



**HAL**  
open science

## Towards a Multi-objective Scheduling Policy for Serverless-based Edge-Cloud Continuum

Luc Angelelli, Anderson Andrei Da Silva, Michael Mercier, Grégory Mounié,  
Denis Trystram, Yiannis Georgiou

► **To cite this version:**

Luc Angelelli, Anderson Andrei Da Silva, Michael Mercier, Grégory Mounié, Denis Trystram, et al..  
Towards a Multi-objective Scheduling Policy for Serverless-based Edge-Cloud Continuum. 2023. hal-  
04177085v1

**HAL Id: hal-04177085**

**<https://inria.hal.science/hal-04177085v1>**

Preprint submitted on 13 Mar 2023 (v1), last revised 3 Aug 2023 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Towards a Multi-objective Scheduling Policy for Serverless-based Edge-Cloud Continuum

Luc Angelelli<sup>\*♦</sup>, Anderson Andrei Da Silva<sup>\*♦♦</sup>, Yiannis Georgiou<sup>\*♦</sup>,  
Michael Mercier<sup>\*♦</sup>, Gregory Mounié<sup>\*♦</sup>, and Denis Trystram<sup>\*♦</sup>  
<sup>♦</sup>Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, Grenoble, France,  
<sup>♦♦</sup>Ryax Technologies, Lyon, France

**Abstract**—The cloud is extended towards the edge to form a computing continuum while managing resources’ heterogeneity. The serverless technology simplified how to build cloud applications and use resources, becoming a driving force in consolidating the continuum with the deployment of small functions with short execution. However, the adaptation of serverless to the edge-cloud continuum brings new challenges mainly related to resource management and scheduling. Standard cloud scheduling policies are based on greedy algorithms that do not efficiently handle platforms’ heterogeneity nor deal with problems such as cold start delays. This work introduces a new scheduling policy that tries to address these issues. It is based on multi-objective optimization for data transfers and makespan while considering heterogeneity. Using simulations that vary workloads, platforms, and heterogeneity levels, we study the system utilization, the trade-offs between the targets, and the impacts of considering platforms’ heterogeneity. We perform comparisons with a baseline inspired by a Kubernetes-based policy, representing greedy algorithms. Our experiments show considerable gaps between the efficiency of a greedy-based scheduling policy and a multi-objective-based one. The last outperforms the baseline by reducing makespan, data transfers, and system utilization by up to two orders of magnitudes in relevant cases for the edge-cloud continuum.

**Index Terms**—Scheduling Policies, Serverless Computing, Edge-Cloud Continuum, Heterogeneous Platforms.

## I. INTRODUCTION

In the last years, we have been witnessing the emergence of edge computing to complement the well-established technology of cloud computing. It led to the paradigm of an edge-cloud computing continuum [1] which can better address the challenges brought by the new generation of applications. These applications involve massive data, a much shorter time for action, along with security and privacy vulnerabilities, that the cloud alone could not handle. The edge-cloud continuum, illustrated in Figure 1, comprises a multi-layer architecture where cloud clusters, edge clusters (also known as Fog) and edge resources are interconnected. This continuum comprises the heterogeneity of resources while proposing common abstractions and mechanisms per cluster enabling a unified control. It provides a platform with heterogeneous resources to better address the needs of modern applications, but it also introduces further complexity, particularly in resource management and scheduling where it requires more efforts

to deal with such heterogeneous multi-layered infrastructures. More specifically, standard cloud scheduling policies are based on greedy algorithms that do not efficiently handle platforms’ heterogeneity and do not optimize data transfers. In parallel, serverless technology has been gaining popularity as the new way to program and deploy applications on clouds [2]. Serverless computing enables lightweight deployments of small functions with a short execution time. It is a perfect fit for the edge-cloud continuum because it allows quick adaptations to any move toward the edge level while keeping the applications’ footprints low. However, serverless also brings new challenges such as managing heterogeneous platforms and applications that deal with massive data, as well as deploying complex software environments such as the ones needed for machine learning and artificial intelligence applications.

In serverless computing, the low latency applications will compete for resources with the data aggregation and batch processing applications like model training. To keep the quality of services, the batch processing applications have to avoid bandwidth saturation and minimize the execution completion time, called makespan. Function as a service (FaaS) is an approach encompassed by serverless, and in the case of complex software stacks such as the ones that execute machine learning workflows, these functions are usually deployed within containers. Whenever a function is triggered, its container image is required. If the computational resource selected to execute this function already has its container deployed, it is called a warm start-up, and the function will be shortly initialized. If not, it is called a cold start-up (or cold start delay), and the container needs to be downloaded from an online repository and deployed on the allocated computational resource. Hence, the initialization of the function takes longer. Shahradeh et al. showed that the deployment of containers can cause an overhead of up to 20x of the platform slowdown, with cold start delays up to 10x longer [3]. Even so, containers are composed of layers. Each layer packs different types of software, such as OS or libraries. Due to this composition, containers can profit from a sharing mechanism of layers (or caching of layers), which can be used to speed up their deployment [4].

We develop a serverless platform in the edge-cloud continuum, and in this paper we evaluate the impacts of considering the heterogeneity of the platform at the scheduling phase while optimizing the deployment of containers and functions

\*Authors’ names are sorted in alphabetical order. Contributions are detailed at the end of the article.

execution time. Our platform manages a small to a medium number of functions per machine. To do so, we propose a multi-objective scheduling policy, called *FOA*, that enables the allocation of batches of serverless functions on heterogeneous edge-cloud platforms with the capability to minimize both makespan and cost. By cost, we consider the sum of the amount of data transferred to download the container images and the amount of data transferred for functions I/O. By reducing the amount of data transferred to download containers, we speed up the container’s deployment so we minimize the cold start delays. This will have a considerable impact on the execution time of a function. Furthermore, by reducing the amount of data transferred for function I/O, we speed up functions’ initialization.

Our scheduling policy is used at the global level of the edge-cloud continuum composed of different clusters (see Figure 1). The policy takes into account the availability and characteristics of all the connected edge-cloud clusters, and the scheduling decision is applied to each cluster or resource at the local level of the continuum. For the experimental campaign, we adapt a set-up composed of a bare-metal infrastructure and a simulated environment. In the bare-metal infrastructure, we deploy, on top of GRID5000 [5], the open-source serverless platform OpenWhisk [6], and execute an adapted version of the serverless functions from FunctionBench [7] to collect measurements of resource usage and consumed time. In the simulated environment, calibrated with the measurements collected with the bare-metal infrastructure, we evaluate and compare two scheduling policies using the Batsim/ Simgrid simulator [8]. The two scheduling policies are *FOA*, and a baseline inspired by a Kubernetes scheduling plugin. With the simulators, we also model serverless-based heterogeneous edge-cloud platforms and batches of workloads with functions that depend on container images. We evaluate both scheduling policies in terms of cost, makespan, system utilization (the number of machines used), and processing time while studying the impacts of considering platforms’ heterogeneity at the scheduling phase. The experimental results show that considering the heterogeneity of platforms at the scheduling phase impacts a lot in the efficiency of serverless platforms in the edge-cloud continuum. *FOA* outperforms the baseline for makespan, cost, and system utilization by up to two orders of magnitude.

The main contributions of this paper are:

- A multi-objective scheduling policy called *FOA*, which in comparison with the greedy-based policy baseline, improves (i) the makespan, (ii) the amount of downloaded data, and (iii) the system utilization.
- The extension of the benchmark FunctionBench [7] to the OpenWhisk [6] platform;
- An evaluation methodology and a setup that is completely reproducible and can be extended to other serverless-based heterogeneous edge-cloud platforms [9];
- A detailed study of the gaps between a greedy and a multi-objective algorithm for scheduling policies in serverless platforms at the edge-cloud continuum;

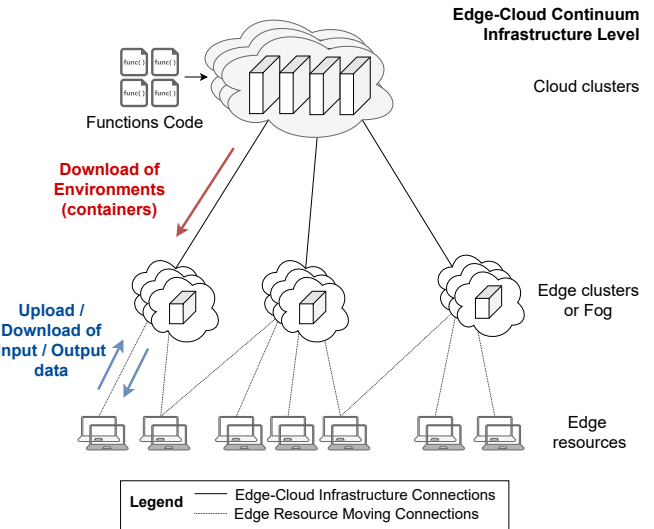


Figure 1: Infrastructure of the Edge-Cloud Continuum. The global level has the total view of the continuum, and the local level only sees the edge clusters and resources.

The paper is organized as follows. Section II presents a few preliminary concepts and definitions used in this paper. Section III presents a background and literature review, where we emphasize our approach against related works. Section IV presents *FOA*, and Section V presents our methodology. Section VI presents the experimental results, and finally, in Section VII we discuss our conclusions and future works.

## II. PRELIMINARY CONCEPTS

In the computing continuum, the infrastructure of a serverless-based heterogeneous edge-cloud platform can be characterized as illustrated in Figure 1, by three levels, namely here as cloud clusters, edge clusters (or Fog), and edge resources. We define the cloud cluster level as a centralized, or on-premise, set of servers performing compute-intensive work, including batch aggregation and data processing; the edge resources level as a composition of mobile edge or IoT devices that are harvesting and preprocessing data before sending it to the higher level for computation; the intermediate level as a set of geographically spread middle-size clusters, sometimes called edge clusters or Fog. As illustrated in Figure 1, the edge cluster level enables communications with the edge resources proposing computation and data treatment with lower latency and lower data transfers (in comparison with the cloud), while assuming the task of data aggregation and processing with more powerful computation resources (in comparison with the far edge resources). The cloud and edge clusters, along with the far edge resources are naturally composed of heterogeneous hardware resources with different computation power (CPUs, RAM), storage, and networks among them.

In the remaining of the section, we present the main terms we use in this paper, defining respectively a) serverless functions, cost, and container layers; b) edge-cloud continuum and local clusters; and c) heterogeneity level:

### **Serverless Functions, Cost, and Container Layers:**

Serverless can be characterized by small functions with a short execution time. In addition, we consider that Serverless functions are executed inside containers, and their deployment brings extra costs for the functions' execution. Such costs can be understood in many ways, such as time, network bandwidth, or image size. In this paper, the cost is the sum of the amount of data transferred for the deployment of containers' images and the amount of data transferred by functions I/O. In addition, containers are built by layers, and such layers can be shared among different containers. Hence, if properly scheduled, even different functions can benefit from sharing container layers.

**Edge-Cloud Continuum and Local Clusters:** The edge-cloud continuum is defined by an architecture with different layers composed of several cloud clusters, edge clusters and possibly edge resources [10]. In this context, each cluster can be independent and self-managed while being connected with the other clusters of the continuum. They can follow similar rules and APIs, which enables them to exchange information and collaborate in the execution of applications. Hence, there are two important levels: a) the high-level global continuum, which has the view to all -possibly- heterogeneous clusters and resources of the continuum; and b) the local cluster level, which consists, in general, of a group of homogeneous resources. Concerning the scheduling, we adopt a one-phase scheduling mechanism, which allows the selection of resources and placement of tasks directly from the high-level global view.

**Heterogeneity Level:** In this paper, we represent the heterogeneity of the platforms by modeling different edge-cloud clusters. We characterize light heterogeneity by different CPU speeds and a homogeneous network. We call heterogeneity level the number of different edge-cloud clusters in our platform. Each cluster is composed of identical machines with the same CPU speed. The CPU speeds are different among the different clusters.

### **III. BACKGROUND AND RELATED WORKS**

Serverless computing has emerged as a new paradigm of abstraction, platform, and implementation of cloud functions [11], [12]. It has been seen as an evolution of the cloud computing model in the sense of using microservices and containers. The FaaS concept was first presented by AWS in 2014 with Lambda [13]. After that, other vendors, such as Google, Microsoft and IBM followed AWS and introduced their platforms as respectively Google Cloud Functions [14], Microsoft Azure Functions [15], and IBM Cloud Function [16]. Several associated challenges in the literature are generally grouped, such as system or programming models, as pointed out in a global view by Baldini et al. [12]. Other groups are proposed by Jonas et al. [2], such as abstract, network, security, and architecture. Despite the different classifications, the raised challenges are common among all these works. In addition, due to the different vendors, migrating solutions from the platforms is not an easy task [17].

Several platforms have emerged in the literature to address the challenges presented above. Considering data management, for example, due to the stateless nature of serverless computing, it is challenging to manage such platforms. Klimovic et al. present Pocket, which provides its efficiency by a design of an elastic, fully managed cloud storage service [18]. Mahmoudi and Khazaei present the SimFaaS platform [19]. It is an open-source serverless platform that allows the study of scheduling policies. Besides many works for serverless exploit the usage of containers, as we do, Utiugov et al. presented a technique to reduce functions' cold start latency based on snapshots [20]. They save the current state of a VM on the disk and use it when needed.

FaaS applications can benefit from containerization, so they also can be managed by Kubernetes-based platforms. Serverless computing extends the FaaS concept by avoiding server infrastructure management. However, Kubernetes was not developed for Edge-Cloud computing scenarios, then it is still sensitive to characteristics, such as multi-tenancy and fast deployment and execution of functions. To address these issues, several serverless platforms were developed on top of Kubernetes, such as Kubeless [21], OpenWhisk [6], and OpenFaaS [22]. These platforms automatically handle the Kubernetes configuration side to make it easy for developers to upload, deploy and execute their functions. In addition, there are available some open-source schedulers for Kubernetes, such as YuniKorn [23], IBM Safe Scheduler [24], and IBM Multiple Cluster Dispatcher [25].

To better understand and exploit the possibilities with the scheduling policies, simulation is a key practice. With simulations, one can easily perform and compare different scheduling policies. However, it is needed sets of workloads to describe different scenarios. Kim and Lee [7], [26] presented micro/ application benchmarks for serverless platforms. The microbenchmarks measure the performance of target resources with a function call, such as matrix multiplication, linpack, chameleon, and iperf3. The benchmarks provide applications with realistic scenarios, dealing with data-oriented flows and several resources. Some examples are image/video processing, logistic regression, face detection, and word generation. SimGrid [8] is a framework that allows the development of simulators to be used for prototyping, evaluating, and comparing system designs, platform configurations, and algorithmic approaches. On top of this framework, Batsim [27] was developed as a resource and job management system (RJMS) simulator.

Scheduling algorithms are used for several reasons, such as to minimize function response time, to save costs, and to reduce data movements or energy consumption. Shmoys and Tardos developed a dual-approximation algorithm [28] to assign independent tasks to unrelated machines and bound their approximation to, at most, the cost and twice the makespan of the optimal solution. It uses a linear program to get an assignment of tasks to machines, but its solutions are not necessarily integral, and hence an integral matching is required. The properties and existence of such a matching

are detailed in depth by Plummer and Lovász [29]. Inspired by the dual-approximation algorithm of Shmoys and Tardos, we propose our scheduling policy, called *FOA*, to assign independent serverless functions to heterogeneous serverless platforms at the edge-cloud continuum.

Rausch et al. propose a container scheduling system called Skippy that enables serverless frameworks to support edge functions [4]. They exploit Kubernetes standard scheduler to attach and tune weights to priority mechanisms that may lead to low function execution time, efficient resource usage, and reduced network traffic and costs. In addition, they leverage the container layer sharing mechanism by estimating a sharing percentage among their Python functions on their workload and using it during the scheduling phase. Differently, with *FOA*'s input, we know in advance the possible performances of all functions in all machines. Hence, we schedule each function in a way to maximize the sharing of containers without compromising the makespan of the platform. In addition, this approach makes our algorithm available for any serverless workload and functions. Li et al. propose Pagurus, a runtime container management system for reducing cold startup [30]. Pagurus is comprised of an inter-action container scheduler that schedules shared containers among actions. They do it by creating what they call zygote containers: containers with common packages among the recurrent functions. These zygote containers are used to speed up the starting time of a function by just installing the different packages that are not yet inside the container. Instead of creating containers with the basis that can be used by different functions that may arrive on the platform, our algorithm decides where to deploy a function knowing in advance which container is already available on the nodes. Aumala et al. proposed a scheduling policy based on the packages needed to execute a function [31]. The package-aware scheduling proposed, PASch, considers the package affinity during scheduling. They map two affinity workers that cache the largest package required by the functions and choose the least loaded one. Similarly, we are concerned about function requirements locality. However, we deal with them at the container. We prioritize the machines with the required container available.

Suresh and Gandhi used OpenWhisk to implement a scheduling policy, named FnSched, which focused on costs (resources) reduction [32]. For that, they developed and combined two algorithms, the first one called cpu-shares regulation and the second one called greedy. The cpu-shares regulation algorithm regulates how much cpu the instances will use. They define a latency ratio and verify it over time. If an instance achieves the latency ratio, it receives more cpu-shares capacity. The greedy algorithm will take care of allocating and scaling up the instances. It checks the available memory of the host, and just if needed, more invokers are used. In our case, we do not need to verify the platform loading because all scheduling decisions are taken at once at the beginning. Zuk et al. proposed some online node-level scheduling policies based on known policies, such as First-In, First-Out (FIFO), Shortest Expected Processing Time (SEPT), and Earliest Expected

Completion Time (EECT). They modified these scheduling policies to use historical data, estimating the frequency and execution of function calls, with which they reduced cold starts [33]. To use *FOA*, we do not gather historical data within the online usage of the platform. However, we assume that the provider may know in advance information, such as the execution time and amount of data downloaded by functions and containers regularly executed on the platforms. With this information, we reduce cold starts by reducing the amount of data downloaded by containers.

#### IV. A MULTI-OBJECTIVE SCHEDULING POLICY FOR SERVERLESS

Inspired by the dual-approximation algorithm of Shmoys and Tardos [28], we propose *FOA*, a scheduling algorithm that enables the allocation of batches of serverless functions on serverless-based heterogeneous edge-cloud platforms. *FOA* minimizes both makespan and cost (the sum of the amount of data transferred to deploy containers and the amount of data transferred by functions I/O). This section details this scheduling policy, its algorithm, and its two main components: the linear program and the integral matching process. In addition, we detail the container layer download optimization model used.

##### A. *FOA*: *Function Orchestration Algorithm*

Our scheduling policy named *FOA* has two objectives, to reduce cost and makespan. It works under the following assumptions: a) functions depend on environments and all dependencies are known; b) functions and environments are independent between themselves and known in advance; c) their cost and processing time on each machine are known. *FOA* expects the following inputs: 1) environment execution time: the time to download and deploy functions environments (containers), in seconds (s); 2) environment cost: the amount of downloaded data per environment, in megabytes (MB); 3) function execution time: the execution time of serverless functions, in seconds; 4) function cost: the amount of data transferred as functions' I/O, in megabytes.

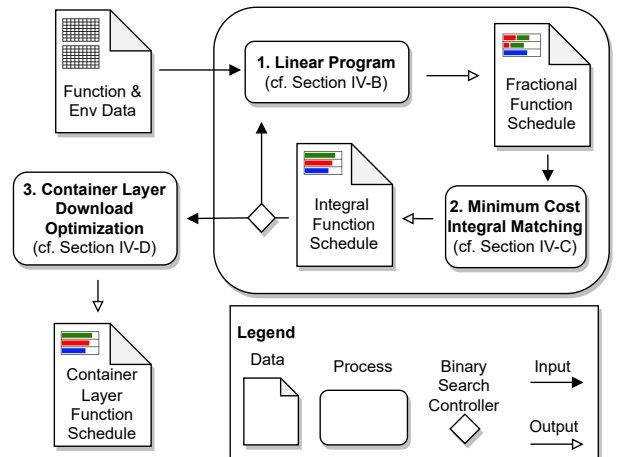


Figure 2: *FOA*'s Algorithm step by step.

Figure 2 illustrates all steps performed by FOA. Step 1. *Linear Program* (detailed in Section IV-B) uses function and environment data to produce a fractional function schedule (to get the results fast enough). However, we need an integral solution for allocating functions in our context. Then, step 2. *Minimum Cost Integral Matching* (in Section IV-C) converts the fractional function schedule into an integral function schedule. Finally, step 3. *Container Layer Download Optimization* (in Section IV-D) models the sharing of container layers done by Docker [34] in Kubernetes, for each function that arrives in the machines. It reduces the amount of data downloaded and execution time based on the cache state of the machines. Our algorithm optimizes the cost of the entire workload, under a constraint on the makespan, of an arbitrary value  $T$ . In order to optimize the makespan, we re-execute steps 1 and 2 of FOA up a given number of times until reaching the expected precision, before starting step 3. These repetitions update the makespan constraint  $T$  at each iteration by a binary search process. The *Binary Search Looping Controller* component, illustrated in Figure 2, decides when this process is finished. The first iteration of the algorithm provides a solution with the best cost because the makespan constraint is fully relaxed. As far as the makespan decreases through the binary search process, the cost increases. Empirically, eight iterations are sufficient.

### B. FOA's Linear Program

FOA's linear program, presented in Figure 2, uses the list of variables and notations presented in Table I. It schedules the functions on the machines minimizing the total cost under a makespan constraint of value  $T$ . We use a single-objective constraint problem (makespan) to exploit the trade-off between both objectives (makespan and cost). This way, we are able to study accurately how each objective affects the optimization. Equation 1 describes such an objective function:

$$\text{Min}(\sum_{i=1}^M \sum_{j=1}^N c_{ij} \times x_{ij} + \sum_{i=1}^M \sum_{k=1}^K d_{ik} \times e_{ik}) \quad (1)$$

Under the following constraints:

- For each function, the function is done completely (the minimization allows to relax the strict equality);

$$\forall j \leq N, \quad \sum_{i=1}^M x_{ij} \geq 1 \quad (2)$$

- For each machine, a machine is not filled with more than  $T$ ;

$$\forall i \leq M, \quad \sum_{j=1}^N p_{ij} \times x_{ij} + \sum_{k=1}^K b_{ik} \times e_{ik} \leq T \quad (3)$$

- For all machines, for all environments, it is pre-computed the maximum number of functions per environment ( $N_{ik}$ , integer) that could be deployed on the machine with a makespan of  $T$ , taking also into account the environment deployment time.

$$\forall i \leq M, \forall k \leq K, \quad \sum_{env[j]=k} x[i][j] \leq N_{ik} \times e[i][k] \quad (4)$$

- An environment is present at most once per machine;

$$\forall i \leq M, \forall k \leq K, \quad e_{ik} \leq 1 \quad (5)$$

Notation	Description
$M$	Number of machines
$N$	Number of scheduled functions
$K$	Number of container environments
$c_{ij}$	Cost of the $j$ -th function on the $i$ -th machine
$p_{ij}$	Execution time of the $j$ -th function on the $i$ -th machine
$d_{ik}$	Cost of the $k$ -th environment on the $i$ -th machine
$b_{ik}$	Execution time of the $k$ -th environment on $i$ -th machine
$env[j]$	Environment id of the $j$ -th function
$x_{ij}$	Placement of the $j$ -th function on the $i$ -th machine
$e_{ik}$	Placement of the $k$ -th environment on the $i$ -th machine

Table I: FOA's list of notation and descriptions.

Going further, Equation 4 is the central point of our linear program. We relaxed the quality of the fractional solution in order to get a much faster linear program computation. To verify that the environment of each function is on the target machine, a simple constraint such as  $x[i][j] \leq e[i][env[j]]$  implies a large set of  $(N \times M)$  difficult constraints to solve. The  $K \times M$  constraints in Equation 4 check only that at least a fraction of the environment is there. The quality of the relaxation depends on the evaluation of the maximum number of functions of a particular environment that fit in time  $T$  on the machine ( $N_{ik}$ ). In our targeted platform, we expect a small number of functions per machine. The computation of  $N_{ik}$  could be done by sorting the execution time of all the functions of one environment on the machine in increasing order, then computing the prefix vector (an entry is the summation of all previous execution times). If at the index  $n + 1$  of the prefix vector, the sum of computation is strictly larger than  $T$  minus the environment deployment time, then  $N_{ik} = n$ . Using this relaxation, the computation of the linear program, even with free solvers, stays in a seconds-to-minutes time interval.

### C. FOA's Minimum Cost Integral Matching

Shmoys and Tardos [28] show that an X-perfect fractional matching can be converted efficiently to an X-perfect integral matching. They do it by constructing a bipartite graph where one side consists of job nodes, the other consists of machine nodes, and the edges between both sides correspond to job assignments to machines. For instance, let us consider one of their examples from [28], with 3 machines, 7 functions, no cost (for simplicity), and processing times equal to 3s for the first function, and 1s for all the others. This way, step 1. *Linear Program* provides the fractional function schedule solution as illustrated in Figure 3. Hence, step 2. *Minimum Cost Integral Matching* constructs an integral function schedule in two sub-steps, by a) constructing a function-perfect fractional matching, as illustrated in Figure 4, with dotted lines weighing  $1/3$  and full lines weighing  $2/3$ ; and b) converting the function-perfect fractional matching in a function-perfect integral matching as illustrated in Fig 5. Finally, the integral function schedule, illustrated in Figure 6, is constructed by allocating functions  $j$  on machines  $i$  if there exist edges between any machine  $i$  and function  $j$ .



$$\begin{bmatrix} 1/3 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Figure 3: Fractional Function Schedule in the *Minimum Cost Integral Matching* step. Each line is a machine and each column is a function.

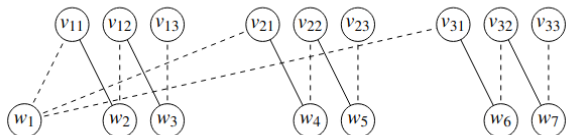


Figure 4: Fractional Matching in the *Minimum Cost Integral Matching* step. Machine  $i$  is represented by the set  $v_{in}$  and function  $j$  by  $w_j$ .

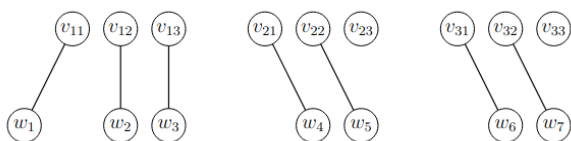


Figure 5: Integral Matching after the *Minimum Cost Integral Matching* step. Machine  $i$  is represented by the set  $v_{in}$  and function  $j$  by  $w_j$ .

Machine V1	$w_1$	$w_2$	$w_3$			
Machine V2	$w_4$	$w_5$				
Machine V3	$w_6$	$w_7$				

Figure 6: *Integral Function Schedule* represented as an allocation matrix.

#### D. A Model for Container Layer Download Optimization

To exploit the composition of containers and their layer-sharing mechanism, we keep a list of container layers for each machine of the platform. For each function that will be executed, we verify its required container and retrieve its list of layers. We cross this list with the list of container layers in the selected machine, and we verify the layers that can be re-used. It speed-up the deployment of the containers and, therefore, the functions that required them. If it is not possible to reuse any container layer present in any machine, the entire container should be downloaded and deployed.

## V. METHODOLOGY

We evaluate *FOA* by comparing it with a baseline policy inspired by the Kubernetes scheduling policy. More specifically, its image locality plugin [35]. To conduct such experiments, we performed simulations on top of Batsim/Simgrid [8], and Figure 7 illustrates this simulation setup. We present in this section how we model our workloads, platforms, and the baseline policy. We also detail the simulated environment that makes possible the study of these scheduling policies, and the grid of experiments conducted.

Function	Input Values	Unit
Chameleon	$2.10^3, 3.10^3$	Matrix size
Float operation	$10.10^7, 20.10^7, 30.10^7$	Operations
Image processing	60, 40	Image size in MB
Linpack	$5.10^3, 6.10^3$	Matrix size
Matrix Multiplication	$3.10^3, 4.10^3$	Matrix size
Model Training	50, 100	Dataset size
Pyaes	$3.10^3, 4.10^3, 5.10^3$	Length of message
Video Processing	30, 50, 100	Video size in MB

Table II: Functions adapted from FunctionBench [7], and the different input values used. Each combination of function name and input value characterizes one profile of our workloads. In total, we have 9 different functions and 19 profiles.

#### A. Modeling Workloads for Serverless

We perform bare-metal executions of functions that we adapted from the FunctionBench [7] benchmark, such as matrix multiplication, linpack, chameleon, modeling training, and image and video processing. We evaluate 9 functions with different inputs. In total, we have 19 combinations of functions and inputs, as presented in Table II. Each function requires a different container, being 9 in total. The container sizes vary from 170 to 2560 MB. All of them are available on a public repository on *DockerHub*. We deploy the serverless platform OpenWhisk on top of GRID5000 [5] and execute our functions there. For each function executed, we obtain its execution time and resource usage measurements (CPU, memory, bandwidth, etc.). We also instrument the functions to extract the time consumed by different phases of serverless functions such as download and deployment of containers, functions execution, I/O transferring, containers deletion, etc. At last, we perform a calibration phase to estimate the number of floating-point operations (flops) necessary to execute each function, which allows Batsim and Simgrid to accurately perform the simulations.

Our workload model is based on such executions of serverless functions on serverless-based heterogeneous edge-cloud platforms. For each function, we translate the data retrieved from the bare-metal executions to the Batsim/Simgrid requirements and format. We use random seeds to randomly select different combinations of functions and inputs for each workload created. In addition, for each function and its required container, we retrieved the container layers' composition. Such a description was also attached to the workloads to reproduce the layers' sharing behavior during our simulations.

#### B. Modeling Platforms for Serverless

To model our platform, we defined a range of valid *CPU* computation power, with a minimum value representing an ARM CPU, up to one of the newest CPUs evaluated in general benchmarks. This range goes from 2000 to 105000 megaflops (MFlops). To reproduce the heterogeneity of an edge-cloud continuum composed of different edge-cloud clusters, we defined the heterogeneity level (see Section II). Each cluster contains only one type of machine, with a fixed CPU computation power, generated randomly after fixing a random seed. For instance, with a fixed platform size of 300, a fixed

heterogeneity level of 3 means that we have 3 different types of machines, one per cluster. Then every 100 identical machines will belong to the same cluster, and we have three different edge-cloud clusters.

### C. A Container Locality Baseline Policy

Our baseline belongs to the high level of the edge-cloud continuum, but it considers the dynamic that exists at the local level of our platform (edge clusters). This baseline is inspired by the Kubernetes scheduler, more specifically, by the Image Locality plugin [35]. The algorithm works as follows: for each function in the queue, it 1) gets the container required; 2) searches the container on the available machines and scores them; 3) sorts the list of available machines by their scores; 4) selects the first machine in the sorted list. If none of the machines has the required container, it will be downloaded in the first one of the list (the selected one). However, at the deployment phase in Kubernetes, managed by Docker [34], if the machine selected to execute a function does not have the container required but has any container layer that can be shared, it will be done, and the download and deployment of the required container will speed up.

### D. A Simulated Environment

As illustrated in Figure 7, Batsim receives as input a workload, a platform description, and connects to scheduling policy. During the simulation, the scheduling policy allocates each function from the workload to the machines described in the platform. When a function allocation decision is taken, Batsim communicates it to Simgrid, which will perform the actual simulation. Finally, to build the evaluated scheduling policies, we used a Python API layer on top of Batsim called PyBatsim.

### E. Design of Experiments

To evaluate the gap of performance and efficiency between *FOA* and *K8S ImageLocality* for different scenarios, we designed a set of experiments, which are presented in Table III. We vary workloads and platform sizes, as well as the slight levels of heterogeneity of the platform. To statistically validate each combination of workload, platform, and heterogeneity level, we use 30 random seeds to create different workloads and platforms for each combination. We execute all scenarios with both scheduling policies, *FOA* and *K8S ImageLocality*. In total, we performed 1620 experiments.

Parameters	Values	Unit
Workload Size	200, 600, 1000	Number of functions
Platform Size	100, 300, 500	Number of machines
Heterogeneity Level	3, 5, 7	Number of levels
Scheduling Policies	<i>FOA</i> , <i>K8S ImageLocality</i>	-
Random Seeds	30	Number of seeds

Table III: Design of Experiments.

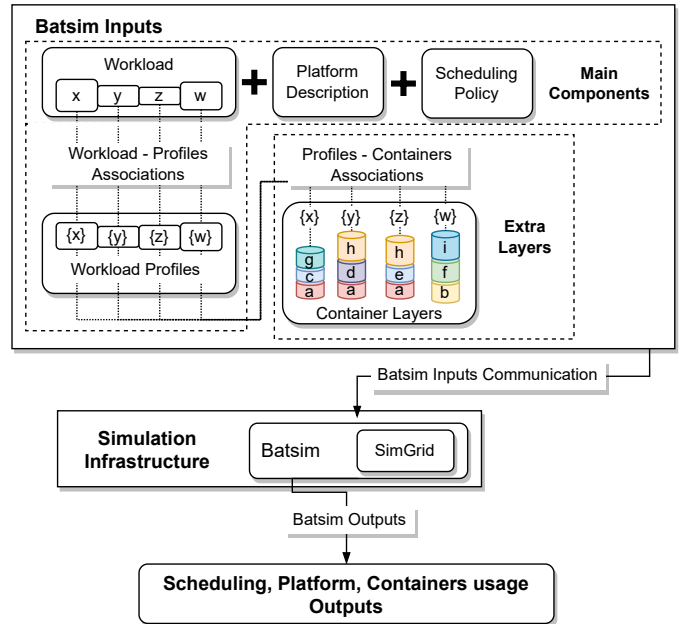


Figure 7: Simulated environment infra-structure illustration. Batsim and Simgrid, as the simulators, communicate between them directly. Batsim receives as input a few main components, and to enrich the model, workloads are described with an extra layer: container layers.

## VI. EXPERIMENTAL RESULTS

The evaluation process analyzes our multi-objective policy, which reduces both cost and makespan. In this section, we present *FOA*'s linear program performance and we compare it against our baseline, *K8S ImageLocality*, in terms of makespan, cost, and the number of machines used. We remark that we are interested in cases where there are not many functions per machine.

### A. *FOA*'s Linear Program Results

*FOA*'s linear program resulted in fractional solutions optimizing cost and makespan. However, it is not possible to optimize both simultaneously. Minimizing the cost is simple by allocating all tasks of the same environment to a single fast machine. But, this solution does not minimize the makespan. Hence, optimizing cost and makespan is a compromise. Figure 8 illustrates this compromise through *FOA*'s binary search process (explained in Section IV) for all combinations of workload size and platform size. The heterogeneity level is not distinguished because the three levels show similar behaviors. The x-axis is the makespan (in minutes), and the y-axis is the cost (the amount of downloaded data in GB). The colors represent the iterations of *FOA*'s binary search process. The first iteration computes the smallest cost and hence the highest relevant makespan. As far as we constrain the makespan over the iterations, the cost increases. We highlight that (i) the 3rd iteration looks to be a good trade-off between both objectives, (ii) the Pareto's shape is quite smooth in relevant scenarios for



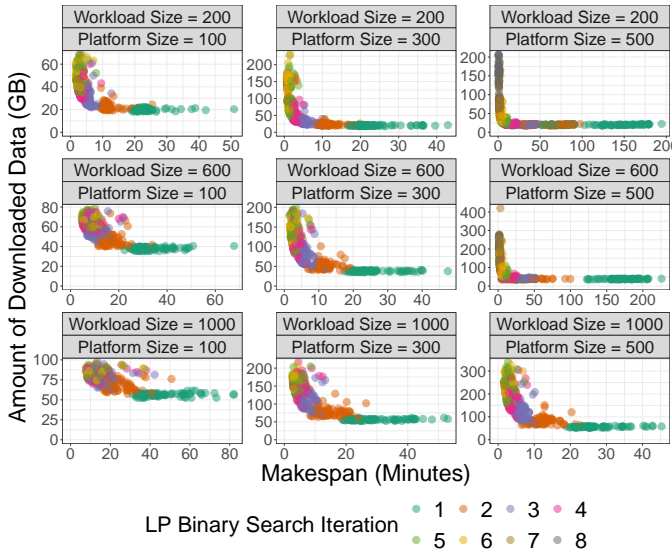


Figure 8: *FOA*'s linear program trade-off between makespan (x-axis) and amount of downloaded data (y-axis). The color of the points represents the iteration-id of the linear program binary search. Each one is one solution of the linear program.

us. In addition, we remark that (a) not all instances found a solution of cost with small values for makespan, and (b) since we need to re-execute the linear program at each iteration, the processing time to produce a final solution is cumulative.

We call the time that a scheduling policy takes to decide the allocation of the functions as processing time. *FOA* is based on a linear program with several constraints, as shown in Equation 1, which is expensive in terms of processing time. Unlike our approach, the *K8S ImageLocality* baseline has a greedy algorithm and does not have a global view of the platform during the scheduling phase. Hence, *K8S ImageLocality* produces a non-optimal allocation decision within a second while *FOA* reaches a better solution within minutes. Our scheduling policy takes a median of 2.69 minutes to produce a decision, while our baseline performed in the median of 0.6 seconds. However, *FOA*'s decision considered the eight repetitions of its binary search process. Each repetition performs with a median of 0.34 minutes. Considering that, and as mentioned above, our results show that three repetitions are enough for good results, then *FOA* may produce them with a median of 1.02 minutes. It is still not as fast as *K8S ImageLocality*, but it is reasonable for evaluating the gap between both scheduling policies. We also remark that in our experiments, we use an open-source solver, *CBC*, through a library to compute our linear program, *python-mip*. Using a commercial solver would reduce the processing time but also would hinder the system administration simplicity of our computations.

### B. Simulation Results: Makespan, Cost, and Resources Usage

Through our simulated environment, we compare *FOA*'s performance against *K8S ImageLocality* in terms of makespan, cost and number of machines used. In addition, we evaluate the average percentage gain of *FOA*, and we emphasize that all



Figure 9: Comparison of *K8S ImageLocality* and *FOA* in terms of makespan (y-axis) against the heterogeneity level (x-axis).

simulations consider the container layer download optimization for both algorithms. Figure 9 presents the experimental results for makespan (in minutes), Figure 10 the experimental results for cost (in MB), and Figure 11 the experimental results for system utilization (in number of machines used). The three figures show with different facets the combined scenarios of workload and platform sizes. The scheduling policies are illustrated with grayscale, the x-axes present the heterogeneity level, and the y-axes present the boxplots of the different analyzed parameters, respectively, makespan, cost and, number of machines used.

1) **Makespan:** Figure 9 shows that both algorithm behaviors are stable in all scenarios, and *FOA* is slightly better, and with less variability, than *K8S ImageLocality* for almost all of them. Besides, as far as the heterogeneity level increases, the small gap increases as well. Nevertheless, in the important scenarios for us, *FOA* outperformed *K8S ImageLocality* by an order of magnitude. To summarize, when the number of functions is large in comparison with the platform size, greedy algorithms focused on container locality achieve good makespan performance. Otherwise, as the heterogeneity level increases or the workload size decrease, the gap becomes significant with an algorithm that manages such heterogeneity.

2) **Amount of Downloaded Data (Cost):** Figure 10 shows that *FOA* and *K8S ImageLocality* behavior are stable in all scenarios. In all cases, the amount of downloaded data of *K8S ImageLocality* is from one or two orders of magnitude more than *FOA*'s solution. Moreover, the gap increases with the increase of workload and platform size. To summarize, as far as we increase the number of functions in the platform, there are more data transfers to be managed. Hence, the placement of the functions is very important to minimize the amount of data transferred. An algorithm that optimizes function placement, such as *FOA*, shows much better management than greedy algorithms such as *K8S ImageLocality*.

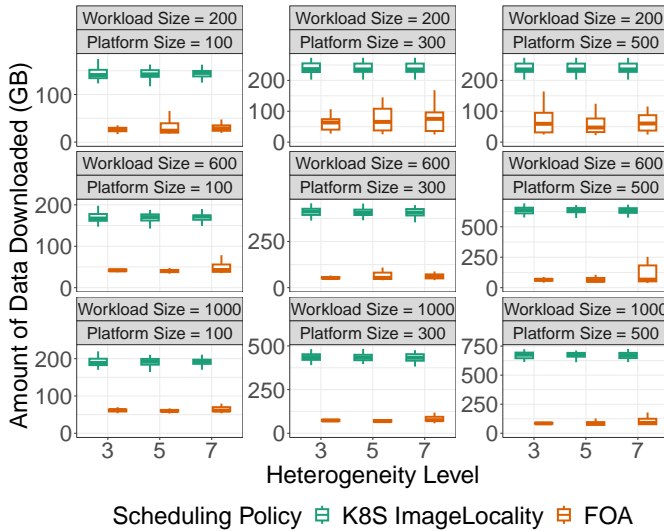


Figure 10: Comparison of *K8S ImageLocality* and *FOA* in terms of the amount of downloaded data (y-axis) against the heterogeneity level (x-axis).

3) **Number of Machines Used:** Greedy algorithms, such as *K8S ImageLocality*, use as many machines as possible. Note that in cases where there are fewer functions than machines (top-right), it is not possible to use all machines. In the other cases, Figure 11 shows that algorithms that optimize placement, such as *FOA*, better choose the machines to be used. Hence, they use fewer machines while achieving better performance (makespan, and amount of data downloaded) and thus, efficiency. Besides, slight levels of heterogeneity do not affect much the number of machines used in our experiments. In summary, when there are many functions per machine, both algorithm uses almost all machines. In the other cases, the important scenarios for us, the experiments show that similar performances can be achieved with significantly less machines.

4) **Number of Machines Used versus Makespan and Amount of Data Downloaded:** Figures 12 and 13 present, respectively, the makespan and the amount of data downloaded against the number of machines used by *FOA* and *K8S ImageLocality* for several scenarios of workload size and platform size. The heterogeneity level is presented by the colors. We added a small jitter to clear the overlapping of points. As *K8S ImageLocality* always uses as many machines as possible, Figure 12 shows all instances of that in the extreme right. In the relevant cases for us, *FOA* uses much fewer machines for comparable makespan. Figure 13 shows that *FOA* reduces the amount of data downloaded from one to two orders of magnitudes. We remark that (i) the greedy algorithm *K8S ImageLocality* is far from the best solution for the amount of data downloaded, (ii) by better choosing the placement of the function, *FOA* also reduces the number of machines used.

## VII. CONCLUSION AND FUTURE WORK

In this work, we study the usage of serverless platforms in the edge-cloud continuum. We evaluate the impacts of considering the heterogeneity of the platforms at the scheduling

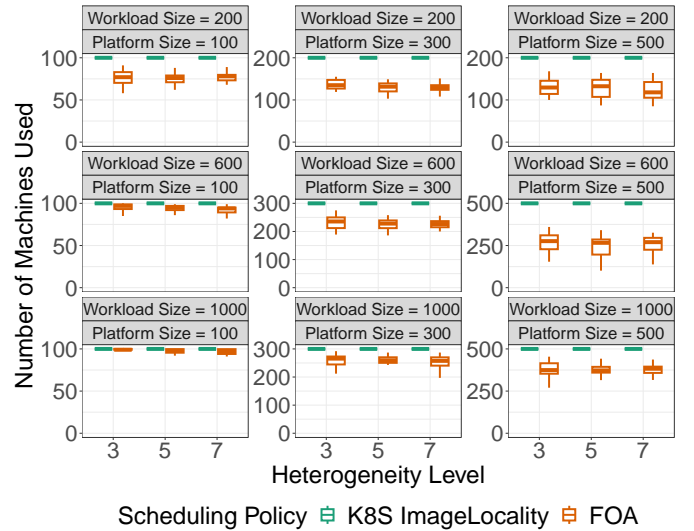


Figure 11: Comparison of *K8S ImageLocality* and *FOA* in terms of the number of machines used (y-axis) against the heterogeneity level (x-axis).

phase while optimizing data transfers and functions' execution time. For that purpose, we develop a multi-objective scheduling policy, called *FOA*, that enables the allocation of batches of serverless functions on heterogeneous edge-cloud platforms with the capability to minimize the makespan and the amount of data downloaded to deploy containers and functions' I/O. We implement a greedy algorithm as a baseline, called *K8S ImageLocality*, inspired by the Kubernetes scheduling policy. We evaluate the efficiency gap between both scheduling policies for the makespan, the amount of data downloaded, and the number of machines used. For the experimental campaign, we adapt serverless functions from the FunctionBench benchmark. We deploy the OpenWhisk serverless platform on the academic cluster GRID5000 and then execute the adapted functions there. With that, we model our workloads, which are used in a simulated environment on top of the Batsim/SimGrid simulators. To study the impacts of the heterogeneity of the platforms, we model slight levels of heterogeneity on top of our serverless platform.

Our experimental results show that standard cloud greedy algorithms, such as our baseline, may not profit from the best efficiency of heterogeneous serverless platforms at the edge. *FOA*, on the contrary, optimizes the placement of functions by its multi-objective policy. It outperforms our baseline for both data transfers and makespan criteria, in addition to the system utilization by up to two orders of magnitudes. We remark that *FOA* is robust regarding the heterogeneity, and it is not affected by the different levels of heterogeneity studied, producing results with the same quality for all of them. However, *FOA* is very time-consuming. It is based on a linear program with several constraints, which is costly in terms of processing time. Our results showed a processing time in orders of minutes for *FOA*, while *K8S ImageLocality* performed in order of a second.

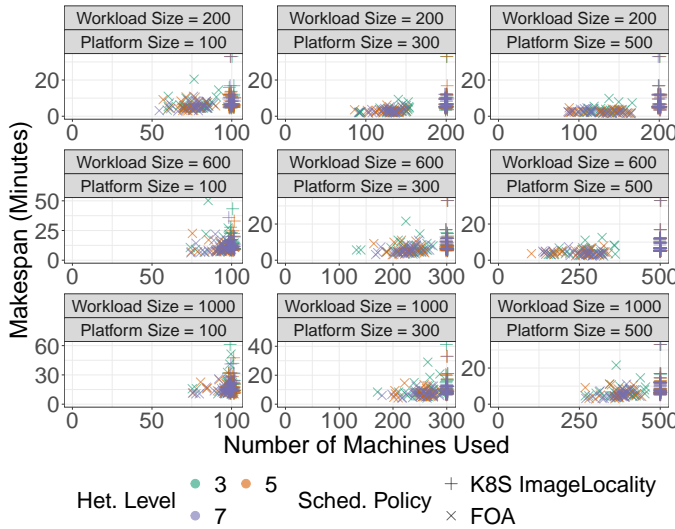


Figure 12: The makespan (y-axis) against the number of machines used (x-axis). Facets combine workload and platform sizes. Shapes represent the scheduling policies, and colors represent the heterogeneity level.

We conclude that even with light levels of heterogeneity, the gains of *FOA* are important in serverless computing at the edge-cloud continuum, and we believe that with a more accurate model of heterogeneity, these gains may increase. By reducing the amount of data transferred to download containers, we minimize cold start delays through faster container deployments. In addition, it also speeds up the functions’ execution time. We emphasize that the results are even better in the scenarios important for us, where there are not many functions per machine. Thus, efficient scheduling policies for serverless computing at the edge-cloud continuum require better management of the heterogeneity to drastically improve the amount of data downloaded and the system utilization.

In the future, we plan to follow three main directions. The first direction is to study an approach with two levels of scheduling that may be more adapted to the edge-cloud continuum. Instead of having one algorithm in the global level of the continuum that allocates the functions to the resource at the local level, as *FOA* does, we will enable the first level to decide the best edge cluster to be used, and the second level will decide the final allocation of the functions. With that, cloud and edge clusters can benefit from eventual mobility. For instance, mobile edge resources will profit from complete autonomy and besides local resource management, they will also have local scheduling and orchestration. In addition, this is important to handle intermittent network communications, which can often happen in mobile edge cases. Within the second direction, we plan to study applications that can be modeled as workflows of serverless functions, which is an increasingly used programming style (i.e. AWS Step Functions) and allows developers to describe more complete and complex application logic. Naturally, the above platforms’ and applications’ characteristics introduce further complexity in

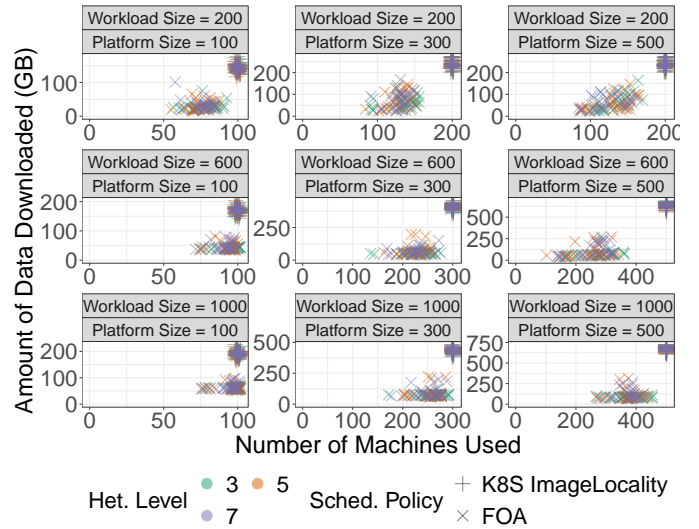


Figure 13: The amount of data downloaded (y-axis) against the number of machines used (x-axis). Facets combine workload and platform sizes. Shapes represent the scheduling policies, and colors represent the heterogeneity level.

resource management and scheduling which will need to be tackled in our future study. Furthermore, in our third direction, we are interested in adding more objectives to our multi-objective scheduling policy such as energy consumption and latency minimization. A complementary direction that will be explored is to experiment *FOA* with other linear program solvers in order to investigate if some implementations can provide faster processing time.

#### AUTHORS CONTRIBUTION

Modeling of serverless edge cloud platforms: DA SILVA, GEORGIU, and MERCIER; Problem modeling: All authors; Design and implementation of algorithms (*FOA*): DA SILVA, ANGELELLI, MOUNIÉ, and TRYSTRAM; Design and implementation of algorithms (*K8S ImageLocality*): DA SILVA, MERCIER, and GEORGIU; Design and execution of experiments: DA SILVA; Analysis and investigation: DA SILVA, MOUNIÉ, and TRYSTRAM; Writing: DA SILVA, MOUNIÉ, TRYSTRAM, GEORGIU, and MERCIER; Reproducible artifact: DA SILVA. All authors have read and agreed to the published version of the manuscript.

#### ACKNOWLEDGMENTS

This work was supported by the research program on Edge Intelligence of the Multi-disciplinary Institute on Artificial Intelligence MIAI at Grenoble Alpes (ANR-19-P3IA-0003), and by the European Union’s Project H2020 PHYSICS (GA 101017047). Experiments presented in this paper were carried out using the Grid’5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see <https://www.grid5000.fr>).

## REFERENCES

- [1] D. Rosendo, A. Costan, P. Valduriez, and G. Antoniu, "Distributed intelligence on the edge-to-cloud continuum: A systematic literature review," *J. Parallel Distributed Comput.*, vol. 166, pp. 71–94, 2022.
- [2] E. Jonas *et al.*, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv:1902.03383 [cs]*, Feb. 2019.
- [3] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," 10 2019.
- [4] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, 08 2020.
- [5] "Grid5000." [Online]. Available: <https://www.grid5000.fr/>
- [6] "Openwhisk." [Online]. Available: <https://openwhisk.apache.org/>
- [7] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. Milan, Italy: IEEE, Jul. 2019, pp. 502–504.
- [8] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014.
- [9] "Foa - gitlab repository." [Online]. Available: <https://gitlab.com/andersonandrei/foa-a-multi-objective-scheduling-policy-for-serverless>
- [10] D. Milojicic, "The edge-to-cloud continuum," *Computer*, vol. 53, no. 11, pp. 16–25, nov 2020.
- [11] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures," in *Proceedings of the 20th International Middleware Conference*. Davis CA USA: ACM, Dec. 2019, pp. 41–54.
- [12] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," *arXiv:1706.03178 [cs]*, Jun. 2017.
- [13] "Aws lambda." [Online]. Available: <https://aws.amazon.com/lambda/>
- [14] "Cloud functions." [Online]. Available: <https://cloud.google.com/functions>
- [15] "Azure functions." [Online]. Available: <https://azure.microsoft.com/en-us/services/functions/>
- [16] "Ibm cloud functions." [Online]. Available: <https://www.ibm.com/cloud/functions>
- [17] V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller, "Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 273–283. [Online]. Available: <https://doi.org/10.1145/3344341.3368813>
- [18] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Elastic Ephemeral Storage for Serverless Analytics," vol. 44, no. 1, p. 6, 2019.
- [19] N. Mahmoudi and H. Khazaee, "SimFaaS: A Performance Simulator for Serverless Computing Platforms," *arXiv:2102.08904 [cs]*, Feb. 2021.
- [20] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots," *arXiv:2101.09355 [cs]*, Feb. 2021.
- [21] "Kubeless." [Online]. Available: <https://kubeless.io/>
- [22] "Openfaas." [Online]. Available: <https://www.openfaas.com/>
- [23] "Yunikorn." [Online]. Available: <https://yunikorn.apache.org/>
- [24] "Ibm safer scheduler." [Online]. Available: <https://github.com/IBM/Kube-Safe-Scheduler>
- [25] "Ibm multicluster dispatcher." [Online]. Available: <https://github.com/IBM/multi-cluster-app-dispatcher>
- [26] J. Kim and K. Lee, "Practical cloud workloads for serverless faas," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 477.
- [27] P.-F. Dutot, M. Mercier, M. Poquet, and O. Richard, "Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator," in *20th Workshop on Job Scheduling Strategies for Parallel Processing*, Chicago, United States, May 2016.
- [28] D. B. Shmoys and E. Tardos, "An approximation algorithm for the generalized assignment problem," *Math. Program.*, vol. 62, no. 1–3, p. 461–474, Feb. 1993.
- [29] M. Plummer and L. Lovász, *Matching Theory*, ser. ISSN. Elsevier Science, 1986.
- [30] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo, "Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 69–84. [Online]. Available: <https://www.usenix.org/conference/atc22/presentation/li-zijun-help>
- [31] G. Aumala, E. Boza, L. Ortiz-Aviles, G. Totoy, and C. Abad, "Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. Larnaca, Cyprus: IEEE, May 2019, pp. 282–291.
- [32] A. Suresh and A. Gandhi, "Fnsched: An efficient scheduler for serverless functions," 12 2019, pp. 19–24.
- [33] P. Žuk, B. Przybylski, and K. Rządca, "Call scheduling to reduce response time of a faas system," 2022. [Online]. Available: <https://arxiv.org/abs/2207.13168>
- [34] "Docker." [Online]. Available: <https://www.docker.com/>
- [35] "Kubernetes plugins." [Online]. Available: <https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins>



## A. Abstract

We provide the source code implementation of all the scheduling policies proposed in the paper, as well as the source code of the simulation experiments used to evaluate our approach. Then, the reader can (i) generate and reproduce the experiments described in the paper for both evaluated scheduling policies, (ii) reproduce the analysis presented in the paper, (iii) generate and run their own workloads, platforms, and experiments, (iv) modify and exploit the linear program that is the basis of our proposed scheduling policy, FOA, and (v) test of pre-designed scenarios.

## B. Description

### 1) Check-list (artifact meta information):

- **Program:** (i) The scheduling policies' source code (including FOA); (ii) The scripts to install the experimental environment, to generate the inputs, and to run the experiments; (iii) The script to pre-process the simulated outputs; (iv) The script to run the analysis, in addition to a reproducible document (Jupyter Notebook); and (v) The script for testing pre-designed scenarios.
- **Compilation:** GCC
- **Data set:** Workload models, container descriptions, platform models, and experiment descriptions. See more details below.
- **Run-time environment:** Python 3.8.
- **Hardware:** Various x86 or x64 CPUs.
- **Experiment workflow:**
  - **Step 1:** Installation;
  - **Step 2:** Inputs generation;
  - **Step 3:** Execution of simulations;
  - **Step 4:** Preprocessing of simulation outputs;
  - **Step 5:** Testing of results;
  - **Step 6:** Analysis of results.
- **Output:**
  - **Step 1:** None;
  - **Step 2:** Workloads and platforms (in JSON format);
  - **Step 3:** Output of the simulations (in CSV format);
  - **Step 2:** (ii) Preprocessed simulation results (in CSV format)
  - **Step 5:** Messages with status.
  - **Step 6:** (iii) Figures (in PNG and PDF format).
- **Experiment customization:** See below.
- **Publicly available?:** Yes.
- **Vocabulary:** At the implementation level, FOA is called *approxAlgo* and *K8S ImageLocality* is called *kubernetesAlgo*.

2) *Software Availability:* All source material can be downloaded at a GitLab repository<sup>12</sup>, which can be downloaded by the command:

```
$ git clone https://gitlab.com/andersonandrei/
  ↪ foa-a-multi-objective-scheduling-policy-
  ↪ for-serverless
```

3) *Organization:* The Git repository is structured by the following main directories:

- **analysis:** It contains all the source material to process the simulated outputs and to perform the analysis. Also, the figures generated are saved there;

<sup>1</sup><https://gitlab.com/andersonandrei/foa-a-multi-objective-scheduling-policy-for-serverless>

<sup>2</sup>The repository will be also uploaded to Zenodo (<https://zenodo.org/>) after the review.

- **experiments:** It contains all the source material to generate and run the simulations, including FOA's linear program. The directories inside this folder are:
  - **scripts:** it contains all scripts necessary to execute the experiments;
  - **simulations:** it contains the files and directories related to the simulations. The directories are: `exp_out`, `platforms`, `schedulers` and `workloads`.
  - **results:** it contains all simulation results grouped by scheduling policies and analysis parameters.
  - **tests:** it contains the results used as a baseline for testing the pre-designed experiments. The outputs of the simulations are deterministic, so we test them by comparing the outputs of the new simulations with validated sets of experiments: (a) `unit_test`: a small set of 6 experiments; and (b) `paper`: the complete set of experiments performed in this paper.

4) *Hardware dependencies:* Any modern x86 or x64 CPU is appropriate to execute the experiments. It is advised, however, that the system has at least 6GB of RAM since some experiments consume significant amounts of memory.

5) *Software dependencies:* The main software requirements are a Linux distribution (preferably Ubuntu or Debian), the GCC C compiler and Python 3.8. Below is presented a list with the specific software requirements:

- **Git:** Download and installation instructions available at <https://git-scm.com/>;
- **Nix:** Download and installation instructions available at <https://nixos.org/download.html>;
- **Batsim:** Download and Installation instructions are available at <https://batsim.readthedocs.io/>;
  - **Batsim extensions:** PyBatsim, BatExpe.
- **Python 3.8:** Download and installation instructions available at <https://www.python.org/downloads/>.
  - **Python libraries:** csv, json, math, cProfile, hashlib, mip, numpy, itertools, networkx, matplotlib.
- **R:** Download and installation instructions available at <https://www.r-project.org>.
  - **R packages:** dplyr, tidyr, ggplot2, data.table, gridExtra, patchwork, RColorBrewer.
- **Jupyter Lab or Notebook:** Download and installation instructions available at <https://jupyter.org/>.

6) *Datasets:* Except for the container description file, all files here are generated with the scripts available. The container description file is generated after benchmarks that are not in this project's scope. The datasets are *containers description*, *workload models*, *platform descriptions*, and *experiment descriptions*.

## C. Installation and Execution

After the repository is downloaded (see Section B2) and the software dependencies are installed (see Section B5), check below how to run the script that manages the whole workflow described in Section B1. We remark that it automatically

installs Batsim, PyBatsim, and the R and Python packages. But the installation of Python, R and Nix are required. To execute the main script, from the root directory:

```
$ cd experiments
$ ./run_all.sh
```

**Customization.** To modify the path where are the parameters and experiment descriptions, modify their values in the `run_all.sh` file:

```
PARAMETERS_FILE="./parameters_small.yaml" "#" ./
  ↪ parameters.yaml"
EXP_DESCRIPTION_PATH="./simulations/
  ↪ exp_description/"
```

#### D. Experiment workflow

In this section, we explain what each workflow step does and how to execute them separately. The commands below assume that all is executed from the folder `experiments/`. By default, all steps are performed using Nix. To not use Nix, all the software dependencies (see Section B2) should be installed manually. Most of the steps use the same input file, `parameters.yaml`, which specifies paths and parameter values for the experiments. For simplicity, we provide a second input file, `parameters_small.yaml`, for a first installation and execution. It produces a small set of 6 experiments that may not take more than 5 minutes for being executed and tested. Besides, the first installation may take longer.

1) **Installation:** The script `run_all.sh` inside the folder `experiments/` will first install the required environment automatically, but it can also be done by:

```
$ nix-build environment.nix
```

2) **Inputs generation:** The script `generate_experiments.py`, inside the folder `experiments/scripts/`, generates the workloads, platforms, and experiment descriptions. To execute it:

```
$ nix-env environment.nix -A pyshell --run '
  ↪ python3 scripts/
  ↪ generate_and_run_experiments.py
  ↪ parameters.yaml' --pure
```

**Customization:** The parameters that define the set of experiments are inside the `parameters.yaml` file. They are:

```
scheduling_policies: ["kubernetesAlgo", "
  ↪ approxAlgo"]
platform_sizes: [100, 300, 500]
workload_sizes: [200, 600, 1000]
heterogeneity_levels: [3, 5, 7]
random_seeds: [i for i in range(0,150,5)]
```

3) **Experiments execution:** The script `run_experiments.sh`, inside the folder `experiments/scripts/`, executes the experiments. To execute it:

```
$ nix-env environment.nix -A batshell --run '
  ↪ bash run_experiments.sh' --pure
```

**Customization:** To modify the path where are the experiment descriptions, modify its value in the `run_experiments.sh` file:

```
EXP_DESCRIPTION_PATH="./simulations/
  ↪ exp_description/"
```

4) **Preprocessing of simulation outputs:** The preprocessing of the simulation outputs groups all output files by the scheduling policies and the metrics that are analyzed. It also retrieves timestamps from the logs of the simulations and converts them to CSV files. The script is the `pre_process_outputs.py`, located in the folder `experiments/scripts/`. To execute it:

```
nix-env environment.nix -A pyshell --run '
  ↪ python3 scripts/pre_process_outputs.py
  ↪ parameters.yaml' --pure
```

5) **Tests:** The two tests available are performed through comparisons with expected validated results: (a) `unit_test`: it was designed for the small scenario that is produced by the `parameters_small.yaml` input file. Its baseline results are located at `tests/unit_test/`; (b) `paper`: it was designed for the whole paper experiments, produced by the `parameters.yaml` input file. Its baseline results are located at `tests/paper/`. They can be executed through the following commands:

```
$ nix-env environment.nix -A pyshell --run '
  ↪ python3 scripts/run_tests.py
  ↪ parameters_small.yaml' --pure
```

**Customization:** To change the scenario that will be tested, modify the parameters above. The `unit_test` is identified by the ID 0 and the `paper` by the ID 1.

```
current_results_path: "../results/"
unit_test_results_path: "../tests/unit_test/"
paper_results_test_path: "../tests/paper/"
scheduling_policies_to_test: ["approxAlgo", "
  ↪ kubernetesAlgo"]
metrics_to_test: ["out_download_data_info", "
  ↪ out_jobs", "out_schedule", "
  ↪ out_valid_solutions"]
test_id: 0
```

6) **Analysis of Results:** We used R to process and produce the figures presented in the paper. The script is called `analysis.r`. In addition, there is a reproducible document (a Jupyter Notebook), `analysis.ipynb`, to do the same, but in an interactive way, allowing easy modifications for further investigation and discussions. Both files are in the same repository `analysis/`, and both should be executed from there. To run the `analysis.r` script:

```
$ nix-shell environment.nix -A rshell --run '
  ↪ Rscript analysis.r' --pure
```

To run the reproducible document:

```
$ jupyter-lab analysis.ipynb
```