FluidFaaS: A Dynamic Pipelined Solution for Serverless Computing with Strong Isolation-based GPU Sharing

Xinning Hui
xhui@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Yuanchao Xu
yxu314@ucsc.edu
University of California, Santa Cruz
Santa Cruz, USA

Xipeng Shen
xshen5@ncsu.edu
North Carolina State University
Raleigh, USA

Abstract

Prompted by the rise of artificial intelligence (AI) or machine learning (ML), more serverless workloads demand efficient GPU support. Recent years have witnessed a shift of interest from weak isolation-based methods, such as Multi-Process Service (MPS), to strong isolation-based methods, such as Multi-Instance GPU (MIG), for GPU support on serverless platforms, thanks to concerns about performance interference and security. The current MIG-based solution for serverless computing is, however, subject to severe GPU resource fragmentation and under-utilization. This paper identifies the reason as the gap between current MIG supports in serverless computing and the rigid constraints in MIG (re)configurations. It proposes FluidFaaS, a solution that enables flexible MIG management for serverless computing. Through a novel programming system support, FluidFaaS enables fine-grained resource assignment to the components within a serverless function, based on which, it equips the invokers with runtime support that constructs pipelines on MIGs on the fly for a serverless function. The innovations, along with a hotness-aware eviction-based time sharing of MIG slices, significantly reduce GPU resource fragmentation and enhance system throughput. Evaluations demonstrate that FluidFaaS outperforms the state-of-the-art solutions by 25%-75% in throughput while achieving up to 90% higher SLO hit rates in various workloads.

CCS Concepts

• **Computer systems organization → Cloud Computing**; • **Computing methodologies → Planning and scheduling**.

Keywords

Cloud computing, Serverless Computing, Quality of Service, Function-as-a-Service, Resource Management, Resource Allocation, Resource Efficiency, Machine Learning for Systems, Deep Learning

ACM Reference Format:
Xinning Hui, Yuanchao Xu, and Xipeng Shen. 2025. FluidFaaS: A Dynamic Pipelined Solution for Serverless Computing with Strong Isolation-based GPU Sharing. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3731545.3731580

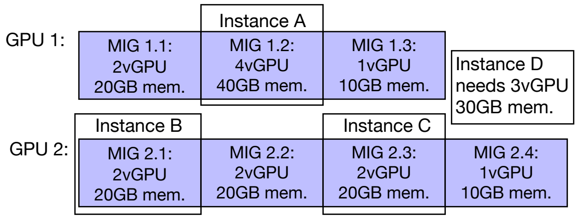


Figure 1: Current MIG support in serverless computing cannot make serverless function "instance D" utilize idle resources fragmented into multiple MIGs.

1 Introduction

Recent years have witnessed an increasing interest in providing efficient Graphic Processing Unit (GPU) support for serverless computing. It is motivated largely by the popular demands for services based on AI and ML capabilities, for which, GPU is essential for delivering high performance and throughput. Serverless platform is appealing for hosting such services, thanks to the ease of programming and maintenance, autoscaling, and pay-as-you-go billing [17, 20, 23, 25–27, 36, 50, 52, 55, 56]. However, effective supporting the use of GPU on serverless remains an open problem.

There have been some recent research efforts trying to narrow the gap. A large portion of them, however, are about how to best leverage NVIDIA Multi-Process Service (MPS) [1] to facilitate GPU sharing across different function instances [23, 27, 45, 55, 57]. However, context sharing in MPS makes it subject to performance interference and security concerns due to its weak isolation. To overcome these limitations, recent studies [32, 36] and industry container orchestration systems (e.g., Kubernetes) [12] shift the focus to Multi-Instance GPU (MIG) technology [13].

MIG offers strong isolation. When a GPU is configured into multiple MIGs, the resource partition between MIGs is thorough, from processing units to cache, memory, and even data paths. The strong isolation avoids both performance interference between MIGs and security hazards, the two fundamental concerns of MPS. An important catch, though: the configuration is hard to change at runtime. After the MIG partition for a GPU is configured and gets deployed, it would take several minutes to change the configuration.

Due to the lack of methods to gracefully deal with the rigid restriction on MIG reconfiguration, the previously proposed supports of MIG in serverless computing are subject to severe GPU resource underutilization. ESG [32], a state-of-the-art work on serverless

HPDC '25, Notre Dame, IN, USA

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3731580

MIGs, demands 167% more than the required resource on average, as detailed in Section 4. The root reason is that the previous solutions view a serverless function as a fixed monolithic unit in GPU usage. In those solutions, for a given serverless function, all the GPU kernels inside it are assumed to run on the same MIG instance (or an entire GPU). The MIG instance size, hence, must be greater than the peak GPU memory required by the serverless function. This monolithic view causes severe GPU under-utilization, unnecessary costs, and long waiting times.

The intuition is illustrated in Figure 1. MIG 1.2 on GPU-1 is occupied by a serverless function (instance A), and MIG 2.1 and MIG 2.3 on GPU-2 are occupied by instance B and C. Even though the total idle resources on either GPU is sufficient for instance D, the idle resources on both GPUs are held in two separate MIG instances—there is no single MIG instance that is large enough to hold it. Because quick MIG reconfiguration is impossible, instance D has to wait for existing instances to complete and a large MIG (MIG 1.2 in this example) becomes available. The consequences are multifold: (i) the long waiting time of instance D, (ii) the fragmentation of the GPU resource, (iii) the under-utilization of the GPU, and (iv) the high unnecessary cost.

In this paper, we introduce FluidFaaS, a solution that bridges the gap between the rigid configuration of MIGs and serverless ML workloads. The central idea is to enable efficient pipelined use of different MIG instances by different components of a serverless function.

Rather than treating an entire serverless function as a single unit for resource configuration, we introduce FluidFaaS function, a new form of serverless function that allows the runtime to automatically split a serverless function into stages. These stages are assigned to available fragmented MIG slices, creating a pipeline that utilizes these slices to serve requests, thereby improving overall GPU utilization. For instance, in Figure 1, application instance D could be divided into two functions: the first requiring 2vGPU-20GB memory, and the second requiring 1vGPU-10GB memory. This division allows for the employment of fragmented MIG slices 1.1 and 1.3 for accommodating instance D.

To materialize the idea into efficient serverless support, there are several challenges. (i) The serverless programming model, in conjunction with resource scheduling, lacks mechanisms for dynamically constructing functions based on available fragmented resources. Existing serverless models require users to develop applications as static functions, which are then encapsulated within containers. These systems allocate resources to deploy instances from these predefined containers, disallowing the dynamic construction of pipeline functions and containers. (ii) Data movement and communication overhead is another challenge. Due to the strong isolation between MIGs, the communications between two pipeline stages on separate MIGs cannot happen on GPU memory directly. They have to go through CPU memory, incurring extra time. (iii) Efficient use of the possibly multiple instances of a serverless function introduces additional scheduling challenges. Instances may need to time-share resources or utilize different resources and pipeline partitions to construct the pipeline. These instances exhibit varied latency and throughput. How to effectively schedule requests and auto-scale to optimize resource utilization while meeting SLOs presents complex, multi-dimensional challenges.

We propose a solution named FluidFaaS, a flexible serverless platform for ML inference on MIGs. FluidFaaS enables the new capability of a serverless programming model by providing programming and runtime support that enables the dynamic construction of pipelines within a container. To address the inherent complexities of scheduling, we equip FluidFaaS with a scheme for hotness-aware eviction-based time sharing of MIG slices. This approach moves away from the rigid one-to-one instance-to-MIG slice binding, introducing a dynamic binding model facilitated by eviction. This model allows multiple instances to time-share a single MIG slice based on requests. By continuously assessing recent instance usage, our system can strategically evict and reload instances to optimize the utilization of an MIG slice while still meeting SLOs.

Overall, this work makes the following major contributions:

- It identifies the fundamental reason for severe GPU resource under-utilization, the gap between current MIG supports in serverless computing and rigid MIG configurations.
- It proposes a novel form of serverless function named FluidFaaS function, and its programming models and runtime support to enable on-the-fly pipeline construction within a serverless function for it to leverage fragmented MIGs on GPU.
- It equips FluidFaaS with a scheme for hotness-aware eviction-based time sharing of MIG slices.
- It empirically confirms the effectiveness of FluidFaas, outperforming the state-of-the-art work by 25%-75% in system throughput while achieving up to 90% higher SLO hit rates in various workloads.

2 Background

2.1 Serverless Platform

Figure 2 illustrates a typical architecture of a serverless platform, such as OpenWhisk, Knative, OpenLambda, or OpenFaaS [3, 4, 7–11, 31]. These platforms consist of centralized control modules, including the Controller and load balancer, that accept function invocations and distribute them to various nodes. The controller reacts to the rate of incoming requests. If there is a surge in demand, the controller will spawn more instances to handle the increased load. Conversely, it will scale down and remove instances during periods of low demand.

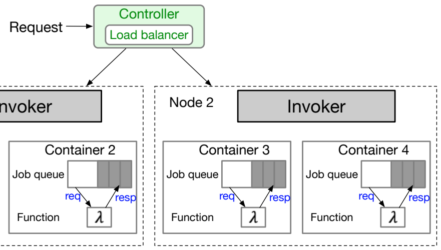


Figure 2: Overview of existing serverless platforms.

Within each node, an invoker module is tasked with initializing instances and executing functions. Each invoker manages a pool of function instances. Upon receiving a request from the controller,

Table 1: Comparison of FluidFaaS with related works.

	INFless [55]	Protean [19]	Miso [36]	ESG[32]	StreamBox [52]	FaaSwap [57]	Antman [54]	TGS [51]	Orion [48]	FluidFaaS
serverless framework	✓	✓	×	✓	✓	✓	×	×	×	✓
GPU sharing	✓	✓	✓	✓	✓	×	✓	×	✓	✓
Sharing Mechanism	MPS	MIG+MPS	MIG	MIG	CUDA stream	N/A	MPS	N/A	CUDA stream	MIG
Strong isolation	×	×	✓	✓	×	✓	×	✓	×	✓
High GPU utlization	✓	✓	×	×	✓	✓	✓	✓	✓	✓
Fragmentation-aware	×	×	×	×	×	N/A	×	N/A	×	✓

Table 2: Complete list of MIG profile on an A100 GPU

Slice	Compute	Memory	Max Count
7g.80gb	7GPC	80gb	1
4g.40gb	4GPC	40gb	1
3g.40gb	3GPC	40gb	2
2g.20gb	2GPC	20gb	3
1g.10gb	1GPC	10gb	7

the invoker directs the request to an available instance. It continues routing to the next available instance once the current one reaches its capacity.

Taking Knative [7, 9] as an example, it is an open-source platform that extends Kubernetes to support serverless workloads. Each container has a job queue, allowing it to handle multiple requests sequentially. This queuing mechanism ensures that requests are managed effectively, reducing the immediate need for container scaling and optimizing resource utilization across the platform.

2.2 GPU Resource Sharing

GPU sharing allows multiple processes to share a single GPU for their executions. Modern GPUs support time sharing (one after the other) and spatial sharing. However, time sharing cannot address the challenge of the GPU underutilization. Spatial sharing is essential for ML inferences because an inference by an ML model often uses only a fraction of the massive parallel computing capacity of a GPU [1, 13]. Allowing multiple processes to execute concurrently on a GPU is essential for turning its computing power into throughput. Modern GPUs from NVIDIA offer two mechanisms for spatial sharing, Multi-Process Service (MPS) [1] and Multi-Instance GPU (MIG) [14]. MPS shares a single GPU context across multiple processes, which can result in resource contention and pose security risks. This approach is unsuited for serverless computing, where isolated execution environments and strong security are essential.

Multi-Instance GPU (MIG) partitions a single GPU into multiple hardware-isolated instances, providing better performance isolation and security. NVIDIA A100 and H100 Tensor Core GPUs support MIG mode. This paper uses the A100 (80GB) as our example. On an A100 GPU, its streaming multi-processors (SM) consist of seven graphics processing clusters (GPC); it is common to regard each GPC as a vGPU in serverless platforms [32]. In MIG mode, a *slice* (used interchangeably with *MIG instance*) contains one or more GPCs and a certain amount of GPU memory. We list the full MIG slice profiles in Table 2. The 'max count' indicates the maximum number of slices of the same type that can coexist on a single GPU. The slice-type notation specifies the number of GPCs and the corresponding amount of GPU memory.

Unlike the MPS approach, arbitrary MIG partitions are not supported due to hardware restrictions. A full A100 GPU can be partitioned only into specific combinations of MIG slices. For example, both (4g.40gb, 2g.20gb, 1g.10gb) and (4g.40gb, 3g.40gb) are valid partitioning options. There are only 18 MIG configurations on an A100 GPU [15].

Dynamic reconfiguration of the MIG partition during runtime is impractical for the serverless platform due to the significant time overhead. Reconfiguring MIG requires several minutes to checkpoint, re-partition, and resume execution [36]. This delay is particularly problematic in serverless environments, where functions are expected to be highly responsive and scale rapidly to handle fluctuating workloads. The lengthy reconfiguration process could introduce unacceptable latencies, disrupting the seamless execution of serverless functions and diminishing the overall efficiency of the platform.

3 Related Work

There are two kinds of GPU sharing, time-sharing and spatial sharing. Effisha [21] by Chen and others pioneered software approaches for enabling efficient preemptive scheduling of GPU for flexible time sharing. Zhu and others [61] first studied co-run scheduling with power cap on integrated CPU-GPU systems. A later work, TGS [51], intercepts system calls from containers to GPUs so that it can regulate the dequeuing rates of tasks to production deep learning (DL) training (performance is critical) and tasks to opportunistic DL training, which helps ensure the performance of the former while allowing the latter still benefit from the GPU resources. FaaSwap [57] is an example of time sharing for serverless computing. It keeps models in main memory and swaps them onto GPUs on demand, allowing many inference functions to share a GPU simultaneously while each function utilizes the full GPU.

Time-sharing cannot address the problem of GPU under-utilization when one job cannot fully utilize the entire GPU. For spatial sharing, the mechanisms are through CUDA streams, MPS, and MIG (or the equivalent schemes in other vendors' GPUs). Many prior studies concentrate on GPU sharing on standalone servers or clusters [28, 29, 39, 49, 51, 53, 54, 58]. AntMan [54] is an example of using MPS in a cluster. It employs an application-level method, modifying DNN frameworks so that multiple DL training jobs can share a GPU effectively. Miso [36] explores using MIG to support multi-tenant ML applications in data centers. It predefines the MIG slices and determines application co-location based on profiling data, offering no dynamic adaptation to the dynamic changes of workload or systems.

Serverless computing differs from computing on a standalone server or a cluster in many aspects, from programming models to the levels of control from users, the execution models, the expected performance, and security guarantees. Recent years have witnessed an increasing interest in GPU spatial sharing in serverless computing. INFless [55], Protean [19], and Llama [45] are examples that use MPS to allow multiple ML containers spatially share a GPU in serverless settings. A significant challenge facing those works is how to mitigate the performance interferences caused by the weak isolation of MPS between co-located jobs. They have attempted various methods to address that, from profiling to building slowdown models and workload classifications. These solutions impose certain applicable conditions (e.g., existence of both compute-intensive and memory-intensive jobs), and even with them, they still cannot fully address the interference concerns or offer strong performance or security guarantees that are essential to serverless computing. Orion [48] and Streambox [52] employ GPU streams for sharing among the components within a workflow. Although they demonstrate the usefulness of GPU streams in reducing communication overhead and improve resource utilization, strong isolation through MIG is still indispensable for general serverless computing platforms where many serverless workflows from many users come and go and share the same set of resources.

Due to those reasons, recent research [32, 36] and industry container orchestration systems (e.g., Kubernetes) [12] have shifted the focus to MIG-based GPU sharing for serverless computing. The state-of-the-art solution, ESG [32], is a representative. It is the work closest to this current study. ESG assumes MIG-based GPU sharing, and concentrates on task scheduling at the serverless controller. Its main contribution is combining A*-search and a novel dual-blade pruning to prune the space of task scheduling dramatically. It, however, takes a monolithic view of a serverless function in MIG assignment and usage, as we have explained in Section 1, causing serious resource underutilization and fragmentation. This current study aims to address the limitation. Table 1 summarizes the differences between this work and the most relevant prior studies.

4 Motivation

This section presents our analysis of the underutilization of GPU MIGs in serverless computing environments. Prior studies have analyzed GPU underutilization in traditional cloud computing setups for ML inference. However, our work identifies distinct factors causing the underutilization of GPU MIGs in serverless frameworks, including resource fragmentation and exclusive keep-alive.

We conducted empirical studies on workflows involving ML inferences within real-world serverless scenarios using ESG [32], a state-of-the-art serverless platform that integrates MIG. Our analysis utilizes DNN applications with invocation frequencies and intervals as the traces from Azure's serverless functions [47] (details in Section 6). A server node includes eight A100_PCIE_80GB GPUs. For lack of space, we show only the observations when each GPU is partitioned into three MIG slices (4g.40gb, 2g.20gb, and 1g.10gb); the conclusions hold on other partitions (confirmed in Section 7).

One important feature of serverless computing is its capability for auto-scaling, which dynamically adjusts resources based on the current request load. We analyze GPU utilization and the required GPU resources through empirical studies, as depicted in Figure 3 (a). In ESG, a GPU is considered utilized if one MIG is processing requests. The required GPU resource reflects the ideal resource consumption necessary to manage fluctuating request volumes. Our results indicate that ESG requires substantially more resources than the ideal consumption levels. For instance, at the 83rd second, ESG's resource demand exceeds the required resource by 167%.

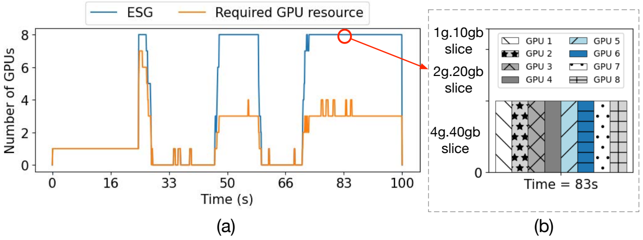


Figure 3: (a) GPU utilization in ESG and the required GPU resources. For ESG, a GPU is considered utilized if it is processing requests. (b) MIG usage at the 83rd second.

Further analysis of the utilization of MIGs is presented in Figure 3 (b). We observe that only the 4g.40gb MIG slices are actively utilized, while the 1g.10gb and 2g.20gb MIG slices remain idle under high workload conditions. This underutilization is caused by the rigid partitioning inherent in MIG, combined with the scheduling algorithms used in ESG and other serverless studies, which all treat the entire serverless function as a single unit for resource configuration. These algorithms estimate the necessary resources for functions to meet SLOs and memory requirements. Although the aggregate resources of the 1g.10gb and 2g.20gb MIG slices are adequate, individually, an individual slice fails to meet requirements, leading to their idleness and the underutilization of the GPU. We refer to such underutilization as resource fragmentation.

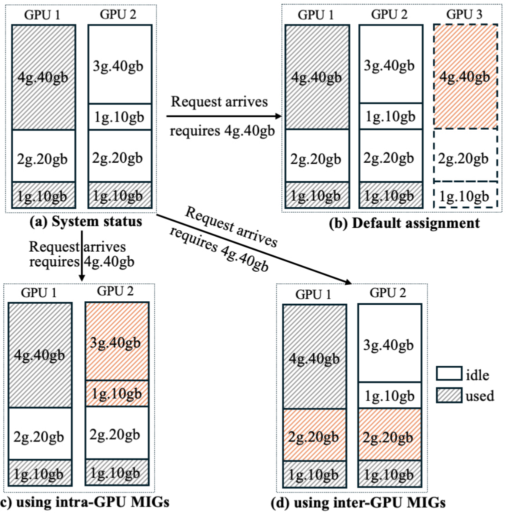


Figure 4: Illustration of GPU resource fragmentation

Resource Fragmentation. We extend our analysis to demonstrate resource fragmentation within a GPU and across multiple GPUs. The system initially allocates two GPUs: GPU 1 utilizes one 4g.40gb MIG and one 1g.10gb MIG, and one 1g.10gb MIG on GPU 2 is in use, as illustrated in Figure 4 (a). When a new function instance requires a 4g.40gb, GPU 2, despite having sufficient total resources, cannot accommodate the instance due to its MIG partitions. In addition, although the combined resources of GPU 1 and GPU 2 are adequate, the separation across two GPUs prevents their unified allocation to the new instance to meet SLOs. Such resource fragmentation requires the allocation of an additional GPU to meet the new instance's requirement, leading to GPU underutilization in serverless systems.

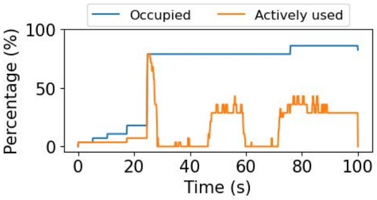


Figure 5: The occupied and actively used GPU percentage.

Exclusive Keep-alive. In addition to resource fragmentation, the underutilization of allocated MIGs is exacerbated by the common exclusive keep-alive policy, which keeps a model active within an MIG slice and precludes its resources from being used by other function instances. Figure 5 illustrates the occupied rates and actively used rates of MIGs across GPUs 1-8 in production trace [47], where the average active percentage is 16.1%, and these MIGs operate at less than 35% percentage for 90% of the time. The exclusive keep-alive policy hinders the reassignment of computational resources to other instances, exacerbating GPU underutilization.

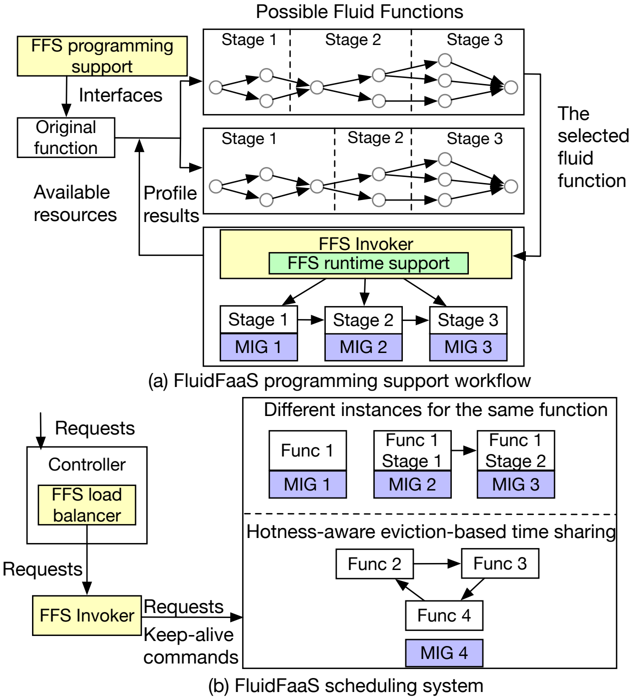
Approaches for Addressing Problems. We propose two approaches: pipeline-based instance construction and eviction-based time sharing, to mitigate the problems of resource fragmentation and the exclusive keep-alive policy.

The pipeline-based instance construction allows the automatic construction of a Directed Acyclic Graph (DAG) that represents the internal components of a function and enables the runtime to allocate MIG slices to the pipeline. This approach facilitates the formation of a processing pipeline that effectively connects and utilizes fragmented MIG slices. For instance, an instance requiring 4g.40gb slice could be built into a two-stage pipeline utilizing a combination of 3g.40gb and 1g.10gb MIG slices, as shown in Figure 4(c), or two 2g.20gb MIG slices, as depicted in Figure 4 (d).

The eviction-based time sharing approach allows the time sharing of a MIG slice by multiple instances, thereby decoupling the rigid one-to-one binding of instances to MIG slices and enabling dynamic rebinding of models. Although eviction introduces overhead, it is a viable solution given that MIGs operate below 35% capacity for 90% of the time, as illustrated in Figure 5. An appropriate design for eviction-based time sharing can improve utilization and reduce costs. Importantly, this method maintains the security principle of exclusive MIG slice usage by allowing only one instance to access a MIG slice at any given time, ensuring secure and isolated computational environments.

5 Design of FluidFaaS

This section introduces the design of FluidFaaS on how to improve the GPU resource utilization under the resource fragmentation and exclusive keep-alive caused by MIG.



(a) FluidFaaS programming support workflow

(b) FluidFaaS scheduling system

Figure 6: Overview of FluidFaaS.

The serverless functions we target in this work consist of multiple Deep Neural Network (DNN) models structured as a Directed Acyclic Graph (DAG), a common practice in serverless ML applications [24, 34, 37, 41, 45, 59, 60]. Although some work assumes each DNN model is made into a serverless function, recent studies advocate putting the entire workflow of an ML application as a serverless function, for the significantly smaller overhead in communications and cold starts [18, 33, 37, 38, 40–42, 46], and the over 95% reduction of memory usage for avoidance of duplicated GPU runtime [52]. It is what our serverless functions are.

5.1 Overview

We introduce FluidFaaS (FFS) to effectively support pipeline-based instance construction and eviction-based time sharing. FFS comprises two primary components: programming support and the FluidFaaS scheduling system.

We propose a new form of serverless function called *FluidFaaS function.* As depicted in Figure 6 (a), the FluidFaaS programming support enables the automatic configuration of a pipelined DAG, based on available resources and profiling results. The selected FluidFaaS (FFS) function has several stages, allowing the runtime to allocate MIG slices to different stages efficiently. The FFS runtime and Invoker use one instance to manage different stages on different MIG slices and the data transfer between them. Details about the programming support are presented in Section 5.2.

We design the FluidFaaS scheduling system to implement hotness-aware eviction-based time sharing and scheduling of function instances. This system incorporates multi-level keep-alive states and corresponding state transitions to materialize hotness-aware eviction-based time sharing in the FFS invoker. Various keep-alive states and different pipelines for the same function create the heterogeneity of instances, imposing significant scheduling challenges as they have different latencies and throughputs. To address these, we co-designed keep-alive state transitions with a heterogeneity-aware traffic routing algorithm in the FFS load balancer and the FFS invoker. This approach aims to meet SLOs and improve resource utilization. Details are discussed in Section 5.3.

5.2 Automatic Pipeline Instance Construction

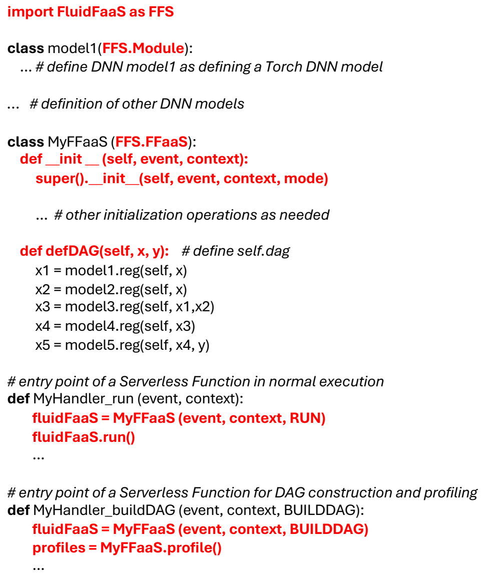
When scaling up requires launching a new instance, the construction of that instance should be flexible and adaptive based on available resources. For example, depending on resource availability, the function might be divided into a two-stage or three-stage pipeline, or configured as a non-pipeline process. However, the rigid, hard-coded nature of serverless functions typically limits dynamic pipeline construction. To address this limitation, we provide programming support that ensures MIG instance usage is both transparent to users and adaptable at runtime. Transparency here refers to an interface that abstracts the complexities of MIG management away from the serverless function developer. It is essential for both ease of use and performance due to the unpredictable availability of GPU resources.

5.2.1 Programming Support. In existing serverless function platforms, a serverless function is regarded as a monolithic unit to invokers. The key to enable flexible MIG-based GPU resource mapping and utilization is to make each component in a serverless function a unit manageable at runtime on invokers. The management includes decisions on what size of the MIG slice and which specific slide is assigned to each component. The design shall also ensure the executions of the serverless function indeed execute each of its components on the MIG slice as specified by the invoker.

Our solution is *FluidFaaS function*. This new form of serverless function extends the current serverless function with a module named *FluidFaaS*. A serverless function written with FluidFaaS (FFS) APIs will be equipped with a DAG, called FFS DAG, which represents the components in the function as nodes and dataflows among them as edges. Please note that this DAG differs from task DAGs in existing serverless computing: The former captures the computation flows *within* a serverless function, while the latter is about relations *among* serverless functions. After DAG construction, each node in the FFS DAG will carry a performance profile of the size of the required memory and the running speeds of the component on each size of MIG.

When an FFS function is being launched by an invoker, based on the FFS DAG, the invoker will first find out the appropriate assignment of MIG slices to the components in the DAG, writes the assignment into the configuration layer of the FFS function, and then launches the FFS function.

The FFS function is structured such that its execution will be done through a FluidFaaS class (a core element in the FluidFaaS module), which ensures that the execution will put each component onto the MIG slice as specified in the configuration layer, construct efficient communication channels among them, and execute the FluidFaaS function efficiently.



```
import FluidFaaS as FFS

class model1(FFS.Module):
   ... # define DNN model1 as defining a Torch DNN model

... # definition of other DNN models

class MyFFaaS (FFS.FFaaS):
   def __init __ (self, event, context):
      super().__init__(self, event, context, mode)

      ... # other initialization operations as needed

   def defDAG(self, x, y):   # define self.dag
      x1 = model1.reg(self, x)
      x2 = model2.reg(self, x)
      x3 = model3.reg(self, x1,x2)
      x4 = model4.reg(self, x3)
      x5 = model5.reg(self, x4, y)

# entry point of a Serverless Function in normal execution
def MyHandler_run (event, context):
     fluidFaaS = MyFFaaS (event, context, RUN)
     fluidFaaS.run()
     ...

# entry point of a Serverless Function for DAG construction and profiling
def MyHandler_buildDAG (event, context, BUILDDAG):
     fluidFaaS = MyFFaaS (event, context, BUILDDAG)
     profiles = MyFFaaS.profile()
     ...
```

Figure 7: Illustration of the programming of a FFaaS function.

Programming Interface. Figure 7 gives an example showing how the programming interface of FluidFaaS can be used in writing a serverless function. FFaaS module contains two most important classes. (i) The first one is *FluidFaaS.Module*, which is a thin wrapper of the DNN model class, *nn.Module*, in PyTorch[1]. The definitions of DNN models in a PyTorch program do not need to be modified except that the superclass is changed from *nn.Module* to *FluidFaaS.Module*. Each of the five DNN models in the example is made into a subclass of *FluidFaaS.Module*. One of the main extensions of *FluidFaaS.Module* over *nn.Module* is that it includes a method "reg", which registers the model in the FFS DAG along with the types and shapes of the model's inputs and outputs (see the DAG definition function "defDAG"). (ii) The second important component in FluidFaaS module is the *FFaaS* class, which offers most of the functions that are essential for the serverless function to execute each of its components based on the MIG mapping specified by the invoker, such as build DAG, import the MIG assignments and the structure of the pipeline determined by the invoker. *FFaaS* also includes functions to profile each component. The initialization of a *FFaaS* object has two modes. In the "BUILDDAG" mode, it calls "defDAG" to build the DAG, while in the "RUN" mode, from the configuration layer, it imports the DAG, along with the MIG configs and stage configs that the invoker has put onto the configuration layer.

Runtime Support of FFaaS. Listing 1 shows the implementation of FluidFaaS.run() to explain how a FluidFaaS function gets executed. As this is "RUN" mode, the __init__() function already retrieves the DAG and the MIG assignments and pipeline configurations. The *run()* function creates a separate process for each MIG, establishes shared memory for data transfer, and sets up queues to trigger the execution of subsequent stages. The module *_run_inference()*

[1] We use PyTorch as a representation; FluidFaaS is extensible to other ML frameworks.

represents the execution process for each stage. In this function, the input tensor is retrieved from shared memory, model inference is performed, and the resulting output tensor is written back to shared memory for use in the next stage. This interface is designed for simplicity, requiring only the DAG registration. The function *_terminate_processes()* monitors changes in instance states and adjusts the states of MIGs (Section 5.2.2). It also responds to termination signals from the serverless platform by stopping the processes on the MIG slices.

Unlike the default serverless function, where each function is hosted by a process that starts execution upon receiving a trigger event, FluidFaaS conducts the execution running on a dedicated MIG instance with a separate process. These pipeline components communicate through the shared memory of the host system. The predecessor process writes its output tensor to the shared memory, as shown in the module *_write_to_shared_memory()*, and successor process reads output tensor using module *_get_from_shared_memory()* as its input tensor, facilitating efficient data flow between pipeline stages. This approach streamlines execution.

The use of shared memory on the host for communications helps keep the communication overhead low. We further discuss the data communication overhead associated with this design in Section 7.3, where we analyze its impact on overall system performance and efficiency.

5.2.2 Runtime Support in Invoker. The programming interface introduced by the FluidFaaS module makes each component in a serverless function a unit in the FFS DAG. Based on it, the runtime support in the invoker figures out the appropriate MIG resource assignment to each of the components and outlines the appropriate pipeline for the serverless function.

Leaving these functionalities to the runtime of invokers is an important design decision of FluidFaaS. It is essential for FluidFaaS to work efficiently despite the dynamic nature of serverless environments. Workloads and system conditions can change dramatically and unpredictably; predefined pipelines for a specific application cannot suit the needs. An unbalanced pipeline can lead to inefficient resource utilization, causing bottlenecks and increased latency as certain stages may become overburdened while others remain underutilized. This solution takes into account both pipeline balance and current resource availability to ensure high performance.

This runtime support is implemented on each invoker, where it functions as a local scheduler. As the workflow depicted in Figure 6 (a), the invoker is responsible for constructing the pipeline and allocating MIG slices based on current resource availability and application knowledge which is the profile information. This decentralized approach allows the scheduler to efficiently build pipelines and allocate resources, adapting to the invoker's current conditions to ensure efficiency and responsiveness to fluctuating workloads. Notably, this runtime system can be integrated into existing serverless architectures without requiring any modifications to the central controller, making it practical and easy to implement.

Balance is key to building a pipeline, which is crucial for performance. To accomplish this, the DAG is divided into several self-contained and balanced groups. Several studies have explored methods for partitioning DAG [32, 37]. Our method is based on the dominator-based method from ESG [32] but extends it with selection with a coefficient of variation (CV). It evaluates all possible partitions of the DAG by calculating the coefficient of variation (CV) [16] to measure pipeline balance. For simplicity, consider a sequential DAG with five models: [1, 2, 3, 4, 5]. There are 2^4 possible consecutive partitions, each representing a different pipeline configuration. For instance, one configuration could be [[1], [2, 3], [4, 5]], which forms three distinct stages. The CV is determined by dividing the standard deviation of the execution times of these stages by their mean, as shown in Equation 1, where t_n denotes the execution time of stage n. If a stage involves multiple MIGs running in parallel, we use the maximum execution time among them as the stage's execution time.

```
class FFaaS:
    def __init__(self, event, context, mode):
      if (mode == 'RUN'):
         self.dag = self.importDAG()
         self.migs = self.importMIGs()
         self.stages = self.importStages()
         self.eviction = [False] * len(self.migs)
      elif (mode == 'BUILDDAG'):
         ...
    def _load_models(self, DAG):
        #load models for all stages
    def _run_inference(self, stage, queue, next_queue, shared_data
     , nextShdata):
        device = torch.device("cuda")
        while True:
            input = self._get_from_shared_memory(shared_data).to(
     device)
            if input not empty:
                # Run all components in stage based on the DAG
                output = model(input)
                self._write_to_shared_memory(nextShdata, output)
                next_queue.put()
            # Placeholder for actual eviction condition
            if self.eviction[stage]:
                model.cpu()
                del model
    def _get_from_shared_memory(self, shared_data):
       #get data from shared memory
    def _write_to_shared_memory(self, shared_data, data):
       #write data back to shared memory
    def _start_processes(self):
        #start a process for each stage in one mig
        for i in range(len(self.stages)):
            os.environ["CUDA_VISIBLE_DEVICES"] = self.migs[i]
            torch.cuda.init()
            p = mp.Process(
                target=self._run_inference,
                args=(self.stages[i], self.queues[i], self.queues[
     i+1], self.shared_data[i], self.shared_data[i+1])
            )
            self.processes.append(p)
            p.start()
    def _terminate_processes(self):
       #terminate processes when eviction happens
       #By modify self.eviction to True
    def profile(self):
       ...
    def _initialize_shared_memory_and_queues(self):
       #initialize shared memory and job queues
    def run(self):
        self._initialize_shared_memory_and_queues()
        self._load_models()
        self._start_processes()
        self._terminate_processes()
        # Cleanup shared memory
```

Listing 1: Implementation of the core runtime support FFaaS

$$CV = std(t_1, t_2, ..., t_n)/mean(t_1, t_2, ..., t_n) \quad (1)$$

After calculating and sorting the CVs for all possible pipeline configurations, which is done once and offline for each application (thus avoiding any runtime overhead), the next step is to assign a pipeline to the available MIGs within the invoker. We rank pipelines based on their CVs: lower CVs represent better balanced configurations. The pipelines are then evaluated in order with the profiles, against the available MIG resources. If a given pipeline can be supported by the available resources, the invoker records the pipeline configuration and corresponding MIGs on the configuration layer of the serverless function and then proceeds to launch the instance with that configuration. If the resources are insufficient, it moves to the next pipeline in the ranked list. This process continues until a suitable pipeline is found and deployed. By following this approach, we ensure that the most balanced pipeline is selected based on the available MIGs, leading to optimized performance and efficiency.

5.2.3 Programming Efforts for Using FluidFaaS. The efforts needed from developers is minimum. They only need to import FluidFaaS and construct FluidFaaS functions as shown in Figure 7. Specifically, during function construction, they use a FluidFaaS.Module class to encapsulate each ML model and all related data processing, and register them and the data flow among them in the defDAG function. The trigger events are defined as in standard serverless function conventions.

FluidFaaS is designed to support a wide range of ML applications, from traditional deep learning models to more complex large language model (LLM) inference. Since FluidFaaS treats each module as an independent, manageable unit, it can efficiently handle different inference patterns, including sequential, parallel, or hybrid execution across GPU resources. For LLM inference, where multi-stage processing (e.g., tokenization, model execution, and response generation) is required, our framework seamlessly maps these stages to the appropriate GPU resources, ensuring efficient utilization while maintaining low latency. The ability to flexibly allocate and manage GPU resources at runtime makes our solution well-suited for both latency-sensitive applications and high-throughput workloads.

5.3 Hotness-aware Eviction-based Time Sharing

Current serverless computing platforms manage CPU resources efficiently by handling cold and warm states. However, this strategy underperforms GPU resources due to its early binding of computational resources and memory. Specifically, keeping a model in a warm state within a Multi-Instance GPU (MIG) slice precludes the use of its computational resources by another instance, resulting in the underutilization of MIGs.

Multi-level Keep-Alive States. To optimize the utilization of warm-keeping MIG slices, we propose a hotness-aware eviction-based time sharing approach. We propose interleaved usage of the MIG slice through eviction to improve overall efficiency. Crucial to this method is the selection of which instance to evict, as frequent evictions and reloads between GPU and CPU memory undermine performance gain from this method. We define two different instance states based on request loads to identify which instances are suitable for eviction, optimizing the eviction process and improving performance.

Exclusive Hot State. An instance that has a high request load will be tagged as an *exclusive hot state.* An instance in this state keeps all necessary data for the instance resides exclusively within a MIG slice. Instances in this state are exempt from eviction. This policy prevents unnecessary evictions for high-load instances, improving the effectiveness of eviction-based time sharing. All pipeline instances are in the exclusive hot state to simplify scheduling.

Time Sharing State. This state is proposed for instances that are not actively busy (i.e., utilization below 30%). An instance in this state indicates that its MIG can be shared with other instances. The data for such an instance may reside in either MIG memory or CPU memory. Instances classified under this time sharing state share MIG resources, which is aimed at enhancing the overall utilization of MIG slices.

We reuse **warm** and **cold** states in CPU. If part or all of the data is evicted from MIG memory, the data are first moved to the CPU, entering a **warm** state for this instance. In this scenario, when a new request arrives, the data can be retrieved from the CPU and transferred back to MIG memory, reducing loading time compared to fetching the model from remote storage. If there are no requests for this instance for 10 minutes, this instance is terminated, termed as a **cold** state.

Resource allocation and eviction processes operate as follows: Each instance in the exclusive hot state fully occupies a MIG slice without being subject to eviction. Instances in the time sharing state share one or more MIG slices. Requests assigned to these instances are queued according to deadlines subtracted by estimated execution and instance load times and are processed in ascending order of these values. When the FFS Invoker schedules a request, if the corresponding time sharing state instance is already on an MIG, the request is executed directly. Otherwise, the FFS Invoker selects the least-recently-used (LRU) instance for eviction, reloads the pertinent instance, and fulfills the request. The number of MIG slices allocated to time sharing state instances increases if they are overloaded (i.e., unable to meet SLOs), or decreases if their average utilization falls below a threshold, similar to auto-scale policy in prior serverless work.

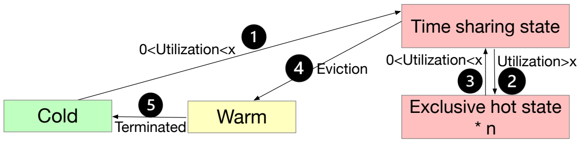


Figure 8: Instance state transition.

Figure 8 illustrates the state transition details. Upon receiving the initial request, the FFS invoker creates an instance and designates it as a time sharing state instance ❶ . As the request volume increases and the instance's utilization exceeds a specified threshold, it transitions to an exclusive hot state ❷ . Conversely, a decrease in request volume reverts the instance to the time sharing state ❸ . Notably, each serverless function is restricted to a maximum of one instance in the time sharing state. In this state, the instance may be evicted to CPU memory, termed as the warm state ❹ . If there is no demand for ten minutes, the instance is terminated, transitioning to a cold state ❺ .

Request Routing. The introduction of multi-level keep-alive states and pipeline functions adds complexity to serverless scheduling, as instances for the same function exhibit varying execution latencies and throughputs. To manage this complexity, we have redesigned the heterogeneity-aware request routing in the FFS load balancer. Our approach optimizes utilization and ensure SLOs by prioritizing request routing based on adjusted deadlines subtracted by estimated execution and instance load times. Requests are then routed in ascending order of these values. The routing begins with *exclusive hot state* instances, utilizing both non-pipeline and pipeline types. For the *exclusive hot state* instances for a function, urgent requests are directed to the instance with the lowest latency until its serving capacity is reached. Subsequent requests are routed to the next lowest latency instance, and so forth. Then the remaining requests are routed to the *time sharing state* instance. This strategy not only maximizes the utilization of *exclusive hot state* instances but also alleviates the load on *time sharing* instances. This request routing algorithm reduces the complexities associated with resource sharing and improves overall system performance.

Pipeline migration. To further enhance performance and adapt dynamically to changes in system resources, FluidFaaS implements a pipeline instance migration strategy that migrates pipeline instances to non-pipeline instances when large MIG slices become available. When a large MIG slice becomes available, the invoker establishes a new non-pipeline instance on this slice. This new instance, once operational, begins handling incoming requests. Concurrently, existing pipeline instances continue to process ongoing requests until completion. Following the fulfillment of these residual requests, the original pipeline instance is gracefully retired. The newly created instance may operate in either an exclusive or a time-sharing state. This migration approach utilizes available resources and avoids pipelines if unnecessary, improving system performance by dynamically responding to resource availability.

6 Evaluation Methodology

Baseline. Our baseline for comparison is the state-of-the-art serverless ML solution supporting MIG, ESG [32]. It has been compared with other representative serverless computing works, INFless [55], FaST-GShare [27], Orion [41], and Aquatope [60], consistently showed a high SLO hit rate with a similar or less cost. ESG assigns functions to specific MIGs based on resource configurations determined by the Controller, which selects the most resource-efficient option that meets the SLO. Besides ESG, we also integrate MIG support into INFless [55] and include it as an additional baseline. As demonstrated in previous work, INFless is one of the most competitive alternatives to ESG.

Table 3: Experimental hardware configuration on each node

CPU device	4 * AMD EPYC 7763 64-Core Processors
CPU Mhz	2445.404
CPU memory	1440GB
GPU device	8 * NVIDIA A100 80GB
GPU memory	80GB

SLO Latency Requirement. SLO latency is an important requirement on a serverless workload. It defines the acceptable latency for the platform to give a response to a request. Let t be the time needed by the application to complete its entire workflow when it runs alone with a unit CPU and the minimum MIG instances, as shown in Table 5. We use *SLO scale* when describing the SLO latency of a workload. It is defined as the ratio between the SLO latency and t. By default, we set the SLO to be 1.5x (SLO scale=1.5), which refers to the case where the acceptable maximal latency is 1.5 times of t. For different workload variants as shown in Table 5, the t is different.

Table 4: Applications

Applications	Composition
Image classification (App 0)	Super resolution [35] ->Segmentation [6, 22] ->Classification [2, 30]
Depth recognition (App 1)	Deblur [5] ->Super resolution ->Depth recognization [44]
Background elimination (App 2)	Super resolution ->Deblur ->Background removal [43]
Expanded image classification (App 3)	Deblur ->(**if** low resolution: ->Super resolution; **else:** pass) ->Background removal ->Segmentation ->Classification

We use the following metrics to examine the performance.

SLO Hit Rate. SLO hit rate is defined as the fraction of requests whose latencies (from the request arrives at the serverless platform and the time when the result is produced) are below the required SLO latency.

GPU time and MIG time GPU time refers to the total time the entire GPU is active, even if only one slice is used. In contrast, MIG time specifically measures the active time of individual MIG slice.

Evaluation environment. The runtime of FluidFaaS is implemented in the invoker node. We have two nodes and each invoker node contains eight A100_PCIE_80GB GPUs. Table 3 reports the node configuration. The MIG partition for each GPU is 1g.10gb, 2g.20gb, and 4g.40gb by default. We test other partitions in the sensitive study in Section 7.4.

Table 5: Application variants and MIG slices to run

Application	Variants	MIG to run (Baseline)	MIG to run (FluidFaaS)
Image classification	small	≥1g.10gb	≥1g.10gb
	medium	≥2g.20gb	≥1g.10gb
	large	≥3g.40gb	≥2g.20gb
Depth recognition	small	≥1g.10gb	≥1g.10gb
	medium	≥2g.20gb	≥ 1g.10gb
	large	≥3g.40gb	≥2g.20gb
Background elimination	small	≥1g.10gb	≥1g.10gb
	medium	≥2g.20gb	≥1g.10gb
	large	≥3g.40gb	≥2g.20gb
Expanded image classification	small	≥ 2g.20gb	≥1g.10gb
	medium	≥4g.40gb	≥1g.10gb
	large	NULL	NULL

Applications and Workloads. For an apple-to-apple comparison, we use the four applications used in ESG [32], with each composed of multiple DNN inferences, as shown in Table 4. Three applications are in a sequence, and one (*Expanded image classification*) is extended with two branches to form a DAG.

Each application is available in three variants—small, medium, and large—determined by memory requirements and batch size. These variants require different sizes of MIG slices to avoid running into out-of-memory errors, as shown in Table 5. Thanks to the pipeline construction of FluidFaaS, the minimum MIG slice needed to run each variant is smaller compared to the Baseline. Notably, the 4g.40gb configuration cannot support the large variant of the

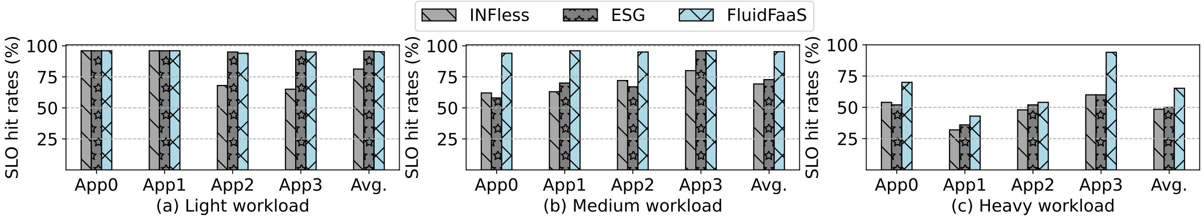


Figure 9: SLO hit rate in different workloads for each application.

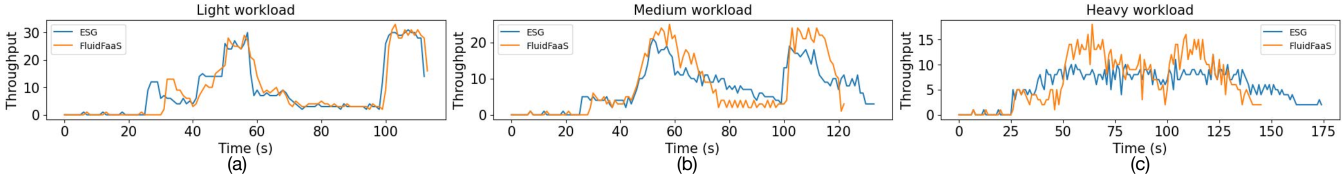


Figure 10: Throughput in different workloads.

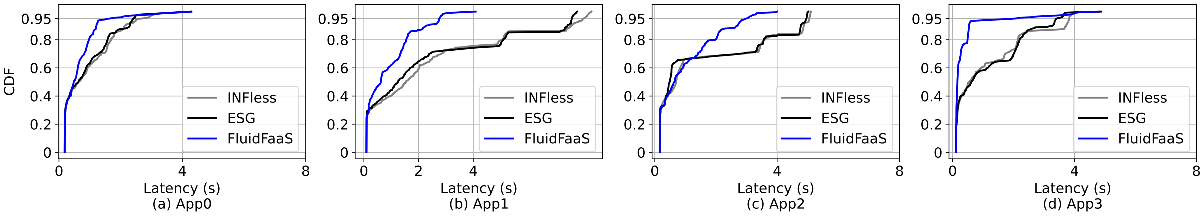


Figure 11: End-to-end latency distribution in the heavy workloads for each application.

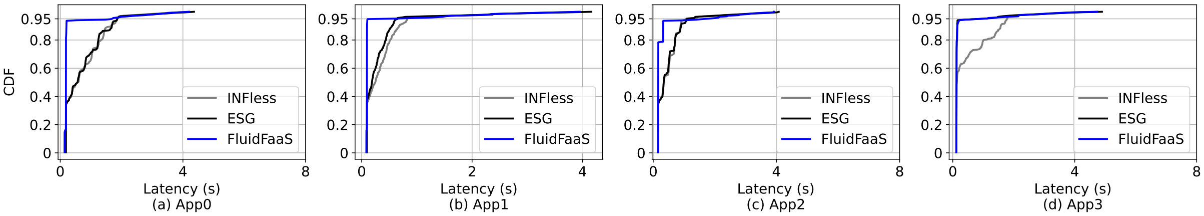


Figure 12: End-to-end latency distribution in the medium workloads for all applications.

Table 6: Resource cost comparison, where the result of FluidFaaS is normalized to 1. The lower, the better. INF is INFless and Fluid is FluidFaas.

Workload		Light workload			Medium workload			Heavy workload		
Method		INF	ESG	Fluid	INF	ESG	Fluid	INF	ESG	Fluid
Norm.	MIG time	0.95	0.96	1	0.93	0.99	1	0.94	0.97	1
	GPU time	1.08	1.07	1	1.06	1.05	1	1.17	0.99	1

expanded image classification application in the Baseline, so this variant is excluded from our study. We evaluate three different workloads, light, medium, and heavy, where each application is in small, medium, and large size respectively.

Traces. Following the prior work [32], we use the real-world traces from Azure Functions [47] to set the invocation frequencies and intervals of the serverless applications.

7 Evaluation Results

This section evaluates (i) the end-to-end performance of FluidFaas, compared with the state-of-the-art ESG and INFless method; (ii) its throughput and resource utilization improvement; (iii) the sensitive study for FluidFaaS with different MIG partitions.

7.1 End-to-End Performance

Figure 9 shows SLO hit rates across various workloads. It proves that FluidFaaS improves the SLO hit rate by 90% in medium workloads and 61% in heavy workloads across all applications while achieving a similar SLO hit rate as ESG in light workloads. ESG and INFLess adopt the non-pipeline execution model and delivery the similar performance in the medium and heavy workload. ESG outperforms INFLess 14% in SLO hit rates average in the light workload due to the advanced scheduling algorithm, as proven in ESG.

Table 6 shows the normalized GPU time and MIG time, as we introduced in Section 6, for all applications. FluidFaaS achieves these SLO hit rates improvements with lower GPU time as shown in Table 6, saving 6-17% than INFless and up to 6% saving than ESG accross various worklaods. The MIG time remains comparable, with a maximum difference of 7%. Since ESG and INFless share the same non-pipeline execution model, and ESG outperforms INFless,

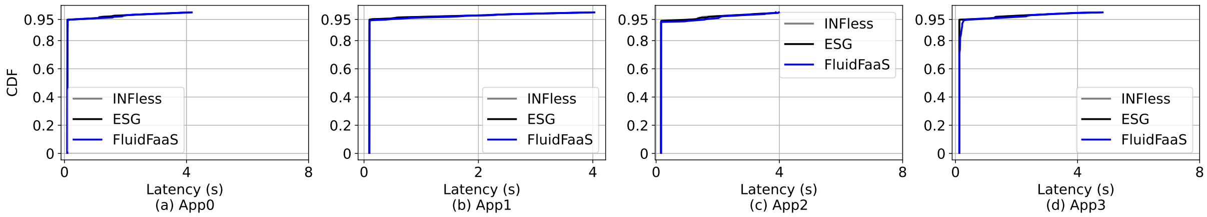


Figure 13: End-to-end latency distribution in the light workloads for all applications.

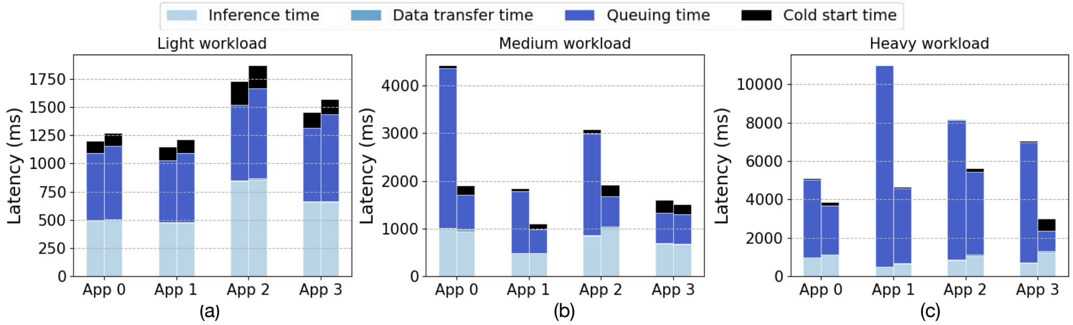


Figure 14: End-to-end latency breakdown. Left bar is for ESG and right one is for FluidFaaS.

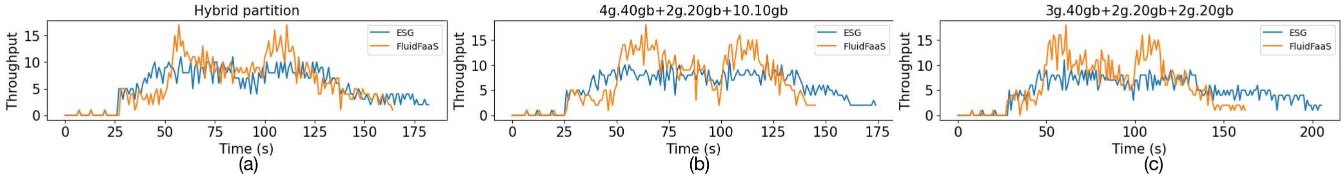


Figure 15: Throughput in different partitions.

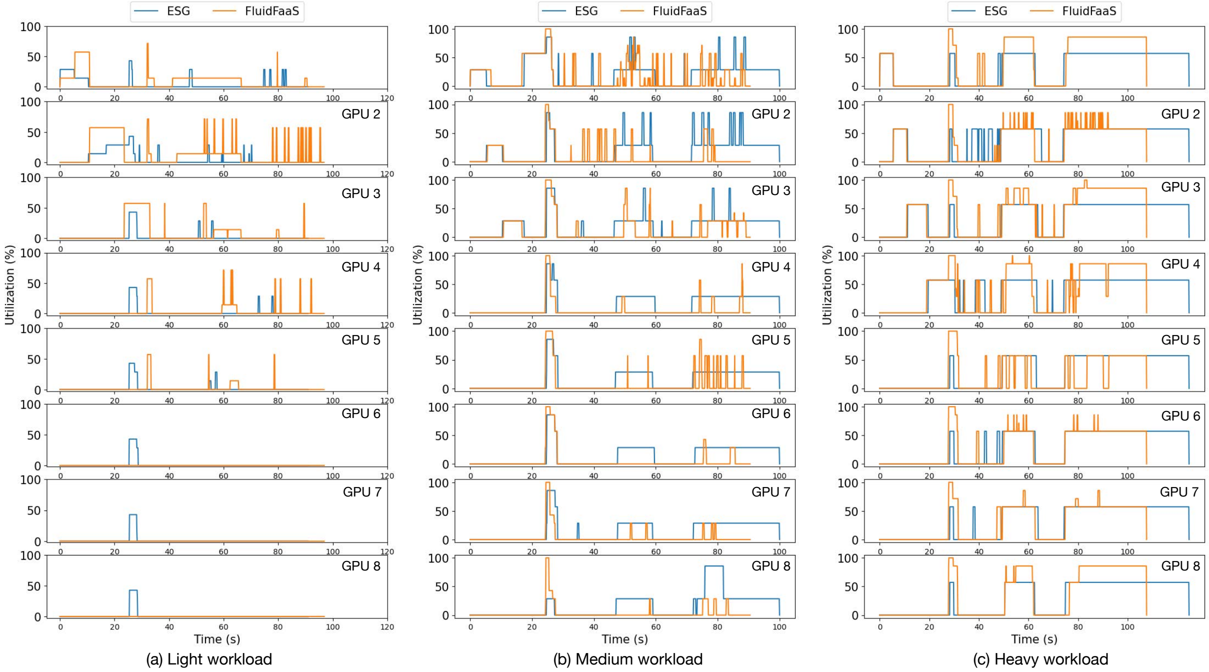


Figure 16: GPU utilization in different workloads.

we use ESG as the baseline for detailed analysis to better illustrate throughput and utilization.

Figure 10 illustrates system throughput across different workloads. FluidFaaS achieves a 75% higher throughput in heavy workloads and a 25% increase in medium workloads while maintaining similar throughput in light workloads. This higher throughput allows FluidFaaS to significantly reduce end-to-end latency, as shown in Figure 11, which illustrates the Cumulative Distribution Function (CDF) across all application in the heavy workload. We can see that FluidFaaS achieves 83.3% reduction in P95 tail latency for depth recognition in heavy workloads compared to ESG and INFless. For other applications, it achieves at least a 50% reduction in the P95th tail latency. Specifically, FluidFaaS improves the P95 tail latency by up to 81% in heavy workloads and 70% in medium workloads while maintaining similar latency in light workloads, detailed latency cdf for medium workloads and light workloads are shown as Figure 12 and Figure 13.

7.2 Detailed Analysis

As described in Sec.6, each GPU is divided into 4g.40gb, 2g.20gb, and 1g.10gb slices. In a light workload, all slices can host functions, resulting in similar performance between FluidFaaS and ESG. In medium workloads, ESG cannot utilize the 1g.10gb slices due to memory limitations, whereas FluidFaaS can leverage all slices thanks to its DAG partitioning and pipeline construction. In heavy workloads, ESG can only use the 4g.40gb slices, while FluidFaaS utilizes both the 4g.40gb and 2g.20gb slices.

Figure 16 shows the GPU utilization in the different workloads. As illustrated in Figure 16 (c), FluidFaaS improves GPU utilization by 75% compared to ESG during task bursts, which enables the higher throughput observed in Figure 10 (c). In this scenario, FluidFaaS can utilize a total of 7g.70gb, while ESG is limited to 4g.40gb, resulting in a 75% increase in throughput. In the medium workload, the ESG is limited to using 4g.40gb and 2g.20gb slices, while FluidFaaS can utilize all available slices, resulting in a 25% increase in throughput. The throughput is similar in the light workload scenario because each GPU has a comparable utilization level.

The high throughput allows FluidFaaS to complete all tasks 10% faster in medium workloads and 17% faster in heavy workloads compared to ESG, as shown in Figure 10(b) and (c). Thus, the earlier finish time leads to the GPU time saving. Additionally, this increased throughput results in shorter queuing times, as indicated in Table6, positively impacting both end-to-end latency and SLO hit rates.

7.3 Breakdown

Figure 14 illustrates the time breakdown of various workloads for ESG (left) and FluidFaaS (right). In the light workload, both FluidFaaS and ESG perform similarly, with execution times within the SLO limit for each application. However, in the medium and heavy workloads, FluidFaaS demonstrates up to 2.36 times lower latency than ESG. This reduction in latency for FluidFaaS leads to higher SLO hit rates, while the longer latency experienced by ESG in these workloads results in lower SLO compliance, as shown in Table 6.

It is evident that FluidFaaS consistently experiences a slightly longer data transfer overhead—ranging from 10ms to 40ms—due to communication between pipeline stages, compared to ESG's 1–5ms. Nevertheless, this overhead is offset by FluidFaaS's significantly shorter queuing times. ESG incurs much longer queuing times than FluidFaaS, with delays up to three times longer, ranging from hundreds to thousands of milliseconds, as depicted in Table 6. Therefore, the data transfer overhead introduced by the pipeline in FluidFaaS is marginal compared to the substantial benefits it brings in reducing overall latency and improving performance under heavier workloads.

Table 7: Different MIG partition

Name	Partition
Hybrid	1 * [1g.10gb *7] 2 * [2g.20gb*3+1g.10gb] 4 * [3g.40gb+4g.40gb] 1 * [4g.40gb+2g.20gb+1g.10gb]
P1	8 * [4g.40gb+2g.20gb+1g.10gb]
P2	8 * [3g.40gb+2g.20gb+2g.20gb]

7.4 Sensitive Study

In this section, we evaluate how FluidFaaS performs under different MIG partitions. In the above evaluations, each GPU is partitioned into one 4g.40gb instance, one 2g.20gb instance, and one 1g.10gb instance. We also experimented with a hybrid partitioning scheme—where each GPU has different partitions—and an alternative uniform partitioning scheme that divides each GPU into one 3g.40gb instance and two 2g.20gb instances, as detailed in Table 7.

As shown in Table 6, both FluidFaaS and ESG perform well under light workloads. However, under heavy workloads, FluidFaaS leverages fragmented resources to achieve better performance. Our sensitivity study demonstrates that FluidFaaS outperforms ESG not only in a single partition but across multiple partitions, especially when system resources are underutilized. Figure 15 illustrates that FluidFaaS surpasses ESG across all partitioning schemes. Specifically, in the hybrid partition, FluidFaaS achieves 70% higher throughput; in partitioning scheme P1, it achieves 75% higher; and in scheme P2, it reaches 78% higher throughput. These improvements stem from the fragmented small MIG slices that cannot be utilized by ESG but are effectively employed in pipelines by FluidFaaS.

8 Conclusion

This paper proposes FluidFaaS, the first solution that enables flexible GPU MIG management of the components within a serverless ML function while minimizing resource fragmentation and underutilization. It features novel programming system support, on-the-fly pipeline construction, and GPU-aware function state management. It demonstrates significant improvements in system throughput and SLO hit rates compared to existing solutions, underscoring the importance of considering GPU-specific characteristics in serverless computing environments. It opens new opportunities for flexible resource management in GPU-based serverless platforms.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Grants No. CNS-2312207. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] 2012. NVIDIA Multi-Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html.

[2] 2015. ResNet50. https://pytorch.org/hub/nvidia_deeplearningexamples_resnet50/.

[3] 2016. Apache OpenWhisk. How OpenWhisk works. https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openWhisk-works.

[4] 2016. OpenWhisk. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[5] 2017. DeblurGAN. https://github.com/pablodz/DeblurGAN.

[6] 2017. DEEPLABV3. https://pytorch.org/hub/pytorch_vision_deeplabv3_resnet101/.

[7] 2017. Fission: Open source Kubernetes-native Serverless Framework. https://fission.io/.

[8] 2017. OpenFaas. https://www.openfaas.com/.

[9] 2018. Knative. https://knative.dev/docs/.

[10] 2018. KNative Serving Activator. https://github.com/knative/serving/tree/main/pkg/activator.

[11] 2018. KNative Serving Autoscaler. https://github.com/knative/serving/tree/main/pkg/autoscaler.

[12] 2018. NVIDIA k8s-device-plugin. https://github.com/NVIDIA/k8s-device-plugin/.

[13] 2020. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[14] 2020. NVIDIA Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[15] 2020. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[16] Hervé Abdi. 2010. Coefficient of variation. *Encyclopedia of research design* 1, 5 (2010), 169–171.

[17] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*. 263–274.

[18] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. 2020. Using application knowledge to reduce cold starts in FaaS services. In *Proceedings of the 35th annual ACM symposium on applied computing*. 134–143.

[19] Vivek M Bhasi, Aakash Sharma, Rishabh Jain, Jashwant Raj Gunasekaran, Ashutosh Pattnaik, Mahmut Taylan Kandemir, and Chita Das. 2024. Towards SLO-Compliant and Cost-Effective Serverless Computing on Emerging GPU Architectures. In *Proceedings of the 25th International Middleware Conference*. 211–224.

[20] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Ra Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.

[21] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. In *The 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[22] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. 2017. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587* (2017).

[23] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. 2022. Sla-driven ml inference framework for clouds with heterogeneous accelerators. *Proceedings of Machine Learning and Systems* 4 (2022), 20–32.

[24] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*. 64–78.

[25] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110* (2020).

[26] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[27] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. 2023. FaST-GShare: Enabling Efficient Spatio-Temporal GPU Sharing in Serverless Computing for Deep Learning Inference. *arXiv preprint arXiv:2309.00558* (2023).

[28] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

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

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[31] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with OpenLambda. In *8th USENIX workshop on hot topics in cloud computing (HotCloud 16)*.

[32] Xinning Hui, Yuanchao Xu, Zhishan Guo, and Xipeng Shen. 2024. ESG: Pipeline-Conscious Efficient Scheduling of DNN Workflows on Serverless Platforms with Shareable GPUs. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 42–55.

[33] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.

[34] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504.

[35] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. 2017. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4681–4690.

[36] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*. 173–189.

[37] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 782–796.

[38] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. 2023. DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 57–72.

[39] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 161–175.

[40] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware data passing for chained serverless applications. In *USENIX Annual Technical Conference (USENIX ATC)*.

[41] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 303–320.

[42] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–28.

[43] Xuebin Qin, Zichen Zhang, Chenyang Huang, Masood Dehghan, Osmar Zaiane, and Martin Jagersand. 2020. U2-Net: Going Deeper with Nested U-Structure for Salient Object Detection. *Pattern Recognition* 106, 107404.

[44] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. 2020. Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer. arXiv:1907.01341

[45] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM symposium on cloud computing*. 1–17.

[46] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. 2024. FUSIONIZE++: Improving Serverless Application Performance Using Dynamic Task Inlining and Infrastructure Optimization. *IEEE Transactions on Cloud Computing* (2024).

[47] 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[48] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1075–1092.

[49] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. 2021. Wavelet: Efficient DNN training with tick-tock scheduling. *Proceedings of Machine Learning and Systems* 3 (2021), 696–710.

[50] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1288–1296.

[51] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 69–85.

[52] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. 2024. StreamBox: A Lightweight GPU SandBox for Serverless Inference Workflow. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 59–73.

[53] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[54] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan : Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[55] 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*. 768–781.

[56] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. 2017. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 1–13.

[57] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. 2023. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. *arXiv preprint arXiv:2306.03622* (2023).

[58] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-grained GPU sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems* 2 (2020), 98–111.

[59] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 30–44.

[60] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2023. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1–14.

[61] Qi Zhu, Bo Wu, Xipeng Shen, Li Shen, and Zhiying Wang. 2017. Co-Run Scheduling with Power Cap on Integrated CPU-GPU Systems. In *31st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Orlando, FL.

Received 6 February 2025; accepted 6 April 2025