HeROcache: Storage-Aware Scheduling in Heterogeneous Serverless Edge - The Case of IDS

Vincent Lannurien, Camélia Slimani, Laurent d'Orazio, Olivier Barais, Stéphane Paquelet, Jalil Boukhobza

▶ **To cite this version:**

Vincent Lannurien, Camélia Slimani, Laurent d'Orazio, Olivier Barais, Stéphane Paquelet, et al.. HeROcache: Storage-Aware Scheduling in Heterogeneous Serverless Edge - The Case of IDS. CCGrid 2024 - 24th IEEE/ACM international Symposium on Cluster, Cloud and Internet Computing, May 2024, Philadelphia, United States. pp.1-11. hal-04571484

HAL Id: hal-04571484
https://hal.science/hal-04571484v1

Submitted on 7 May 2024

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.

Distributed under a Creative Commons Attribution 4.0 International License

HeROcache: Storage-Aware Scheduling in Heterogeneous Serverless Edge – The Case of IDS

Vincent Lannurien*‡, Camélia Slimani‡, Laurent d'Orazio*†, Olivier Barais*†,
Stéphane Paquelet*, Jalil Boukhobza*‡
*b<>com Institute of Research and Technology, †Univ. Rennes, Inria, CNRS, IRISA.
‡ENSTA Bretagne, Lab-STICC, CNRS, UMR 6285, F-29200 Brest. France
Email: vincent.lannurien@ensta-bretagne.org, stephane.paquelet@b-com.com,
{laurent.dorazio, olivier.barais}@irisa.fr, {camelia.slimani, jalil.boukhobza}@ensta-bretagne.fr

***Abstract*—Intrusion Detection Systems (IDS) are time-sensitive applications that aim to classify potentially malicious network traffic. IDSs are part of a class of applications that rely on short-lived functions that can be run reactively and, as such, could be deployed on edge resources, to offload processing from energy-constrained battery-backed devices. The serverless service model could fit the needs of such applications, given that the platform allows adequate levels of Quality of Service (QoS) for a variety of users, since the criticality of IDS applications depends on several parameters. Deploying serverless functions on unreserved edge resources requires to pay particular attention to (1) initialization delays that could be significant on low resources platforms, (2) inter-function communication between edge nodes, and (3) heterogeneous devices. In this paper, we propose both a storage-aware allocation and scheduling policy that seek to minimize task placement costs for service providers on edge devices while optimizing QoS for IDS users. To do so, we propose a caching and consolidation strategy that minimizes cold starts and inter-function communication delays while satisfying QoS by leveraging heterogeneous edge resources. We evaluated our platform in a simulation environment using characterization data from real-world IDS tasks and execution platforms and compared it with a vanilla Knative orchestrator and a storage-agnostic policy. Our strategy achieves 18% fewer QoS penalties while consolidating applications across 80% fewer edge nodes.**

***Index Terms*—serverless, orchestration, scheduling, edge, cloud, IDS, cache, consolidation, heterogeneous computing**

I. Introduction

IDS, a time-sensitive and critical application: A wide range of embedded systems that operate in static and controlled (*e.g.* sensors in a factory) or dynamic and uncontrolled environments (*e.g.* moving drone swarms) can be temporarily or constantly exposed to critical attacks through network links. As these attacks might jeopardize their execution and seriously damage the related infrastructures, considering them is a critical issue. To mitigate these threats, Intrusion Detection Systems (IDS) are used to analyze network traffic and detect patterns of potentially malicious activities. Machine Learning (ML) models are particularly relevant for a timely classification of the traffic, but are computationally intensive. As a consequence, running them directly on the embedded platform is not a safe solution, as it can affect their lifespan if operating on a battery [8], interfere with other critical tasks, or even be downright impossible to run due to resource shortage.

This work was supported by the Institute of Research and Technology b<>com, dedicated to digital technologies, funded by the French government through the ANR Investment referenced ANR-A0-AIRT-07.

IDS on the edge: A solution to offload these resources-hungry algorithms from deployed embedded systems while keeping the system reactive to attacks is to run IDS in the cloud, and in particular on edge devices [9]. IDS must satisfy variable Quality of Service (QoS) requirements and might be needed only during critical periods, identified beforehand. As a consequence, running IDS on reserved edge devices could be inefficient from a cost perspective. In fact, different types of attacks might have different impacts on the underlying infrastructure. In addition, the risk of attack could change in time and place (according to application domain). We argue that deploying IDS on unreserved low-energy resources on the edge could provide the benefit of a cost-effective solution for running such applications, while keeping the latency lower than when relying on the cloud.

Serverless computing for IDS on the edge: One of the main cloud computing paradigms that makes it possible to run event-driven applications on unreserved resources with fine resource allocation granularity is serverless computing [10]. Deploying serverless computing on the edge for IDS, and more generally for time-sensitive and critical applications, is cost-effective as it opens up optimization opportunities for service providers: dynamic scaling of resources following load peaks in interactive applications, as well as fine and measured allocation granularity for limited edge resources.

Challenges of serverless on the edge for time-sensitive and critical applications: To deploy time-sensitive applications composed of short-lived functions in heterogeneous serverless edge computing, three challenges should be addressed: (1) reduce initialization delays, (2) avoid high communication delays, and (3) leverage heterogeneous resources to satisfy variable QoS. **Initialization delays.** IDS functions are short-lived, and serverless computing relying on unreserved resources implies a higher rate of function initializations, each requiring pulling the function image from a dedicated image storage node for deployment on the edge nodes [11]. Edge devices expose low-capacity, low-performance storage devices behind network links limited in reliability and speed, hence this issue needs to be considered closely to satisfy users' QoS. **Communication delays.** In a

Table I
STATE OF THE ART WORK ON DATA-AWARE AUTOSCALING PLATFORMS

	Function chains	QoS-aware	Hardware heterogeneity	Programming constraint	Energy consumption	Function cache	Function communications
Cypress [1]	✓	✓	✗	✓	✓	✗	✓
FaDO [2]	✗	✗	✗	✓	✗	✗	✓
FaasFlow [3]	✓	✗	✗	✗	✗	✗	✗
FIRST [4]	✗	✗	✗	✓	✓	✗	✗
HeROfake [5]	✗	✓	✓	✓	✓	✗	✗
Netherite [6]	✓	✗	✗	✓	✗	✗	✓
Palette [7]	✓	✗	✗	✗	✗	✓	✓
Target solution	✓	✓	✓	✓	✓	✓	✓

distributed infrastructure such as serverless edge, the functions of the same application can be deployed on several nodes far from each other, implying the use of the network when these functions need to communicate intermediate results. This causes delays that can lead to QoS violations [12].
Heterogeneous resources. The serverless platform cannot consider all placements equal because they will yield various levels of performance. However, the affinity of a function to a specific execution platform cannot alone guide scheduling decisions, because functions can belong to different chains depending on the requested application.

Problem statement: The problem we address is how to account for **initialization and communication delays** when deploying **chains of short-lived serverless functions** on **edge cloud**, leveraging **heterogeneous hardware** to optimize time sensitive applications that require **variable QoS**, while limiting the number of edge nodes used.

State-of-the-Art: Previous studies have explored the need for orchestration platforms that support scheduling function chains on unreserved resources. Table I summarizes to what extent these solutions are not applicable in our case study, and Section VII gives further details. These contributions generally target cloud deployments where the issue is to fit as many tasks as possible in an always-on homogeneous infrastructure of nodes, so as to maximize resource efficiency. The scope of our study is to show that with adequate allocation and scheduling policies, we can fit well-defined applications on a limited number of heterogeneous edge nodes and reduce the overall energy consumption of the cluster through consolidation.

Contribution: HeROcache, a QoS-aware Heterogeneous Resources Orchestration Platform for Serverless Edge Computing based on caching and consolidation: In this paper, we present a solution that addresses the three challenges mentioned above. HeROcache: (1) leverages a caching mechanism on the edge nodes that reduces **initialization delays** without saturating their storage capacity; (2) consolidates tasks on an application basis to limit the number of slow inter-node **communication delays**; (3) manages to respect QoS requirements for critical tasks by using metadata collected from the applications and the heterogeneous platforms used for deployment. These data include performance and energy metrics that guide the orchestrator in making informed decisions when scheduling tasks on **heterogeneous resources**.

Results: We evaluated HeROcache in the context of a real-word IDS application, characterized on various execution platforms. This evaluation was carried out with an ad hoc simulator. We also implemented the behavior of a vanilla Knative [13] orchestrator. HeROcache manages to outperform Knative, keeping QoS violations under 28% while consolidating tasks on 80% less edge nodes in the infrastructure. Powering off these nodes would result in a drastic reduction in static energy consumption.

The paper is organized as follows: Section II gives some background knowledge; Section III explains the overall project; Section IV details our offline metadata collection approach; Section V describes our online orchestration strategy; Section VI discusses evaluation results; Section VII discusses state-of-the-art work; Section VIII concludes the paper.

II. BACKGROUND AND MOTIVATION

A. Serverless challenges

Serverless is a trending service model for the cloud [10]: by shifting the resource allocation responsibility from customers to service providers, it alleviates an important part of the complexity from application developers and opens new opportunities of optimization and cost control for the infrastructure manager. In a serverless architecture, developers design their applications as a composition of stateless functions. Stateless means that the outcome of the computation depends exclusively on the inputs [6]. These functions take a payload and an invocation context as input, and produce a result that is stored in a persistent network-accessible storage tier.

When an event triggers their execution, functions are deployed on nodes in the infrastructure, in execution environments called **replicas**. As functions are stateless, requests can be mapped to any available replica. Scaling a serverless application consists in growing or shrinking the pool of replicas for the functions following the load peaks. Kubernetes-based serverless platforms such as Knative [13] or OpenWhisk [14] proposed a threshold-based model for rightsizing the pool of replicas. For any function, an **autoscaler** can deploy multiple *replicas* to absorb the load. Each replica is allocated on an execution platform (*e.g.* one CPU core, one GPU, etc.) and has a request queue of fixed length for incoming requests. The number of replicas for a given function at any moment determines its concurrency level. A **scheduler** places user requests in queue on function replicas. When a replica has no more requests, it is deallocated. When a function is requested while no replica exists, it goes through a **cold start**.

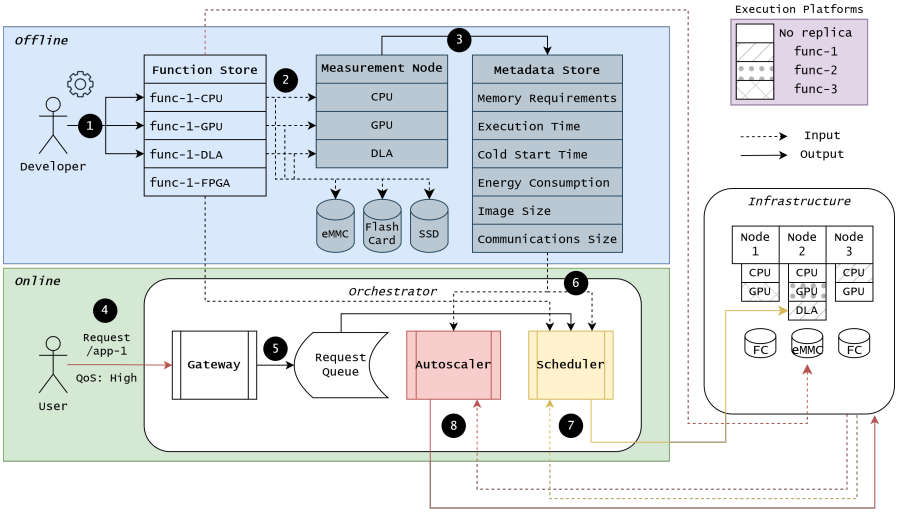


Figure 1. Serverless IDS platform, system overview

This cold start presents a risk of increased latency, as the provider has to allocate hardware resources and instantiate the application before responding to the request. The more complex the application, the higher the risk of large delays [15]. Providers usually pre-allocate some resources to avoid cold starts, which comes with a cost in resources provisioning. Commercial actors such as AWS, Google and Microsoft all reuse function instances to some extent, keeping them running during a timeout period in order to circumvent latency costs incurred by cold starts [16].

A recent study has shown that 50% of serverless applications deployed at Microsoft Azure Durable Functions [1] consist of 3 or fewer functions, with 65% of the applications exhibiting a simple DAG of functions arranged as linear chains [17]. Our IDS application consists of different chains that are two functions long, as described in Section IV-B. Workload characterization work showed that 25% of the functions deployed at Microsoft Azure Functions [2] execute in 100 ms or less [18]. The functions that make up our IDS application run for hundredths to tenths of a second, which makes them particularly prone to critical slowdowns in the context of dynamically allocated resources.

B. Function cache

Function replicas are initialized from **function images** (*e.g.* a Docker image). These are stored in an image registry. Such registries can be remotely accessible through the Internet. However, numerous previous studies [1]–[4] only consider best-case scenarios in which function images are already available on edge nodes. This does not reflect the reality where function images are stored in registries on dedicated nodes and pulled by edge nodes where and when functions are deployed.

[1]https://learn.microsoft.com/en-US/azure/azure-functions/durable/durable-functions-overview

[2]https://azure.microsoft.com/en-us/products/functions/

In fact, pulling images on the edge nodes can account for more than 80% of function response time [11] since the cold start latency dominates the function's total response time. This is not acceptable when the platform has to meet stringent QoS requirements, as is the case for critical tasks such as IDS.

C. Inter-function communications

As it is necessary to support dynamic scaling of the functions, each invocation of a serverless function is self-contained and does not carry information or context from previous invocations. This allows replicas to queue user requests and handle them sequentially without the need to go through a cold start between requests. This introduces a constraint on the serverless platform: if an application is composed of several functions that form a processing pipeline, the output of each function must be stored in persistent storage to be fed as input to the next function in the chain [19].

State-of-the-art work showed that serverless functions that communicate through remote storage can suffer up to 11x slowdown compared to functions using direct communications [12]. The functions of our IDS application need to communicate intermediate results at each stage of the application's DAG. When functions are deployed on different edge nodes, inter-function communications will have to be achieved through the use of remote storage. This introduces slowdowns that can deteriorate QoS.

III. IDS Applications on Serverless Edge

Orchestrating serverless applications while achieving SLA requires carefully modeling application characteristics and taking these into account when allocating resources and scheduling user requests on the serverless platform. Figure 1 gives an overview of the overall lifecycle of a request on our serverless platform. It is divided in two phases; an **offline phase** that consists in characterizing the applications deployed by the users on edge platforms, and an **online phase** where the requests to these applications are scheduled on the platform.

Offline phase. In our platform, the lifecycle of the application starts during an offline phase, where the developer provides the code for their functions for different hardware architectures (GPU, CPU, DLA, etc.) ❶. This code is stored by the service provider in a function registry. The functions are then deployed on a measurement node ❷ where they are run to generate metadata relative to the execution of the functions on heterogeneous edge nodes. Memory requirements, execution time, cold start time, energy consumption, function size, and communication size for each function are written to a metadata store ❸. Running the offline phase is required once for a given function on a given platform, as described in Section IV.

Online phase. Requests are sent to the IDS applications with a payload of TCP traffic (serialized packets) to analyze ❹, and an associated desired QoS level for request response time. The request is appended to a request queue ❺ at the orchestrator level. When the scheduler pops the request from the queue, the metadata store is queried to retrieve the appropriate function metadata ❻.

The scheduler will then try to find an available replica of the first function in the application to handle the request ❼. If such a replica does not yet exist, the autoscaler will be asked to initialize a new instance of the function ❽. During the lifecycle of the application, the autoscaler periodically checks the average load of each function to adjust the number of replicas deployed on the platform, depending on the concurrency threshold set by the service provider.

When the application completes, it returns a classification vector to the user that gives the probabilities that the traffic is malicious, exhibiting patterns of a potential attack.

IV. Offline Phase: IDS Characterization

A preliminary stage of platform and workload characterization is necessary to achieve adequate resource allocation and task placement for the execution of IDS models. To this end, we benchmarked several IDS models in terms of performance and energy on heterogeneous edge platforms that are representative of edge devices [20]. This section describes our methodology and results.

A. Execution platform benchmarks

We used platforms that are representative of what one can find in the edge [8], [20]: **(1) Raspberry Pi 4B** equipped with a quad core ARM Cortex-A72, 4 GB LPDDR4 main memory and a 16GB SD Card. It runs on Linux Raspbian 5.4. **(2) Nvidia Jetson Xavier AGX** composed of three processing elements: an 8 core NVIDIA ARM Carmel CPU, an NVIDIA Volta GPU with 512 CUDA cores, and a Deep Learning Accelerator (DLA), which is a fixed-function hardware accelerator designed for Convolutional Neural Networks (CNN). It is assumed to be more energy efficient than the GPU. The NVIDIA Xavier AGX is equipped with 16 GB LPDDR4 and a 32 GB eMMC 5.1 Flash Storage. It runs on Linux Tegra 4.9.10. The 15 Watts Desktop power mode was used. **(3) PYNQ-Z2 Development Board**, a board based on the Xilinx Zynq XC7Z020 System on Chip. It is equipped with the Artix-7 FPGA and 512 MB DDR3 memory and a 16GB SD card.

Table II
IDS models architectures and sizes

Model	Architecture	Model Size on CPUs (MB)	Model Size on GPU (MB)
NoFS-RF	5 trees of 100 maximum depth	28	15.4
AE-RF	5 trees of 50 maximum depth	-	32.9
ES-RF	10 trees of 10 maximum depth	9.1	5.5
NoFS-DNN1	4 Dense Layers (128x64x32x10)	0.144	
AE-DNN1		0.321	
ES-DNN1		0.053	
NoFS-DNN2	5 Dense Layers (7024x704x288x64x10)	3.33	
AE-DNN2		2.96	
ES-DNN2		2.61	
NoFS-CNN	2 Conv1D (x64) - MaxPool	4.77	
AE-CNN	3 Conv1D (x256) - MaxPool	2.9	
ES-CNN	3 Dense Layers (100x20x10)	2.6	

B. Workload characterization

Our application consists of different preprocessors and classifiers. The preprocessor selects a subset of relevant features of the TCP packets. 3 different preprocessing approaches were used: (1) using all the packet features without any selection (NoFS: No Feature Selection); (2) using a DNN auto-encoder to project features in a smaller latent space (AE: Auto-Encoder); and (3) expertly selecting a subset of the features (ES: Expert Selection). For the classifier part, we used Random Forest (RF), two different Dense Neural Network (DNN) architectures, and a CNN.

Table II shows the IDS models considered in this study and some of their characteristics. These models were trained and characterized on the reference network intrusion dataset UNSW-NB15[3] where each observation represents statistical, content and time features on data traffic during a time window, and tagged as "normal" or "attack". The dataset includes 9 attack categories. The neural network models were exported and optimized using TensorFlow Lite and TensorRT when intended for CPU and GPU/DLA platforms, respectively. Regarding Random Forest, the models were exported using the Emlearn and HummingBird.ml frameworks when targeting CPU and GPU platforms, respectively. hls4ml was used to export neural network models for the FPGA target.

C. Performance measurements results

Each of the IDS models was deployed on the target platforms and inferences were run with a set of 80,000 packets from the UNSW-NB15 dataset to characterize inference latency. The results are shown on Figure 2. Only one model (ES-DNN1) has been characterized on the FPGA platform since the other HLS models could not be accommodated on the target. The conclusion that was drawn from these results is that for neural networks, the Xavier CPU achieves the best performance in the majority of cases, except for NoFS-CNN which takes advantage of the GPU capabilities due to its high number of parameters and GPU efficiency for convolution operations. For Random Forest models, the fastest processing element is the GPU. In terms of cost and availability, the Xavier AGX

[3]https://research.unsw.edu.au/projects/unsw-nb15-dataset

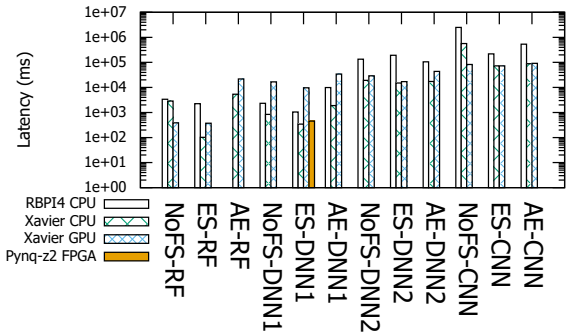


Figure 2. Latency characterization of IDS models

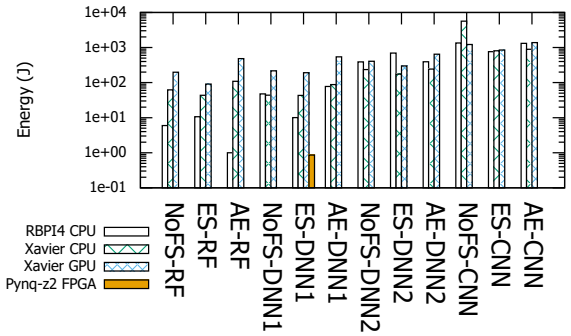


Figure 3. Energy characterization of IDS models

is respectively around 20x and 10x more expensive than the RBPI4 and the Pynq-Z2 platforms, respectively. We size our infrastructure accordingly by providing more RBPI4 platforms than Xavier AGXs to be representative of real deployments.

D. Energy consumption measurements results

We run inferences on IDS models on each processing element and measured the energy consumption of the platform using the N6705A DC Power Analyzer. The results are shown in Figure 3. For the same reasons mentioned above, only ES-DNN1 was characterized on FPGA. We observe that the CPU processing elements show a lower energy consumption than the GPU in the majority of cases. The only case where the GPU shows better results is when the speedup it achieves as compared to CPUs is high. For instance, this occurs for NoFS-CNN where RBPI4 CPU is more than 30x slower than GPU. Even if Pynq-Z2 shows the best energy efficiency with the ES-DNN1 model, since it is more expensive and exhibits a limited design genericity, we assume it less available than RBPI4.

V. Online Phase: HeROcache Orchestration

A. Overview of HeROcache

The HeROcache orchestrator is mainly composed of two modules, the **autoscaler** and the **scheduler** (see Figure 1). The autoscaler is in charge of dynamic resource allocation: it assigns execution platforms to function replicas. The scheduler handles the placement of user requests on the replicas.

Table III
Notation dictionary

Notation	Description
x_a	Allocation of resource for application a
y_a	Invocation of application a
z_i	Placement of task for function i
$f_{N,P}$	A function f scheduled to run on a platform P available on node N
f_a	A function f that belongs to application a
A	Total number of applications to be scheduled on the platform
F_a	Total number of functions that belong to an application a
$RT_{f_{N,P}}$	Time to retrieve function image for f to run on a platform P available on node N
NB_N	Network bandwidth between node N and the infrastructure
SMT_N	Storage medium throughput on node N
SML_N	Storage medium latency on node N
QP	QoS penalty
QD	QoS deviation
WET	Worst execution time
TT	Task total time
CST	Cold start time
ST	Storage time
ET	Execution time
EC	Energy consumption
IS	Image size
HP	Hardware price
TC	Task consolidation
Q	Task queue on a replica
CP	Cache proportion
SIS_a^f, SOS_a^f	Size of resp. input, output state of function f that belongs to application a
$threshold_{f,h}$	Concurrency threshold for a function f on a replica of hardware type h
$scaleCost_a^{f_{i_{N,P}}}$	Cost of creating a new replica for function f_i from application a on a platform P available on node N
$schedCost_a^{f_{i_{N,P}}}$	Cost of scheduling an execution of function f from application a on a platform P available on node N

HeROcache addresses the three above-mentioned challenges through the design of complementary greedy cost-minimization strategies at the autoscaler and scheduler levels. HeROcache minimizes **initialization delays** by considering the latencies of image extraction at the autoscaler level. Prefetching strategies are also implemented for function image caching. **Inter-function communication** costs are considered mainly in the scheduler part, which naturally tends to consolidate functions from the same application. The autoscaler participates indirectly in this consolidation by prefetching the next functions of the DAG of the application on the same edge node. Finally, **heterogeneous platforms** are taken into account as the different execution costs extracted during the offline phase (see Section IV) are considered throughout the entire autoscaling and scheduling process. The next sections describe both the autoscaling and scheduling strategies.

B. Autoscaling cost minimization strategy

We formulate resource allocation as an optimization problem and solve it with a simple greedy algorithm. The objective of the autoscaler is to minimize the cost of the sum of allocations $scaleCost_a$ for y_a invocations of application a (Equation 1) for all applications in A, under the constraint of a finite infrastructure with x_a being the allocation of resources for application a (Equation 2).

$$\forall A,\ \min \sum_{a=0}^{A} y_a \cdot scaleCost_a \tag{1}$$

$$\text{s. t.} \sum_{a=0}^{A} x_a \leq TotalResources \tag{2}$$

The cost of resource allocation for an application a is the sum of the allocation costs for its functions (Equation 3). One replica is allocated to one execution platform.

$$scaleCost_a = \sum_{i=0}^{F_a} scaleCost_a^{f_{i_{N,P}}} \tag{3}$$

Each function replica has an associated allocation cost.

We designed a cost model (Equation 4) for the resource allocation needed to deploy one function of a given application. It is composed of four components, the sum of which we need to minimize:

- the *cache proportion* CP translates the scattering of the functions on the different edge nodes. The higher the score, the more scattered the functions on the nodes. Minimizing this term helps in consolidating the functions;
- the *total time* TT represents the total execution time of the function. It consider the QoS of the application, the heterogeneity of the platform and the deployment cost (whether the image is cached or distant). The higher this cost, the lower the QoS;
- the *energy consumption* EC translates the energy consumption of the function deployment. The higher EC, the higher the cost;
- the *hardware price* HP describes the Total Cost of Ownership (TCO) supported by service providers related to the execution time. This translates the cost of deployment on a given hardware platform. The higher HP the higher the cost of the solution.

The overall objective of the cost model is to deploy a function with the lowest cost possible, that is an increased consolidation, a reduced makespan, a reduced energy consumption, and a reduced cost of ownership. We will detail each part of Equation 4 in the next paragraphs. Each component of the equation is weighted to allow flexible tuning; the values we chose for the deployment of the IDS application are specified in the evaluation part (Section VI).

$$\begin{aligned} \forall N, \forall P \in N, scaleCost_a^{f_{i_{N,P}}} = k_{CP} \cdot CP_{a_N} + \\ k_{TT} \cdot TT_{f_{N,P}} + k_{EC} \cdot EC_{f_{N,P}} + k_{HP} \cdot HP_{f_{N,P}} \end{aligned} \tag{4}$$

Cache Proportion. As seen earlier, enforcing task (a function execution) consolidation among applications should help minimize communication and delays in the function chains. HeROcache pushes toward deploying replicas of a function on nodes where other functions belonging to the same application are already deployed.

To do so, HeROcache keeps track of $CF_a^{f_{i_{N,P}}}$ the number of function images f_i of application a deployed on node N on a given execution platform P (e.g. GPU) available in cache on node-local storage. The proportion of cached functions is computed for each application (Equation 5) and then averaged over all applications running on a given node and inverted to give a high value for non-consolidated functions (as the objective is to minimize this proportion), see Equation 6.

$$\forall a \in A,\ \forall f \in a,\ CF_a^{f_{i_{N,P}}} = \frac{\sum_{i=0}^{F_a} isCached(f_i, N, P)}{F_a} \tag{5}$$

$$\forall N, \forall P \in N,\ CP_{a_N} = \frac{A}{\sum_{i=0}^{F_a} CF_a^{f_{i_{N,P}}}} \tag{6}$$

In addition to the cost minimization, in order to reduce deployment delays, the autoscaler prefetches images for function chains when deploying a new replica on a node. It inspects function chains and sequentially pulls missing function images from the remote registry to node-local storage asynchronously.

Total Time. The second component of scaling cost is the *total time*. Minimizing total time should prevent initialization delays snowballing throughout function chains, thus preventing SLA violations.

Thanks to the metadata collected about each function during the offline phase, the autoscaler is able to predict the time to completion $TT_{f_{N,P}}$ of the first request that will be scheduled onto a new function replica (Equation 7).

$$TT_{f_{N,P}} = RT_{f_{N,P}} + WT_{f_{N,P}} + CST_{f_{N,P}} + ET_{f_{N,P}} \tag{7}$$

- $RT_{f_{N,P}}$ is the duration of the image retrieval time of the function. If the function's image is already cached on the compute node, this duration is zero; otherwise, it depends on the image size IS and is influenced by the network link bandwidth NB, as the image will be read from a remote image registry, and by the node storage medium throughput SMT and latency SML, as the image will be written and stored locally for further use (Equation 8);

$$RT_{f_{N,P}} = \frac{IS_{f_{N,P}}}{\min(NB_N, SMT_N)} + SML_N \tag{8}$$

- $WT_{f_{N,P}}$ is the time that the task will spend waiting in the queue of the platform. At the time of replica creation, this will be equal to zero as we only predict the latency of the first request on the replica;
- $CST_{f_{N,P}}$ is the cold start time required to initialize the function's instance (*i.e.* decompressing the image, preparing the container, initializing the libraries, etc.). It is measured in function of extracted metadata;
- $ET_{f_{N,P}}$ is the duration of the function execution, including the time of communications with its potential predecessors and successors in the DAG. This time considers the platform metadata extraction (Equation 9).

$$ET_{f_{N,P}} = CT_{f_{N,P}} + ST_{f_{N,P}} \tag{9}$$

$CT_{f_{N,P}}$ is the *compute time* of the function – the expected time for the function to complete its execution once fully initialized. The value depends on the performance and availability of the execution platform. $ST_{f_{N,P}}$ is the *storage time* of the function – the expected time for the function to retrieve its

input data and store its output data. The value depends on network link and storage devices performance.

The function storage time $ST_{f_{N,P}}$ depends on the size of its state, *i.e.* its input and output data. Retrieving the input and storing the output of each function in the chain depend on the performance of the network link and the selected storage medium throughput and latency, as shown in Equation 10.

$$ST_{f_{N,P}} = \frac{SIS_a^{f_{i_{N,P}}} + SOS_a^{f_{i_{N,P}}}}{\min(NB_N, SMT_N)} + SML_N \tag{10}$$

Energy Consumption and Hardware Price. Finally, accounting for energy consumption and hardware price should help breaking ties when multiple possible allocations seem to be yielding the same cost (providing the same level of QoS). $EC_{f_{N,P}}$ and $HP_{f_{N,P}}$ correspond respectively to (a) the dynamic energy consumption generated by this allocation obtained thanks to the offline workload and platform characterization phase and (b) the Manufacturer's Suggested Retail Price (MSRP) of the hardware mobilized $HardwarePrice_P$ to deploy the function on said node and platform with regards to the function execution time $ET_{f_{N,P}}$ (Equation 11).

$$HP_{f_{N,P}} = \frac{HardwarePrice_P}{ET_{f_{N,P}}} \tag{11}$$

C. Scheduling cost minimization strategy

As with autoscaling, we formulate an optimization problem to find the optimal scheduling configuration for each user request (as the QoS is to be guaranteed on a user request basis) and solve it with a simple greedy algorithm. The objective of the scheduler is to minimize the cost of z_i tasks placement on R_i replicas of function i for y_a invocations of application a (Equation 12), under the constraint of a finite set of function replicas R_i (Equation 13) previously deployed by the autoscaler. We assume that applications always run to completion and that nodes do not fail, hence there is no cost associated to task migrations or retries.

$$\min \sum_{a=0}^{A} y_a \cdot schedCost_a \tag{12}$$

$$\text{s. t.} \forall a \sum_{i=0}^{F_a} z_i \leq \sum_{i=0}^{F_a} R_i \tag{13}$$

As the platform operates at the granularity of functions, the cost of scheduling an application a is the sum of the scheduling cost for its function chain (Equation 14).

$$schedCost_a = \sum_{i=0}^{A} schedCost_a^{f_i} \tag{14}$$

Each function scheduled in the chain has an associated cost computed for each possible placement on an existing replica. We designed a cost model (Equation 15) for the placement of tasks required to complete a user request for an application.

$$schedCost_{f_{i_{N,P}}} = k_{QP} \cdot QP_{f_{N,P}} + k_{EC} \cdot EC_{f_{N,P}} + k_{TC} \cdot TC_{f_{N,P}} \tag{15}$$

It is composed of three components, the sum of which we need to minimize:

- the *QoS penalty* QP is incurred by the service provider when a user request is not handled in a timely manner. It is a boolean value that determines whether or not a given placement will make the application miss its deadline;
- the *energy consumption* EC translates the energy consumption of the function execution. The higher EC, the higher the cost;
- the *task consolidation* TC describes resource usage for a given placement. The lower TC, the more a replica queue is close to its concurrency threshold, maximizing the hardware utilization.

The overall objective of the cost model is to place tasks in function replicas at the lowest possible cost, that is avoiding penalties suffered by the service provider for missing the application's deadline set by user request, using the less power-hungry execution platforms as possible, and enforcing a high usage ratio for the resources allocated to each function. We describe each part of Equation 4 in the next paragraphs. Each component of the equation is weighted to allow flexible tuning; the values we chose for the deployment of the IDS application are specified in the evaluation part (Section VI).

QoS penalty. The scheduler selects incoming tasks by **earliest deadline first**, leveraging the function metadata to compute a worst-case execution time noted WET (Equation 16). The user request is associated with a QoS level that sets a variable QoS deviation QD applied to the application execution time. This constitutes the application deadline.

$$\forall (N,P),\, WET_f = \max ET_{f_{N,P}} \tag{16}$$

We can predict the application penalty by summing the expected total time for its tasks and comparing it with the application's deadline (sum of functions deadlines), see Equation 17. We re-use Equation 7 to compute the total time for a function execution; however, here, RT and CST will be zero as the replica has already been initialized by the autoscaler during allocation. WT will be equal to the sum of the execution time of tasks of higher priority currently in queue on the replica.

$$QP_a = \sum_{i=0}^{F_a} TT_{f_{i_{N,P}}} > \sum_{i=0}^{F_a} WET_{f_i} \cdot QD_a \tag{17}$$

By factoring storage time in the scheduling cost, we seek to nudge the scheduler into placing tasks as close as possible to the data they operate on. To prevent saturating node-local storage, the platform proceeds to cleanup the intermediate data as soon as the application finishes its execution, *i.e.* when the last function in the chain returns its value.

Energy consumption. $EC_{f_{N,P}}$ corresponds to the dynamic energy consumption generated by this scheduling configuration. It is related to the execution time of the function. The offline measurement results are used for this term.

Task consolidation. We want function replica queues to reach their maximum length: the worst case is to have an empty queue, meaning that hardware resources would have been allocated for nothing. We also want to prevent replica

queues from growing too fast beyond this threshold, otherwise it could generate QoS violations because of long waiting times.

We start by establishing the *platform usage* ratio PU of each replica for the function we try to schedule (Equation 18): the closest the replica queue length Q is to the concurrency threshold ($threshold$ in the equation), the lower the score.

$$PU_{f_{N,P}} = \frac{Q_{N,P}}{threshold_{f,P}} \tag{18}$$

Then, we compute a task consolidation score TC by applying an exponential function to PU (Equation 19). This makes TC the lowest for placements in idle replicas, and this cost increases sharply as the queues are filled up, resulting in the scheduler prioritizing placements on empty replicas and disregarding replicas where the request queue is saturated.

$$TC_{f_{N,P}} = exp(PU_{f_{N,P}}) \tag{19}$$

VI. Evaluation

This section presents our evaluation methodology and results obtained in an IDS deployment scenario on edge devices. The evaluation is done in two phases: we compare HeROcache with several baselines, then we evaluate the impact of each of its components (autoscaler and scheduler) taken apart.

A. Experimental setup

Offline characterization metadata. To evaluate our contribution, we ran measurements for three IDS applications (see Section IV-B). These applications consist of different preprocessing and inference functions that have been implemented on heterogeneous hardware (see Section IV-A). These metadata served as input for a simulator [4] we built using SimPy [21].

Online scheduling. We generated synthetic scenarios by modeling user requests as a Poisson process, following a uniform distribution across application invocations as devised in [22]. By tweaking the λ parameter of the Poisson process, we can generate various traces with different rates of Requests per Second (RPS). We considered a scenario with 10 edge nodes communicating through 4G (LTE) connectivity. The bandwidth for 4G LTE depends on various factors ranging from antenna coverage, to communications service provider's QoS, to receiver quality. We chose to use broad values of 100 Mbps (12.5 MB/s). TCP packets to be analyzed are 1.5 KB size, and are sent in batches of 100 units to the IDS applications. This results in a rate of 83 RPS in our scenario, for 10 minutes of user requests.

Weights for the autoscaling decisions (Equation 4) have been set to $k_{CP} = \frac{3}{8}$, $k_{TT} = \frac{3}{8}$, $k_{EC} = \frac{1}{8}$ and $k_{HP} = \frac{1}{8}$. Weights for the scheduling decisions (Equation 15) have been set to $k_{QP} = \frac{2}{3}$, $k_{EC} = \frac{0.5}{6}$ and $k_{TC} = \frac{1.5}{6}$. We use values inspired from [5] so as to be comparable.

In our experiments, we make it possible to evaluate the autoscaler and the scheduler separately to better understand their behavior. We evaluated different combinations to show which part of each policy is relevant to address the various challenges in our problem. We implemented three autoscalers in our simulator:

- HeROcache (HRC) – Our autoscaling policy;
- HeROfake (HRO) [5] – Enforces a policy similar to HRC, but is oblivious of storage costs;
- Knative (KN) [23]– We modeled the Knative autoscaler behavior to the best of our knowledge. It deploys function replicas on the most available node.

On top of these autoscalers, we used four schedulers:

- HeROcache (HRC) – Our scheduling policy;
- HeROfake (HRO) [5] – Enforces a policy similar to HRC, but is oblivious of storage costs;
- Knative (KN) [13] – Knative considers execution platforms as homogeneous and does not enforce QoS. Replicas are sorted by in-flight requests count; the replica with the shortest queue is selected;
- Bin-Packing First Fit (BPFF) [24] – Tasks are consolidated on the minimum number of nodes and execution platforms. Nodes are sorted by available memory; the first function replica on a node with available memory will be selected for the user request. BPFF is likely to be the scheduling policy for AWS Lambda;
- Random Placement (RP) – Tasks are scheduled on a randomly selected replica.

The naming of each scenario consists of two parts divided by a dash symbol. The first part corresponds to the autoscaling policy; the second part corresponds to the scheduling policy.

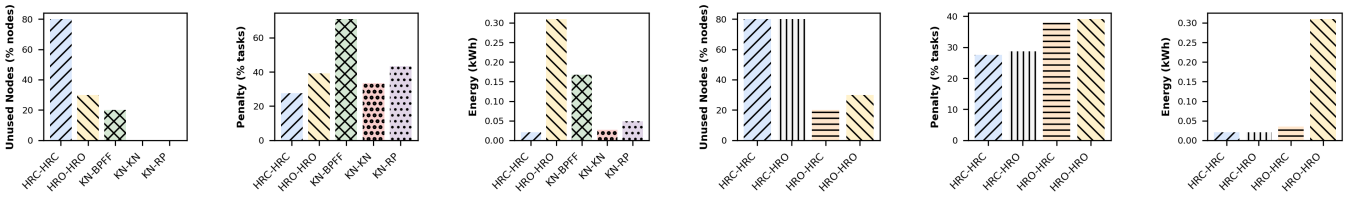
We designed a two-step performance evaluation:
(1) **Comparison against baselines**: we compare full-featured HeROcache (HRC-HRC) to: (1) full-featured Knative (KN-KN), (2) full-featured HeROfake (HRO-HRO), (3) Knative autoscaler with BPFF scheduler (KN-BPFF), (4) Knative autoscaler with RP scheduler (KN-RP).
(2) **Impact of HeROcache components on the overall performance**: we discuss the individual impact of the autoscaler and the scheduler in different strategies: (1) HeROcache autoscaler with HeROfake scheduler (HRC-HRO), and (2) HeROfake autoscaler with HeROcache scheduler (HRO-HRC), comparing them to full-featured HeROcache and HeROfake.

We evaluate HeROcache on the basis of three metrics: (1) the number of unused nodes in the infrastructure, which measures the consolidation level reached; (2) QoS penalties, which expresses the capability for our strategy to meet user requirements; (3) energy consumption, which is a salient challenge in resource-constrained edge computing.

B. Experimental results

1) Comparison against baselines: **Tasks consolidation**. Figure 4a shows that our combination of autoscaler and scheduler achieves the best task consolidation, utilizing only 20% of the edge infrastructure for the execution of the scenario. Knative behaves as expected, spreading the load across the entire infrastructure. Note that BPFF under Knative produces slightly different results: as task queues are maximized, the autoscaler does not need to allocate as many replicas. In this

[4] https://github.com/b-com/HeROsim



(a) Consolidation (b) QoS (c) Energy (d) Consolidation (e) QoS (f) Energy

Figure 4. Evaluation – Comparison against baselines (a-c) and impact of individual components (d-f)

scenario, if the unused edge nodes were powered off instead of sitting idle, our strategy would allow the service provider to save almost 100 Wh (that is 80% of the static energy and more than 83% of the total energy) by turning off 80% of the infrastructure, while still guaranteeing the application response time for 72% of user requests.

Quality of Service. Figure 4b illustrates how relevant taking resources heterogeneity into account is. Indeed, our policy manages to keep QoS violations at 27.5% while leaving 80% of the infrastructure unused. Knative violates just over 30% of the user requests QoS while spreading the load over all the available edge nodes, which is counterintuitive. This is explained by the dependencies between tasks that Knative does not take into account. As a consequence, tasks communicate over slow network storage. While tasks in Knative may spend less time in queue, they still exhibit higher latency than in HeROcache. When using the BPFF policy, violations go up to almost 70%: in this situation, replica queues are too long for tasks to complete within their deadline. For comparison's sake, Knative using the RP scheduler keeps QoS violations around 50%. HeROfake generates 39% QoS violations.

Our policy keeps the proportion of cold starts below 0.011% of user requests, whereas Knative suffers from 4 times more cold starts. In HeROcache, node-local image cache is hit in 33% of function initializations, reducing initialization delays by 17.6%. With HeROcache, 30% of the tasks manage to communicate through node-local storage, speeding up application execution by reducing communications latency by 88.4%.

Energy consumption. Figure 4c shows that HeROcache manages to cut dynamic energy consumption by a third: with a makespan of 1505 seconds for the scenario, the infrastructure consumes 0.0088 kWh, as compared to 0.0266 kWh for 2193 seconds of execution time under Knative. Not only does HeROcache's consolidation strategy allow for power-off policies that could provide important reductions in static energy requirements for running IDS applications on the edge, but by selecting adequate execution platforms, it also reduces the overall consumption of the edge cluster. HeROfake consumes the most energy at 0.31 kWh because of a much longer execution time for the scenario.

2) Impact of each component: **Tasks consolidation**. Figure 4d shows that strategies that are oblivious to storage costs do not manage to consolidate tasks as well as HeROcache: HRO-HRC and HRO-HRO respectively use 80% and 70% of the infrastructure. We explain these results as follows: as dependencies are not satisfied in time, load keeps on growing for the various functions, leading the autoscaler to increase the number of replicas, thus enroling more nodes for the duration of the scenario.

Quality of Service. Figure 4e illustrates the consequence of the previous point: QoS penalties are higher with an autoscaler that does not factor in the delays introduced by function image pulling and function communications. While HRO-HRC is indeed aware of hardware and request heterogeneity, it still finishes at 37.9% of applications missing their deadline.

Energy consumption. Figure 4f displays that while HRO-HRC allocates 70% of the infrastructure, it still manages to keep energy consumption almost as low as HRC-HRC. This is because it chose the nodes that were the least energy-hungry, at the expense of penalties that it could not predict since it is not storage-aware.

Note on complexity: HeROcache employs a greedy optimization technique comparable to HeROfake. In HeROcache, the complexity is bounded by the number of applications A, their size f_a and the size of the infrastructure N (Equation 20): in the worst-case scenario where all the resources are available, the autoscaler scans through the whole infrastructure N to score each node for replica creation.

$$\mathcal{O}_{autoscaling}(A \cdot f_a \cdot N) \tag{20}$$

As the scheduler works with already created replicas R_f of functions, its complexity is lower (Equation 21).

$$\mathcal{O}_{scheduling}(A \cdot f_a \cdot R_f) \tag{21}$$

As our current case study implies a limited subset of IDS functions with a reasonable number of edge nodes, the scalability was not an issue. However, this overhead should be considered for wider deployments of different case studies.

VII. Related Work

Previous work focused on autoscaling platforms for the deployment of short-lived tasks, comprised in applications exhibiting unpredictable load patterns (see Table I).

[2] proposes a data-aware orchestrator, but does not consider the snowballing of delays across function chains. [4] does not support the scheduling of these function chains. All of these contributions consider a homogeneous infrastructure [1]–[4], [7]. This is not representative of our use case, where edge

devices are highly heterogeneous. HeROfake [5] leverages hardware heterogeneity in their orchestration policy, but does not integrate inter-function dependencies nor image caching in their cost model. It was chosen for evaluation purposes to highlight the need to consider such costs. Some of these contributions optimize energy consumption at the autoscaler level [1], [4]. However, they focus on the dynamic part of energy consumption: they do not consider the possible impact of consolidation towards static energy consumption. We argue that service providers should seek task consolidation as a means to power off as many nodes as possible, dramatically lowering the overall infrastructure energy requirements. In [25], the authors investigated the various overheads inflicted by serverless orchestration. This element has not been taken into account in our study, as we are targeting an edge infrastructure limited in size for the deployment of a single application.

VIII. Conclusion

In this work, we presented an allocation and scheduling policy for serverless edge computing. This policy seeks to optimize time-sensitive applications deployment for QoS on energy-constrained devices. By leveraging workload characterization, hardware heterogeneity and local storage devices on the edge nodes, HeROcache enforces applications consolidation and manages to reduce average initialization delays by 17.6% and communication delays by 88.4%. This results in reducing the static energy consumption of the platform by 80% while maintaining under 28% of QoS violations. We plan to generalize the HeROcache approach for case studies including several types of application on larger edge or cloud infrastructures. For such a sake, machine learning strategies or metaheuristics could be used for scaling purposes.

References

[1] V. M. Bhasi, J. R. Gunasekaran, A. Sharma, M. T. Kandemir, and C. Das, "Cypress: Input Size-Sensitive Container Provisioning and Request Scheduling for Serverless Platforms," in *SoCC '22*. San Francisco California: ACM, Nov. 2022, pp. 257–272.

[2] C. P. Smith, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict, "FaDO: FaaS Functions and Data Orchestrator for Multiple Serverless Edge-Cloud Clusters," in *ICFEC 2022*. Messina, Italy: IEEE, May 2022, pp. 17–25.

[3] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service," in *ASPLOS '22*. New York, NY, USA: Association for Computing Machinery, 2022, p. 782–796.

[4] L. Zhang, C. Li, X. Wang, W. Feng, Z. Yu, Q. Chen, J. Leng, M. Guo, P. Yang, and S. Yue, "FIRST: Exploiting the Multi-Dimensional Attributes of Functions for Power-Aware Serverless Computing," in *IPDPS 2023*. St. Petersburg, FL, USA: IEEE, May 2023, pp. 864–874.

[5] V. Lannurien, L. D'Orazio, O. Barais, E. Bernard, O. Weppe, L. Beaulieu, A. Kacete, S. Paquelet, and J. Boukhobza, "HeROfake: Heterogeneous Resources Orchestration in a Serverless Cloud – An Application to Deepfake Detection," in *CCGrid '23*, 2023, pp. 154–165.

[6] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient Execution of Serverless Workflows," *Proc. VLDB Endow.*, vol. 15, no. 8, p. 1591–1604, apr 2022.

[7] M. Abdi, S. Ginzburg, X. C. Lin, J. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette Load Balancing: Locality Hints for Serverless Functions," in *EuroSys '23*. Rome Italy: ACM, May 2023, pp. 365–380.

[8] C. Slimani, L. Morge-Rollet, L. Lemarchand, D. Espes, F. Le Roy, and J. Boukhobza, "Characterizing Intrusion Detection Systems On Heterogeneous Embedded Platforms," in *DSD '23*, Durres, Albania, Sep. 2023.

[9] M. Eskandari, Z. H. Janjua, M. Vecchio, and F. Antonelli, "Passban IDS: An Intelligent Anomaly-Based Intrusion Detection System for IoT Edge Devices," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6882–6897, 2020.

[10] V. Lannurien, L. D'Orazio, O. Barais, and J. Boukhobza, *Serverless Cloud Computing: State of the Art and Challenges*. Cham: Springer International Publishing, 2023, pp. 275–316.

[11] B. Yan, H. Gao, H. Wu, W. Zhang, L. Hua, and T. Huang, "Hermes: Efficient Cache Management for Container-based Serverless Computing," in *Internetware '20*. Singapore Singapore: ACM, Nov. 2020, pp. 136–145.

[12] M. Wawrzoniak, I. Müller, R. Fraga Barcelos Paulus Bruno, and G. Alonso, "Boxer: Data Analytics on Network-enabled Serverless Platforms," in *CIDR 2021*. www.cidrdb.org, Jan. 2021.

[13] C. N. C. Foundation. (2022) Knative. [Online]. Available: https://knative.dev/

[14] Apache. (2022) OpenWhisk. [Online]. Available: https://openwhisk.apache.org/

[15] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile Cold Starts for Scalable Serverless," in *HotCloud 19*. Renton, WA: USENIX Association, Jul. 2019, p. 6.

[16] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold Start in Serverless Computing: Current Trends and Mitigation Strategies," in *COINS 2020*. Barcelona, Spain: IEEE, Aug. 2020, pp. 1–7.

[17] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs," in *OSDI 22*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 303–320.

[18] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *USENIX ATC'20*. USA: USENIX Association, 2020, p. 14.

[19] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure," in *SIGMOD 2020*. Portland OR USA: ACM, Jun. 2020, pp. 115–130.

[20] L. Kljucaric, A. Johnson, and A. D. George, "Architectural Analysis of Deep Learning on Edge Accelerators," in *HPEC 2020*, 2020, pp. 1–7.

[21] T. SimPy. (2022) SimPy. [Online]. Available: https://simpy.readthedocs.io/

[22] S. Li, E. Baştuğ, and M. Di Renzo, "On the Modelling and Analysis of Edge-Serverless Computing," in *MeditCom 2022*, 2022, pp. 250–254.

[23] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments," in *ACSOS 2020*. Washington, DC, USA: IEEE, Aug. 2020, pp. 1–10.

[24] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *USENIX ATC 18*. Boston, MA: USENIX Association, Jul. 2018, pp. 133–146.

[25] A. Fuerst, A. Rehman, and P. Sharma, "Ilúvatar: A Fast Control Plane for Serverless Computing," in *HPDC '23*. New York, NY, USA: Association for Computing Machinery, 2023, p. 267–280.

Appendix

A. Artifact description

We developed a Python simulator [5] that models a generic serverless platform. The software design given in Figure 5 follows the reference architecture of state-of-the-art orchestrators such as Google Knative [13] or Apache OpenWhisk [14]. This section describes the general organization of the artifact; more details are given in the `README.md` file at the root of the software repository.

[5]We used the SimPy [21] library (MIT licensed) for discrete-event simulation.

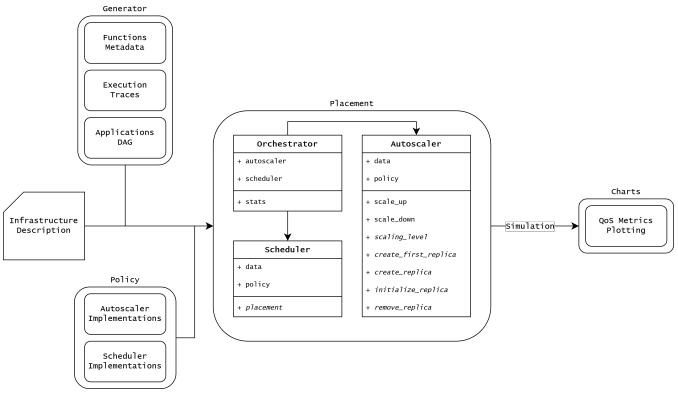


Figure 5. High-level view of the simulator's architecture

1) Simulation tool: The source code for the simulator and the input data used for the experiments conducted in our paper are available in the following repositories:

- Persistent identifier: https://hal.science/hal-04468894;
- GitHub repository: https://github.com/b-com/HeROsim.

2) Simulator usage: HeROsim replays an allocation and placement scenario under different orchestration policies. A simulator run requires the following inputs:

1) a **workload description** – found under `data/ids`; details on the **characteristics of the functions** that will be invoked during the scenario, **i.e.** their execution time, cold start time, memory requirements, energy consumption, etc.;
2) an **infrastructure description** – found under `data/ids/infrastructure.json`; the listing of the different **nodes available**, their different execution platforms (*i.e.* hardware resources), storage devices, network bandwidth, etc.;
3) an **execution trace** – found under `data/ids/traces/workload-83-600.json`; the arrival times for all **user requests**, associated with the requested application and desired Quality of Service (QoS) level.

The user chooses their desired orchestration policy for the run and executes the main program. The simulator will:

- initialize the infrastructure as described: the scenario starts with all the nodes idle, waiting for new requests;
- follow the arrival times of events from the execution trace, and pass the user requests to the orchestrator;
- let the scheduler try to place these requests on function replicas;
- let the autoscaler allocate and deallocate hardware resources that will execute user requests.

The simulation advances when functions are invoked: it is a called a **task execution**. The simulator knows how long a function's response time is thanks to the metadata measured beforehand. These metadata concern the specific hardware and workloads the user is interested in scheduling. Details on the methodology we used to characterize various platforms and workloads can be found in our paper.

During the simulation, logs are written under the `log` directory. When all the user requests have been processed, the simulation stops and returns results and charts summarizing the simulation run, respectively in the `result` and `chart` directories.

B. Experiments replication

1) Prerequisites: The simulator has been tested using Ubuntu 22.04 under WSL2 [6], but should work in any GNU/Linux environment, provided it ships with Python 3.12 or any tool that allows installing Python 3.12. Note that it has not been tested under native Windows.

Also note that the process will write approximately 4 GB of logs to the disk (two times 2 GB), and 2 GB of results files.

Please follow the instructions found in the `README.md` file at the root of the project's Git repository. In particular, the *Usage* section of the file will guide you through the steps necessary for artifacts installation and evaluation.

2) Instructions: Please refer to the detailed `README.md` file at the root of the repository for complete replication instructions.

Using consumer-grade hardware (*i.e.* a Dell laptop with an Intel i5-1145G7 CPU and 16 GB RAM), the simulation scenario finishes in under an hour, including charts generation.

Overall, counting the environment setup steps, you should not have to dedicate more than two hours for the replication of the results.

C. Results

The main results presented in the paper that should be reproduced with this artifact are six charts presented in Figure 4. Once the scenario has been successfully executed, these charts should be available under the `chart` directory.

The evaluation in the paper is done in two parts: (1) evaluation of our policy against baselines, and (2) evaluation of the impact of individual components. Running the scenario script will produce two subdirectories, named after the date of execution, under the root `chart` directory. In these two directories, you will find a `figures` directory that contains the following files:

- `2-unused-nodes.png` (Figure 4 (a) and (d));
- `3-penalty-proportions.png` (Figure 4 (b) and (e));
- `6-energy-consumption.png` (Figure 4 (c) and (f)).

D. License

This software is released under the Apache License, Version 2.0. Feel free to modify, distribute, and use the software in accordance with the terms of the license. Contributions to the project are also welcome.

E. Acknowledgments

The authors want to give special thanks to the reviewers, for their hard work on both the paper and the artifacts evaluation.

[6] https://learn.microsoft.com/en-us/windows/wsl/install