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

Xinning Hui, Xipeng Shen (North Carolina State University)
Yuanchao Xu, (University of California, Santa Cruz)

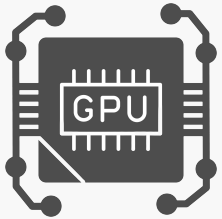
HPDC' 2025 Jul

01 Introduction and Problem Overview

Introduction

Growing Interest in ML Inference on Serverless Platforms with GPU support:

- Ease of programming, maintenance, autoscaling, and pay-as-you-go billing



GPU Sharing with MPS and MIG:

MPS (Multi-Process Service):

- Suffers from weak isolation (share GPU's memory and compute resources)
- Causes performance interference and security concerns

MIG (Multi-Instance GPU):

- Provides strong isolation (separate compute, memory, data paths)
- No performance interference from other process

- To overcome these issues with MPS, recent research and industry (e.g., Kubernetes) have shifted focus to MIG (Multi-Instance GPU), which offers strong isolation by hardware partitioning.

Problem Statement

LImitations of Previous Research/MIG:

- Fragmented GPU Resources
    a. MIG slices across GPUs are often small and scattered
    b. No single MIG slice may be large enough when needed → function has to wait.

- Rigid MIG Reconfiguration
    a. Changing MIG layout takes several minutes
    b. Incompatible with the dynamic and bursty nature of serverless workloads

- Monolithic Function Design
    a. Prior systems (e.g., ESG) require: Entire function must fit on one MIG slice
    b. This leads to: Wasted resources, Long delays, Poor scalability

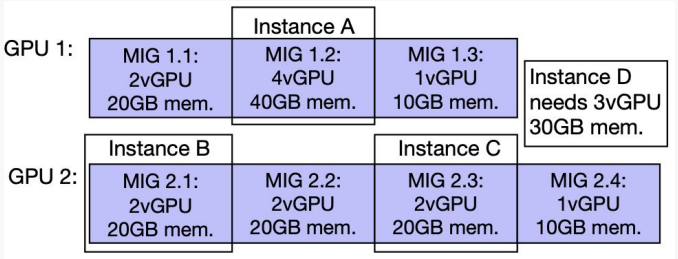


Figure : Current MIG support in serverless computing cannot make serverless function "instance D" utilize idle resources fragmented into multiple MIGs.

02 Background and Related Work

Serverless Platform

Serverless Platform

- **Platforms:** OpenWhisk, Knative, OpenLambda, OpenFaaS
- **Controller:** Scales functions up/down based on load
- **Load Balancer:** Routes requests to available invoker nodes
- **Invoker:** Runs function instances; manages job queues

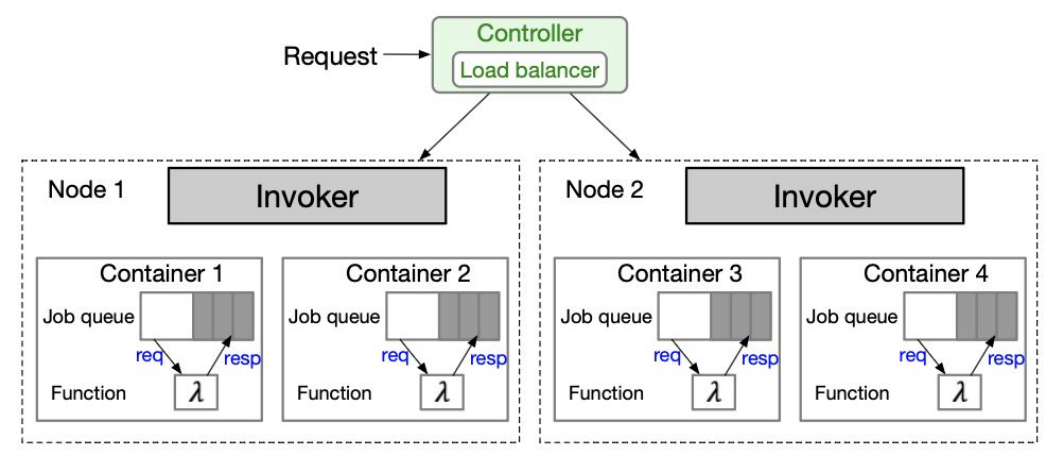


Figure: Overview of existing serverless platforms.

Related Work – Why Existing Solutions Fall Short

INFless, Protean, Llama:

- Use MPS for GPU sharing
- Suffer from interference due to weak isolation

FaaSwap

- Uses model swapping to time-share GPUs
- Incurs warm-start delays and reloading overhead

ESG, Miso:

- Use MIG for stronger isolation
- Treat functions as monolithic units
- Can’t combine idle fragmented slices (e.g., 1g.10gb + 2g.20gb)

StreamBox, Orion:

- Stream-level or application-level GPU sharing
- Optimized for DAG-based inference workflow composed of fine-grained functions
- No isolation

03 Motivation Analysis

Underutilization of GPU Resources in MIG-based Serverless

Setup:

- Workloads: DNN inference pipelines from Azure Functions trace
- Platform: ESG (MIG-enabled serverless scheduler)
- Hardware: A100 GPU with 3 MIG slices – 4g.40gb, 2g.20gb, 1g.10gb

Key Observation:

- ESG considers a GPU "utilized" if any one MIG is active
- At peak load (83s), ESG's resource demand exceeds ideal by 167%

Root Cause:

- Fragmented small slices (e.g., 1g, 2g) remain idle
- ESG's static, monolithic scheduling can't utilize them

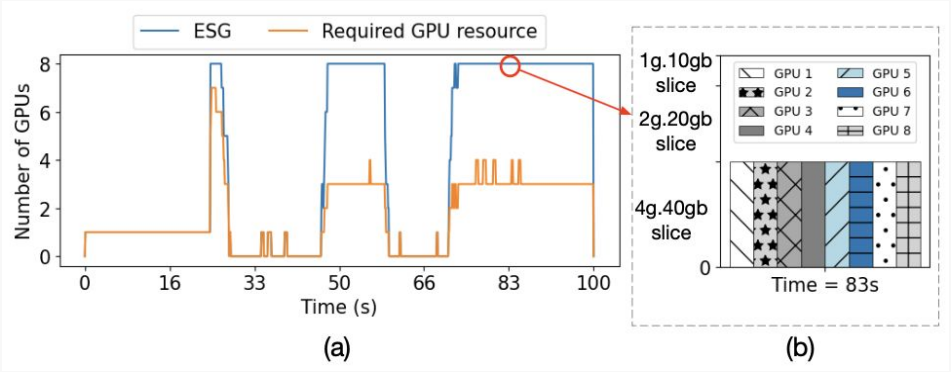


Figure: (a) GPU utilization in ESG and the required GPU resources. For ESG, a GPU is considered utilized if it is processing requests. (b) MIG usage at the 83rd second.

Problem: Fragmentation

Resource Fragmentation:

- MIG slices can't be flexibly recombined at runtime
- Even if aggregate capacity is enough, function fails if no single slice matches
- Example: 2g.20gb + 1g.10gb idle, but request needs 3g.40gb → must wait

Illustrated in Figure:

- GPUs with idle slices can’t serve new requests
- Forces unnecessary allocation of additional GPUs

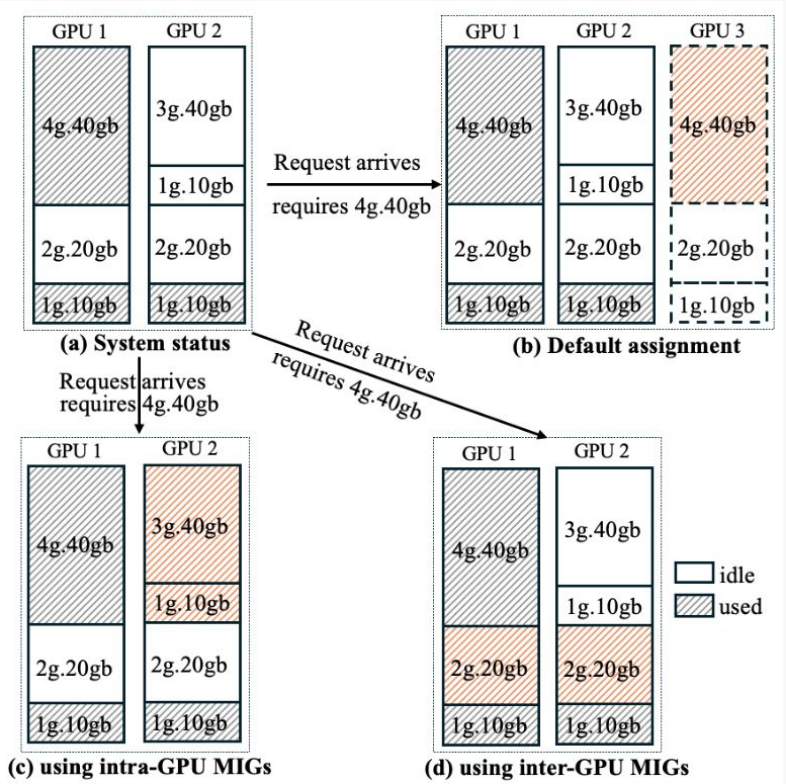


Figure: Illustration of GPU resource fragmentation

Problem: Exclusive Keep-Alive

Exclusive Keep-Alive Policy

- Once a model is loaded, its MIG slice is locked — even when idle
- Prevents sharing and reuse by other functions

Empirical Finding:

- Avg. active MIG usage = 16.1%
- For 90% of time, MIGs used <35% of their capacity

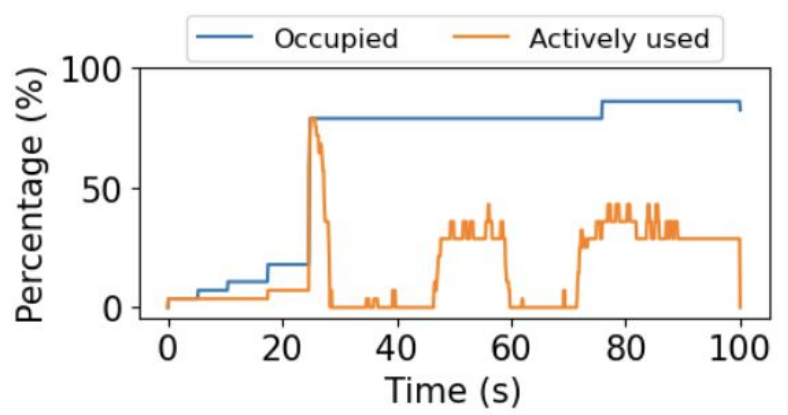


Figure: The occupied and actively used GPU percentage

Proposed Solution and Key Challenges

Solution:

- FluidFaaS dynamically splits a serverless function into pipeline stages, each assigned to a fragmented MIG slice.
- This enables fine-grained GPU utilization without requiring costly MIG reconfiguration.

Challenges:

1. **Lack of Dynamic Pipeline Support**
    - Existing serverless platforms require static function containers.
    - No native support for runtime DAG construction and resource-aware execution.
2. **Cross-MIG Communication Overhead**
    - MIG slices are hardware-isolated.
    - Communication between stages must go through CPU memory, adding latency.
3. **Complexity in Scheduling & Scaling**
    - Functions may run in multiple forms (pipeline vs. full).
    - Scheduling must handle eviction, instance heterogeneity, and meet SLOs.

04 Design of FluidFaaS

FluidFaaS Overview

Break a serverless function into smaller stages and run them on fragmented MIG slices as a dynamic pipeline

- Splits functions into DAG-based pipeline stages
- Maps each stage to available fragmented MIG slices
- Dynamically constructs and schedules GPU pipelines at runtime
- Addresses MIG fragmentation and rigidity in GPU resource allocation

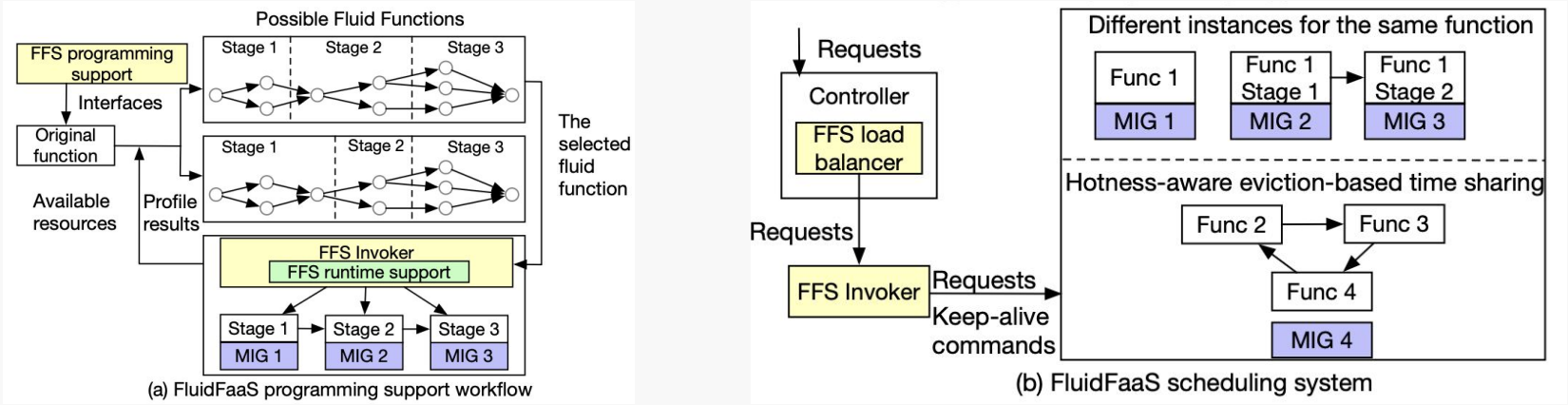


Figure 6: Overview of FluidFaaS.

Automatic Pipeline Instance Construction

A. Programming Support

- Breaks monolithic functions into components using a DAG.
- Each node in the DAG is profiled for memory and execution time on MIG slices.
- Invoker assigns available MIGs to nodes at runtime and stores this config.

a. Programming Interface

- Use **FluidFaaS.Module** instead of **nn.Module**.
- Register components with **.reg()** to build the DAG.
- Use **FFaaS** to manage DAG creation and execution.
- Modes:
  - **BUILDDAG**: Build DAG and profile DAG offline.
  - **RUN**: Load DAG and execute with MIG mapping.

```
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
def MyHandler_buildDAG (event, context, BUILDDAG):
    fluidFaaS = MyFFaaS (event, context, BUILDDAG)
    profiles = MyFFaaS.profile()
    ...
```

Figure: Illustration of the programming of a FFaaS function.

Automatic Pipeline Instance Construction

b. Runtime Support of FFaaS

- Loads DAG and MIG assignments in RUN mode.
- Spawns one process per stage on assigned MIG slices.
- Uses shared CPU memory to pass data between stages.
- Manages execution and termination with minimal developer code.

```
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
```

Each stage runs as an independent process, passing data via shared memory like steps in a pipeline — all managed by FFaaS internally.

Figure: Implementation of the core runtime support FFaaS

Automatic Pipeline Instance Construction

B. Runtime Support in Invoker

Goal: Dynamically construct and schedule balanced pipelines using available MIG slices.

Invoker as Local Scheduler:

- Runs on each GPU node independently
- Constructs pipelines based on real-time resource availability and offline profile data
- Assigns MIG slices to DAG components
- Operates without relying on the central controller

Pipeline Construction Strategy:

- All pipeline partitions are pre-evaluated offline for various combinations of resources
- Applies Coefficient of Variation (CV) for balance: *CV = std(t1,...,tn)/mean(t1,...,tn)*
- At runtime, selects the most balanced, MIG-fit pipeline

[DAG + Profile] + [Available MIGs] → [Select best-fit pipeline] → [Write Config for each Slices]

Hotness-aware Eviction-based Time Sharing

Problem: GPU slices remain underutilized due to fixed warm bindings.

Solution: Dynamically evict low-usage instances to share MIGs.

Multi-level Keep-Alive States:

Exclusive Hot:

- High-load instances stay fully loaded on MIG.
- Not evicted; ensures fast access and SLO compliance.
- All pipeline stages are in this state.

Time Sharing:

- Low-utilization (<30%) instances.
- Stored in CPU memory if idle (warm)
- Can be evicted and reloaded as needed.
- Terminated if inactive (cold).

Eviction Policy:

- Evict least recently used (LRU) time-sharing instance if MIG is needed.
- Reload instance from warm state (CPU) if accessed again.

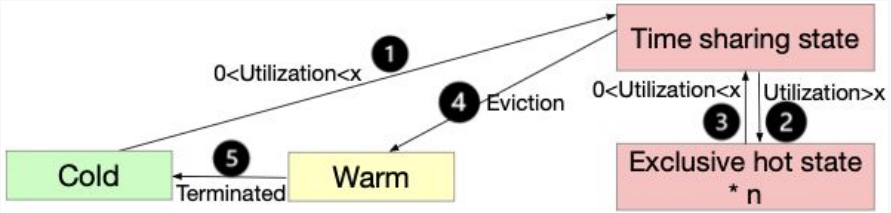


Figure: The state transition details.

05 Evaluation & Results

Evaluation Methodology

Baseline:

- **ESG:** State-of-the-art MIG-aware scheduler
- **INFless:** Strong MPS-based GPU sharing for inference (added MIG support)

Testbed Setup:

- 2 nodes, each with:
  - 8 × NVIDIA A100 80GB GPUs
  - 4 × AMD EPYC 7763 CPUs, 1440 GB RAM
- MIG Partition (default): 4g.40gb, 2g.20gb, 1g.10gb per GPU

Workload Traces:

- Real Azure Functions invocation traces
- Each app tested in 3 sizes: small, medium, large

Applications (from ESG)

- App 0: Image classification
- App 1: Depth recognition
- App 2: Background elimination
- App 3: Expanded image classification (DAG with branching)

SLO Configuration:

- **SLO Scale =** 1.5× the runtime of isolated execution
- Measures whether inference finishes within latency targets

End-to-End Performance

SLO Hit Rate:

- FluidFaaS consistently outperforms both baselines in medium and heavy workloads:
  - Up to 90% improvement in SLO hit rate (vs. INFless)
  - Similar performance to ESG in light workloads

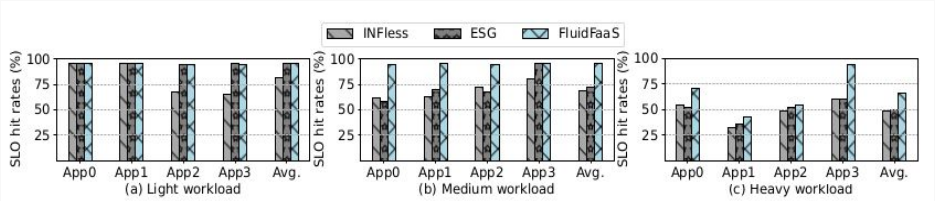


Figure: SLO hit rate in different workloads for each application

GPU Time & MIG Time:

- FluidFaaS uses less GPU time than INFless across all workloads:
  - Up to 17% reduction (vs. INFless in heavy workload)
  - Up to 6% reduction compared to ESG
- MIG time is similar across all systems
  - Max difference: ≤7%
  - Confirms FluidFaaS maintains efficiency without increasing slice usage

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

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

End-to-End Performance

Throughput

- FluidFaaS improves throughput by:
  - +75% in heavy workloads
  - +25% in medium workloads
  - Similar performance as ESG in light workloads

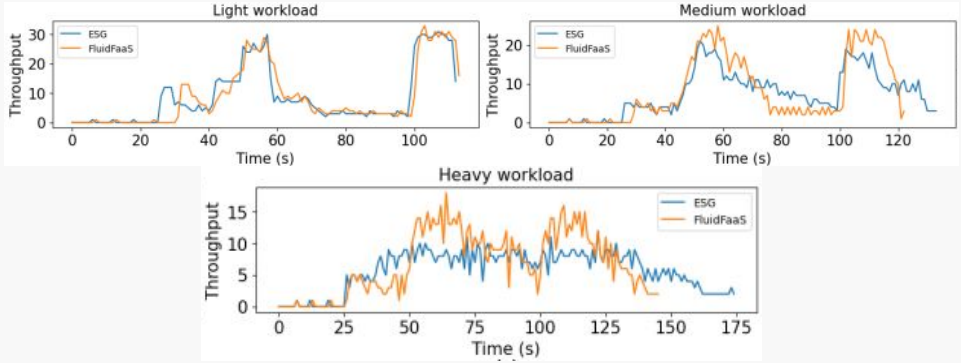


Figure: Throughput in different workloads

P95 Latency:

- 83.3% lower for depth recognition (App 1)
- ≥50% lower for other applications
- Overall: up to 81% reduction (heavy), 70% (medium)

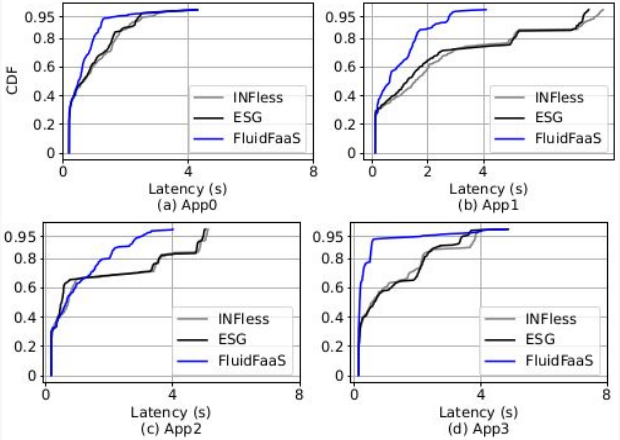


Figure: End-to-end latency distribution in the heavy workloads for each application

Improved GPU Utilization with FluidFaaS

In heavy workloads, FluidFaaS achieves 75% higher utilization than ESG

- Medium workloads: +25% higher
- Light workloads: similar across all systems

Why?

- ESG can only use large MIG slices (e.g., 4g.40gb)
- FluidFaaS utilizes all slices (4g, 2g, 1g) via DAG pipelining

Result:

- Tasks complete 17% faster in heavy workloads
- Less queuing, more throughput, better GPU efficiency

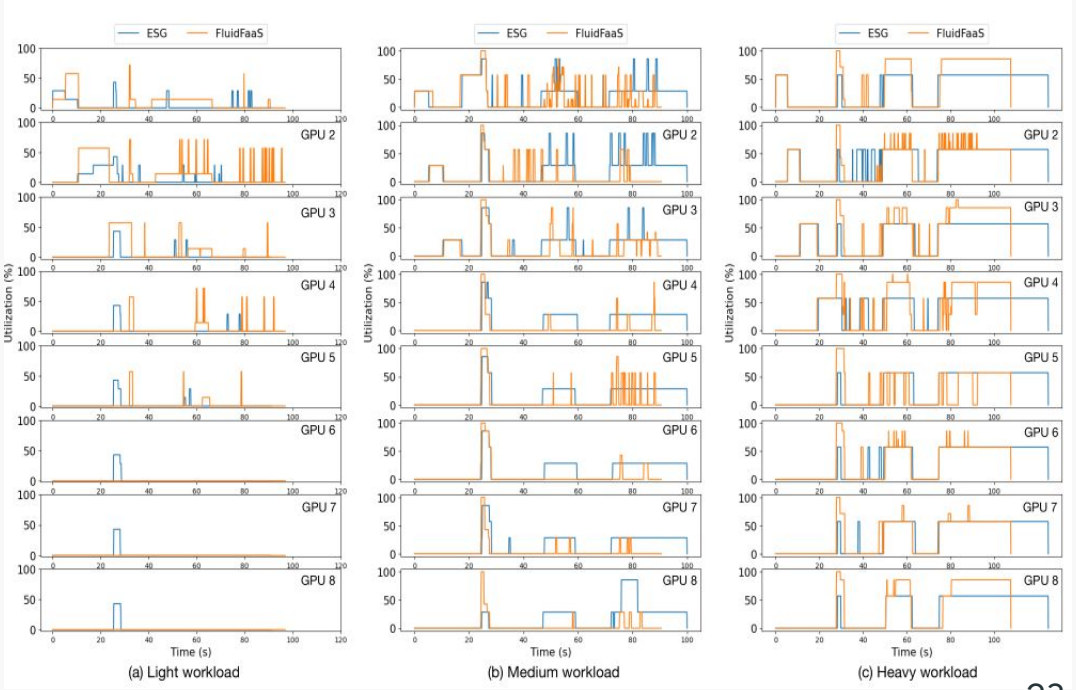


Figure: GPU utilization in different workloads

End-to-End Latency Breakdown

Workload Behavior:

- Light workload: FluidFaaS and ESG perform similarly (both meet SLOs)
- Medium & heavy workloads: FluidFaaS shows up to 2.36× lower latency

Why FluidFaaS is Faster:

- Slightly higher data transfer overhead (10–40ms)
- But much lower queuing delay (up to 3× shorter than ESG)
- Pipelines allow faster response despite stage-wise execution

Net Benefit:

- Reduced queuing outweighs transfer overhead → higher SLO compliance

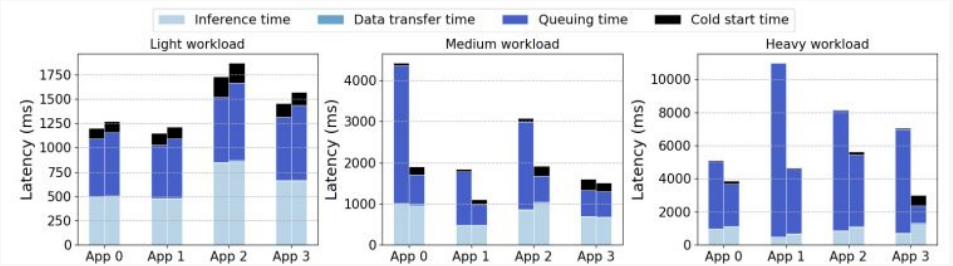


Figure: End-to-end latency breakdown. Left bar is for ESG and right one is for FluidFaaS

05 Conclusion & Takeaways

Conclusion & Takeaways

Problem:

- MIG-based GPU sharing suffers from fragmentation and rigid allocation
- Serverless workloads need dynamic, fine-grained GPU usage

FluidFaaS Solution:

- Enables pipeline-based execution across fragmented MIG slices
- Introduces hotness-aware eviction for time-sharing without compromising isolation

Results:

- Up to 90% higher SLO hit rate
- 75% higher throughput, 81% lower P95 latency
- Significant GPU time savings vs. ESG and INFless

Key Takeaway:

- FluidFaaS achieves high resource efficiency and performance in serverless ML inference by embracing flexible, runtime-adaptive GPU scheduling

Thanks!