Planning toward Fast Function Startup in Serverless Edge Cloud. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*. 1–9.

[20] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022*

USENIX Annual Technical Conference (USENIX ATC 22). USENIX Association, Carlsbad, CA, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[21] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. 2022. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Surv.* 54, 10s, Article 220 (sep 2022), 34 pages. https://doi.org/10.1145/3508360

[22] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[23] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 115–130. https://doi.org/10.1145/3318464.3389758

[24] NVIDIA. 2023. NVIDIA Container Toolkit. Retrieved March 20, 2023 from https://github.com/NVIDIA/nvidia-docker

[25] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 57–69.

[26] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[27] Google Cloud Platform. 2021. Machine Learning on Google Cloud Platform. Retrieved March 20, 2023 from https://github.com/GoogleCloudPlatform/ml-on-gcp

[28] Bartłomiej Przybylski, Maciej Pawlik, Paweł Zuk, Bartłomiej Łagosz, Maciej Malawski, and Krzysztof Rzadca. 2022. Using Unused: Non-Invasive Dynamic FaaS Infrastructure with HPC-Whisk. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 40, 15 pages.

[29] Bartłomiej Przybylski, Maciej Pawlik, Paweł Zuk, Bartłomiej Łagosz, Maciej Malawski, and Krzysztof Rzadca. 2022. Using Unused: Non-Invasive Dynamic FaaS Infrastructure with HPC-Whisk. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 40, 15 pages.

[30] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[31] Kaspar Riesen and Horst Bunke. 2009. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing* 27, 7 (2009), 950–959.

[32] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 753–767. https://doi.org/10.1145/3503222.3507750

[33] Amazon Web Services. 2022. Machine learning inference at scale using AWS serverless. Retrieved March 20, 2023 from https://aws.amazon.com/cn/blogs/machine-learning/machine-learning-inference-at-scale-using-aws-serverless/

[34] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Comput. Surv.* 54, 11s, Article 239 (nov 2022), 32 pages. https://doi.org/10.1145/3510611

[35] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 14, 14 pages.

[36] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[37] Oleg Sémery. 2021. Sandbox for training deep learning networks. Retrieved March 20, 2023 from https://github.com/osmr/imgclsmob

[38] TensorFlow. 2022. Save and load Keras models. Retrieved March 20, 2023 from https://www.tensorflow.org/guide/keras/save_and_serialize

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[40] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[41] Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2022. Serverless Data Science - Are We There Yet? A Case Study of Model Serving. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1866–1875. https://doi.org/10.1145/3514221.3517905

[42] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 768–781. https://doi.org/10.1145/3503222.3507709

[43] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[44] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*. ACM. https://www.microsoft.com/en-us/research/publication/faster-and-cheaper-serverless-computing-on-harvested-resources/

,