Exploring Function Granularity for Serverless Machine Learning Application with GPU Sharing

XINNING HUI, North Carolina State University, USA
YUANCHAO XU, University of California, Santa Cruz, USA
XIPENG SHEN, North Carolina State University, USA

Recent years have witnessed increasing interest in machine learning (ML) inferences on serverless computing due to its auto-scaling and cost-effective properties. However, one critical aspect, function granularity, has been largely overlooked, limiting the potential of serverless ML. This paper explores the impact of function granularity on serverless ML, revealing its important effects on the SLO hit rates and resource costs of serverless applications. It further proposes adaptive granularity as an approach to addressing the phenomenon that no single granularity fits all applications and situations. It explores three predictive models and presents programming tools and runtime extensions to facilitate the integration of adaptive granularity into existing serverless platforms. Experiments show adaptive granularity produces up to a 29.2% improvement in SLO hit rates and up to a 24.6% reduction in resource costs over the state-of-the-art serverless ML which uses fixed granularity.

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

Additional Key Words and Phrases: Cloud computing, Serverless Computing, Quality of Service, Function-as-a-Service, Function Granularity, Machine Learning for Systems, Deep Learning

ACM Reference Format:
Xinning Hui, Yuanchao Xu, and Xipeng Shen. 2025. Exploring Function Granularity for Serverless Machine Learning Application with GPU Sharing. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 1, Article 6 (March 2025), 28 pages. https://doi.org/10.1145/3711699

1 Introduction

Recent years have witnessed increasing interest in leveraging serverless platforms [15, 17, 21, 22, 77, 78] to provide a machine learning inference service (called *serverless ML* for short) thanks to its cost-effectiveness and autoscaling properties. Many studies have proposed various methods to improve performance and reduce cost [15, 27, 34, 37, 68, 69, 77, 78, 81] of serverless ML. Supporting ML inference on GPUs has been challenging in industrial serverless platforms due to the preliminary support of GPU sharing and intra-GPU isolation. Multi-Instance GPUs (MIG) [10], a feature that allows partitioning one GPU into multiple mutually isolated instances, was recently introduced to mitigate the limitation. It has prompted more interest in serverless ML [38, 46].

Despite the promising outcomes from these studies, one critical aspect, *function granularity*, has been overlooked. Function granularity refers to the size and scope of the serverless functions that make up an application, which acts as the unit of functionality and scheduling on a serverless

Authors' Contact Information: Xinning Hui, North Carolina State University, Raleigh, USA, xhui@ncsu.edu; Yuanchao Xu, University of California, Santa Cruz, Santa Cruz, USA; Xipeng Shen, North Carolina State University, Raleigh, NC, USA.

ACM 2476-1249/2025/3-ART6
https://doi.org/10.1145/3711699

platform. The function granularity could directly influence the scheduling flexibility, Service Level Objective (SLO), and resource usage, making it a pivotal factor in optimizing serverless computing environments. All prior studies ignore the impacts of function granularity, typically selecting a fixed granularity for all applications in their designs without fully understanding its impacts. Prior studies have left several important research questions (RQ) open:

- RQ1: Is the granularity choice important for the efficiency of serverless ML? How much does it affect the performance and resource cost, especially on modern GPUs in the presence of MIG?
- RQ2: From the perspectives of both users and cloud providers, how do different granularities affect the scheduling of serverless ML? How would the impact change with different SLOs, workloads, or an overall goal that weighs SLO and cost differently?
- RQ3: If granularity is crucial and no single granularity fits all situations, is it possible to predict the appropriate granularity, both spatial (across different applications) and temporal (across different requests for the same application), for each application request based on factors such as the application, workload, SLOs, job arrival rates, and system status?
- RQ4: What programming and runtime mechanisms are needed to incorporate adaptive granularity into serverless ML?
- RQ5: How much benefit can adaptive granularity bring to serverless ML?

This paper presents the first known in-depth exploration of function granularity of serverless ML, aiming to answer these research questions through a three-fold exploration.

First, we conducted an empirical study on the impact of function granularity on serverless ML, giving answers to RQ1 and RQ2. We explored the performance and cost implications of bursty and steady workloads across a spectrum of SLO settings, ranging from strict and moderate to relaxed, for four applications using six ML models. The exploration reveals that employing different granularities can lead to discrepancies of up to 53% in Service Level Objectives (SLO) and up to 54% in resource costs across various scenarios. This study confirms that varying function granularities result in significantly different scheduling outcomes, establishing that no granularity is suitable for all situations. Further in-depth analysis is conducted to uncover the reasons why granularity significantly influences serverless scheduling outcomes. This analysis identifies the fundamental factors that affect these outcomes and guides system design in selecting appropriate granularity.

Second, we explored the feasibility of creating a predictive model to dynamically predict the appropriate granularity for each application request under various conditions, addressing RQ3. Specifically, we investigated three types of predictors that have shown promising results in many system problems: deep reinforcement learning (DRL) [50], linear regression [58], and random forests [26]. The results confirm the feasibility of efficient prediction and demonstrate that adaptive granularity can accurately predict the appropriate granularity in different scenarios.

Third, we developed programming tools and runtime extensions to facilitate the adoption of adaptive granularity within existing serverless platforms, addressing RQ4. These programming tools are designed to automatically partition the original application into functions at various granularities, significantly reducing the programming efforts needed to develop functions at different granularities. The runtime extension provides adaptive granularity prediction and delivers functions at the predicted granularity to the existing serverless scheduler. This extension requires only minor modifications to the current serverless schedulers.

Finally, for RQ5, we integrate our programming tools and runtime extensions into two state-of-the-art serverless schedulers on GPUs [38, 78] to demonstrate the improved scheduling outcomes achieved by using adaptive function granularity. We evaluate these two schedulers, featuring

adaptive function granularity, across four ML inference applications on various workloads to quantify the impact of adaptive granularity.

Overall, this paper makes the following contributions.

- It provides the first in-depth study on the impact of function granularity on serverless ML and reveals a set of insights on the impact.
- It gives the first known exploration of the construction of predictive models for function granularity of serverless ML and confirms the feasibility through supervised and reinforcement learning approaches.
- It proposes a set of programming and runtime support to enable the integration of adaptive granularity for ML models in the current serverless platforms.
- It evaluates the effectiveness of adaptive granularity through a set of experiments and demonstrates significant benefits; compared to the state-of-the-art serverless ML, which uses fixed granularities, the SLO hit rates increase by as much as 29.2% and the cost decreases by as much as 24.6%.

2 Background

2.1 Serverless Platform

We use Apache OpenWhisk [11], a popular open-source serverless platform, to illustrate the architecture of serverless platforms. OpenWhisk's architecture [1, 70], depicted in Figure 1, comprises a frontend with RESTful interfaces and a Controller with a Load Balancer. The front end receives function invocations (i.e., function requests), while the Controller handles their distribution to Invokers.

Upon receiving a user request, the Controller retrieves the corresponding functions from the database, assesses the required resources for each function, and assigns these functions to the appropriate Invokers. The Invoker then initializes the execution environment within a container (i.e., function instance), after which the function is executed in this container. Each container is dedicated to a function. To execute multiple requests for the same function concurrently, OpenWhisk dynamically spawns multiple containers as needed.

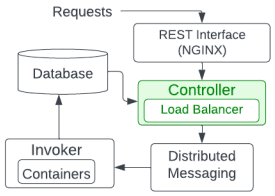

Fig. 1. OpenWhisk architecture.

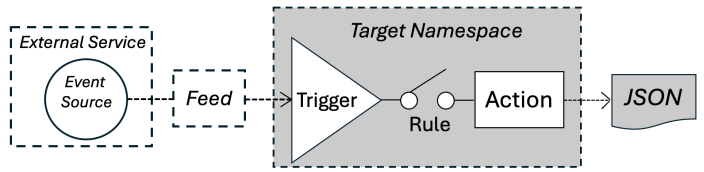

Fig. 2. Programming model of OpenWhisk.

2.2 Serverless Programming

The execution of a serverless application is event-driven. The development of a serverless application specifies the event triggers and corresponding actions. As illustrated in Fig. 2, a trigger represents an event from various event sources and is linked by a rule to an action, which embodies functional logic.

A real-world serverless application is typically implemented as a multi-stage workflow or a directed acyclic graph (DAG) [43]. Incoming requests invoke some serverless functions, the executions of which trigger other actions. Serverless platforms provide programming frameworks to specify the workflows, such as AWS Step Functions [2], Azure Durable Functions [3], or Google Cloud Workflows [6].

Take AWS Step Functions as an example. The user defines the workflow, also known as a state machine, in JSON by following the Amazon States Language (ASL). This JSON structure defines the various states of the workflow, their sequence, and how data transitions between them. States encompass various activities such as executing a Lambda function, branching decisions, wait periods, and parallel processing paths, and so on.

A state machine is usually invoked by a lambda function. We name the function as *bridge function*, which defines the input data, output data, start execution, and state machine ARN (Amazon Resource Name). ARN is a unique identifier assigned by AWS to a specific state machine. OpenWhisk is similar. Its Composer serves a similar purpose by creating a composer akin to a state machine to orchestrate workflows. Our work uses the bridge function to create functions with adaptive granularity, as explained in Section 5.1.

2.3 GPU Sharing and Multi-Instance GPU (MIG)

Modern GPUs (by NVIDIA) offer two mechanisms for spatial sharing: Multi-Process Service (MPS)[9] and Multi-Instance GPU (MIG)[8]. GPU sharing allows multiple processes to execute concurrently on a single GPU. MPS shares a single GPU context across multiple processes, which can result in resource contention and pose security risks for week isolation. It is not ideal for serverless computing, where isolated execution environments and security are crucial concerns. MIG partitions a single GPU into multiple hardware-isolated instances, providing better performance isolation and security. For instance, an A100 GPU's streaming multi-processors (SM) consist of seven graphics processing clusters (GPC). In MIG mode, each *slice* (used interchangeably with *MIG instance*) includes one or more GPCs and a certain amount of GPU memory. The A100 GPU, featuring seven GPCs, enables the following MIG instance configurations: 1g.10gb, 2g.20gb, 3g.40gb, 4g.40gb, and 7g.80gb, where '1g' stands for one GPC and '10gb' for its associated 10GB of GPU memory.

2.4 Existing Studies All Use a Fixed Function Granularity on GPUs

Several studies have explored serverless system designs on GPUs [38, 46, 48, 62, 78, 79] for ML inference. They however all employed a fixed function granularity. For example, some studies treat each ML model as a function. Examples include Infless [78], which explores dynamic batching and resource assignment with a heuristic algorithm, and ESG [38], which focuses on GPU sharing with a scalable and efficient resource assignment scheduler. Other studies regard the entire application as a single serverless function. Examples include Miso [46], which examines container placement strategies to enhance GPU utilization, and Simppo [62], which uses the entire application, consisting of multiple ML models to reduce data transfer and improve resource efficiency. Additionally, some studies use model partitioning to refine function granularity. For instance, Gillis [79] serves large neural networks by automatically partitioning models at the block level, while Tetris [48] improves memory efficiency by splitting models at the tensor level to reduce redundancy.

All prior work, however, applies a fixed function granularity to all applications and requests on GPUs, leaving the impact and possibility of adaptive selection of different granularities yet to be understood. This work attempts to fill the void.

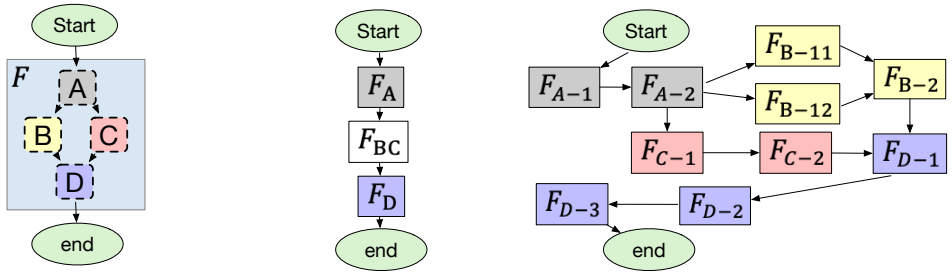

Fig. 3. The example of different granularities for one DAG application with four ML models. (a) application granularity, the whole application as one function; (b) stage granularity, each model or a group of models as one function; (c) block granularity, the model parallelism is explored and each block as one function.

3 Empirical Studies on the Impact of Granularity

We attempt to answer RQ1 and RQ2 (as listed in Section 1) about the impact of granularity through empirical studies.

3.1 Experiment Design

Our experiments compare the execution of various serverless ML workloads using different function granularities.

Serverless Platform. We use the popular open-source platform, OpenWhisk [11], as the experimental platform. More details are in Section 2. Similar to other serverless platforms, by default, OpenWhisk does not support GPU sharing or MIG. To overcome this limitation, we adopt ESG [38], a state-of-the-art scheduling algorithm designed for MIG-enabled serverless computing. ESG adopts dynamic batching and minimizes data transfer and cold start delays through data-locality-aware scheduling and pre-warming techniques, helping us assess the full potential of different granularities. Without these optimizations, the overhead from data transfers and cold starts could substantially throttle the performance benefits of fine granularities.

Granularities. We evaluate three representative granularities: **application**, **stage**, and **block**, as shown in Figure 3. *Application* is the largest granularity, making the entire application a single serverless function, as shown in Figure 3 (a). *Stage* is smaller, making each model or a group of models in the pipeline (or DAG) of the application workflow a serverless function, as shown in Figure 3 (b). We adopt the dominator-based methods for DAG partition from ESG [38] and divide the DAG into several self-contained stages.

We also introduce a third granularity, *block*, which further breaks down each ML model into smaller serverless functions based on the model architecture, as shown in Figure 3 (c). There are many possible ways to do that [32, 40, 75, 80, 82]. In our experiments, we try to partition an ML model evenly for load balance. The number of blocks affects the communication overhead and resource demand per function. Empirical measurement can help identify the right block size that gives a good overall tradeoff. In our measurements, we find the numbers shown in Table 1 give good tradeoffs to those models used in our experiments. For the interest of space, we will concentrate our discussions of *block* level on those settings.

Applications and Workloads. We use four Deep Learning-based applications as shown in Table 2. These applications were used in the state-of-the-art work [38]; using them allows us to compare head-to-head to prior work. Among these applications, three are compositions of multiple DNN

Table 1. ML models and the number of blocks they are partitioned into that give good tradeoffs in experiments.

Model	Source	Total blocks
Super-resolution	SRGAN [45]	2
Deblur	DeblurGAN [4]	3
Background removal	U^2 Net [61]	6
Segmentation	Deeplabv3_resnet50 [5, 24]	5
Classification	ResNet50 [12, 36]	4
Depth recognition	Midas [63]	6

processing stages in a sequence, and one (*Expanded image classification*) is extended with two branches to form a DAG. The DNN models used in those applications are already listed in Table 1.

Table 2. Applications

Applications	Composition
Image classification	Super resolution ->Segmentation ->Classification
Depth recognition	Deblur ->Super resolution ->Depth recognization
Background elimination	Super resolution ->Deblur ->Background removal
Expanded image classification (DAG)	Deblur ->(**if** low resolution: ->Super resolution; **else:** pass) ->Background removal ->Segmentation ->Classification

Following the practice in the recent studies [15, 51, 65], we use two workloads, **bursty** and **steady**, for the experiments. These workloads were generated based on the frequencies of job arrivals in the real-world Azure serverless traces [70]. For the **bursty** workload and **steady** workload, intervals are set such that they put the resource utilization in our serverless environment to 100%, 50%, and 25%, respectively, levels that are representative of what prior studies have used [28, 35, 54, 64]. The requests to the four applications arrive in a round-robin manner.

SLO Latency Requirement. SLO latency is an important requirement on a serverless workload. It defines the acceptable latency for the platform to respond 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 a unit GPU. We use *SLO level* when describing an SLO latency of a workload. It is defined as the ratio between the SLO latency and t. An SLO latency level of 0.8× refers to the case where the acceptable maximal latency is 0.8 times t. The settings of SLO latency follow the practice in recent literature [38]: For bursty workload periods, we adopt a relaxed SLO setting (1.2× and 1.5×); for normal workload conditions, more moderate and stringent SLO settings are used (0.8× and 0.5×), aligning with typical user expectations. Note that 0.5× and 0.8× performance targets are reasonable and achievable because the base is the performance when only one unit CPU and one unit GPU are used for the application. DNN applications typically have high parallel speedups.

Platform. We adopt an experimental framework used in a prior work [38] as the platform for this study. The framework can emulate various serverless workloads and scenarios. The emulations are based on the actual performance of the serverless functions measured on actual machines in various configurations (MIGs, batch size, CPU, and GPU resource allocations). The machine is as specified in Table 3. There is a 16-core CPU and an NVIDIA A100 GPU in each node. By default, the serverless platform regards a node consisting of 16 vCPUs and 7 vGPUs, with one vCPU corresponding to one CPU core, and one vGPU to one MIG instance. It is worth noting that multiple vGPUs (MIGs) can be used as a group to serve one serverless function through software methods [39], which is the base for our experiments when the scheduler assigns one lambda function to multiple vGPUs. The network bandwidth is 5Gbps, according to the bandwidth of single-flow traffic to and from AWS EC2 instances [7]. The hardware resource considered testbed in the emulations consists of 32 nodes, each equipped with 16 vCPUs and one A100 GPU (up to 7 vGPUs by MIG). To accommodate the impact of other runtime factors on the performance, the emulations add Gaussian noises to the

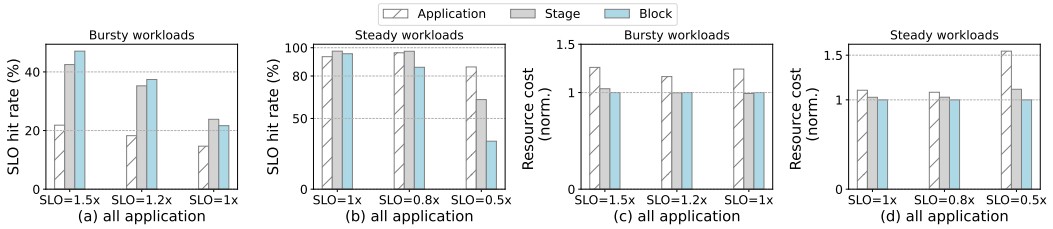

Fig. 4. The SLO hit rates and normalized resource costs for all applications in different SLO requirements and workloads.

performance. The emulation is equipped with a workload generator, which generates workloads by sampling the set of serverless functions randomly based on a specified arrival rate. The set of workloads we considered in the evaluation is detailed earlier. The scheduler and job dispatching implementation are based on the controller in OpenWhisk [1]. The framework uses proxy threads to monitor the function call intervals, predict subsequent invocations, and proactively warm-up instances so that most function calls (except first-time calls) can avoid cold-start delays.

Table 3. Experimental hardware configuration

CPU device	AMD EPYC 7302P 16-Core Processor
CPU Mhz	1499.866
CPU memory	128GB DDR4 3200MHz ECC DRAM
GPU device	NVIDIA A100 80GB
GPU memory	80GB
MIG instances	Up to 7 MIGs@10GB

3.2 Metrics and Measurement

We repeat the measurement 10 times and report the average.

Resource Cost. It is defined as execution time multiplied by the allocated resource and the unit price of a resource. Following AWS EC2 pricing [78], we set the unit price of a vCPU to 0.034$/hour (including the memory). Based on the pricing of an entire GPU on AWS, we divide it by 7 vGPUs and get the unit price of a vGPU as 0.67$/hour.

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

Our analysis also examines other metrics, such as GPU usage and job queueing time, for in-depth analysis.

3.3 Observations and Insights

3.3.1 Overall Observation. Fig. 4 shows the total SLO hit rates and resource costs of the workloads, for both bursty (a & c) and steady cases (b & d). In each case, there are three SLO latency requirements.

On the bursty workloads: When the SLO requirement is relaxed (1.2× and 1.5×), the *application* granularity uses the largest amount of resource but delivers the lowest SLO hit rates. In contrast, the *block* granularity achieves the highest SLO hit rates with the least resource consumption. It doubles the hit rates of the *application* granularity while reducing the resource cost by about 20-25%. When the SLO requirement is strict (1×), the *stage* granularity becomes the most favorite choice. It

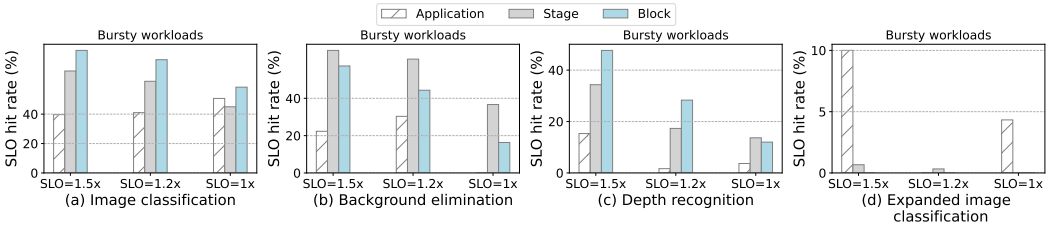

Fig. 5. The SLO hit rates for each application in the **bursty** workloads.

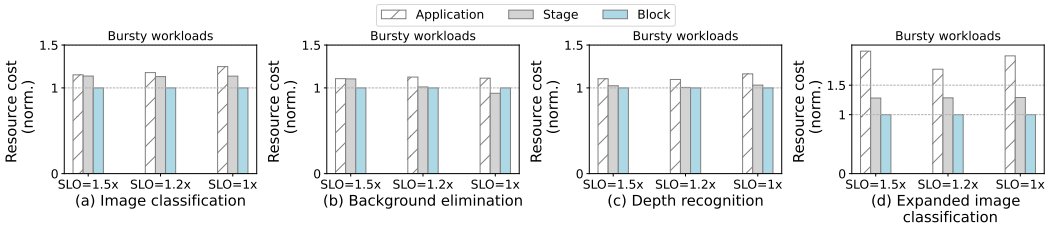

Fig. 6. The normalized cost for each application in the **bursty** workloads.

delivers the highest SLO hit rate (9% higher than *application*) while consuming only 79% of the resource used at the *application* granularity.

On the steady workloads: The differences between the different granularities are less pronounced except at the lowest SLO level (0.5×), where the *block* granularity is the least favorable choice. Compared to the *stage* granularity, it saves 10% resource but degrades the SLO hit rates by 30%.

More detailed results are shown in Figs. 5, 6, 7, and 8. The first two show the SLO hit rates and resource cost of each application in the bursty workloads, and the latter two show those in the steady workloads. The granularity preferences vary across the applications in both the bursty and steady workloads. In the bursty case, for instance, *image classification* achieves both the highest SLO hit rates and the lowest resource cost when it uses block granularity, while it is not the case for the *background elimination* application, for which, the stage granularity works substantially better. The applications also show different responses to the changes in the intensity of the workloads. For application *image classification*, for example, block granularity stays preferable as the workloads change from bursty to steady when the SLO factor is 1× or 0.8×. For *extended image classification*, the block was not delivering any SLO hits in the bursty case but becomes the most favorable choice in the steady case when the SLO factor is 1×.

The application, *expanded image classification*, is worth more discussions. It shows very low SLO hit rates in the busty case. The reason is that this application consists of the largest number of DNN models and, hence, the longest pipeline. In the stage and block cases, it gets the most serverless functions. As a result, for it to serve one request, it is subject to the longest waiting times in the queues. It hence gets minimum or no SLO hits, especially when the SLO factor is small. In the steady workloads, the queues are less occupied, and hence, the waiting time is lower, especially when the SLO factor is not too small, where, all granularities start to get most SLO hits.

We next provide an in-depth analysis of the impacts of function granularity.

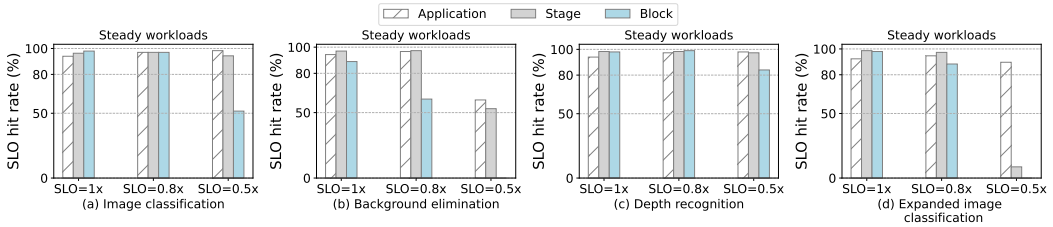

Fig. 7. The SLO hit rates for each application in the **steady** workloads.

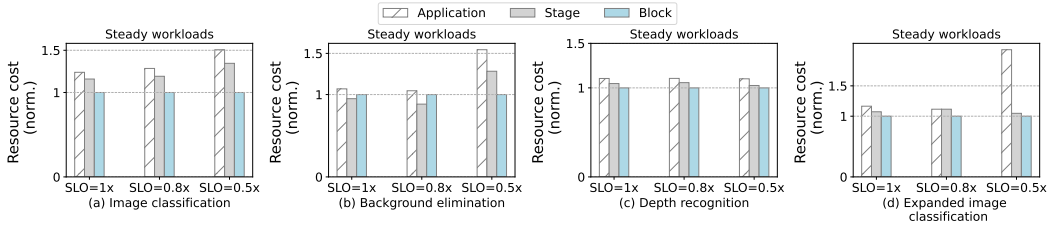

Fig. 8. The normalized cost for each application in the **steady** workloads.

3.4 In-Depth Analysis of the Impact of Function Granularity

The aforementioned results indicate that varying granularities can significantly affect SLO hit rates and resource consumption across different applications. We conduct an in-depth analysis to uncover the fundamental reasons behind, providing insights for system designs.

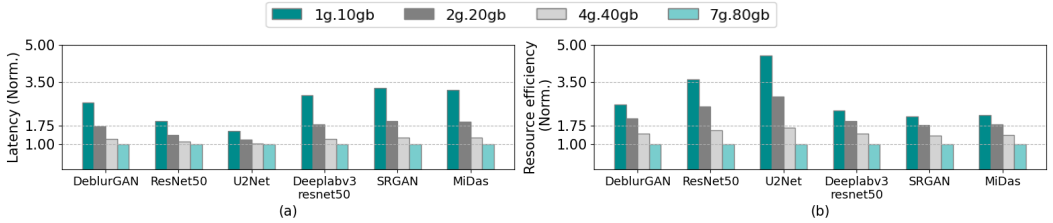

Fig. 9. The normalized execution latency (a) and resource efficiency (b) of different models. Resource efficiency is the number of requests that can be completed on the entire A100 GPU with various MIG slices for the same time period.

3.4.1 Variations across Models and Blocks. One explanation for a finer granularity outperforming a coarser granularity is the variability in scalability across ML models and model blocks on different MIG instances.

Figure 9 illustrates the execution latency and resource efficiency of the models on a full A100 GPU across different MIG slices. Resource efficiency is measured by the number of requests that can be completed on the entire A100 GPU with various MIG slices within the same time period. The execution latency represents the average latency of all requests completed during this time period.

The execution latencies of SRGAN and MiDas exhibit higher sensitivity to computing resources, with a 3.2x slowdown when resources are reduced to 1/7. In contrast, U2NET experiences only a 1.53x slowdown under the same conditions. From the resource efficiency perspective, as illustrated

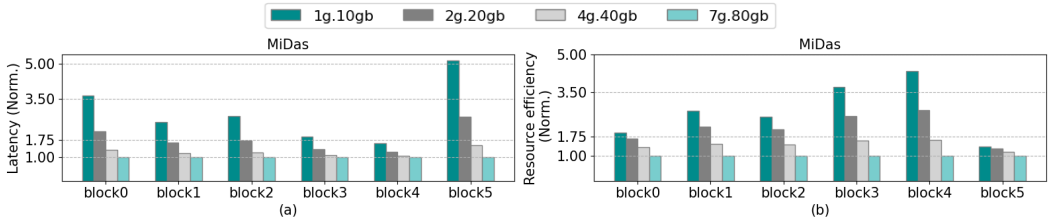

Fig. 10. The normalized execution latency and resource efficiency for each block of the depth recognition model.

in Figure 9 (b), all models demonstrate improved resource efficiency with a finer granularity. Notably, U2NET achieves significantly higher resource efficiency in the 1g.10gb setting, due to its small slowdown in execution latency on reduced resources.

Figure 10 shows the execution latency and resource efficiency of each block in MiDas, a depth recognition model. Similar variations are also observed across different blocks. The resource efficiency of block 4 is three times higher than that of block 5 on the 1g.10gb MIG slice. Other models exhibit similar patterns, but they are not depicted for the sake of space.

These variations suggest that using a coarser granularity may lead to resource over-provision. For example, the serverless platform assigns a fixed amount of resources to an invocation of a serverless function. If the amount of resource best fits the need of one part of the serverless function (e.g., one of the DNN models in an *application*-wise serverless function), it could be a less efficient setting for another part.

3.4.2 Analysis of Various Scheduling Outcomes. The above evaluation shows that different function granularities lead to different scheduling outcomes. In this part, we conduct a deeper analysis of how function granularity influences the scheduling outcome by analyzing GPU resource utilization, queueing time, and data transfer overhead.

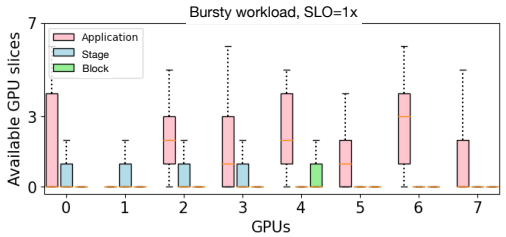

Fig. 11. The available GPU slices distribution of each GPU under the bursty workloads.

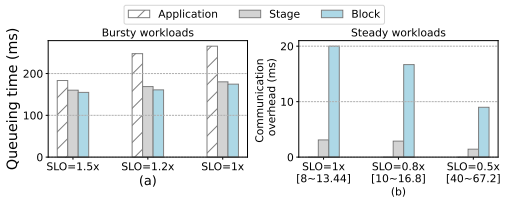

Fig. 12. (a) The average queueing time for all tasks. (b) The data transfer overhead. No data transfer in the application granularity, thus the overhead is 0.

Figure 11 illustrates the availability of vGPU slices on eight randomly sampled GPUs in the cluster under a bursty workload. Due to variations in availability over time during workload execution, boxplots are used to represent the distributions. The results indicate that the *application* granularity results in the highest number of idle vGPUs, whereas the *block* granularity leaves almost no vGPUs idle. This occurs because functions at the *application* granularity require large resources initially; if the currently available resources are insufficient, the scheduler may leave fragmented resources unused. Additionally, the *application* granularity often experiences longer queuing times as it requires accumulating enough resources.

Figure 12 (a) displays the average queueing time for a job before the controller can have enough resources and dispatch it. The strong resource constraints associated with *application* granularity result in a 19% longer queueing time compared to the *block* granularity. This increased queueing time adversely affects SLO hit rates.

The finer-grained functions are not always beneficial. Fig. 12 (b) shows the data transfer overhead of different granularity. The application granularity incurs no data transfer overhead, whereas the *block* granularity experiences a data transfer overhead of up to 100ms. This data transfer overhead increases the execution latency of applications and negatively affects the SLO hit rates.

3.5 Summary of Insights

In summary, the studies have revealed the following insights:

- Function granularity is crucial to the performance and cost of serverless ML, causing 10%–53% differences in SLO hit rates and up to 54% differences in costs. (Answers to RQ1)
- No granularity fits all situations. For a given system workload, the best granularity changes when the SLO level changes or the arrival rates of requests change. (Part-1 answer to RQ2)
- Different models show different scaling as the number of vGPUs increases, and similar variations manifest across different blocks of an ML model as well. It suggests that the best granularity differs across different applications. (Part-2 answer to RQ2)
- There is a tradeoff among the various effects of granularity. Based on the observations, we qualitatively summarize the effects in Fig. 13. As granularity increases, data locality becomes better, and hence, data transfer overhead decreases, but at the same time, flexibility for scheduling worsens, job waiting time lengthens, and resources become underutilized. The exact effects are sensitive to the SLO level and workload properties (e.g., job arrival rates). (Part-3 answer to RQ2)

It is worth noting that SLO hit rates and resource costs form a tradeoff. Sometimes, one would use a single *reward* function to combine them into a single reward score for assessment [83]. An example is a weighted sum of the two:

$$reward = \alpha \cdot SLO\ hits - costs$$

where, α is called *reward factor*. It is easy to see that in a given setting, different reward factor values could lead to different rankings of the granularities. The factors in Fig. 13 hence also include reward function as one of the influencing factors. We show how the reward factor affects the granularity choice in Section 6.

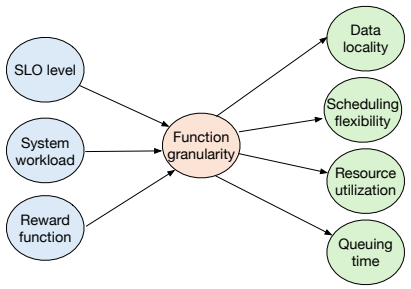

Fig. 13. Impacts and factors of granularity explored.

4 Predicting Granularity

The findings in the previous section suggest that the best function granularity varies across applications, and is affected by many runtime factors, such as current system workloads, resource availability, and the SLO requirement. The second exploration of our study is to investigate whether it is feasible to predict the appropriate granularity for each invocation of a serverless ML application during runtime. We use the three granularities mentioned in the previous section as the potential choices.

We have designed three ML-based predictive models. The first is multi-agent deep reinforcement learning (DRL), which does not require pre-training. This method emulates the human learning process, continuously exploring the space of choices and learning the relations between the state

of the environment and the best option to make. The other two are supervised learning methods: linear regression for its simplicity and interpretability, which provides a fast and efficient method to establish predictions [58]; and random forests for their robustness and capability of handling complex nonlinear relationships [26]. We choose these methods because (i) they have shown lots of success in handling runtime predictions in solving system problems [29, 49, 57, 73], (ii) unlike DNNs, they do not need a large amount of labeled data to train, (iii) they are lightweight models fast to run, which is important for our task as the prediction must happen on the fly. We next explain them.

4.1 Multi-agent Reinforcement Learning

It is based on Deep Q-Network (DQN) [31], a deep reinforcement learning (DRL) method. It does not require manual labels, unlike supervised learning, but actively explores the environment, learns the relations between actions and rewards automatically, and uses the learned model to predict the following suitable action. This nature makes it a good fit for the dynamic environment in our problem. Our design creates a DQN agent for each serverless ML application for the granularity decision.

The DQN-based RL agent is lightweight. It requires less than 250KB of additional memory, making the storage overhead minimal [73]. Computationally, the network's inference time is under 1 ms, and one learning step takes only 1 ms even with all (four) DQN agents operating concurrently. The computational overhead introduced by the model is negligible.

Applying DRL requires defining three key components: States, Rewards, and Actions. We define them in the serverless execution environment as follows:

States. The States in a DRL are supposed to capture the status of the environment and the application. We use the currently available resources (# vCPUs, # vGPUS) in each machine to represent the environment states and the status of the application with the SLO, and the time interval since its last call.

Actions. The Actions in a DRL consist of all the possible outputs of its prediction. In our problem, they are 0 for application granularity, 1 for stage granularity, and 2 for block granularity.

Rewards. The Rewards function in a DRL specifies the rewards that one action could bring. For many problems, the final rewards may not be known until a series of actions are taken. Our problem is such a case: it is not possible until an invocation of an application finishes running to tell the total costs and latency. To help DRL effectively learn, it is common to assign a small reward to intermediate progress. In our design, we define the award of the finish of one stage (i.e., one DNN inference or one block) of an ML application as follows:

$$Reward = \begin{cases} -p + 1(l \leq SLO) * \alpha & , \text{ if } app\ is\ done \\ 1\ (model\ wise) & , \text{ otherwise} \\ 0.33\ (block\ wise) & , \text{ otherwise} \end{cases} \quad (1)$$

where, p is the total cost, l is the total latency, SLO is the SLO latency, α is a constant (based on the user's preference, see Section 3.5) called *reward factor*, which balances hit rates and cost. We assign a small reward (1 for a model, 0.33 for a block) to an intermediate progress.

Neural Networks. DQN includes a *policy neural network* and a *target neural network* inside, which learn about the relations between states, actions, and rewards. We use the default neural network architectures in DQN [57], but adjusted the input and output channels to align with the dimensions of the States and Actions in our problem.

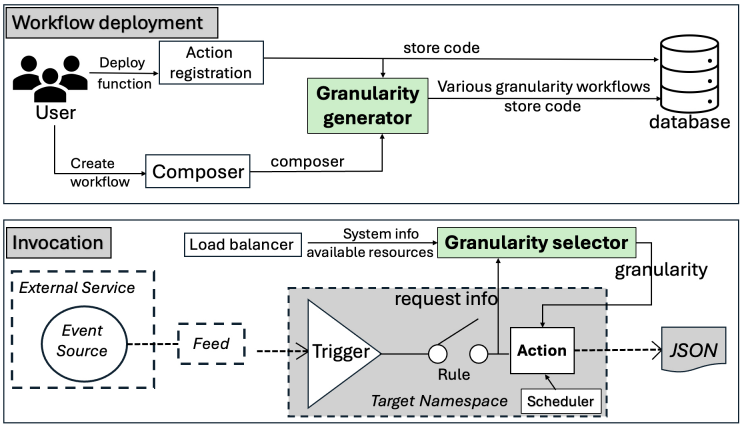

Fig. 14. Expanded OpenWhisk workflow with adaptive granularity enabled.

4.2 Supervised Learning

In addition to the reinforcement learning method, we also explore supervised learning methods. Given the uncertain relationship between system workload status and granularity, we investigate two supervised machine learning techniques to forecast granularity. We employ linear regression for its straightforward approach, and XGBoost, which usually excels on structured data.

Linear Regression. Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables [58]. The equation of a linear regression model with multi-independent variables is $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n + \varepsilon$, where, y is the dependent variable, x_n is the independent variables. The goal is to find the linear equation that best predicts the dependent variable based on the independent variables. In this context, the dependent variable represents the predicted granularity, while the independent variables include current system resource availability (# vCPUs, # vGPUS), workload status, and SLO settings.

XGBoost. Xtreme Gradient Boosting (XGBoost) operates on the principle of gradient boosting, constructing an ensemble of decision trees sequentially, where each subsequent tree aims to correct the errors made by the previous ones, thereby improving the model's accuracy. XGBoost works well with structured data and can be used for classification. It has built-in features for preventing overfitting. Our objective function *multi:softprob* is tailored to the nature of the granularity prediction task.

The training processes use a robust training-validation strategy, employing a split of training and validation sets, where the ratio is 80:20, to evaluate the model's generalizability and prevent overfitting. The input includes current system resource availability (# vCPUs, # vGPUS), the job arrival interval length, and the SLO levels, and the output is the predicted granularity. We created a set of synthesized workloads with various settings and SLO levels and empirically found the best granularities through many trials. These data are used to train the predictive model. The real-world traces are used as the testing data for evaluations.

5 Programming and Runtime Support

The third exploration in this work is to design the programming and runtime supports to ease the adoption and integration of adaptive granularity in serverless ML.

Before describing the support, we first note the high-level strategy in constructing functions of various granularities. At first glance, it might seem desirable to construct serverless functions of

various levels of granularity on the fly based on the dynamic situations of the serverless platform. The construction, however, incurs runtime overhead. Moreover, it worsens cold start overhead. When a new container is initiated for the newly constructed serverless function, it needs to create a new container image based on the selected function granularity rather than reusing an existing image. It makes it hard for cold-start minimization techniques to apply, which typically rely on reusing or prewarming techniques for ahead-of-time created images [20, 28, 53, 67, 70, 74]. The strategy we find effective is to predefine functions with representative granularities ahead of time and adaptively invoke the proper one(s) at runtime based on the dynamic situation.

Fig. 14 shows the expanded OpenWhisk workflow with adaptive granularity enabled. During the deployment stage, the user first creates the functions used in the workflow. User-created functions prompt OpenWhisk to establish a namespace, register the action, and store its code and configuration in a database. We enhance this process with a *granularity generator* that helps users create functions of different granularities, which are also stored in the database. In the invocation phase, events trigger action invocations. *Granularity selector* residing in the Controller determines the optimal granularity based on the runtime information. The Controller retrieves the function of the selected granularity from the database and allocates resources for functions. Next, we will explain the added support in more detail.

5.1 Creating Serverless Functions with Adaptive Granularity

The creation of serverless functions happens offline. The tools we have designed, in the form of a Python module, help the process. It has three parts: bridge function generation, composer generation, and function splitting.

As discussed in Section 2.2, a serverless application is implemented as a workflow, represented as a composer in OpenWhisk, and invoked through a *bridge function*. Listing 1 illustrates the *bridge function* template our tool creates for adaptive granularity. It defines different branches to call the application in different granularities. For *application* granularity, this call will be a function call. For the other granularities, this call will be a composer call. When the user creates various granularity applications, they either return the function name or the composer name. Our tool takes in the function name and composer name to generate the *bridge function* for an application. In execution, the *bridge function* will take the granularity from the granularity selector as the input to decide which branch to execute.

Application-wise workflow. The user takes the whole application program and defines it as a lambda function, and returns the function name to the *bridge function* to fill in the application-wise branch.

Stage-wise workflow. In Section 2.2, we discussed how users create serverless workflows by defining a composer. For a serverless function to be referenced within this composer, it must be predefined. The composer's name is then utilized in the relevant branch in the *bridge function*.

Block-wise workflow. When a user creates the function for the block-wise workflow, our splitting function module will generate the block-wise function, and the composer generation module generates the composer.

As we introduced in Section 3.1, we evenly divide the model into several blocks for load balance. Our tool retrieves the source code and analyzes it to extract the model architecture and the forward relations between layers. It then determines the topological order of the layers and evenly segments them into distinct blocks that fully exploit the parallel execution opportunity in the original model architecture, creating a separate function for each block and storing them in the database.

After creating all the block-wise functions for a given stage function, our tool can then construct the composer specific to that stage. The composer of all the stages is put together into the composer for the application. Since only tensor data is exchanged between blocks, it is straightforward to

generate these composers. The composer's name is then put into the relevant branch in the *bridge function*.

5.2 Runtime Support

Besides the offline automatic generator, we need to provide the runtime extension to make the adaptive function invocation in OpenWhisk.

As introduced, the controller handles requests and fetches that function from the database. In our extended platform, the invocation becomes a call to the *bridge function*. Before fetching the function, the granularity selector residing in the controller will check the load balancer to get the available resource information. Proxy threads are employed to track the intervals between application requests. Then the granularity selector, through its encompassed granularity predictor, uses the data on available resources, request intervals, and SLO levels to determine the optimal granularity. Subsequently, the controller fetches the *bridge function* from the database and gives the granularity as the input. The *bridge function* takes the granularity and calls the corresponding branch.

```
const openwhisk = require('openwhisk');
function main(params) {
    #get the granularity from input
    granularity = console.log(params.granularity);
    #application wise
    if granularity==0:
        name = 'ApplicationName'
        # Invoke the TargetFunction
        data =  ow.actions.invoke({name, result, params})
    #stage wise or block wise
    if granularity==1 or granularity==2:
        # Initialize the Openwhisk Composer
        const composer = require('@openwhisk/composer')
        # Define composer name here
        if granularity==1:
            composer_name = 'StageName'
        if granularity==2:
            composer_name = 'BlockName'
        # Start execution of the composer
        module.exports = composer.sequence(composer_name)
    return data
}
```

Listing 1. Bridge function template

6 Evaluation of Adaptive Granularity

This section evaluates (i) the effectiveness of adaptive granularity, (ii) the detailed analysis of the effectiveness, and (iii) the generality evaluation. We use the same methodology as described in Section 3. One difference is that we use real-world traces from Azure [70] with mixed SLO levels—1.2× and 1.5× SLO requirements during the bursty stage, and 0.45×, 0.5×, and 0.8× SLOs during the steady stage.

We assess the effectiveness of adaptive granularity using three predictors: multi-agent deep reinforcement learning (DRL), linear regression, and XGBoost, as detailed in Section 4. These methods are evaluated against the state-of-the-art serverless ML, ESG [38]. To test generality, we also experiment with another recent serverless ML, INFless [78]. Both use stage granularity. We add the other two fixed granularities and implement our adaptive granularity upon those two state-of-the-art serverless ML frameworks. The results with fixed granularities are labeled as ESG-application, ESG-stage, ESG-block, INFless-application, INFless-stage, and INFless-block. Results

using adaptive function granularity with three predictors are denoted as ESG+DRL, ESG+Linear Regression, ESG+XGBoost, INFless+DRL, INFless+Linear Regression, and INFless+XGBoost.

We use reward score as the overall metric, defined as $reward = \alpha \cdot SLO\ hits - costs$ as mentioned in Section 3.5. To illustrate adaptivity, we use three different reward factors (1, 50, 100) for α to represent different user preferences on the emphasis given to resource cost and SLO hit rate.

6.1 End-to-end Performance

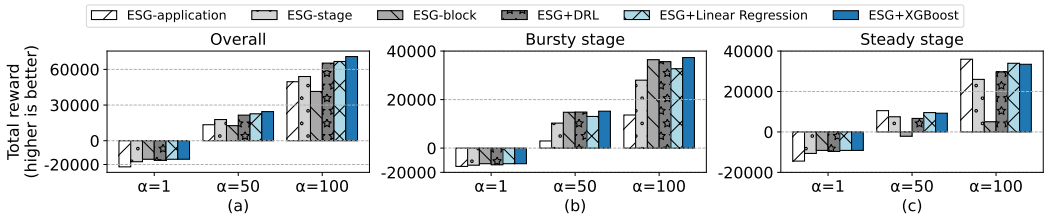

Fig. 15. The rewards (higher is better) with different factor settings in (a) overall, (b) bursty stage, and (c) steady stage.

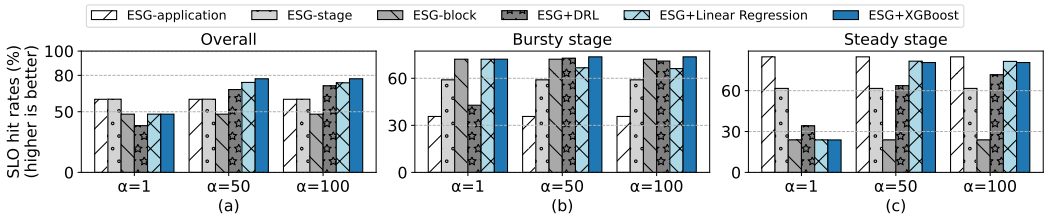

Fig. 16. The SLO hit rates (higher is better) with different factor settings in (a) overall, (b) bursty stage, and (c) steady stage.

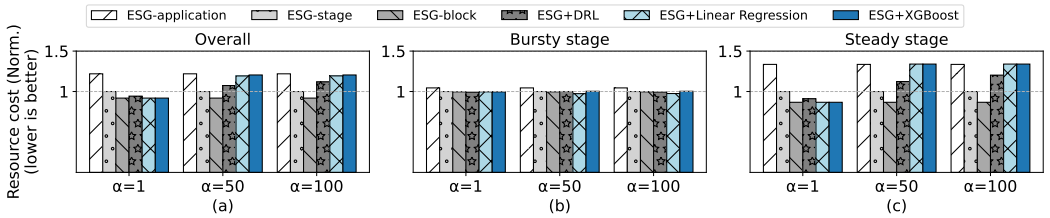

Fig. 17. The resource cost (lower is better) with different factor settings in (a) overall, (b) bursty stage, and (c) steady stage.

Fig. 15 (a) displays the total reward scores for various reward factors when different granularities are applied. When the factor is set to 50 and 100, emphasizing the importance of SLO hit rate over resource cost, ESG+XGBoost achieves 31.1%-93.5% improvement in rewards compared to fixed granularity. Conversely, with the reward factor at 1, which gives some more emphasis on resource cost over SLO hit rate, ESG+XGBoost outperforms both ESG-application and ESG-stage by 29.11% and 11.9%, respectively, showing that using the adaptive function granularity leads to significant better scheduling outcomes for different scenarios. Among all adaptive schemes,

ESG+XGBoost surpasses ESG+RL by 3%-25% and ESG+Linear Regression by 0%-8.24%, delivering the best performance.

Fig. 16 (a) and 17 (a) illustrate SLO hit rates and resource costs, respectively. When the reward factor is 1, ESG+XGBoost consumes smaller resource costs by up to 24.6% compared to the fixed granularity. When the reward factor is set to 50 and 100, Fig.16(a) demonstrates that adaptive granularity outperforms fixed granularity in terms of the SLO hit rate, showing up to 29.2% SLO hit rate using XGBoost.

Fig. 18 presents the total rewards for each application. All applications achieve the highest rewards using ESG+XGBoost. The highest improvement is on the expanded image classification workload, 436% higher than the fixed granularity. The reason is that this application consists of the largest number of DNN models and, hence, the longest pipelines. As shown in Figure 18 (d), this application achieves the highest reward when using application-level granularity compared to all fixed granularities. Fine-grained approaches introduce data transfer overhead and queuing delays for each function in the pipeline, which increase the end-to-end latency and lead to the poorest performance. Our dynamic granularity successfully identifies the best granularity and yields significant improvement.

Fig.19 displays the SLO hit rates, showing that ESG+XGBoost yields the best SLO hit rates for all applications when the reward factor is 50 and 100, at most 45% higher than the fixed granularity for expanded image classification. Fig. 20 illustrates the resource cost, where ESG+XGBoost consistently results in the lowest resource cost across all applications when the reward factor is set to 1, at most 58% lower resource cost for the expanded image classification.

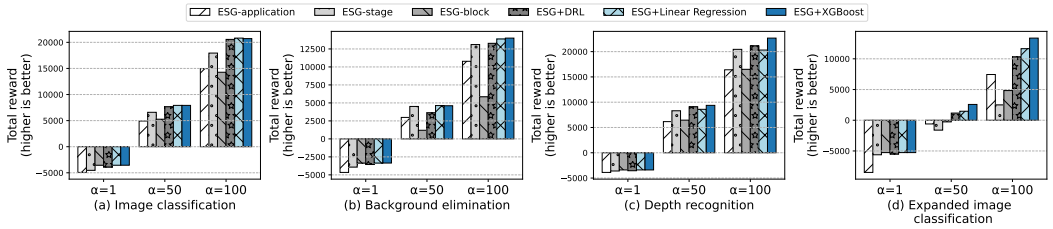

Fig. 18. The rewards (higher is better) for each application in the real-world trace.

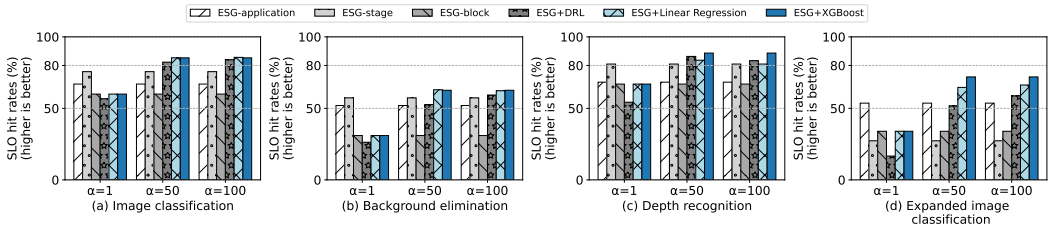

Fig. 19. The SLO hit rates (higher is better) for each application in the real-world trace.

6.2 Performance Breakdown Analysis

Figs. 15 (b), 16 (b), and 17 (b) show the performance during the bursty stage, whereas the (c) parts of these figures show the results in the steady stage.

The figures show that SLO hit rate and resource cost remain the same across different factor settings for a fixed granularity level (Application, Stage, or Layer). This consistency is caused by

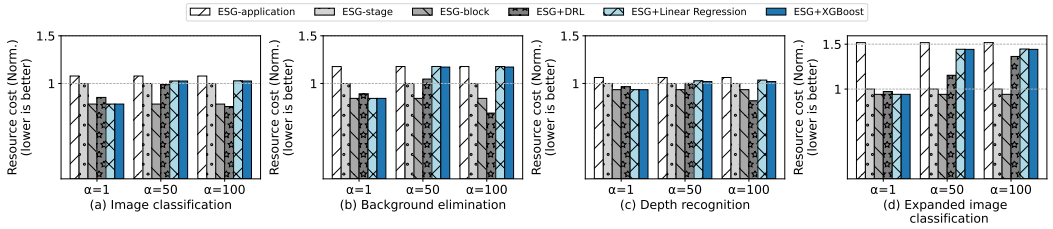

Fig. 20. The resource cost (lower is better) for each application in the real-world trace.

the same granularity being utilized irrespective of all factor settings. In the bursty case, ESG-block achieves a higher SLO hit rate (22%-102%) and utilizes fewer resources (0.3%-5%) over the other two fixed granularity, thanks to its high resource efficiency, which aligns with the analysis presented in Section 3. In the steady case, because of the stringent SLO requirements, ESG-application gets the highest SLO hit rate but at the expense of more significant resource costs.

When the factor is set at 1, indicating more emphasis on resource cost savings, ESG-block demonstrates the lowest resource cost during both bursty and steady cases for all requests, as shown in Fig. 17. It is observed that ESG+XGBoost consistently selects this granularity, matching the performance of block granularity in each stage.

When the factor is adjusted to 50 and 100, prioritizing the SLO hit rate becomes crucial. As Fig. 16 shows, during the bursty phase, ESG-block granularity achieves the top SLO hit rate, and the adaptive granularity approach delivers comparable results. In the steady phase, application granularity gets the highest SLO hit rate, and the performance of the adaptive granularity closely aligns with it.

Figure 16 (c) reveals that the SLO hit rate for ESG-application is slightly better than ESG+XGBoost when the factor is set at 50 or 100. This difference is primarily due to variations in runtime system resources, which depend on the granularity method employed. Specifically, during the bursty phase, the granularity choices between ESG+XGBoost and ESG-application differ, impacting resource availability in subsequent normal phases and influencing detailed performance metrics. Despite these variations, Figure 16 (a) and Figure 17 (a) show that ESG+XGBoost outperforms fixed granularity approaches in terms of SLO hit rates while incurring similar resource cost.

Given the varying nature of workloads and factor settings, the adaptive granularity can adapt to the change and choose suitable granularity, thus it can outperform fixed granularity. This suggests that considering the system's current state and user preferences is crucial for optimizing the performance of serverless ML workflows.

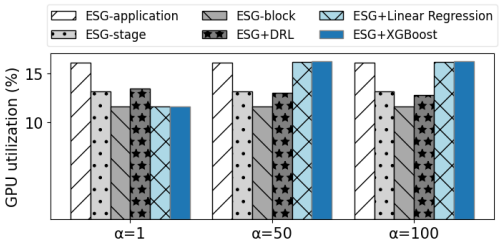

Fig. 21. The average GPU resource utilization.

6.3 Resource Utilization Analysis

Fig.21 shows the average GPU resource utilization. It shows that finer function granularity facilitates cost savings due to the variability in the scalability of ML models and blocks (as discussed in Section3.4.1), while coarser granularity consumes more resources but achieves higher SLO hit rates (see Fig. 16). The figure also reveals that when the factor is set to 1, ESG+XGBoost consumes the fewest resources, similar to ESG-block.

However, as the factor increases to 50 and 100, ESG+XGBoost utilizes more resources, akin to ESG-application, to achieve higher SLOs (as depicted in Fig. 16).

Figure 22 displays the GPU resource utilization for each time step, with different αs. It shows the period when the bursty stage ends and the system shifts into a steady stage. During the bursty stage, ESG+XGBoost exhibits increased resource utilization to reduce queuing time and improve SLO. In the steady stage, when the factor is 50 and 100, prioritizing SLO, as illustrated in Figure 22 (b) and (c), ESG+XGBoost achieves high SLO hit rates through increased resource usage. Conversely, when the factor is set to 1, emphasizing resource conservation, ESG+XGBoost reduces resource utilization, as shown in Figure 22(a). In conclusion, ESG+XGBoost can make suitable decisions based on system and workload conditions, as well as performance objectives.

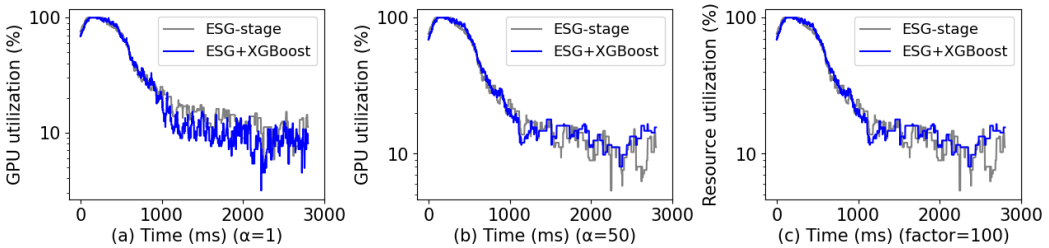

Fig. 22. The GPU resource utilization during which the burst stage ends and the system transitions to a steady stage for (a) factor is 1, (b) factor is 50, and (c) factor is 100.

6.4 Generality Study

6.4.1 Different MIG Partitions on A100. We conducted experiments on two additional MIG partition settings, as detailed in Table 4. Figure 23 presents the results for partition P1, which consists of three 2g.20gb instances and one 1g.10gb instance. Among the fixed granularity choices, block granularity consistently achieves the highest performance, with SLO hit rates that are 17% and 7% higher than those of application and stage granularities, respectively. Figure 24 presents the results for partition P2, which consists of one 4g.40gb instance, one 2g.20gb instance, and one 1g.10gb instance. For this setting, it is the stage granularity that achieves the highest SLO hit rates among all fixed granularity choices, outperforming application granularity by 5%. Block granularity shows the lowest performance. We attribute the reasons for the different performances of the three fixed granularities in these two settings to the different restrictions imposed by the partitions. The lower performance of block granularity in P2, for instance, is due to the tension between its need to schedule more functions and the very limited MIG instances in P2. That leads to significant resource contention, causing many functions to wait for available resources.

Our dynamic granularity, ESG+XGBoost, consistently brings the largest rewards in both settings. In partition P2, for instance, it achieves SLO hit rates that are 12% to 19% higher than fixed granularity methods when prioritizing SLO adherence and reduces costs by 17% when emphasizing cost savings. It is worth noticing that in terms of the overall results, the best are achieved by ESG+XGBoost when the GPUs have the finest MIG partitions (as shown in Section 6.1). The reason is that that setting gives the largest granularity selection space (recall that multiple vGPUs can be used as a group by a serverless function) and, hence, the largest performance potential.

6.4.2 NVIDIA H100. We also conducted experiments on the NVIDIA H100, a more recent GPU in Hopper architecture. It has the same set of MIG instance partitions as A100. We use the finest MIG partitions for this experiment for its superior performance.

Table 4. Additional partition settings of NVIDIA A100.

Partitions	NVIDIA MIGs
P1	2g.20gb + 2g.20gb + 2g.20gb + 1g.10gb
P2	4g.40gb + 2g.20gb + 1g.10gb

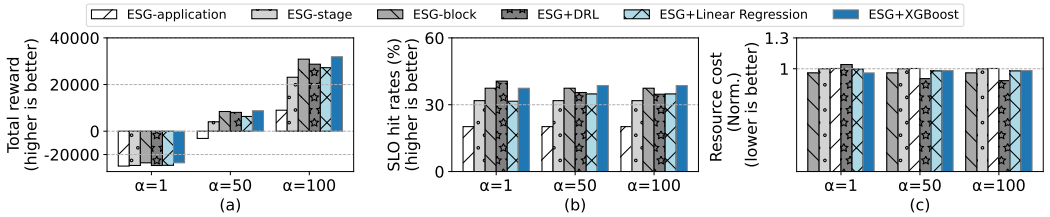

Fig. 23. The (a) total reward, (b) SLO hit rates, and (c) resource cost enabled on partition P1 in Table 4. The ESG-stage in (c) is 1.

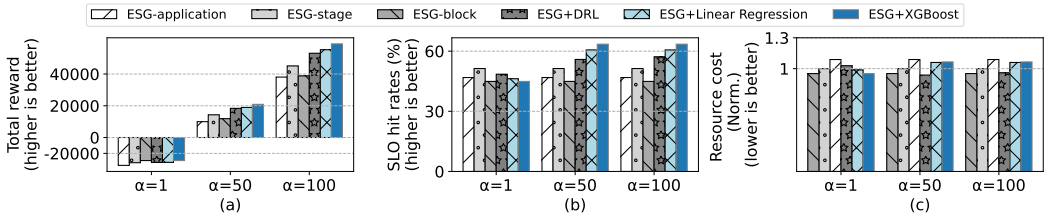

Fig. 24. The (a) total reward, (b) SLO hit rates, and (c) resource cost enabled on partition P2 in Table 4. The ESG-stage in (c) is 1.

Fig. 25 (a) displays the total reward scores for various reward factors when different granularities are applied. When the factor is set to 50 and 100, emphasizing the importance of SLO hit rate over resource cost, ESG+XGBoost achieves 15.3%-88.7% improvement in rewards compared to fixed granularity. Conversely, with the reward factor at 1, which gives some more emphasis on resource cost over SLO hit rate, ESG+XGBoost outperforms both ESG-application and ESG-stage by 37.6%, showing that using the adaptive function granularity leads to significantly better scheduling outcomes for different scenarios. Among all adaptive schemes, ESG+XGBoost surpasses ESG+RL by 15% and ESG+Linear Regression by 11%, delivering the best performance.

Fig. 25 (b) and (c) illustrate SLO hit rates and resource costs, respectively. When the reward factor is 1, ESG+XGBoost consumes smaller resource costs by up to 29.8% compared to the fixed granularity. When the reward factor is set to 50 and 100, ESG+XGBoost shows up to 36.7% SLO hit rate improvement. H100 delivers better SLO hit rates overall than A100 thanks to its greater computing power.

6.4.3 On Other Serverless Framework. To check the generality, Fig. 26 shows the results on another serverless ML framework, INFless [78]. The results show that adaptive function granularity also improves INFless; XGBoost gives the best performance.

6.5 Overhead Analysis

Table 5 reports the average queueing delay and data transfer overhead of different methods. The ESG-block yields the least queuing delay in the bursty phase while it has the largest data transfer overhead. The ESG-application has the largest queuing overhead but no data transfer overhead.

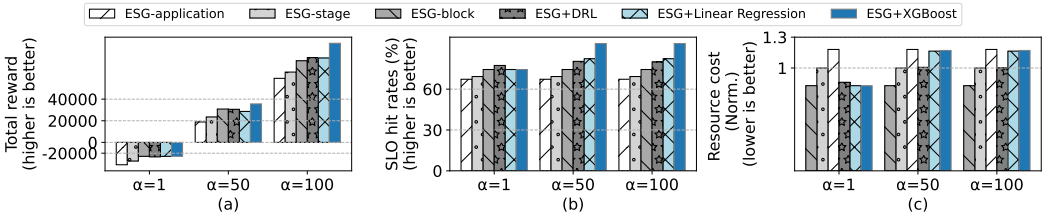

Fig. 25. The (a) total reward, (b) SLO hit rates, and (c) resource cost on NVIDIA H100. The ESG-stage in (c) is 1.

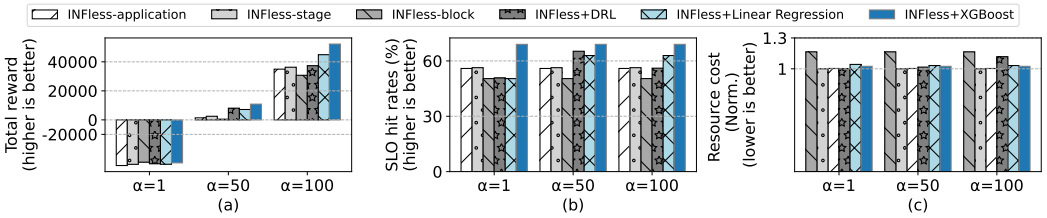

Fig. 26. The (a) total reward, (b) SLO hit rates, and (c) resource cost enabled on INFless[78]. The INFless-stage in (c) is 1.

Table 5. The queueing time and data transfer overhead.

Workloads		Total			Bursty			Steady		
α		1	50	100	1	50	100	1	50	100
Queuing time (ms)	ESG-application	25	25	25	49	49	49	1	1	1
	ESG-stage	7	7	7	12	12	12	2	2	2
	ESG-block	9	9	9	11	11	11	7	7	7
	ESG+DRL	19	7	12	32	12	21	6	2	4
	ESG+Linear Reg.	9	6	6	11	12	12	7	0.5	0.5
	ESG+XGBoost	9	6	6	11	12	12	7	0	0
Data transfer overhead (ms)	ESG-application	0	0	0	0	0	0	0	0	0
	ESG-stage	6	6	6	9	9	9	4	4	4
	ESG-block	33	33	33	55	55	55	10	10	10
	ESG+DRL	22	24	22	33	45	37	10	3	7
	ESG+Linear Reg.	33	18	15	55	35	30	10	0	0
	ESG+XGBoost	33	28	28	55	56	56	10	0	0

ESG+XGBoost results in shorter queuing delays compared to ESG-application and reduces data transfer compared to ESG-block, enhancing its overall scheduling outcomes. These benefits are derived from its adaptive decision-making. As demonstrated in Table 5, XGBoost selects block granularity during the burst stage, leveraging its scheduling flexibility and shorter execution times to reduce queuing. During the steady stage, it opts for application granularity, taking advantage of sufficient system resources to handle applications more efficiently and minimize data transfer overhead.

Fig. 27 presents the 95th percentile tail latency for all applications. The ESG-application experiences higher tail latency primarily due to cold start delays, whereas the ESG-stage and ESG-block achieve lower latency by loading only parts of the models each time. Additionally, the pre-warming policy initializes non-first function instances within the workflows, which further reduces tail latency for the ESG-stage and ESG-block. When the factor is set to 50 and 100, the adaptive methods choose block granularity during the bursty stage and application granularity during the steady stage (details in Fig. 28 and Fig. 29). Consequently, the adaptive methods exhibit tail latency similar to that of the ESG-application. However, when the factor is set to 1, the adaptive methods opt for block granularity, resulting in tail latency comparable to the ESG-block.

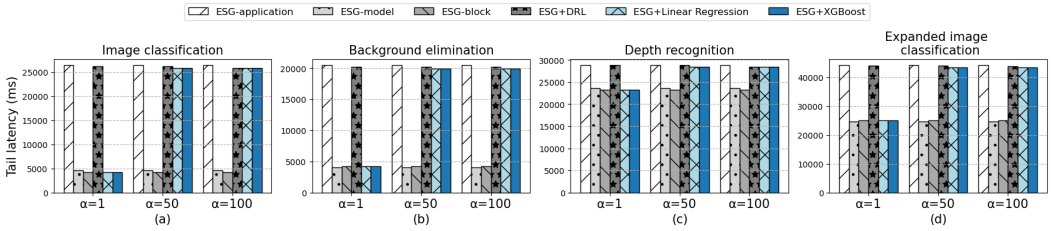

Fig. 27. The 95th percentile tail latency for all applications.

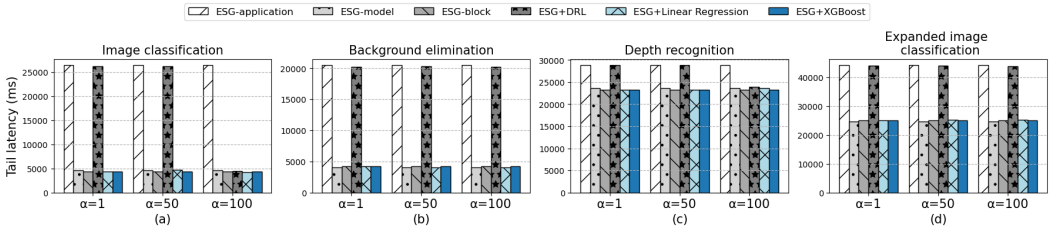

Fig. 28. The 95th percentile tail latency for all applications in the bursty stage.

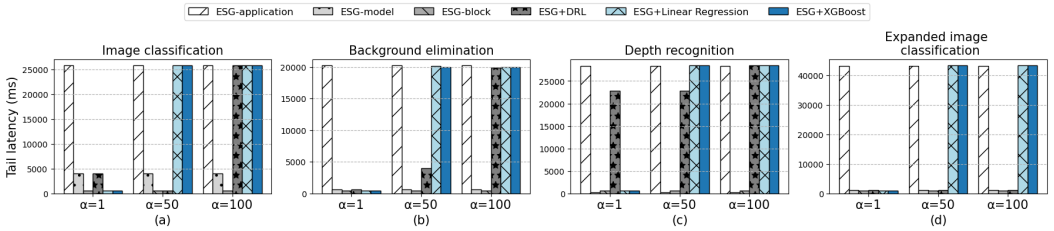

Fig. 29. The 95th percentile tail latency for all applications in the steady stage.

Table 6. Scheduling overhead and percentage.

	Scheduling overhead (ms)			Percentage (%)		
α	**1**	**50**	**100**	**1**	**50**	**100**
ESG-application	0.34	0.34	0.34	0.0	0.0	0.0
ESG-stage	1.23	1.23	1.23	0.05	0.05	0.05
ESG-block	2.55	2.55	2.55	0.17	0.17	0.17
ESG+RL	1.49	1.59	2.71	0.02	0.02	0.11
ESG+Linear Reg.	3.05	2.38	2.03	0.2	0.08	0.07
ESG+XGBoost	3.45	3.48	3.48	0.22	0.15	0.15

Table 6 presents the scheduling overhead associated with different granularities. Finer granularity incurs higher scheduling overhead than coarse granularity because of the increased search space from more functions. Adaptive granularity results in slightly longer overhead, approximately 1 ms, compared to fixed granularity. However, this overhead constitutes a small fraction of the total end-to-end latency, less than 0.3%.

6.6 Discussion

The current results reveal that XGBoost surpasses both linear regression and the multi-agent RL approach. The superior performance over RL is due to the challenges of sparse and delayed rewards in reinforcement learning: rewards are distributed at each step as functions are executed,

yet comprehensive data such as end-to-end latency and exact resource usage of an application are not clear until the final function completes, complicating the task of accurate reward evaluation. XGBoost's superiority over linear regression highlights that the correlation between input features and predicted granularity is too intricate for linear models to capture.

This study explores three given granularities to demonstrate how function granularity affects serverless scheduling outcomes. One could use automated ML partitioning techniques, such as Alpa [82], to partition the original application into even more possible granularities. Our current design can be extended to handle that, which is left as future work.

This paper demonstrates the enhancements in performance and cost efficiency achieved through the adoption of varying function granularities in MIG. Incorporating different function granularities could also improve performance and cost efficiency in MPS by broadening the selection space, thereby accommodating dynamic resource availability and diverse user needs. One recent study, PROTEAN [19], have investigated the integration of MIG and MPS on serverless computing to augment resource efficiency and satisfy SLOs. Our research contributes are orthogonal to PROTEAN by introducing dynamic granularities. Incorporating a broader spectrum of function granularities could potentially further enhance PROTEAN's improvements if the increased scheduling complexity can be effectively addressed.

7 Related Work

Explorations to enable efficient sharing of a GPU for multiple co-running jobs trace back to the pioneering work by Wu and others [76], who proposed a technique called *SM-centric program transformations* to allow flexible control of the placements and scheduling of GPU threads at program level. It opens up opportunities for optimizing co-runs of GPU kernels through code transformations. Based on the technique, Chen and others [23] designed the first software-enabled preemptive scheduler for GPUs regardless of whether the GPUs have built-in hardware support for kernel preemption. These techniques circumvent the limitations of GPU hardware, making it possible for programs to conduct flexible controls and optimizations of threads, scheduling, and kernel co-runs. Recent GPUs are equipped with new hardware features, such as MIG, to better support co-runs. Those software methods still have their value in enabling flexible software-level optimizations.

In serverless computing, several recent studies have started investigating the segmentation of functions to enhance resource utilization. For instance, Splitwise [59] decomposes large generative language models (LLMs) into memory-intensive and compute-intensive segments to optimize the use of resources; this approach ensures that compute and memory resources are not unnecessarily tied up when they are not in active use. Gillis [79] introduces automatic model partitioning for large neural networks within serverless functions, leveraging the potential for parallel execution to enhance performance and reduce costs. While these studies acknowledge the advantages of fine granularity in serverless functions concerning resource use and flexibility, they do not delve into the impacts of varying granularities.

Other studies employed the coarse fixed function granularity, such as application and model. For example, some studies treat each ML model as a function. Infless [78] explores dynamic batching and resource assignment with a heuristic algorithm, while ESG [38] focuses on GPU sharing, developing a scalable and efficient resource assignment scheduler. Other research considers the entire application as a single function. Miso [46] examines container placement strategies to enhance GPU utilization, and Simppo [62] uses the entire application, consisting of multiple ML models, to reduce data transfer and improve resource efficiency.

There are numerous studies trying to improve the efficiency of the serverless computing by optimizing scheduling algorithm [16, 25, 42, 44, 48, 55, 56, 60, 62, 66, 72, 78, 83]. Some of them, such

as [38, 55, 78], employ search-based scheduling algorithms, while others (e.g., [62]) utilize reinforcement learning for resource management. There are also approaches [83] that adopt Bayesian optimization-based scheduling algorithm, which builds upon IceBreaker [67] and CLITE [60] and extends BO in new ways than what was previously done in other BO-inspired solutions, such as SATORI [66], Ribbon [47], and OLPart [25]. Several studies have investigated the efficient scheduling of function Directed Acyclic Graphs (DAGs) on CPUs. Orion [55] proposes assigning multiple instances of a function to a single virtual machine (VM) to improve resource utilization. WiseFuse [56] suggests assigning in-series functions to a VM, aiming to reduce communication overhead. Although WiseFuse dynamically assigns in-series functions to leverage some benefits akin to our exploration of varying function granularities, its focus remains on CPU-based environments and does not explore the potential resource efficiency improvements that different function granularities might yield on GPUs. ProPack [18] investigates the assignment of an optimal number of function instances of a function to a VM to address the significant scaling latency observed when numerous functions are invoked concurrently.

Besides, there are many works optimizting the cold start [14, 30, 33, 71] and improve data locality [13, 41, 52]. However, none of them change the function granularity to further optimize serverless computing. Our research introduces an innovative dimension, granularity, that is complementary to existing strategies. To the best of our knowledge, this is the first study to explore granularity as a distinct dimension to improve serverless computing, offering potential synergies that could amplify the benefits of prior solutions.

8 Conclusion

This paper presents three-fold explorations in function granularity for serverless ML. It confirms the importance of granularity and uncovers a series of insights on the impact of granularity on the SLO and resource costs. It describes our three attempts to create predictive models for function granularities and the associated programming and runtime support for integrating adaptive granularity into serverless platforms. The results show that adaptive granularity can achieve significant benefits compared with the conventional workflow using single fixed granularity.

Acknowledgments

We thank our shepherd, Dr. Mahmut Taylan Kandemir, and the anonymous reviewers for their insightful feedback that helped improve our work. This material is based upon work supported by the National Science Foundation (NSF) under Grants No. CNS-2312207. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Apache OpenWhisk. How OpenWhisk works. https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openWhisk-works.
[2] AWS Step Functions. https://aws.amazon.com/step-functions/.
[3] Azure Durable Functions. https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc.
[4] DeblurGAN. https://github.com/pablodz/DeblurGAN.
[5] DEEPLABV3. https://pytorch.org/hub/pytorch_vision_deeplabv3_resnet101/.
[6] Google Could Workflows. https://cloud.google.com/workflows.
[7] Improving Performance on AWS and Hybrid Networks. https://aws.amazon.com/blogs/networking-and-content-delivery/improving-performance-on-aws-and-hybrid-networks/#:~:text=AWS%20Site%2Dto%2DSite%20VPN%20over%20the%20Internet&text=However%2C%20this%20type%20of%20connectivity,to%2DSite%20VPN%20performance%20tuning.
[8] NVIDIA 2020, Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[9] NVIDIA 2020, Multi-Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html.
[10] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.
[11] OpenWhisk. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.
[12] ResNet50. https://pytorch.org/hub/nvidia_deeplearningexamples_resnet50/.
[13] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 365–380, 2023.
[14] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.
[15] Sohaib Ahmad, Hui Guan, Brian D Friedman, Thomas Williams, Ramesh K Sitaraman, and Thomas Woo. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 318–334, 2024.
[16] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.
[17] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.
[18] Rohan Basu Roy, Tirthak Patel, Richmond Liew, Yadu Nand Babuji, Ryan Chard, and Devesh Tiwari. Propack: Executing concurrent serverless functions faster and cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 211–224, 2023.
[19] Vivek M Bhasi, Aakash Sharma, Rishabh Jain, Jashwant Raj Gunasekaran, Ashutosh Pattnaik, Mahmut Taylan Kandemir, and Chita Das. Towards slo-compliant and cost-effective serverless computing on emerging gpu architectures. In *Proceedings of the 25th International Middleware Conference*, pages 211–224, 2024.
[20] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.
[21] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Ra Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.
[22] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Ran Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, pages 2–8, 2018.
[23] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 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)*, February 2017.
[24] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.
[25] Ruobing Chen, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. Olpart: Online learning based resource partitioning for colocating multiple latency-critical jobs on commodity computers. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 347–364, 2023.
[26] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.
[27] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. Sla-driven ml inference framework for clouds with heterogeneous accelerators. *Proceedings of Machine Learning and Systems*, 4:20–32, 2022.
[28] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.
[29] Sukhpreet Singh Dhaliwal, Abdullah-Al Nahid, and Robert Abbas. Effective intrusion detection system using xgboost. *Information*, 9(7):149, 2018.
[30] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.
[31] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning. In *Learning for dynamics and control*, pages 486–489. PMLR, 2020.
[32] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 431–445, 2021.

[33] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[34] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference. *arXiv preprint arXiv:2309.00558*, 2023.

[35] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the international symposium on quality of service*, pages 1–10, 2019.

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

[37] Sungho Hong. *GPU-enabled Functional-as-a-Service.* PhD thesis, Arizona State University, 2022.

[38] Xinning Hui, Yuanchao Xu, Zhishan Guo, and Xipeng Shen. 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*, pages 42–55, 2024.

[39] Xinning Hui, Yuanchao Xu, and Xipeng Shen. Efficient serverless support for multi-instance gpus through pipelining. Technical Report TR-2025-1, Computer Science Department, North Carolina State University, 2025.

[40] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[41] Zhipeng Jia and Emmett Witchel. 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*, pages 152–166, 2021.

[42] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 289–305, 2022.

[43] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[44] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

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

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

[47] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. Ribbon: cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2021.

[48] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[49] Wei Li, Yanbin Yin, Xiongwen Quan, and Han Zhang. Gene expression value prediction based on xgboost algorithm. *Frontiers in genetics*, 10:484931, 2019.

[50] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[51] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[52] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 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*, pages 782–796, 2022.

[53] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221*, 2019.

[54] Qixiao Liu and Zhibin Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 347–360, 2018.

[55] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. Orion and the three rights: Sizing, bundling, and prewarming for serverless dags. In *16th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 22), pages 303–320, 2022.

[56] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, 2022.

[57] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[58] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[59] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *ISCA*, June 2024.

[60] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[61] Xuebin Qin, Zichen Zhang, Chenyang Huang, Masood Dehghan, Osmar Zaiane, and Martin Jagersand. U2-net: Going deeper with nested u-structure for salient object detection. volume 106, page 107404, 2020.

[62] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Simppo: a scalable and incremental online learning framework for serverless resource management. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 306–322, 2022.

[63] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer, 2020.

[64] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*, pages 1–13, 2012.

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

[66] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Satori: efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 292–305. IEEE, 2021.

[67] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[68] Justin San Juan and Bernard Wong. Reducing the cost of gpu cold starts in serverless deep learning inference serving. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 225–230. IEEE, 2023.

[69] Justin David Quitalig San Juan. Flashpoint: A low-latency serverless platform for deep learning inference serving. Master's thesis, University of Waterloo, 2023.

[70] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[71] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[72] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. Sponge: Fast reactive scaling for stream processing with serverless frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 301–314, 2023.

[73] Hsin-Hsuan Sung, Jou-An Chen, Wei Niu, Jiexiong Guan, Bin Ren, and Xipeng Shen. Decentralized application-level adaptive scheduling for multi-instance dnns on open mobile devices. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 865–877, 2023.

[74] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.

[75] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[76] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the ACM International Conference on Supercomputing*, 2015.

[77] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. Streambox: A lightweight gpu sandbox for serverless inference workflow. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages

59–73, 2024.

[78] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 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*, pages 768–781, 2022.

[79] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 138–148. IEEE, 2021.

[80] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. Autosync: Learning to synchronize for data-parallel distributed deep learning. *Advances in Neural Information Processing Systems*, 33:906–917, 2020.

[81] Ming Zhao, Kritshekhar Jha, and Sungho Hong. Gpu-enabled function-as-a-service for machine learning inference. *arXiv preprint arXiv:2303.05601*, 2023.

[82] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[83] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 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*, pages 1–14, 2023.

Received 20 October 2024; revised 13 January 2025; accepted 15 January 2025