Function Offloading and Data Migration for Stateful Serverless Edge Computing

Matteo Nardelli
matteo.nardelli@bancaditalia.it
Bank of Italy
Italy

Gabriele Russo Russo
russo.russo@ing.uniroma2.it
Tor Vergata University of Rome
Italy

ABSTRACT

Serverless computing and, in particular, Function-as-a-Service (FaaS) have emerged as valuable paradigms to deploy applications without the burden of managing the computing infrastructure. While initially limited to the execution of stateless functions in the cloud, serverless computing is steadily evolving. The paradigm has been increasingly adopted at the edge of the network to support latency-sensitive services. Moreover, it is not limited to stateless applications, with functions often recurring to external data stores to exchange partial computation outcomes or to persist their internal state. To the best of our knowledge, several policies to schedule function instances to distributed hosts have been proposed, but they do not explicitly model the data dependency of functions and its impact on performance.

In this paper, we study the allocation of functions and associated key-value state in geographically distributed environments. Our contribution is twofold. First, we design a heuristic for function offloading that satisfies performance requirements. Then, we formulate the state migration problem via Integer Linear Programming, taking into account the heterogeneity of data, its access patterns by functions, and the network resources. Extensive simulations demonstrate that our policies allow FaaS providers to effectively support stateful functions and also lead to improved response times.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **General and reference** → General conference proceedings; • **Computing methodologies** → *Distributed computing methodologies.*

KEYWORDS

serverless, scheduling, data migration, edge computing, cloud computing

ACM Reference Format:
Matteo Nardelli and Gabriele Russo Russo. 2024. Function Offloading and Data Migration for Stateful Serverless Edge Computing. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3629526.3649293

ICPE '24, May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0444-4/24/05…$15.00
https://doi.org/10.1145/3629526.3649293

1 INTRODUCTION

Serverless computing enables the deployment of applications without the burden of managing the computing infrastructure (e.g., [5]). In the last few years, we are witnessing the increasing adoption of serverless for organizations using cloud services; for example, in 2023, Datadog declared that up to 70% of its customers use at least one serverless function[1]. The diffusion of serverless has been also boosted by the Function-as-a-Service (FaaS) offering by cloud providers, where customers can execute code (or functions) in response to events, thus drastically simplifying the infrastructure operations typically associated with microservices applications.

Recently, two major trends have involved serverless computing. First, besides traditional cloud environments, serverless has been increasingly adopted at the edge of the network as well (e.g., [20, 22]). Second, its adoption is not limited to stateless functions any more, with functions often recurring to external data stores to exchange partial computation outcomes or to persist their internal state (e.g., [5, 19, 31]). Function state usually consists of key-value pairs (e.g., [13, 26, 30, 31]). While in cloud environments it is reasonable to rely on centralized storage services to save state information (e.g., object storage, in-memory stores), in edge-cloud environments state can be distributed across edge and cloud nodes to increase data locality. Therefore, traditional cloud serverless platforms, such as OpenWhisk and OpenFaaS, do not well fit the features of the emerging environment (e.g., [8]). First, they do not consider latency and bandwidth between edge-cloud resources, which can be particularly relevant for functions with stringent latency requirements. Second, by not explicitly considering data dependencies of functions, they do not often optimize for data locality or data movement. Most of recent research efforts on stateful serverless focus on running functions in logically centralized clouds, thus neglecting the impact of heterogeneous resources on performance (e.g., [9, 26, 27]). As functions usually recur to external data stores to persist their state, other research efforts propose to make explicit data intents of functions (e.g., [20, 27]), to co-locate functions with their data (e.g., [26, 28]), or to optimize over an explicit data dependency model (e.g., [2, 19]). While the first approach moves the complexity of managing data dependencies to users and developers, the others leverage rather simple data models, which do not consider the heterogeneity of data and access patterns, as well as its geographic distribution. To the best of our knowledge, the efficient execution of stateful serverless edge functions is still an open issue.

[1] https://www.datadoghq.com/state-of-serverless/

In this paper, we study the allocation of functions and associated state in geo-distributed environments. Since the performance uncertainty affecting FaaS platforms is a frequent concern for developers, we envision a setting where developers and serverless providers stipulate a Service Level Agreement (SLA), enabling the autonomous execution and management of stateful serverless functions with possibly stringent performance requirements. The SLA helps define the target function performance, through Service Level Objectives (SLOs), to be met at run-time as well as the penalties paid by the platform provider whether these expectations are disregarded. We focus on policies for such a serverless edge computing environment. Differently from existing works (e.g., [2, 19, 20]), we explicitly model the heterogeneity of computing, storage, and networking devices as well as data dependencies of functions. The core contributions of this paper are as follows. First, we present a conceptual *Decentralized Stateful Serverless Platform* (DS2P) that includes the mechanisms where the proposed policy can be installed. Second, we present a *SLO-aware offloading* strategy for allocating functions to edge and cloud nodes that explicitly considers the time to access their data and aims to meet the SLOs stipulated in SLAs. Then, we also propose a *state-aware data migration* policy that relocates data at run-time, aiming to further improve application performance while limiting penalty costs paid by the platform provider. Ultimately, we extensively evaluate the proposed policies by investigating the impact of different configuration parameters as well as their scalability.

The remainder of the paper is organized as follows. We present the system model, the problem under investigation, and the main assumptions in Sect. 2. In Sect. 3, we overview the proposed solution by presenting DS2P, which offers mechanisms for running the policy we focus on in this paper. The function offloading and data migration policies are then presented in Sect. 4 and evaluated in Sect. 5. In Sect. 6, we discuss related works and conclude in Sect. 7.

2 SYSTEM MODEL AND PROBLEM STATEMENT

We consider serverless applications, where user-defined code can be executed without allocating and managing virtualized servers and resources, or being concerned about other operational aspects. The responsibility for operational aspects is offloaded to the service provider. Serverless applications consist of multiple functions, whose trigger is usually an HTTP request, a cloud event, or a scheduler. Functions are commonly used to implement APIs, asynchronous processing, batch tasks, or operations tasks [5]. Most functions are very short-lived, running for less than 1 minute (also due to execution time constraints imposed by service providers). Functions can be stateful or stateless, whether they need persistent data to answer requests. Hereafter, we focus on stateful functions.

A serverless platform manages serverless applications to make them available to users. Users from different network locations can request function execution to obtain a service (e.g., [3, 18]) or to manipulate data (e.g., [4, 11, 29]). To preserve the state across invocations, serverless functions usually rely on external store to persist data (e.g., [5, 11, 19]). According to [5], data volumes handled by serverless applications follow a bimodal distribution: although 53% of applications handle less than 1 MB, 16% between 1 MB and 10 MB, a second peak of 16% of them handle more than 1 GB.

To run function instances, the serverless platform could leverage geographically distributed edge-cloud computing environment, where edge and cloud nodes provide computing and storage resources. Let F be the set of all functions. Each function $f \in F$ is characterized by a memory demand $m_f \in \mathbb{R}$ and an execution time on a reference architecture, $r_f \in \mathbb{R}$. We model application state as key-value pairs, where different keys can be allocated to different storage nodes (e.g., [1, 14, 26, 31]). Let K be the set of key-value pairs, where each key $k \in K$ is associated with a value, whose size is l_k. We assume that each function $f \in F$ can access a subset of key-value pairs in K (with some keys possibly accessed by multiple functions). As computing environment, we consider a system comprising N distributed cloud and edge nodes. Each node in $i \in N$ has limited computing C_i, memory M_i, and storage S_i capacity. Nodes $i, j \in N$ are interconnected with non-negligible network latency $d_{i,j}$, i.e., the logical network link (i, j) has $d_{i,j} \geq 0$. We denote by $b_{i,j}$ the bandwidth of the logical network link (i, j). Different disjoint subsets of keys $K' \subseteq K$ can be hosted on different nodes, whereas a single key is hosted on a single node. We consider a per-key placement, e.g., through client-side partitioning[2], enabling keys to be possibly distributed independently from one another. Although sophisticated replication strategies can be designed, during our first iteration of this study, we assume no data replication. Each node handling a portion of the state can be used for both reading and writing operations.

Problem Statement. Determining the allocation of functions and data is critical to run latency-sensitive serverless functions. In the computing environment under investigation, users request function execution from different locations. These functions can access different portions of the data store, i.e., subsets of the keys in K, which can be possibly located on far away nodes. Moving data across different network locations introduces latency, which, in turn, increases the overall function completion time. Moving large data volumes might also introduce prohibitively long delays, calling for adaptive relocation of function code and its execution on a node close to the data store. Each function can exhibit its data access pattern, which may also vary depending on the geographical zone where the function is executed. The key issue relates to the hardness of statically (i) distributing keys to data store instances and (ii) deciding where function execution requests should be scheduled (i.e., locally or to a remote computing node via offloading).

We assume that function developers stipulates a Service Level Agreement (SLA) with the serverless platform provider, including Service Level Indicators (SLIs), i.e., metrics, and Service Level Objectives (SLOs), i.e., predicate over SLIs. Data location impacts on the function running time. Therefore, we consider that each function $f \in F$ exposes a SLO on the maximum data access latency, i.e., $T_f^{\max}$, that should be met at run-time.[3] Otherwise, the serverless platform provider incurs in a penalty cost $\rho_f \in \mathbb{R}^+$ per unit of extra time to access data that exceeds $T_f^{\max}$.

[2] https://redis-doc-test.readthedocs.io/en/latest/topics/partitioning/

[3] As it will be evident in the following, our approach can be readily adapted to cope with different requirements (e.g., maximum response time).

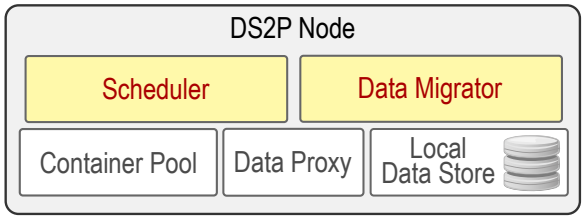

Figure 1: Overview of the architecture of a DS2P node.

Two key problems are considered. First, we need to identify a *function execution and offloading* policy that determines where to run a function instance among all the available (possibly geo-distributed) computing resources. Second, we need to define a *data migration* policy that possibly relocates data at run-time, so as to satisfy the SLA between the function developer and the serverless platform provider, in face of the current workload.

Assumptions. Any client can request the execution of any function in F. To run a function, a client sends an execution request to the closest node $n \in N$ in the computing environment (likely, an edge node in their proximity). Therefore, assuming that invocation requests are always directed to the closest available node, we do not explicitly model client locations in this work. Instead, we only account for the locations of nodes in N, in terms of network latency among them.

The node $n \in N$ receiving the client request is responsible for scheduling the execution of the requested function: the node can possibly decide to offload the execution to another node and, hence, determine how data should be retrieved. To this end, we assume that each node can instantiate any function as it caches a copy of the code for all functions. To support offloading, nodes periodically exchange state information, e.g., using a gossiping protocol (e.g., a similar approach is implemented in [22]).

3 SOLUTION OVERVIEW

We propose Decentralized Stateful Serverless Platform (for short, DS2P), an abstract decentralized FaaS platform designed for edge-cloud computing environments. DS2P manages resources for function execution upon invocation and data storage. As most of FaaS platforms, including OpenWhisk and OpenFaaS, functions run within software containers, which are spawned as needed and initialized with the required code and libraries. A DS2P node can run on edge and on cloud resources; it implements the functionality to receive execution requests, run functions, and interact with other DS2P nodes. In DS2P, there are not privileged entry points for function invocation: every node is able to schedule the execution of incoming requests. Since edge nodes can have limited resource capacity, DS2P allows nodes to *offload* execution requests to other nodes, when needed.

The architecture of a DS2P node is depicted in Fig. 1. Its main components include a Scheduler, a Local Data Store, a Data Proxy, a Data Migrator, and a Container Pool. The *Scheduler* oversees resource allocation for function execution, as it decides whether to execute requests locally, to offload them to other nodes, or to drop them (e.g., during heavy-load periods). To this end, the Scheduler can be equipped with custom policies, such as the one we present in Sect. 4.1. The *Local Data Store* is a key-value store aimed to store function data. It can be implemented using, e.g., Redis, Hazelcast, or Anna. The *Data Proxy* has the responsibility of (i) proxying data access operations, and (ii) characterizing the workload of each function (i.e., which keys they use, type of operation, frequency). The Data Proxy internally exposes a *data distance table*, which reports the estimated latency due to data read/write operations, to the Scheduler and Data Migrator. The *Data Migrator* can periodically relocate key-value pairs by interacting with other DS2P nodes. In the decentralized architecture of DS2P, each Data Migrator handles a subset of keys, i.e., those physically co-located on the DS2P node. Nonetheless, in the following, we assume a logically centralized Migrator that manages all keys stored in the system. Sect. 4.2 proposes a policy to determine when and which key should be more conveniently relocated.

Table 1: Main notation adopted in the paper

Symbol	Description
F	Set of functions
m_f	Memory demand of $f \in F$
r_f	Execution time on a reference architecture of $f \in F$
$T_f^{\max}$	Maximum data access latency expressed by $f \in F$
ρ_f	Unit penalty cost paid when $T_f^{\max}$ is exceeded
K	Set of keys
l_k	Size of value associated to key $k \in K$
$p_{f,k}$	Estimated key access probability $k \in K$ by $f \in F$
N	Set of edge nodes
C_i	Computing capacity of node $i \in N$
M_i	Memory capacity of node $i \in N$
S_i	Storage capacity of node $i \in N$
s_i	Computation speed-up of node $i \in N$
$d_{i,j}$	Network latency between $i, j \in N$
$b_{i,j}$	Network bandwidth between $i, j \in N$

4 FUNCTION AND DATA ALLOCATION POLICIES

In this section, we propose the policies for solving the function execution and offloading problem, as well as the data migration problem. Main notation is reported in Table 1.

4.1 Function Scheduling and Offloading

We propose a *SLO-aware offloading* policy that aims to allocate functions by minimizing their completion time and by fulfilling the SLO on the data access time. Basically, SLO-aware offloading estimates the function completion time both on local node and on other nodes. Since we consider stateful functions, this time is influenced by the access latency of keys used by the function. Then, the policy selects the configuration leading to minimum function completion time, among those satisfying the SLO.

For each function $f \in F$, we maintain an estimate of its key access probability, i.e., $p_{f,k} \in [0, 1]$ with $k \in K$. Such information can be estimated by tracing function data accesses (either in historical traces or at run-time) or, if possible, through static code analysis.

Local Execution. When a node $i \in N$ receives an execution request for function $f \in F$, it checks the execution configuration (i.e., local or offloading) leading to lower response time. First, it estimates the completion time T_i^f assuming that the function f will be executed locally on i:

$$T_i^f = T_{f,i}^{\text{exec}} + T_{f,i}^{\text{data}} \tag{1}$$

this term consists of two contributions, respectively the code execution time of f, $T_{f,i}^{\text{exec}}$,[4] and the average data access time $T_{f,i}^{\text{data}}$, which can be defined as follows:

$$T_{f,i}^{\text{exec}} = \frac{r_f}{s_i} \tag{2}$$

$$T_{f,i}^{\text{data}} = \sum_{k \in K} p_{f,k} T_{f,i,\text{loc}(k)}^{\text{data}(k)} \tag{3}$$

where r_f is the function execution time on a reference architecture, s_i is the computation speed-up of node i, $p_{f,k}$ is the access probability of k by the instance of f, and $\text{loc}(k)$ returns the node hosting key k. $T_{f,i,j}^{\text{data}(k)}$ is the access time of state k by f, when f runs on i and k is stored on $j \in N$, with $j = \text{loc}(k)$; it models the delay to copy the value corresponding to the key from its location j to i:

$$T_{f,i,j}^{\text{data}(k)} = d_{i,j} + \frac{l_k}{b_{i,j}} + d_{j,i} \tag{4}$$

where $d_{i,j}$ is the network delay between i and j, l_k is the size of data associated to k, and $b_{i,j}$ is the bandwidth between nodes i and j.[5]

Local execution of f is *admissible*, if $T_{f,i}^{\text{data}} \leq T_f^{\max}$ and its required memory fits the spare capacity of i, i.e., $m_f \leq \bar{M}_i$.

Offloading. Afterwards, the node i estimates the time to run $f \in F$ when the computation is offloaded to another node. Due to data dependencies, we only consider as candidate the nodes hosting at least a key accessed by f. Having to transfer function inputs and later collect its output, the completion time of f, when offloaded from i to j, with $i, j \in N$, is defined as follows:

$$T_{i,j}^f = T_{i,j}^{in} + T_{j,i}^{out} + T_{f,j}^{\text{exec}} + T_{f,j}^{\text{data}} \tag{5}$$

where $T_{i,j}^{in}$ is the time to move the function input from i to j, $T_{i,j}^{out}$ is the time to collect the computation results from j back to i, $T_{f,j}^{\text{exec}}$ is the time to execute f on j (as in (2)), and $T_{f,j}^{\text{data}}$ is the average data access time of f in j (as in (3)). The first two terms can be formally defined as follows:

$$T_{i,j}^{in} = \frac{l_f^{\text{in}}}{b_{i,j}} + d_{i,j} \tag{6}$$

$$T_{j,i}^{out} = \frac{l_f^{\text{out}}}{b_{j,i}} + d_{j,i} \tag{7}$$

where l_f^{in} and l_f^{out} represent respectively the size of input and output data.

Offloading $f \in F$ to $j \in N$ is *admissible*, if $T_{f,j}^{\text{data}} \leq T_f^{\max}$ and $m_f \leq \bar{M}_j$, where $\bar{M}_j$ is the spare memory capacity of j.

Scheduling. At this point, the node i has all the ingredients to determine where to schedule the function. Among admissible configurations, the policy chooses the one leading to lower expected completion time: the best candidate for offloading is the node j^* with minimum completion time T_{i,j^*}^f, i.e., $j^* = \arg\min_{j \in N'} \left(T_{i,j}^f\right)$, among admissible nodes N'. If local execution is eligible and $T_i^f \leq T_{i,j*}^f$, then f is executed on i; otherwise, f is offloaded to j^*. If no configuration is admissible, the node i returns an error as f cannot be executed while meeting the agreed SLA.

4.2 Data Migration

The Data Migrator can periodically relocate keys among nodes so to reduce the completion time of functions by lowering their data access time. We formulate *state-aware data migration* as an Integer Linear Programming (ILP) problem.

Let $x_i^{(k)} \in \{0, 1\}$ be a binary variable such that $x_i^{(k)} = 1$ if $k \in K$ is allocated on $i \in N$; 0 otherwise. The data migration policy can relocate keys across resources aiming to fulfill the SLA of functions. We consider SLAs with a SLO on the average data access time.

Data Access Time. We define as $T_f^{\text{data}(k)}$ the average data access time of state $k \in K$ experienced by function $f \in F$:

$$T_f^{\text{data}(k)} = \sum_{i \in N} \sum_{j \in N} p_{f,k} T_{f,i,j}^{\text{data}(k)} x_j^{(k)} \tag{8}$$

where $T_{f,i,j}^{\text{data}(k)}$ is the average access time of state $k \in K$ stored in $j \in N$, when f runs on $i \in N$; it is defined in (4).

Migration Time. Data relocation does not come for free, as we need to transfer one or more key-value pairs across the network. In some cases, this activity may be too expensive and we may prefer to completely avoid it. In the data migration policy we model migration time as follows:

$$T_{i,j}^{\text{mig}(k)} = \left(d_{i,j} + \frac{l_k}{b_{i,j}} + d_{j,i}\right) y_{i,j}^{(k)} \tag{9}$$

where $y_{i,j}^{(k)}$ is a binary variable indicating whether $k \in K$ has to migrated from $i \in N$ to $j \in N$.

We can define $y_{i,j}^{(k)}$ through auxiliary constant terms, $\bar{x}_i^{(k)}$, indicating where k was hosted on i before solving the ILP. Conceptually, $y_{i,j}^{(k)}$ represents the logical AND between $\bar{x}_i^{(k)}$ and $x_j^{(k)}$; formally, we define $y_{i,j}^{(k)}$ as:

$$y_{i,j}^{(k)} \leq x_j^{(k)} \qquad \forall i, j \in N \tag{10}$$

$$y_{i,j}^{(k)} \leq \bar{x}_i^{(k)} \qquad \forall i, j \in N \tag{11}$$

$$y_{i,j}^{(k)} \geq \bar{x}_i^{(k)} + x_j^{(k)} - 1 \qquad \forall i, j \in N \tag{12}$$

[4] In this work, we assume that invocation requests do not experience any queueing delay, similarly to what happens in commercial FaaS platforms (e.g., AWS Lambda). Moreover, we assume that performance interference among functions is negligible in the considered computing environment. Note that—with no loss of validity of the proposed approach—more complex models of the execution time can be considered instead.

[5] We consider a simple model of data transfer times. More accurate models accounting, e.g., for possible packet re-transmissions and pipelining issues, can be plugged into our approach with no loss of validity.

SLA Violations. To identify the SLA violations for function f, we resort to *excess* variables $\delta_f \in \mathbb{R}^+$ and $\delta_{f,k} \in \mathbb{R}^+$, and *slack* variables $z_{f,k} \in \mathbb{R}^+$; they indicate how much the data access time is, respectively, above or below the SLO. Formally, we have:

$$T_f^{\text{data}(k)} + z_{f,k} = T_f^{\max} + \delta_{f,k} \qquad \forall k \in K, f \in F \tag{13}$$

$$\delta_f \geq \delta_{f,k} \qquad \forall k \in K, f \in F \tag{14}$$

We assume that, when the SLA is violated, the serverless platform provider pays, to the owner of f, a penalty P_f proportional to δ_f:

$$P_f = \rho_f \delta_f \qquad \forall f \in F \tag{15}$$

where ρ_f is the unit of penalty cost. Other formulations of P_f could be easily considered.

Objective Function. To compute the migration strategy, we need to express the goal function, which formally defines the idea of "best" data placement among all possible solutions. We resort to a simple function $\mathcal{F}$ that minimizes the violations of the functions' SLA and the time to migrate key-value pairs (9). The latter controls how often data should be relocated: ideally, we want to avoid moving *heavy* data too often. We formulate the objective function of state-aware data migration as:

$$\textbf{min} \quad \mathcal{F} + Z \tag{16}$$

with:

$$\mathcal{F} = \sum_{f \in F} w_p P_f + \sum_{k \in K} \sum_{i \in N} \sum_{j \in N} w_m \frac{T_{i,j}^{\text{mig}(k)}}{T_{\max}^{\text{mig}(k)}} \tag{17}$$

$$Z = \sum_{f \in F} \sum_{k \in K} z_{f,k} \tag{18}$$

where $\mathcal{F}$ is the objective function and Z is a technical term to correctly define the slack and excess variables. Moreover, $w_p, w_m \geq 0$, $w_p + w_m = 1$, are weights for the different contributions of the objective function, and $T_{\max}^{\text{mig}(k)}$ is a normalization term defined as $T_{\max}^{\text{mig}(k)} = \max_{i,j \in N \times N} \left(T_{i,j}^{\text{mig}(k)} \right)$.

Constraints and Domain. Storing a key-value pair on a node requires modeling capacity constraints, namely:

$$\sum_{k \in K} l_k x_j^{(k)} \leq \bar{S}_j \qquad \forall j \in N \tag{19}$$

$$\sum_{j \in N} x_j^{(k)} = 1 \qquad \forall k \in K \tag{20}$$

where $\bar{S}_j$ is the spare storage capacity of node j before solving the optimization problem, and (20) ensures that a specific key is allocated on a single node at a time. $\bar{S}_j$ can be readily defined as $\bar{S}_j = S_j - \sum_{k \in K} l_k \bar{x}_j^{(k)}$.

Table 2: Average execution time, memory demand and default arrival rate of the functions used in the experiments.

Function f	r_f	m_f	λ_f
f_1	0.40 s	512 MB	8 req/s
f_2	0.20 s	512 MB	16 req/s
f_3	0.30 s	128 MB	42 req/s
f_4	0.25 s	1024 MB	6 req/s
f_5	0.45 s	256 MB	14 req/s

The ILP formulation also includes domain constraints for each variable used in all the previous equations:

$$\delta_f \in \mathbb{R}^+ \qquad \forall f \in F \tag{21}$$

$$\delta_{f,k} \in \mathbb{R}^+ \qquad \forall k \in K, f \in F \tag{22}$$

$$z_{f,k} \in \mathbb{R}^+ \qquad \forall k \in K, f \in F \tag{23}$$

$$y_{i,j}^{(k)} \in \{0,1\} \qquad \forall k \in K, i, j \in N \tag{24}$$

$$x_i^{(k)} \in \{0,1\} \qquad \forall k \in K, i \in N \tag{25}$$

Equations (8)–(25) formulate the state-aware data migration problem.

5 EVALUATION

We evaluate the proposed policies by simulation. We implement the simulator in Python.[6] In this section, we first describe the experimental setup and then present the results.

5.1 Experimental Setup

Infrastructure. We consider a FaaS system comprising a total of 10 nodes, divided as 5 cloud nodes and 5 edge nodes, which can both execute functions and store key-valued state. The infrastructure comprises an additional cloud node, indicated as *data store*, which only stores state information. Memory capacity is set to 64 GB for cloud nodes and 4 GB for edge nodes. For simplicity, we consider an identical CPU speedup value for all the nodes. We randomly generate network latency values for all the pairs of nodes in the infrastructure. Specifically, edge-to-edge latency is uniformly sampled from $[1, 20]$ ms; latency between edge and cloud nodes (including the data store) is uniformly sampled from $[5, 100]$ ms; cloud-to-cloud latency is uniformly sampled from $[1, 10]$ ms for FaaS nodes, and from $[1, 20]$ ms for cloud-to-data store communication. As regards network bandwidth, we set it to 100 Mbps for edge-to-edge and edge-to-cloud communication, and 1 Gbps for cloud-to-cloud communication.

Functions and invocations. We consider 5 functions, with different resource demands, as indicated in Table 2, and exponentially distributed execution times. Invocation requests to the functions are modeled as independent Poisson arrival processes, where the invocation rate (reported in Table 2) is chosen to have equal expected memory utilization for all functions.

We consider two workload settings, where (i) invocation requests for all the functions are uniformly distributed across edge nodes (**W1**); and, (ii) invocations request for each function are directed

[6]The software is publicly available: https://zenodo.org/doi/10.5281/zenodo.10590302

Table 3: Experimental results in the workload scenario W1.

T_f^{max} (s)	Policies		Avg. Penalty ($/req) $\times 10^{-1}$	SLO Viol. (%)	Migr. Data (MB)	Resp. Time (s)	
	Data Placement	Offloading				P_{50}	P_{95}
0.1	No Migration	Greedy-local	1.7522	61.3518	0.0	0.5833	1.6209
0.1	No Migration	Greedy-data	0.0093	5.3667	0.0	0.3112	1.0684
0.1	No Migration	SLO-aware	0.0087	5.9523	0.0	0.3154	1.0630
0.1	Random	Greedy-local	1.2317	69.8047	384.0	0.5783	1.4500
0.1	Random	Greedy-data	0.7868	56.0006	388.4	0.4748	1.2800
0.1	Random	SLO-aware	0.0835	15.0189	345.9	0.3461	1.1689
0.1	Greedy	Greedy-local	1.7504	65.7991	254.6	0.6242	1.5632
0.1	Greedy	Greedy-data	0.0006	0.2587	204.2	0.3194	1.0763
0.1	Greedy	SLO-aware	0.0005	0.3752	232.6	0.2834	1.0426
0.1	State-aware	Greedy-local	1.4318	53.2727	7.4	0.5386	1.5652
0.1	State-aware	Greedy-data	0.0015	0.7404	7.4	0.3187	1.0765
0.1	**State-aware**	**SLO-aware**	0.0004	0.5055	8.3	0.2900	1.0482
0.2	No Migration	Greedy-local	1.5213	47.2753	0.0	0.6083	1.6715
0.2	No Migration	Greedy-data	0.0000	0.0000	0.0	0.3337	1.0918
0.2	No Migration	SLO-aware	0.0000	0.0000	0.0	0.3188	1.0754
0.2	Random	Greedy-local	0.7323	46.5192	339.1	0.5797	1.4774
0.2	Random	Greedy-data	0.5394	38.8161	390.4	0.5113	1.3215
0.2	Random	SLO-aware	0.0620	11.8700	307.5	0.4011	1.2429
0.2	Greedy	Greedy-local	1.1246	48.9581	206.2	0.6159	1.5650
0.2	Greedy	Greedy-data	0.0000	0.0000	256.3	0.3147	1.0715
0.2	Greedy	SLO-aware	0.0122	1.4703	225.9	0.2928	1.0497
0.2	State-aware	Greedy-local	1.0295	40.3417	7.4	0.5415	1.5803
0.2	State-aware	Greedy-data	0.0000	0.0000	7.9	0.3131	1.0699
0.2	**State-aware**	**SLO-aware**	0.0000	0.0000	7.5	0.2888	1.0473

to a single edge node, with functions assigned to edge nodes in a round-robin fashion (**W2**).

Policies. For comparison against our proposed policies, we consider the following baseline approaches for function offloading:

- **Greedy-local** (**GrL** for short): local execution is always preferred; if the local node runs out of memory, requests are offloaded to the closest cloud node;
- **Greedy-data** (**GrD**): a simplified variant of our state-aware approach, where we always schedule function execution so as to minimize the amount of state data that must be retrieved through the network.

For data migration, we consider the following baselines:

- **No Migration**: keys are never migrated;
- **Random**: randomly assign keys to nodes;
- **Greedy**: move each key to the node that ranks first in remote accesses to that key; to compute the ranking, we use a custom metric defined as the product of the data requests rate coming from the node and the estimated network latency between the node and the current key location.

State. We consider a total of 100 state keys in the system, and assume that each function accesses (i.e., reads or updates) up to 5 keys during execution. The set of keys accessed by each function is randomly sampled, and we consider two different scenarios where the probability of accessing a key follows, respectively, (i) the Zipf and (ii) the uniform distribution. In the latter scenario, the same key is rarely accessed by more than a single function, whereas few popular keys exist in the former. Each state access for every function occurs with a probability value uniformly sampled from $\{0.1, 0.25, 0.5, 0.75, 0.9, 1.0\}$. As regards the size of the data associated with each key, inspired by the observations reported in [21], we consider a bimodal distribution defined as a mixture of two Gamma random variables with shape $k = \mu 10^{-4}$ and scale $\theta = 10^4$, with $\mu \in \{10^4, 10^7\}$ bytes. At the beginning of each experiment, we assume all the key-value pairs to be stored in the data store node in the cloud.

Other parameters. We simulate system execution for one hour, replicating every experiment 10 times. Data migration policy is activated every 120 seconds. We consider different values for the maximum data access time requirement $T_f^{max} \in \{0.1, 0.2\}$s, assuming it to be identical for all the functions. The penalty cost unit associated with SLO violations is set as $\rho_f = 1$ \$/s.

5.2 Offloading and Data Migration Policies

Table 3 reports the simulation results for the first considered workload scenario (W1) and the Zipf distribution for key popularity. In this workload scenario, invocation requests are uniformly distributed to edge nodes, meaning that the keys needed by each function will be necessarily accessed from multiple locations. Considering the stricter SLO requirement of having data accessed within 100 ms (also depicted in Fig. 2), we observe that guaranteeing the

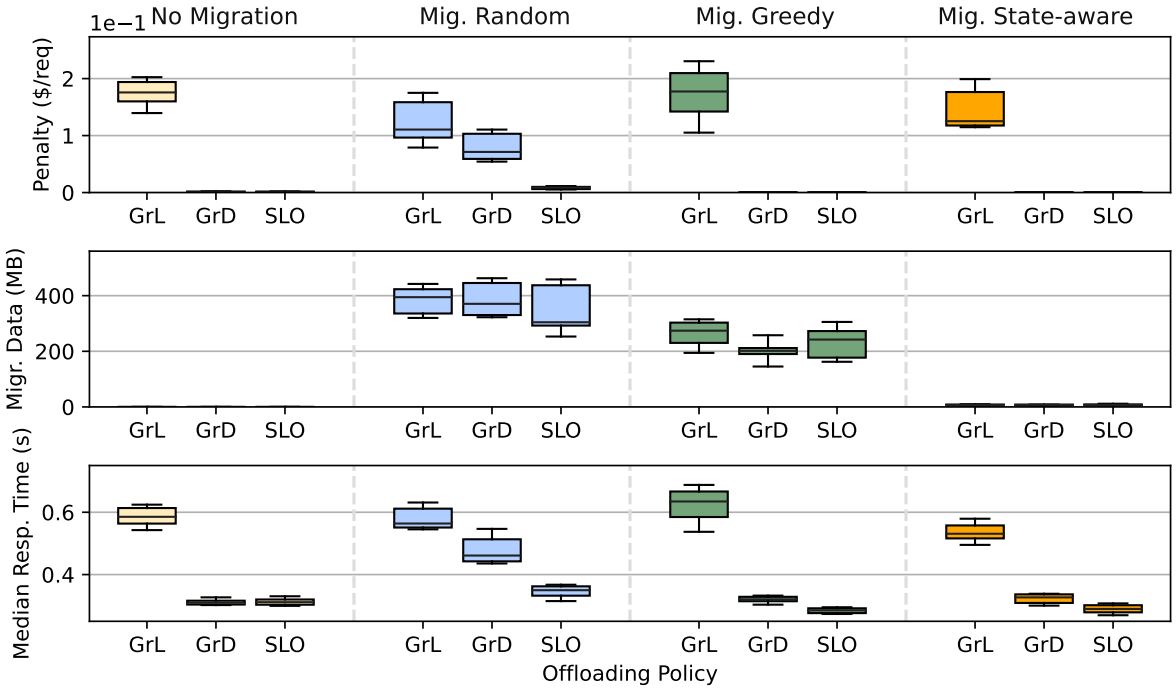

Figure 2: Results in the workload scenario W1, with $T_f^{max} = 0.1$s.

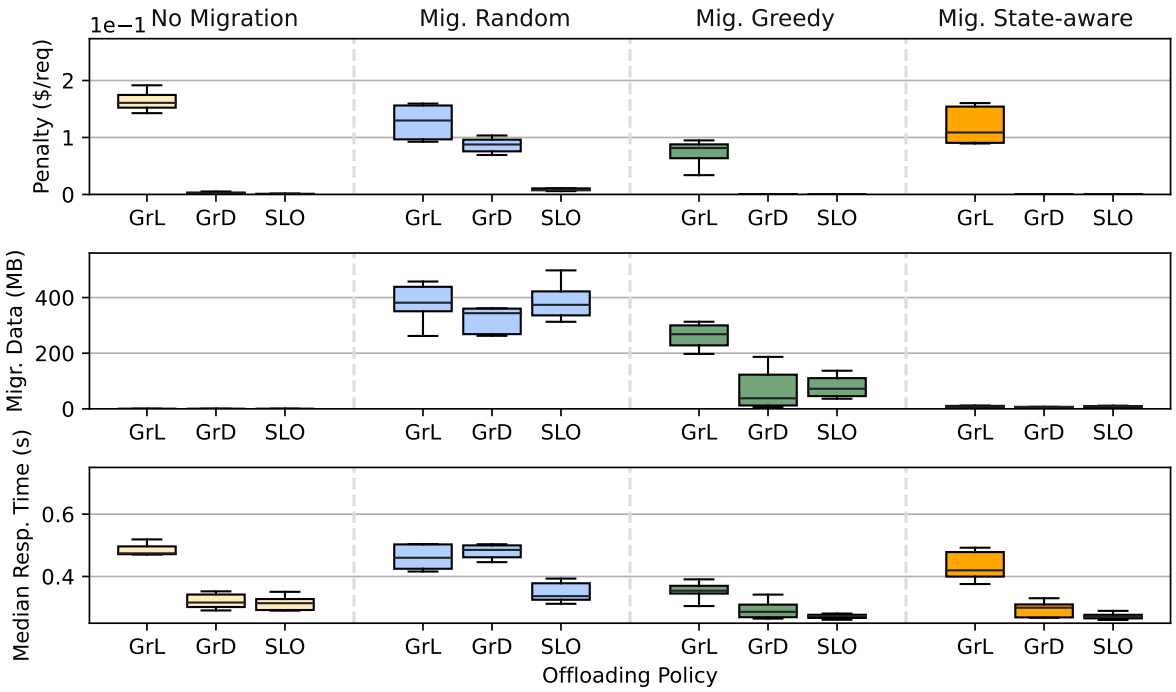

Figure 3: Results in the workload scenario W2, with $T_f^{max} = 0.1$s.

desired service level cannot be simply achieved using any set of policies. Specifically, the experiments demonstrate the importance of devising and adopting state-aware policies.

With no data migration policy enabled, it is particularly evident the impact of state-aware offloading. Indeed, the Greedy-local policy fails to meet the SLO for more than 50% of the served requests in this setting, demonstrating that simply ignoring data locality is not a successful approach in presence of stateful functions. State-aware policies (i.e., Greedy-data and SLO-aware) manage to significantly reduce the number of SLO violations, keeping them below 10% of the served requests.

To further improve the results, we need the ability to relocate data out of the centralized data store. For this purpose, we consider the baseline case of a random migration approach. However, random migrations—as expected—only worsen the situation for all the offloading policies. They also cause a significant amount of data to be migrated throughout the experiment (up to 400 MB in 75% of the cases).

Interestingly, the Greedy migration policy fails to improve the situation compared to the no-migration scenario when coupled with Greedy-local offloading. Conversely, when paired with state-aware offloading policies, SLO violations are less than 1% and the paid violation penalty significantly decreases as well, both for Greedy-data and SLO-aware. Compared to the Greedy migration approach, our State-aware approach achieves even better results. While it also keeps violations below 1%, it reduces the amount of migrated data from about 200 MB to less than 10 MB, avoiding the oscillations of the Greedy data migration policy.

Looking at the overall request response times, we can further observe that the SLO-aware offloading policy performs slightly better than Greedy-data, avoiding unnecessary execution offloading based on expected SLO satisfaction. Therefore, the combination of State-aware data migration and SLO-aware offloading (i.e., the two policies we propose in this paper) leads to the best results in this scenario.

When considering a less strict SLO requirement, i.e., $T_f^{max} = 200$ ms, all the policies experience fewer SLO violations and the impact of data migration decreases. Nevertheless, adopting a proper offloading policy is still fundamental to achieve acceptable performance, with Greedy-local largely failing to meet SLO requirements even in this scenario. State-aware offloading policies completely avoid paying SLO violation penalties in this scenario. The impact of data migration is still visible looking at overall response times. Compared to the baseline scenario with no migrations, the state-aware key placement reduces median response times by about 10%.

Similar results are observed in the alternative workload scenario W2, where invocation requests for each function are directed to a single edge node. As illustrated in Fig. 3, the different workload dynamics do not alter the relative performance of the policies we compare. It is only worth observing that the Greedy-local policy performs slightly better, especially in terms of overall response times. This is explained by the fact that having a limited number of functions to handle at each edge node favors container reuse and, hence, reduces memory contention for local execution. However, this policy still fails to meet SLO expectations in terms of data access time and causes the provider to pay a significant amount of penalties.

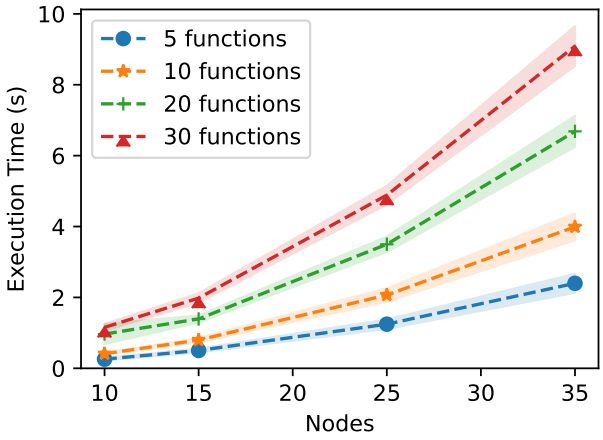

Figure 4: Execution time of the State-aware migration policy.

Due to space limitations, we do not show the results obtained with reduced key contention among functions (i.e., using uniform key popularity rather than Zipf), as they lead to the same considerations reported above.

Overall, our experiments demonstrate that the proposed State-aware data migration and SLO-aware offloading represent the best policies, allowing the provider to minimize the monetary penalties associated with SLO violations, while also leading to the lowest response times.

5.3 Scalability

Throughout the experimental campaign, we kept track of the computational overhead imposed by policy execution. As expected, the State-aware data placement policy, based on ILP resolution, is the most demanding one. Figure 4 shows the execution time of the policy varying the number of functions and computing nodes in the system, with the ILP resolved using IBM CPLEX® (version 12.6.2.0).

It can be observed that the computational overhead is lower than 2 s with 10 computing nodes and grows to less than 10 s with 35 nodes. The observed overhead looks acceptable, especially as the computation of data migrations can be performed asynchronously with respect to function execution. Nonetheless, we plan to explore heuristic resolution strategies in the future to further reduce the execution time of the policy.

6 RELATED WORK

The serverless paradigm requires platform providers to transparently manage infrastructure operations, possibly in face of (stringent) SLAs. We classify the existing research efforts into two categories, according to their main focus. A large set of papers considers functions and their scheduling on computing resources, whereby data strictly depends on functions; data dependencies are modeled at different degrees of granularity, including also implicit modeling. Stressing the importance of data, other works treat data as first-class

citizens, whereby it is assumed to be more convenient to execute or relocate functions whenever needed. In this case, the key topics are caching, replication, and consistency of data. In the following, we analyze the key results, aiming to highlight the core contribution of our work. To the best of our knowledge, the execution of stateful serverless functions in edge-cloud environments is still an open issue.

Function scheduling. Different solutions have been proposed so far to allocate stateful functions. Basically, four main approaches are pursued to model the dependency between functions and their data: make explicit the data intent of functions through APIs or annotations; co-locate functions with their data; optimize data passing among functions; and model data dependencies. Hence, different function scheduling policies are designed.

To make explicit the data dependencies of functions, developers enrich the metadata of functions with references to data (or bucket of data) required during execution (e.g., [20, 25, 27]). Lambdata [27] is the first approach that makes the intent of a function's input and output explicit for improved locality. Although the framework works in a cloud environment, such information is exploited to optimize functions execution, e.g., by running functions that read same data on the same worker. Similarly, to ensure that functions are scheduled close to their data, FaDO [25] provides a unified interface to the various storage services, and defines a special HTTP header to specify the required bucket on the function invocation requests. In this case, functions are allocated on the requested bucket, if possible. Data intent information is especially useful in geographically distributed environments. To determine a trade-off between data and computation movement in edge systems, Rausch et al. [20] propose a greedy multi-criteria policy that selects the best node for function scheduling by leveraging a scoring function with four contributions: (1) proximity to data storage nodes at edge locations; (2) proximity to the cloud-based container registry; (3) availability of specialized hardware; and (4) matching of user-annotated scheduling preferences between edge and cloud locality.

The second class of approaches resorts to co-location, thus considering data dependencies only implicitly. Co-location can be implemented either through replication or caching (e.g., [26]), or by running the function code on the storage node (e.g., [28, 32]). For example, Cloudburst [26] is a popular stateful serverless platform that co-locates compute and data. It employs an architecture with distributed storage and caching on machines hosting functions. In a preliminary work, Tiwary et al. [28] propose using WebAssembly to run serverless functions: exploiting lightweight virtualization, such functions can be more efficiently executed directly on data sources. The idea of pushing WebAssembly functions into storage has been also exploited by Shredder [32], which realizes a single-node multi-tenant cloud store that allows small units of computation to be performed directly within storage nodes. Hetzel et al. [9] suggest using an actor-based model, where each actor has its own locally stored state. Actors only interact by exchanging messages.

Serverless applications are usually composed of multiple functions that exchange intermediate (ephemeral) data. To this end, three general approaches emerged: leverage a shared (remote) storage (e.g., [6, 11, 15, 18]), exploit data locality to improve performance (e.g., [10, 12, 14, 24, 26–28]), and perform direct transfer (e.g., [14]). For example, SyncMesh [7] stores and processes data locally, so as to provide it to other nodes only on-demand. Conversely, Sonic [14] optimizes application performance and cost by selecting among different data-passing methods for each (namely, local storage, direct passing, remote storage). To this end, they model cost and time needed to exchange data and use a Viterbi algorithm, which guarantees to find the true maximum a-posteriori solution. Wukong [3] enhances the locality of DAG-based parallel workloads, by explicitly considering data that a function passes to downstream functions directly connected in the graph. Xu et al. [30] propose a more general approach, where functions can access data generated by upstream function (even if not directly connected). To rule the complexity of model and uncertainty of data volumes and delays, the authors resort to an online learning heuristic that aims to co-locate functions and their data on groups of edge resources. A slightly different perspective is presented in Zion [23], which considers data flowing from storage gateways to storage nodes and enables triggering functions on a data-driven basis (instead of an event-driven basis).

To conclude, a few works explicitly model the relationship between functions and data (e.g., [2, 19]). For example, Puliafito et al. [19] distinguish between stateless functions executed on serverless, which can access a remote cloud storage, and stateful containers, with a locally attached persistence volume. The scheduling problem, formulated as a mixed ILP (MILP), aims to find a trade-off between the cost of transferring input and output data using the network and a data access cost. First, stateful containers are allocated; then, stateless functions are placed so to balance load among nodes. The data model is rather simple, as it assumes that functions exchange a fixed amount of data with a centralized cloud storage system at each invocation. NEPTUNE [2] is a K3s-based platform for running latency-sensitive serverless applications on geo-distributed edge topologies. The authors formulate the application placement as a MILP, considering CPU and GPU requirements and minimizing the overall network delay. Currently, NEPTUNE only models network traffic exchanged between functions, whereas their access to external data stores is only partially modeled: the time to read from and write on a database is modeled as non-controllable stationary disturbance of the response time.

Data placement. The advent of serverless fostered the development of novel approaches to store data. A few works build persistent storage leveraging serverless, being cheaper than cloud computing resources (e.g., [4, 11, 29]). These works do not explicitly consider the presence of functions for computation, but provide convenient storage abstractions.

Other works focus on data and consider serverless functions as volatile, lightweight, and easily relocatable (e.g., [18, 31]). These works usually do not optimize the allocation of resources for functions, but mainly focus on data placement (e.g., [1]), replication (e.g., [16, 17]), and caching (e.g., [26]). Building on Cloudburst [26], Pheromone [31] proposes a data bucket abstraction that can be configured with triggers specifying *when* the target functions should be invoked and *how* their output should be passed to the next functions in a workflow. To improve data locality, Pheromone uses a two-layered scheduler, with a global coordinator that balances load across nodes and drives the execution of function across multiple

buckets, and a local scheduler that triggers functions execution on each node. Pfandzelter et al. [17] propose a data replication middleware that operates in a geographically distributed environments and with heterogeneous resources. Although the middleware can be used to optimistically replicate data on cloud and edge devices running serverless functions [16], currently it only blindly replicates key-groups on every replica node (thus introducing inefficiencies for writing operations). Conversely, the policy we propose helps to identity the locations where it is more convenient to move data.

Contribution. To the best of our knowledge, policies for SLA-aware execution of stateful functions in geo-distributed environments are missing. We consider serverless functions accessing external data stores, dealing with heterogeneous data access patterns and non-uniform data size for different keys. This model agrees with the evidences in [21].

The closest works to ours are [2, 19, 20]. As in [20], we consider an edge-to-cloud computing environment, where computing and storage nodes are interconnected with non-negligible network delay. Nevertheless, we do not require the user to manually indicate the data intent of functions. These access patterns can be estimated (or learned) at run-time. Moreover, we propose a state-aware data migration policy. Differently from [2, 19], we propose a more detailed formulation of the data dependency model, by explicitly considering the data access patterns of functions. We leverage this information to solve two main tasks, namely function offloading and data migration. Both tasks consider that functions expose requirements through a SLO on data access latency: hence, functions and data cannot be located on distant nodes, otherwise the serverless platform provider will pay penalties for SLA violation. To the best of our knowledge, this is the first contribution that considers a SLA-aware allocation of functions and data in edge-cloud environment.

Finally, we acknowledge that our approach is far from being fully-fledged. Indeed, we postpone to future work the integration of more sophisticated techniques for improving replication, fault-tolerance, and caching of data.

7 CONCLUSION

We investigated the problem of executing stateful serverless functions in edge-cloud environments, providing guarantees about the time required to access key-valued state. We proposed a set of policies to control both *function offloading* (i.e., when and where function execution should be offloaded to a remote node), and *data placement* (i.e., how to relocate state data to better accommodate the current workload). Simulated experiments demonstrate that our approach provides the desired SLO guarantees, while avoiding frequent data migrations and improving overall response times.

For future work, we plan to integrate our approach in an existing FaaS framework, and investigate efficient heuristics for the resolution of the data migration problem.

ACKNOWLEDGMENTS

This work has been partially supported by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di "campioni nazionali" di R&S (M4C2-19) - Next Generation EU (NGEU).

REFERENCES

[1] Koichiro Amemiya and Akihiro Nakao. 2020. Layer-Integrated Edge Distributed Data Store for Real-time and Stateful Services. In *Proc. of IEEE/IFIP NOMS '20*. 1–9.
[2] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. 2022. NEPTUNE: Network- and GPU-Aware Management of Serverless Functions at the Edge. In *Proc. of SEAMS'22*. ACM, 144–155.
[3] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, et al. 2020. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In *Proc. of ACM SoCC '20*. ACM, 1–15.
[4] Marcin Copik, Alexandru Calotoiu, Konstantin Taranov, and Torsten Hoefler. 2023. FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example. arXiv:2203.14859 [cs.DC]
[5] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, et al. 2022. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Trans. Softw. Eng.* 48, 10 (2022), 4152–4166.
[6] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, et al. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proc. of USENIX ATC '19*. USENIX Association, 475–488.
[7] Daniel Habenicht, Kevin Kreutz, Soeren Becker, Jonathan Bader, et al. 2022. SyncMesh: Improving Data Locality for Function-as-a-Service in Meshed Edge Networks. In *Proc. of EdgeSys '22*. ACM, 55–60.
[8] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, et al. 2018. Serverless Computing: One Step Forward, Two Steps Back. arXiv:1812.03651 [cs.DC]
[9] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. 2021. μActor: Stateful Serverless at the Edge. In *Proc. of MobileServerless'21*. ACM, 1–6.
[10] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proc. of ACM ASPLOS'21*. ACM, 152–166.
[11] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, et al. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proc. of OSDI'18*. USENIX Association, 427–444.
[12] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *Proc. of USENIX ATC '21*. USENIX Association, 805–820.
[13] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, et al. 2023. Serverless Computing: What It Is, and What It Is Not? *Commun. ACM* 66, 9 (2023), 80–92.
[14] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, et al. 2021. {SONIC}: Application-aware data passing for chained serverless applications. In *Proc. of USENIX ATC '21*. 285–301.
[15] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proc. of ACM SIGMOD '20*. ACM, 131–141.
[16] Tobias Pfandzelter and David Bermbach. 2023. Enoki: Stateful Distributed FaaS from Edge to Cloud.
[17] Tobias Pfandzelter, Nils Japke, Trever Schirmer, Jonathan Hasenburg, and David Bermbach. 2023. Managing data replication and distribution in the fog with FReD. *Softw. - Pract. Exp.* 53, 10 (2023), 1958–1981.
[18] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proc. of USENIX NSDI '19*. USENIX Association, 193–206.
[19] Carlo Puliafito, Claudio Cicconetti, Marco Conti, Enzo Mingozzi, and Andrea Passarella. 2023. Balancing local vs. remote state allocation for micro-services in the cloud–edge continuum. *Pervasive Mob. Comput.* 93 (2023), 101808.
[20] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. 2021. Optimized container scheduling for data-intensive serverless edge computing. *Future Gener. Comput. Syst.* 114 (2021), 259–271.
[21] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, et al. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proc. of ACM SoCC'21*. ACM, 122–137.
[22] Gabriele Russo Russo, Tiziana Mannucci, Valeria Cardellini, and Francesco Lo Presti. 2023. Serverledge: Decentralized Function-as-a-Service for the Edge-Cloud Continuum. In *Proc. of IEEE PerCom '23*. 131–140.
[23] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. 2017. Data-Driven Serverless Functions for Object Storage. In *Proc. of ACM/IFIP/USENIX Middleware '17*. ACM, 121–133.
[24] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proc. of USENIX ATC '20*. USENIX Association, 419–433.
[25] Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. 2022. FaDO: FaaS Functions and Data Orchestrator for Multiple

Serverless Edge-Cloud Clusters. In *Proc. of IEEE ICFEC '22*. 17–25.
[26] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, et al. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (2020), 2438–2452.
[27] Yang Tang and Junfeng Yang. 2020. Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit. In *Proc. of IEEE CLOUD '20*. 294–303.
[28] Mayank Tiwary, Pritish Mishra, Shashank Jain, and Deepak Puthal. 2020. Data Aware Web-Assembly Function Placement. In *Companion Proc. of WWW '20*. ACM, 4–5.
[29] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, et al. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *Proc. of USENIX FAST '20*. USENIX Association, 267–281.
[30] Zichuan Xu, Lizhen Zhou, Weifa Liang, Qiufen Xia, Wenzheng Xu, Wenhao Ren, Haozhe Ren, and Pan Zhou. 2023. Stateful Serverless Application Placement in MEC with Function and State Dependencies. *IEEE Trans. Comput.* (2023), 1–14.
[31] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *Proc. of USENIX NSDI '23*. USENIX Association, 1489–1504.
[32] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and Its State with Storage Functions. In *Proc. of ACM SoCC'19*. ACM, 1–12.