2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC) | 978-1-6654-6087-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/UCC56403.2022.00023

Mobility-Aware Serverless Function Adaptations Across the Edge-Cloud Continuum

Philipp Raith
Thomas Rausch
Schahram Dustdar
Distributed Systems Group, TU Wien
Vienna, Austria
lastname@dsg.tuwien.ac.at

Fabiana Rossi
Valeria Cardellini
DICII, University of Rome Tor Vergata
Rome, Italy
{f.rossi, cardellini}@ing.uniroma2.it

Rajiv Ranjan
School of Computing, Newcastle University
Newcastle upon Tyne, England
raj.ranjan@newcastle.ac.uk

***Abstract*—Serverless functions have emerged as a useful abstraction to manage the complexity of distributed and heterogeneous edge-cloud infrastructure. Current cloud-centric orchestration services and serverless platforms are not suitable for the edge-cloud continuum due to their mobility-unawareness. In this paper, we present a mobility-aware framework for edge-cloud systems, where the operational mechanisms of placement, scaling, and routing of serverless functions work in tandem. To that end, we formulate the concept of *pressure* that captures complex system behavior in a single metric. The pressure-based framework handles geo-distributed workload and user mobility by reducing overall function latency and increasing data throughput while efficiently using edge resources. Our contributions include a novel framework revolving around pressure and a real-world proof of concept evaluation. Our approach combines the benefits of a centralized control-plane with a decentralized data-plane. The results show the efficacy of the platform to address operational goals and make effective and deterministic tradeoffs between system utilization and application performance. The novel concept of pressure shows great extensibility and builds the base for a plethora of future works.**

***Index Terms*—Edge Cloud Continuum, Serverless Computing, Function as a Service, Edge Intelligence**

I. Introduction

Serverless functions encapsulate business logic into atomic units of deployment that are autonomously managed by platforms to hide computing infrastructure from application developers. The platform's main responsibilities of scaling, placing, and routing requests need to consider the mobility of users in modern edge-cloud scenarios [1]. The serverless paradigm can help manage the complexity of operating applications on the edge-cloud continuum, where the geo-distributed and heterogeneous nature of infrastructure presents unique challenges and opportunities [1]. Applications across the edge-cloud continuum include low latency applications spanning various domains, such as autonomous vehicles [2], or augmented urban reality [3]. These use cases share two common characteristics that are particularly challenging to address: geo-distribution and user mobility [4]. Service migration tackles the problem of user mobility and requires platforms to constantly re-allocate resources [5]. Edge-cloud systems are composed of multiple compute units to enable computation in proximity independent from where the requests come [6]. Thus, users constantly re-connect to the nearest access point, making mobility-aware serverless function adaptations necessary for future applications across the edge-cloud continuum. Existing efforts to extend serverless systems (e.g., Kubernetes) to the edge [7] have only recently begun to address how to adapt cloud-based serverless adaptations [6], [8] and cloud-centric solutions are not feasible [5]. Serverless adaptations include scaling and placement of application instances (i.e., function replicas) and the routing of incoming user requests.

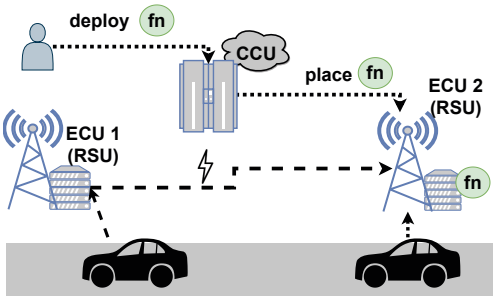

Fig. 1: Scenario

Figure 1 presents a scenario from the domain of autonomous vehicles in which cars are connected to the closest Road Side Unit (RSU). Specifically, the system is composed of different compute units. Namely a central Cloud Compute Unit (CCU) and two Edge Compute Units (ECU). In our case the ECUs are RSUs and offer limited hosting capabilities. Cars automatically connect to the closest one and offload tasks to the ECU. Tasks include assisted driving, traffic management, and infotainment [9]. In our case, tasks are processed by stateless functions that are deployed by a developer in the CCU and placed from there across the edge-cloud continuum. The issue is that functions need to be adapted to migrate along the users' trajectory. Specifically, ECU 1 does not host the needed function and needs to re-route the task to ECU 2, where it can be processed. Naturally, if functions do not migrate, the invocation suffers from long network latency trips. Thus, serverless platforms require scaling, placement and routing strategies to migrate

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871403.

978-1-6654-6087-3/22/$31.00 ©2022 IEEE
DOI 10.1109/UCC56403.2022.00023

functions according to the users' mobility. This entails the consideration of non-negligible network delays, varying workload conditions, resource usage, and user mobility. To that end, we propose a mobility-aware framework based on the novel concept of *pressure* - a metric that measures the force between compute units based on locality and demand. Our framework implements a network-aware offloading strategy and pressure-based scaling and placement that work in tandem. The pressure is highly extensible, and we show this using two pressure formulations that deterministically yield expected results. Additionally, the pressure-based approach significantly reduces communication between compute units, which otherwise might result in a bottleneck on the network layer [10], [11]. Evaluation on a testbed shows that our performance-oriented pressure reduces the 90*th* percentile Round-Trip Time (RTT) by up to 77%, and network latency can be reduced by up to 92% compared to the default Kubernetes Autoscaler. The resource-efficient pressure reduces the average number of running containers by 83% in comparison to the performance-oriented variant while still reducing the RTT by 60% and the network latency by 74%. Inter-network traffic is reduced by 92% and 67%, respectively.

The main contributions of this paper are as follows.

- We propose a novel joint scale & placement and routing framework for serverless functions across the edge-cloud continuum.
- We implement two different pressure calculations showing deterministic behavior and good results.
- Our open source proof-of-concept implementation[1] works in tandem with Kubernetes.

The rest of the paper is organized as follows. Section II discusses related works and highlights issues around them and differences to our work. Afterward, Section III briefly introduces Function-as-a-Service (FaaS) and our system model. Section IV introduces the *pressure*, and Section V the pressure-based serverless platform, of which we describe our proof-of-concept (PoC) implementation in Section VI. Section VII outlines our methodology for evaluating the proposed system. Section VIII highlights the key performance indicators, displays the results of our experiments on the testbed, and discusses advantages and disadvantages. Section IX concludes our work and gives a perspective for future works.

II. Related Work

The related work is split into three categories. First, we look into works that investigate offloading and service migration solutions. Afterward, we focus on solutions that propose edge-cloud serverless function platforms. Then, we consider works that tackle mobility issues.

A. Mobility-aware offloading and service placement

Labriji et al. [9] propose a proactive VM service migration scheme in the Internet of Vehicles. Tang et al. [12] investigate Quality of Service (QoS) requirements in vehicle fleets. Using virtualization, they reduce resource provisioning costs while ensuring QoS requirement satisfaction. Sun et al. [13] formulate a mix-integer non-linear stochastic optimization problem to optimize resource usage and task placement jointly. Maia et al. [14] tackle the joint issue of request routing and service placement using limited look-ahead control and a genetic algorithm. These approaches and more [15]–[17] propose mobility-aware solutions to routing and placement, but evaluate their approaches solely in simulations. While they show promising results, our work differentiates by solving these issues for serverless functions using an integration into Kubernetes.

B. Serverless adaptations in edge-cloud systems

The authors of [8] use a distributed Reinforcement Learning (RL) approach to manage applications in Kubernetes using decentralized request dispatchers and a centralized service orchestrator. This design is similar to ours but differs from the approach to minimize data communication between edge and cloud. Their approach uses Graph Neural Networks to encode the system's state while our scale and placement components work on the aggregated complex pressure metric. Rossi et al. [6] use an RL agent to scale applications and schedule on a geo-distributed system using a network-aware placement heuristic, but do not propose a solution to request routing. Tamiru et al. [18] extend Kubernetes to manage multiple clusters and re-route traffic across all clusters, but focus neither on low latency applications nor the characteristics of edge computing. Other papers that evaluated orchestration services lack edge-based load balancing strategies [19], [20]. Baresi et al. [21] propose a serverless platform for edge computing for which they also build a prototype based on the open-source FaaS platform OpenWhisk. Their architecture is based on a decentralized approach containing a self-contained serverless platform per region (e.g., per city). In contrast to our work, they do not explicitly tackle placement, scaling, and routing but resort to measuring the idle memory footprint and performance in a static environment (i.e., without scaling). Further, these works have not been evaluated in the context of mobility in edge-cloud systems.

C. Mobility-aware serverless function deployments

Few works have approached the task of implementing and evaluating mobility-aware serverless function solutions. Wang et al. [22] propose an RL approach that explicitly tackles user mobility issues but do not jointly investigate placement and routing. Baresi et al. [23] present NEPTUNE, a network and GPU-aware serverless function adaptation platform which is location-aware and built on top of Kubernetes. They use Mixed Integer programming to perform placement decisions and split the system into separate communities (similar to our compute unit definition). In contrast to our work, communities are not aware of each other. That causes the system to fail if a community is overloaded and has to be reconfigured. Our approach differentiates us by presenting a novel pressure-based framework for serverless functions.

[1] https://github.com/phip123/mobility-aware-faas

III. Background

FaaS platforms offer convenient deployment of serverless functions by requiring only a containerized stateless application to scale, place and route requests autonomously. While there are many issues regarding serverless computing (e.g., cold starts [24]), we specifically focus on these three base function adaptations. We build our platform on Kubernetes, a commonly used orchestration service among open-source serverless frameworks [25] and research [6], [18], [23]. This paradigm requires that applications offer a single stateless function that can be invoked over the network (i.e., HTTP). While serverless platforms abstract away the details from users, default components are not suited for modern edge-cloud scenarios. Especially when facing mobility, cloud-centric approaches (i.e., scaling and request routing) fail to ensure user requirements. The serverless paradigm promises to abstract the underlying infrastructure away from the platform users and therefore deployment should only require minimal deployment information. Therefore, we provide a novel serverless platform to perform dynamic mobility-aware function adaptations. Based on the example in Figure 1, we introduce our notation for the system. As mentioned, we envision edge-cloud systems to consist of one or multiple CCUs with unlimited hosting capabilities but experience non-negligible network delays to users (i.e., cars). ECUs are deployed at the edge, comprised of resource-constrained devices, and in proximity to users. Our approach is dynamic and does not differentiate between these two compute units. Therefore, the platform manages a set of compute units C. Each compute unit $c \in C$ contains multiple nodes $n \in N$ and exactly one gateway. The gateway acts as a request router and has function adaptation capabilities, such as the aggregation of fine-grained monitoring data and the local placement of functions. Platform customers deploy functions with resource requirements for CPU and memory (CPU_f^{req} and MEM_f^{req}, respectively). A function replica running on node n is denoted as f_i, where i is a unique index, and we can look up the node for each f_i. Users $u \in U$ generate requests and invoke the deployed functions. A compute unit c receives requests from user u of the given function f ($R_{u,c,f}$). We also denote requests that are being re-routed from compute unit x to compute unit y with $R_{x,y,f}$. Table I lists all symbols.

IV. Pressure

Pressure is an abstract metric composed of multiple low-level metrics and does not impose requirements regarding its definition. Therefore it is versatile and can be formulated toward specific use cases. As our evaluation shows, the pressure can be composed of well-known low-level metrics (i.e., resource utilization). The combination of them positively influences dynamic function adaptations. In the following, we introduce our vision of pressure based on an example scenario.

A. Example scenario

Figure 2 depicts a system consisting of three compute units: $ECU1$, $ECU2$, and a cloud (CCU). The scenario highlights how the pressure can guide function adaptations and make them mobility-aware. It is based on our introductory example in Section I. At the beginning (t_1), one function replica is available in CCU. Thus, the car is offloading requests to $ECU1$ that are being re-routed to CCU, because no function replicas are running on $ECU1$. The network latency negatively impacts different factors (i.e., performance, bandwidth usage), and *ECU1* should process requests. The platform has to adapt the function deployment and place a new function replica in $ECU1$. The pressure captures this mismatch, resulting in a high value between $ECU1$ and CCU. At step t_2, the system starts a function replica on $ECU1$ and incoming requests can be processed internally (resulting in low pressure). As no requests are being processed in CCU, the replica can be shut down. In the next step, t_3, the car moves away from $ECU1$ and connects to $ECU2$. $ECU2$ does not host the function and therefore has to re-route them to $ECU1$. In contrast to cloud-centric load balancers, a mobility-aware platform has to consider network latency between compute units when re-routing requests. As depicted in the figure, $ECU2$ is closer to $ECU1$ than CCU (e.g., consider replicas running in CCU). In t_3, the pressure between $ECU2$ and $ECU1$ is high again, and the platform has to adapt the function deployment dynamically.

This scenario highlights the following important aspects: (1) the platform efficiently places the function from the cloud along the car's path, (2) unused functions are shut down, and (3) the pressure gives direction and enables a joint scale and place algorithm, allowing us to make mobility-aware decisions.

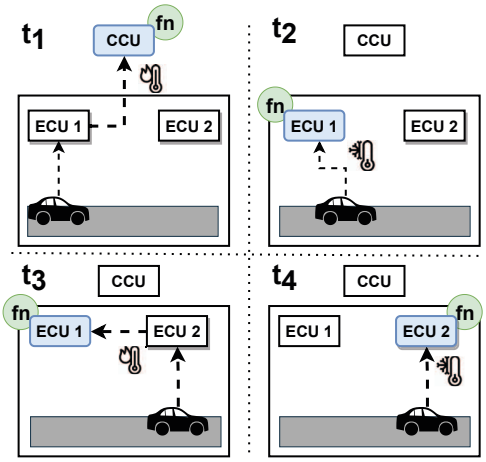

Fig. 2: Scenario

B. Definition

The pressure is calculated for each pair of compute units (i.e., $ECU1$, $ECU2$ and CCU) and for each function f, where requests were re-directed from one compute unit to another. We denote the pressure as $p_{ECU1,ECU2,f}$, which translates to requests being sent from the origin compute unit

(i.e., $ECU1$) to the target compute unit (i.e., $ECU2$). The pressure is directed (i.e., $p_{ECU1,ECU2,f} \neq p_{ECU2,ECU1,f}$). Note that the same compute unit puts pressure on itself (i.e., $p_{ECU1,ECU1,f}$). This is important to adapt compute units independently and ensures requirement satisfaction. Generally, we expect a high pressure between two compute units when the latency between them is high (which has to be defined for each function individually) and a relatively large quantity is re-routed, causing performance degradation for many users. On the other hand, if the network latency is negligible, the pressure should be low (i.e., $p_{ECU1,ECU1}$). Additionally, each compute unit should be able to calculate it based on local compute unit metrics and requests. This minimizes communication costs between compute units and the centralized control plane. Before introducing our pressure-based platform, we give details about the pressure formulations used in our evaluation.

Variable	Description
C	Set of compute units
F	Set of deployed functions
N	Set of nodes
CPU_f^{req}	CPU requirement of function f
MEM_f^{req}	Memory requirement of function f
$CPU_{f_i}^{rel}$	CPU usage of function f_i
f_i	Function replica f with index i
U	Set of users sending tasks (i.e., a car)
$R_{u,c,f}$	Set of requests from user (or compute unit) u received by c
$p_{c_o,c_t,f}$	The pressure from c_o on c_t for function f
$d(n_1, n_2)$	Network latency between nodes n_1 and n_2
t_f^{rtt}	Target round-trip time of function f
$p_{x,y,f}^{REQ}$	Request pressure between compute unit x and y
$p_{x,y,f}^{RES}$	Resource pressure between compute unit x and y
$p_{x,y,f}^{RTT}$	Performance pressure between compute unit x and y
p^{perf}	Performance-oriented pressure definition
p^{eff}	Resource efficient pressure definition
$w_{f_i,c}$	Routing weight of replica f_i for gateway in compute unit c

TABLE I: Symbols

C. Pressure formulations

We introduce two different pressure formulations: p^{eff} and p^{perf} that are composed of three and two low-level components respectively: (1) the request distribution ($p_{x,y,f}^{REQ}$), (2) the performance ($p_{x,y,f}^{RTT}$) and (3) the resource usage (i.e., average CPU usage) of the target compute unit y ($p_{x,y,f}^{RES}$).

$p_{x,y,f}^{REQ}$ captures the request distribution, the relative amount of requests spawning from x and being processed by y. It is formulated as follows, C_y contains all compute units (including y) that have sent function requests to y and $R_{x,y,f}$ is the number of requests sent from compute unit x to y. We divide the result by the highest ratio to normalize values between $[0, 1]$.

$$R_{y,f} = \sum_{c \in C_y} R_{c,y,f}; u_{y,f} = \frac{R_{y,f}}{|C_y|} \tag{1}$$

$$w^{max} = max_{c \in C_y}(\frac{R_{c,y,f}}{u_{y,f}}) \tag{2}$$

$$p_{x,y,f}^{REQ} = (\frac{R_{x,y,f}}{u_{y,f}})/(w^{max}) \tag{3}$$

$p_{x,y,f}^{RTT}$ anticipates applications that require a specific RTT. As network and performance fluctuate during workloads, we formulate a pressure that models this circumstance. It is based on the RTT and a user-specified requirement (t_f^{rtt}). Specifically, we use the 99th percentile of the RTT of function f from x to y ($R_{x,y,f}^{P99}$). The pressure increases when compute unit x sends requests to far away, or overloaded compute units. This allows to fine-tune each function and gives the ability to adapt the importance of performance dynamically. We use a logistic-based function (Eq. 4) to act as a cost function to assess the current state.

$$l(d) = \frac{(L-b)}{1 + e^{-k(d - t_f^{rtt})}} + b \tag{4}$$

$$p_{x,y,f}^{RTT} = \frac{l(R_{x,y,f}^{P99})}{L} \tag{5}$$

We use the following parameter values:

$$L = 250, b = 1, k = 0.2, t_f^{rtt} = 70 \tag{6}$$

Figure 3 shows the resulting pressure function $p_{x,y,f}^{RTT}$ which stays in the range $[0, 1]$. In case no traces exist, we set the pressure to 0. The parameters were chosen to penalize (i.e., calculate a high pressure) for requests with an RTT that violates the threshold and favor (i.e., calculate a low pressure) if the RTT is below the threshold. Future works can include investigating which parameters are suitable for different applications.

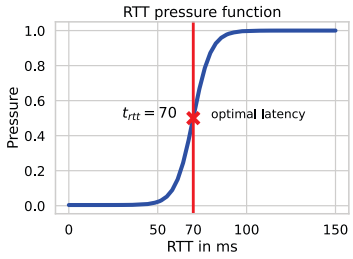

Fig. 3: p^{RTT} - pressure function

$p_{x,y,f}^{RES}$ is based on the ratio between requested CPU usage and current average CPU usage over all function replicas in the target compute unit y ($CPU_{f,y}^{mean}$). This approach is similar to how Kubernetes' Horizontal Pod Autoscaler (HPA) calculates the ratio for scaling applications and is in contrast to the others not influenced by the origin compute unit x. $p_{x,y,f}^{RES}$ can yield higher values than 1. We want to note here that this pressure component can be easily extended to include other resources (e.g., GPU, I/O), but our evaluation application mainly uses CPU.

$$p_{x,y,f}^{RES} = \frac{\text{CPU}_{f,y}^{mean}}{\text{CPU}_f^{req}} \tag{7}$$

We calculate each pressure component and multiply them to create the final pressure value. The pressure formulation can be easily adapted and weighted towards different goals and allows generalization across different infrastructures. To demonstrate the adaptability, we use in our experiments two different formulations: a pressure that contains all components (**eff**icient), and one that omits the $p^{RES}_{x,y,f}$ component to scale and place with a focus on **perf**ormance (see equations 8 and 9, respectively). The presented pressure formulation is our initial evaluation of this concept and can be extended to more sophisticated variants in future work (e.g., weighted sum).

$$p^{\text{eff}}_{x,y,f} = p^{REQ}_{x,y,f} \cdot p^{RTT}_{x,y,f} \cdot p^{RES}_{x,y,f} \tag{8}$$

$$p^{\text{perf}}_{x,y,f} = p^{REQ}_{x,y,f} \cdot p^{RTT}_{x,y,f} \tag{9}$$

V. Pressure-based serverless platform

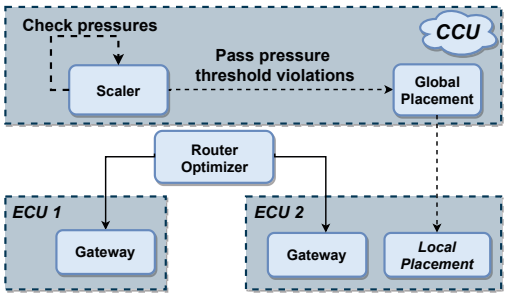

Fig. 4: Framework

Figure 4 shows the components of our pressure-based serverless platform and highlights interactions between them. We employ a joint scale and placement system to dynamically manage the function lifecycle and a network-aware router optimization component. The centralized scale component finds pressure threshold violations. These are passed to the placement component, which consists of a global and local component. The global placement component determines in which compute unit a new function replica should be spawned, and the local one selects the node. The router optimizer frequently updates the gateways in the system.

A. Scaling

The framework uses a reactive threshold-based strategy to scale functions. The component periodically receives the pressure ($p_{c,x,f}$) for each pair of function and compute unit and checks for any threshold violations. In case the pressure is too high (i.e., $p_{c,x,f} > thr_{max}$), it will scale up, and in case the threshold falls below (i.e., $p_{c,x,f} < thr_{min}$) it will trigger a scale down action. The number of replica to scale up or down is currently statically set to one. If the pressure is 0, but replicas are pending to start, we do not take any scale down actions. Additionally, we identify *zero sum actions* to re-use existing resources efficiently. This lets the system identify scaling decisions that would scale down a replica in a compute unit in which we want to schedule a new one. The scaler outputs a list of tuples containing the violating compute unit (c) and the corresponding function (f), which is passed on to the global placement component.

B. Placement

Our work proposes a two-level decentralized placement approach. The global placement component determines the compute unit, and the local placement one selects a specific node. The *global placement component* has to resolve two kinds of events: scale-up and scale-down. In case of a scale-down event, the pressure $p_{c,x,f}$ was lower than thr_{min}, and the current selection strategy chooses the replica with the lowest resource usage (i.e., CPU) in c. In the case of a scale-up event, the pressure $p_{c,x,f}$ was higher than thr_{max}. Therefore, the global placement policy receives a list of pressure violations S and the set of compute units C (see Algorithm 1). Each pressure violation in S consists of the violating compute unit c and the function f. First, it decides whether c can host another replica (line 4). Otherwise, it selects the next best compute unit using latency and pressure (lines 6-20). First, we sort all compute units in ascending order based on the network latency relative to compute unit c (line 8). We assume that the global component is aware of the underlying network topology. In the next step, we check for each possible target compute unit x if it can host another replica of f and check if the pressure threshold (thr_{max}) is violated based on $p_{c,x,f}$. If not, we select x to spawn a new replica. If no compute unit was found, we select the nearest one that can fit another replica (line 17). The pressure allows us to avoid overloading compute units by selecting the nearest one. Afterward, the target compute units and functions (stored in M^*) are passed on to the local placement component. The *local placement component* receives the functions and compute units to spawn new replicas and prepares for each pair one replica. This component, situated directly in the compute unit (i.e., on the gateway), can make fine-grained decisions to place the replica on the best-fitting node. We do not propose a sophisticated local placement heuristic in this work and resort to the default Kubernetes scheduler. Our framework intends to focus on the global level leaving fine-grained decisions open for custom solutions. Note that the local placement component has much more information available, and fine-grained telemetry data does not have to be propagated to the global components (as depicted in Figure 4). This also increases the system's scalability as the global component does not have to compute the pressure but instead selects the compute unit to place the next replica. The routing optimizer includes the newly spawned replica and routes traffic to it.

C. Router Optimizer

The gateways route, based on a weighted-round-robin technique, and the router optimizer periodically updates the weights based on network latency and resource usage. It calculates the weights for all replicas running in the compute unit and for external compute units hosting the function. For

Algorithm 1: Placement

Function `globalPlacementPolicy(`S, C`)`**:**
  Input: S, C: S contains the pressure violations, and C is the set of compute units
  Output: M^*: set containing the target cluster for each function
  $M^* \leftarrow \emptyset$
  foreach *(c, f)* $\in S$ **do**
    if *canHost(c, f)* **then**
      $M^* \leftarrow M^* \cup \{(min_c, f)\}$
    else
      $min_c \leftarrow$ *null*, $C^* \leftarrow C \setminus c$
      $C^* \leftarrow$ sortByLatencyAscending(c, C^*)
      foreach $x \in C^*$ **do**
        if *canHost*(x, f) **then**
          if $p_{c,x,f} < thr_{max}$ **then**
            $min_c \leftarrow x$
            break
        end
      end
      if $min_c =$ *null* **then**
        $min_c \leftarrow$ firstThatCanHost(c, C^*, f)
      end
      $M^* \leftarrow M^* \cup \{(min_c, f)\}$
    end
  end
  return M^*

each deployed function f, compute unit c, and function replica f_i, it calculates the weight $w_{f_i,c}$ as follows:

$$cpu_{f_i}^{diff} = 1 - CPU_{f_i}^{rel} \tag{10}$$

$$lat_{f_i,c} = max(0.01, l(d(n_{f_i}, n_c))) \tag{11}$$

$$w_{f_i,c} = (cpu_{f_i}^{diff} \cdot lat_{f_i,c}) \cdot 100 \tag{12}$$

$CPU_{f_i}^{rel}$ denotes the replica's CPU usage relative to the number of available cores and is normalized in the range of $[0, 1]$. In case no CPU usage is available for a replica, we assume the load is 0. Equation 11 uses the previously defined logistic-based cost function (see Equation (4)). t_f^{rtt} is a parameter that users have to provide. Equation (11) uses the dynamically updated mean network latency, obtained via the distance function d, between the replica's node (n_{f_i}) and the compute unit's gateway node (n_c). We avoid weights of 0 and use 0.01 as a fallback value. The final weight for replica f_i ($w_{f_i,c}$) is calculated in equation 12. We multiply by 100 because our weighted round-robin implementation expects integers. The presented strategy favors replicas with low CPU usage and low network latency.

$$w_{x,c} = \text{ceil}[w_x^{mean} \cdot lat_{x,c}] \tag{13}$$

Equation 13 shows the weight calculation to re-route requests from one compute unit c to another compute unit x. It takes the average over all weights (w_x^{mean}) in compute unit x and weights it with the network latency between the compute units x and c. This optimization requires compute units to interact with each other.

VI. Proof-of-concept implementation

All components run as Python daemons and cache any events published through Redis. The gateway router is implemented in Go[2] and listens on weight changes via etcd, a distributed key-value storage. The PoC implementation does not deploy multiple local schedulers but uses a centralized unit (default Kubernetes scheduler). It also does not calculates the pressure decentralized, in contrast to our vision that dictates each compute unit's gateway to calculate it. However, this does not influence our evaluation, and we consider this as future work. Resource usage is pushed from each node via *telemd*[3] through Redis, and the workload is generated by the distributed load testing framework *galileo*[4] [26]. In the following, we present the RTT and network latency details. Figure 5 depicts a request that is forwarded from the *ECU 2*'s gateway to the *ECU 1*'s gateway and finally arrives at a replica, which contains an OpenFaaS watchdog[5]. The request is forwarded to an internal Python-based Flask HTTP server that can process up to four requests in parallel. Based on the automatically set timestamps, we can calculate RTT ($t_{\text{start}} - t_{\text{end}}$), an estimate for the network latency between nodes ($t_1 - t_0$) and the execution time ($t_3 - t_2$). Further, the figure highlights the different network connections: internet, inter-network, and intra-network. We consider the internet to be all external traffic paid by the client. Inter-network is the most expensive for the platform provider and is the traffic between two networks (i.e., compute unit-to-compute unit connection). Intra-network traffic stays in one network and is cheap.

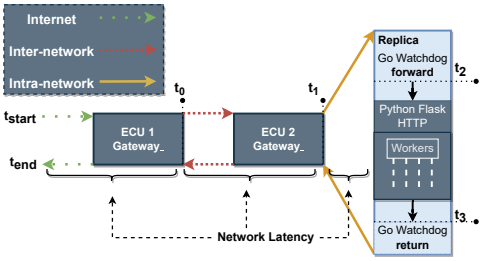

Fig. 5: Lifecyle of a trace and network types

A. Testbed

The testbed includes a variety of devices, and each is assigned to one compute unit, see Table II for further information. It is set up using the lightweight Kubernetes distri-

[2]https://github.com/edgerun/go-load-balancer/
[3]https://github.com/edgerun/telemd/
[4]https://github.com/edgerun/galileo
[5]https://github.com/openfaas/of-watchdog

TABLE II: Testbed

Device	CPU	RAM	Cluster
1x AsRock	8x Ryzen @ 2 GHz	32GB	IoT Box
1x RPI 4	4x Cortex @ 1.5 GHz	1 GB	IoT Box
1x TX2	4x Cortex @ 2 Ghz	8 GB	IoT Box
1x Nano	4x Cortex @ 1.4 GHz	4 GB	IoT Box
1x Xeon	4x Xeon @ 4.6 GHz	16 GB	Cloudlet
4x Xeon	4x Xeon @ 3.4 GHz	8 GB	Cloudlet
4x VM	4x vCPU @ 2 Ghz	8 GB	Cloud
2x NUC	4x i5 @ 2.2 GHz	16 GB	*Clients*

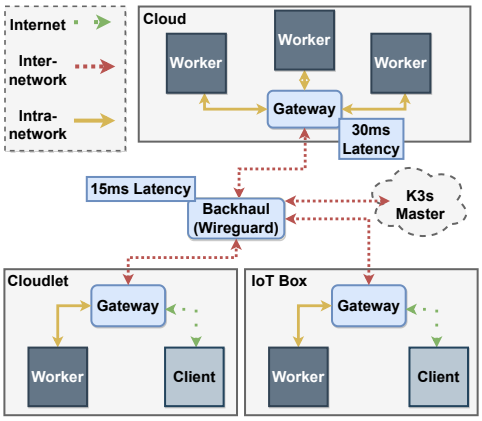

Fig. 6: Testbed setup

bution K3s[6]. There are three compute units in total: *IoT Box*, *Cloudlet*, and *Cloud*. *IoT Box* and *Cloudlet* are considered to be ECUs and the *Cloud*, a CCU. The edge-based compute units are derived from [27]. We emulate the network latency between the compute units in our system using the Linux network traffic shaping tool *tc*. Network latency is based on a study by Braud et al. [28]. A request from the edge to the cloud has a network latency of 60ms, and between the ECUs 30ms. The intra-network latency remains untouched, as well as the connection between users and ECUs ($\leq$ 10ms). Figure 6 depicts the basic structure of our testbed.

VII. Experimental Setting

A. Workload

The generated workload emulates a typical scenario for our use case: cars move around and connect to different compute units over time. This is done by first sending requests to the *IoT Box* and afterward to the *Cloudlet*. Figure 7 depicts the requests over time and highlights the origin compute unit (i.e., which compute unit the users are connected with and send the request to). First, users send requests to the *IoT Box* and then move on to the *Cloudlet*. In the end, both compute units receive requests. The experiments last around 9 minutes, processing 6000 requests (up to 35 per second). The deployed function busy waits 50ms and consumes 100% CPU during execution. The 50ms stem from our preliminary results that investigated the function execution time using a container that utilizes a modern hardware accelerator to execute an AI-based object detection (Google's EdgeTPU[7]). This work disregards the performance heterogeneity of compute units and focuses on evaluating our initial pressure definitions in a mobility-driven scenario.

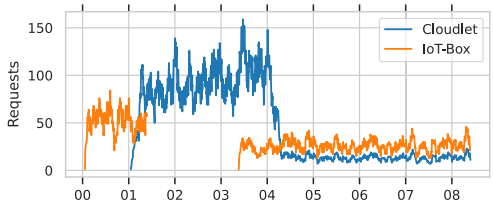

Fig. 7: Workload over a 5 second rolling window during the 9 minute experiment.

B. Comparison strategies

We compare our solution with four different scaling and placement strategies. They are split into two groups, which affects the replica they observe: *central* and *compute unit-level*. The approaches are based on the official Kubernetes algorithms but are re-implemented in Python. This decision was made in order to guarantee equality concerning the monitoring data that is used to evaluate the compute unit state. In the *central* approach, the scaling component observes all replicas, no matter the compute unit they are in. This is the typical compute unit-unaware approach that scales and places without considering the difference in network latency between the compute units. The formulation to get the *target* number of replicas required to run to satisfy the CPU and RTT thresholds is as follows:

$$target = ceil[|\text{replicas}| \cdot (\frac{\text{currentMetric}}{\text{targetMetric}})] \quad (14)$$

We denote these approaches as ***HPA*** (CPU) and ***HLPA*** (RTT). In contrast to the compute unit-unaware *central* approach, the *compute unit-based* approach evaluates the *HLPA* per compute unit. This approach is denoted as ***HLCPA*** and scales each compute unit individually, thus unaware of other compute units. Again, we implemented this approach for our testbed in a centralized manner, but in a real world scenario, a scaling component runs in each compute unit. Further, each compute unit runs at least one function instance to avoid cold startup (because the minimum of running replicas is set to 1). It does not re-route across compute units which entails that if the compute unit is full, there is no mechanism to allocate new resources in another compute unit. If the compute unit is full, it would need additional logic to select the next "best" compute unit. During the evaluation, all approaches use the presented network-aware routing strategy. This evaluation setup is similar to Baresi et al. [23], who also propose a serverless platform for edge-cloud systems.

[6]https://k3s.io/

Parameter	Value	Description
thr_{min}	0.1	Minimum pressure threshold
thr_{max}^{perf}	0.5	Max. pressure threshold for p^{perf}
thr_{max}^{eff}	0.3	Max. pressure threshold for p^{eff}
t^{rtt}	70	Target RTT in ms
t^{rtt}_{router}	35	Target RTT in ms for router optimization
CPU_f^{req}	1000	CPU millis requested from eval. function
MEM_f^{req}	512	MB requested form eval. function
t^{CPU}	60	Target CPU usage of HPA

TABLE III: Evaluation parameters

C. Parameters

We repeat each experiment five times and set the scaler's reconciliation interval to 15 seconds while the router optimizer runs every second. In contrast, the *Perf.* experiments run the optimization every 5 seconds to overcome the limitation of only scaling up functions by one replica. The reconciliation interval for the scale/placement components is based on the Kubernetes default. All other parameters are shown in Table III.

VIII. Results

A. Resource Usage and Scheduling

It is essential to understand when and where replicas were placed to understand the experiments' performance and data throughput aspects. The left side of Figure 8 shows the CPU usage per replica over all experiments as boxplots. We observe that the *HLCPA*, *HLPA*, and *Perf.* approaches have the lowest CPU utilization. The low CPU usage can be explained due to the short function execution time (50ms) and the high number of replicas deployed. The *Eff.* approach has on average a 20% CPU usage while the *HPA* approach has the highest one with around 40-50% on average.

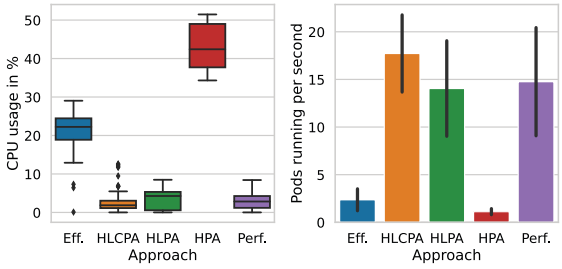

Fig. 8: CPU usage and mean number of replicas

Figure 8 also depicts the average number of replicas running per second in the right plot. This figure explains the low and high CPU usage for each approach. For example, *HPA* and *Eff.* have less than three replicas running per second, whereas the other approaches use more than 12. That means requests in the former cases are distributed over a few instances, causing them to process requests concurrently. While in the latter, the request distribution leads to a lower invocation rate per replica. It also shows that even though our application uses 100% CPU during execution, the *HPA* approach did not scale often and makes CPU-based scaling not feasible for the evaluated application. A scaling approach based on the queue length of the functions might be a better indicator.

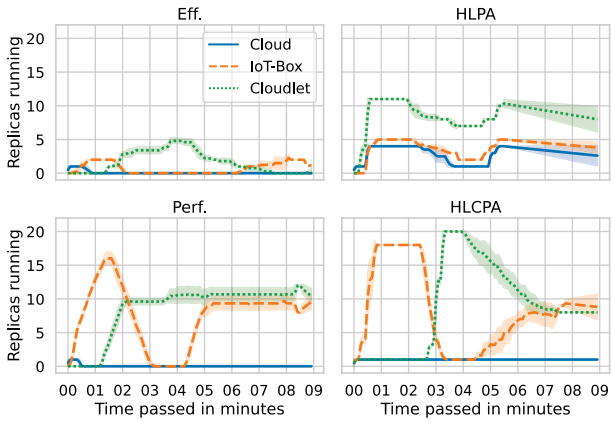

Fig. 9: Replicas running per cluster

To deepen the analysis w.r.t. placement, we include Figure 9 that shows the number of containers during the experiments. We exclude the *HPA* approach because it scaled up the function only once and placed the container in the *Cloudlet*. The results demonstrate the dynamic aspect of our pressure-based approaches as they remove replicas from the *IoT-Box* while users move to the *Cloudlet* (visible around minute 2). The *HLPA* approach also scales down during this time but randomly selects replicas to remove. This highlights a pressure-based framework's ability to adapt functions regarding user mobility. While *HLPA*, *HLCPA* and *Perf.* mostly scale up to the maximum number of replicas, the *Eff.* approach keeps the number of containers below five replicas while maintaining good performance.

B. Performance

Figure 10 displays on the left side the average 90*th* percentile of RTT and on the right side, the network latency across all experiments. The error bars show the 95% confidence interval in which the true mean lies based on a bootstrap sample of 1000. Note that we exclude a small percentage of outliers (around 30 from 6000 requests) that were caused by timeouts through the early tear-down of replicas that were still processing requests. While *HPA* had the highest RTT and network latency, the *Eff.* approach still violates the latency by having requests longer than 150ms. Interestingly the *HLCPA* experiments had trouble maintaining a low RTT. We argue that this circumstance happened because it took the system some time to scale up in the *IoT Box* because resources were slowly released from the *Cloudlet* (see Figure 9 from minute 4). In contrast, the *Perf.* approach was able to scale up resources quicker in the *IoT Box* and also has a much

[7]https://coral.ai/

lower RTT. The *HLPA* has the best results in terms of RTT, and network latency, followed closely by the performance-oriented pressure-based approach (*Perf.*). While *HLPA* had the best performance, the resource usage results showed that it randomly tore down replicas across all clusters. Thus, it does not adapt efficiently to user mobility as pressure-based approaches.

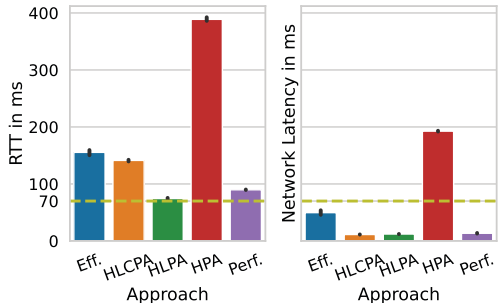

Fig. 10: P90 RTT and network latency over all experiments

C. Network traffic

In terms of cost, the communication between clusters is the most expensive and therefore is examined in the following. The network latency (see Figure 10) of *HPA* is the highest because only the initial replica in the *Cloud* and one in the *Cloudlet* were running, leading to many requests being forwarded to the *Cloud*. *HPA* also has the highest number of inter-network requests (5542). The *Eff.* approach also experienced high network latency caused by the relatively slow resource migration but was able to select clusters deterministically. The average number of inter-network requests are: *Eff.* (1783), *Perf.* (426), *HLPA* (300) and *HPA* (5542). The *HLPA* can only maintain the low network latency because of deploying as many replicas as possible. Due to the testbed's size, it is enough to serve the workload adequately. Our approaches show that mobility-aware scaling and placement reduce network latency and cost. These results also prove the effectiveness of our network-aware routing scheme used throughout the experiments. Even though function replicas are running across the system, inter-network traffic is kept to a minimum.

D. Pressure

In the following, we discuss the different pressure formulations in terms of interpretability and offer a detailed comparison between the approaches. Figure 11 shows various metrics across evaluation scenarios and the pressure in relation to other metrics concisely. The figure shows from top to bottom: the RTT, the performance-oriented pressure (p^{perf}), the efficient-based pressure (p^{eff}), total number of replicas, and the request pattern for each approach across the experiments (i.e., the total number of requests sent from that compute unit). Note that the shown values are aggregated over the whole system and contain values across clusters. Figure 9 shows that metrics vary

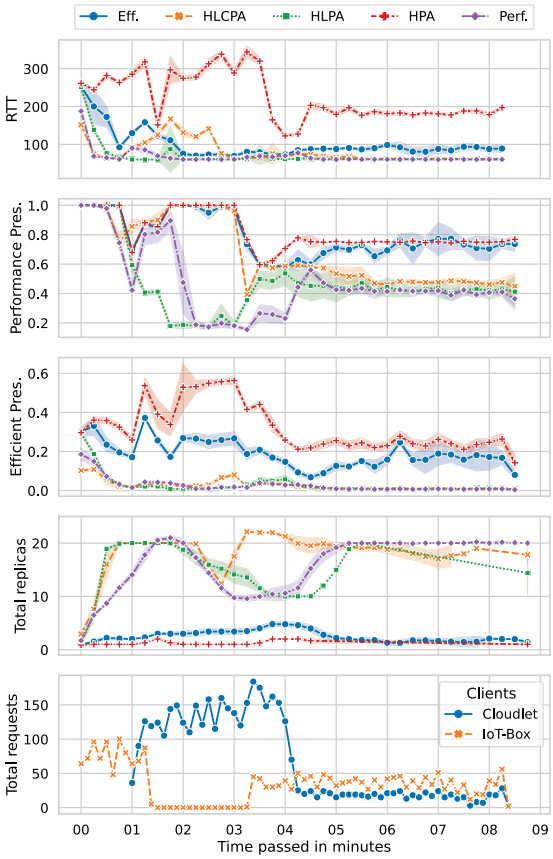

Fig. 11: Metrics across all experiments.

heavily between clusters. First, the number of total replicas and the RTT highlight differences between the two approaches (*Eff.* and *Perf.*). The *Eff.* approach tried to keep a balance between total replicas and RTT, while the *Performance* approach focused solely on performance and disregarded resource efficiency. The *Eff.* approach strikes a good balance between the number of running replicas and the performance. It can keep the pressure below the used maximum threshold of 0.3 and has a low number of replicas running throughout the experiment. This shows that the pressure-based approach can minimize the number of replicas while maintaining good performance. The *Perf.* approach keeps a low number of replicas during the peak workload at around 4 minutes, whereas the *HLCPA* uses up all available resources. The *HLPA* approach reduces the number of replicas, but this happens in a seemingly random fashion. Looking at the *efficient pressure* plot, we can see that the *HPA* and *Eff.* approaches have the highest pressure throughout the experiments, and the *Eff.* approach can quickly reduce it. Other approaches have a very low *efficient pressure* due to the low CPU usage caused by deploying many replicas. We think a different resource pressure component might be better suited (i.e., the number of replicas). On the other hand, the *performance pressure* shows that the *HPA* and *Eff.* approaches

violate the RTT constraints while the other approaches keep it at around 0.5. This shows that we can use the pressure as a valid estimation for the system's performance and state. The reason lies in the RTT pressure component (p^{RTT}), as it quickly penalizes requests that take longer than the target. Both pressures show the inability of the default cloud-centric *HPA* approach to adapt the system experiencing high pressure across the experiments successfully. We conclude that each pressure definition can reach its respective goal and serve as a complex metric to evaluate the system's state. Further, results also showed that the dynamic function adaptations are mobility-aware and suitable for scenarios with moving users.

IX. Conclusion

Managing hybrid edge-cloud systems, where mobility, latency, and data throughput play an important role, is complex and, therefore, should be done autonomously by platform providers. Current operational mechanisms of cloud-centric serverless computing lack awareness of geo-distributed setups and the problem of fluctuating workload due to user mobility. We presented a pressure-based joint scaling and placement framework and a network-aware routing strategy. We evaluated a PoC of our framework using two different pressure formulations. They have shown deterministic behavior regarding the final system performance based on simple low-level metrics. The efficiency-oriented pressure reduced resource usage by up to 83%, while the performance-oriented pressure used the available resources to reduce the RTT by up to 92% to cloud-centric solutions. Showing that the pressure can act as a viable proxy that encapsulates cluster state and workload, minimizing communication costs towards a central management unit. We conclude that the pressure can efficiently guide function adaptations and enables mobility-aware scaling and placement of serverless functions. The novel concept of pressure is highly extensible and can be adapted for various application types. Besides sophisticated local placement strategies, future work comprises proactive approaches based on historical pressure data and large-scale simulations.

References

[1] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[2] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[3] Q. Liu, S. Huang, J. Opadere, and T. Han, "An edge network orchestrator for mobile augmented reality," in *Proc. of IEEE INFOCOM'18*, 2018, pp. 756–764.

[4] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *Proc. of IEEE INFOCOM'16*, 2016.

[5] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2333–2345, 2018.

[6] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[7] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *Proc. of IEEE/ACM SEC'18*, 2018, pp. 373–377.

[8] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. M. Leung, "Tailored learning-based scheduling for Kubernetes-oriented edge-cloud system," in *Proc. of IEEE INFOCOM'21*, 2021, pp. 1–10.

[9] I. Labriji, F. Meneghello, D. Cecchinato, S. Sesia, E. Perraud, E. C. Strinati, and M. Rossi, "Mobility aware and dynamic migration of mec services for the internet of vehicles," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 570–584, 2021.

[10] D. Garg, P. Shirolkar, A. Shukla, and Y. Simmhan, "Torquedb: Distributed querying of time-series data from edge-local storage," in *European Conference on Parallel Processing*. Springer, 2020, pp. 281–295.

[11] A.-V. Michailidou, A. Gounaris, M. Symeonides, and D. Trihinas, "Equality: Quality-aware intensive analytics on the edge," *Information Systems*, vol. 105, p. 101953, 2022.

[12] G. Tang, D. Guo, K. Wu, F. Liu, and Y. Qin, "Qos guaranteed edge cloud resource provisioning for vehicle fleets," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 6, pp. 5889–5900, 2020.

[13] X. Sun, J. Zhao, X. Ma, and Q. Li, "Enhancing the user experience in vehicular edge computing networks: An adaptive resource allocation approach," *IEEE Access*, vol. 7, pp. 161 074–161 087, 2019.

[14] A. M. Maia, Y. Ghamri-Doudane, D. Vieira, and M. F. de Castro, "Dynamic service placement and load distribution in edge computing," in *Proc. of IEEE CNSM'20*. IEEE, 2020, pp. 1–9.

[15] S. Ge, M. Cheng, and X. Zhou, "Interference aware service migration in vehicular fog computing," *IEEE Access*, vol. 8, pp. 84 272–84 281, 2020.

[16] E. F. Maleki, L. Mashayekhy, and S. M. Nabavinejad, "Mobility-aware computation offloading in edge computing using machine learning," *IEEE Transactions on Mobile Computing*, 2021.

[17] Q. Yuan, J. Li, H. Zhou, T. Lin, G. Luo, and X. Shen, "A joint service migration and mobility optimization approach for vehicular edge computing," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 8, pp. 9041–9052, 2020.

[18] M. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *Proc. of ICCCN'21*, 2021.

[19] S. Pasteris, S. Wang, M. Herbster, and T. He, "Service placement with provable guarantees in heterogeneous edge computing systems," in *Proc. of IEEE INFOCOM'19*, 2019, pp. 514–522.

[20] D. Haja, M. Szalay, B. Sonkoly, G. Pongracz, and L. Toka, "Sharpening kubernetes for the edge," in *Proc. of the ACM SIGCOMM'19*, 2019, pp. 136–137.

[21] L. Baresi and D. F. Mendonça, "Towards a serverless platform for edge computing," in *Proc. of IEEE ICFC'19*, 2019, pp. 1–10.

[22] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, 2019.

[23] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, "NEPTUNE: Network- and gpu-aware management of serverless functions at the edge," in *Proc. of ACM SEAMS'22*, 2022, p. 144–155.

[24] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[25] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *Proc. of IEEE SERVICES'19*, vol. 2642, 2019.

[26] T. Rausch, P. Raith, P. Pillai, and S. Dustdar, "A system for operating energy-aware cloudlets," in *Proc. of ACM/IEEE SEC'19*, 2019, pp. 307–309.

[27] T. Rausch, C. Lachner, P. A. Frangoudis, P. Raith, and S. Dustdar, "Synthesizing plausible infrastructure configurations for evaluating edge computing systems," in *Proc. of USENIX HotEdge'20)*, 2020.

[28] T. Braud, Z. Pengyuan, J. Kangasharju, and H. Pan, "Multipath computation offloading for mobile augmented reality," in *Proc. of IEEE PerCom'20*, 2020, pp. 1–10.