Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# Optimized container scheduling for data-intensive serverless edge computing

Thomas Rausch [*], Alexander Rashed, Schahram Dustdar

*Distributed Systems Group, TU Wien, Austria*

ARTICLE INFO

*Article history:*
Received 21 February 2020
Received in revised form 13 June 2020
Accepted 6 July 2020
Available online 7 August 2020

*Keywords:*
Edge computing
Serverless
Container scheduling
Machine learning

ABSTRACT

Operating data-intensive applications on edge systems is challenging, due to the extreme workload and device heterogeneity, as well as the geographic dispersion of compute and storage infrastructure. Serverless computing has emerged as a compelling model to manage the complexity of such systems, by decoupling the underlying infrastructure and scaling mechanisms from applications. Although serverless platforms have reached a high level of maturity, we have found several limiting factors that inhibit their use in an edge setting. This paper presents a container scheduling system that enables such platforms to make efficient use of edge infrastructures. Our scheduler makes heuristic trade-offs between data and computation movement, and considers workload-specific compute requirements such as GPU acceleration. Furthermore, we present a method to automatically fine-tune the weights of scheduling constraints to optimize high-level operational objectives such as minimizing task execution time, uplink usage, or cloud execution cost. We implement a prototype that targets the container orchestration system Kubernetes, and deploy it on an edge testbed we have built. We evaluate our system with trace-driven simulations in different infrastructure scenarios, using traces generated from running representative workloads on our testbed. Our results show that (a) our scheduler significantly improves the quality of task placement compared to the state-of-the-art scheduler of Kubernetes, and (b) our method for fine-tuning scheduling parameters helps significantly in meeting operational goals.

## 1. Introduction

The requirements of data-intensive applications that process data located at the edge of the network are challenging the prevalent cloud-centric compute model [1–3]. Consider an urban sensing scenario [4] where sensor nodes deployed throughout a city provide applications, such as machine learning workflows, with real-time access to sensor or camera feeds. It may be impractical or infeasible to offload compute tasks to the cloud, because data would have to leave the edge network, causing privacy issues, or incurring high latency and bandwidth use. To enable this emerging family of edge-native applications, compute resources are placed at the network edge and pooled together to form a diverse and distributed compute fabric. While traditional cloud-native approaches to resource management, service orchestration, and scheduling have reached a high level of maturity, they are challenged when dealing with key characteristics of distributed edge systems: compute device heterogeneity, geographic dispersion, and the resulting operational complexity [5].

Serverless edge computing has emerged as a compelling model for dealing with many of these challenges associated with edge infrastructure [6–11]. It expands on the idea of serverless computing, which first drew widespread attention when Amazon introduced in 2015 its AWS Lambda service [12]. It allowed users to develop their applications as composable *cloud functions*, deploy them through a Function-as-a-Service (FaaS) offering, and leave operational tasks such as provisioning or scaling to the provider. Analogous to the idea of serverless cloud functions, we imagine that *edge functions* can significantly simplify the development and operation of certain edge computing applications. Operating data-intensive edge functions, and the limiting factors of state-of-the-art serverless platforms in supporting them, is the focus of this paper.

Current serverless platforms exhibit several limitations for enabling data-centric distributed computing [13], that are further exacerbated by the operational properties that underpin edge systems, in particular when making function placement decisions. Specifically, this manifests as follows. First, they do not consider the proximity and bandwidth between nodes [6,13], which is particularly problematic for edge infrastructure where the distance between compute nodes, data storage, and a function code repository (e.g., a container registry), incurs significant latencies [14]. Second, fetching and storing data is typically part of the function

[*] Corresponding author.
*E-mail address:* t.rausch@dsg.tuwien.ac.at (T. Rausch).
*URL:* https://dsg.tuwien.ac.at/team/trausch (T. Rausch).

https://doi.org/10.1016/j.future.2020.07.017
0167-739X/© 2020 Published by Elsevier B.V.

code and left to application developers (e.g., manually accessing S3 buckets), which makes it hard for the platform to reason about data locality and data movement trade-offs [6,15]. Third, they provide limited or no support for specialized compute platforms or hardware accelerators such as GPUs [13,16], leaving potential edge resources that provide such capabilities underutilized.

We present Skippy, a container scheduling system that facilitates the efficient placement of serverless edge functions on distributed and heterogeneous clusters. Skippy interfaces with existing container orchestration systems like Kubernetes, that were not designed for edge computing scenarios, and makes them sensitive to the characteristics of edge systems. The core component of Skippy is an online scheduler, modeled after the Kubernetes scheduler, which implements a greedy multi-criteria decision making (MCDM) algorithm. We introduce four new scheduling constraints to favor nodes based on (1) proximity to data storage nodes, (2) proximity to the container registry, (3) available compute capabilities (e.g., for favoring nodes that have hardware accelerators), and (4) edge/cloud locality (e.g., to favor nodes at the edge). We have found that these constraints are a critical missing piece for making an effective trade-off between data and computation movement in edge systems. Furthermore, we recognize that tuning scheduler parameters for effective function placement is challenging, as it requires extensive operational data and expert knowledge about the production system. Instead, we propose a method that leverages the tight integration of the scheduler with a simulation framework, in combination with existing multi-objective optimization algorithms, to optimize high-level operational goals such as function execution time, network usage, edge resource utilization, or cloud execution cost. We show how the scheduler and optimization technique work in tandem to enable serverless platforms to be used in a wide range of edge computing scenarios.

The contributions of this paper are as follows:

- Skippy: a container scheduling system that enables existing serverless frameworks to support *edge functions* and make better use of edge resources. Our scheduler introduces components and constraints that target the characteristics of edge systems.
- A method to tune the weights attached to low-level constraints used by the scheduler, by optimizing high-level operational goals defined by cluster operators. To compute the optimization we introduce a serverless system simulator we have developed.
- We demonstrate Skippy's performance in various scenarios using data from our testbed and running trace-driven simulations. We analyze emerging edge computing scenarios to synthesize edge topologies.
- Open dataset of traces from extensive profiling of our edge testbed, and synthetic traces from our simulations of different infrastructure scenarios [17].

## 2. Related work

Serverless computing in the form of cloud functions is seen by many in both industry and academia as a computational paradigm shift [12,18–20]. Only recently has the serverless model, and in particular the FaaS abstraction, been investigated for edge computing. Gilkson et al. [7] proposed the term *Deviceless Edge Computing*, to emphasize how serverless edge computing helps to hide the underlying compute fabric. However, the characteristics of edge infrastructure exacerbate the challenges of serverless computing, such as platform architecture [8,9], runtime overhead [21], cold starts [22], or scheduling [6]. In a recent effort, Baresi and Mendonça [9] proposed a serverless edge computing platform based on OpenWhisk. They focus on the complete system architecture design and the implementation of load balancer that considers distributed infrastructure. In industry, AWS IoT Greengrass [18] enables on-premises execution of AWS Lambda functions, Amazon's serverless computing platform. AWS IoT Greengrass currently allows machine learning inference on edge devices, using models trained in the cloud. However, the configuration of AWS IoT Greengrass devices is highly static, since the functions running on a device are defined using a local configuration file. In an effort to extend existing serverless runtimes to enable serverless edge computing, Xiong et al. [23] implemented a set of extensions to Kubernetes called *KubeEdge*. Its most important component, the *EdgeCore* node agent, manages networking, synchronizes state, and potentially masks network failures. Our approach is complementary, in that Skippy provides an edge-enabled scheduling system for making better function placement decisions on edge infrastructure.

There is a strong relation between serverless function scheduling and the service placement problem (SPP). Many variants of the SPP for different edge computing system models and operational goals exist [24–28]. Typically, the problem is formulated as an optimization problem, and an algorithm is implemented to solve an instance of the problem heuristically by leveraging assumptions within the system model. Gog et al. [29] map the service placement problem to a graphic data structure and model it as a min-cost max-flow (MCMF) problem. Hu et al. [30] pursue a similar approach by modeling the service placement as a min-cost flow problem (MCFP) which allows encoding multi-resource requirements and affinities to other containers. Their scheduler considers the costs for offloading tasks from the edge nodes to rented cloud resources. Aryal and Altmann [31] map the service placement problem to a multi-objective optimization problem (MOP) and propose a genetic algorithm to make placement decisions. Bermbach et al. [10] propose a distributed auction-based approach for a serverless platform in which application developers bid on resources. These and other approaches [32,33] have in common that the constraints considered by the schedulers are defined a priori. Generally, scheduling algorithms described in academic literature often assume very detailed information about the system state and service requirements, whereas in production systems, both may not be available.

Many online container schedulers, such as the ones from Docker Swarm, Kubernetes, or Apache Mesos, implement a greedy MCDM procedure. A key phase in this procedure is *scoring*, i.e., calculating the score of a feasible node by invoking a set of *priority functions*, building a weighted sum of priority scores, and selecting the highest scoring node for scheduling. The Kubernetes scheduler implements this procedure in a very general and flexible way [34], which is why we build on its model, as it generalizes to many other container schedulers. It allows to dynamically plug in and configure different hard- and soft-constraints, theoretically even at runtime. It is unclear whether and how existing SPP approaches could be implemented in this framework. Our work is an effort to examine how ideas from service placement in edge computing, such as using latency and proximity awareness for placement decision, can translate to, or be implemented in, these types of schedulers.

## 3. Background & application scenarios

This section outlines the domain for which we have developed our system. We discuss different application and infrastructure scenarios to motivate serverless edge computing and highlight systems challenges we uncovered during initial experiments. Furthermore, we provide background on the operational underpinnings of serverless platforms using as examples Kubernetes and OpenFaaS.

### 3.1. Data-intensive serverless edge computing

Many modern application scenarios require data processing at the edge of the network, close to where data is generated. Typical characteristics and requirements associated with data-intensive edge computing applications can be summarized as follows:

- heterogeneous workloads: the application is composed of multiple functions that have different computational requirements (e.g., GPU acceleration)
- locality sensitive: some parts of the application are locality sensitive, e.g., should not be executed in the cloud because consumers are located at the edge
- latency sensitive: some parts of the application are required to provide service quality at low latency
- high bandwidth requirements: some parts of the application may exchange large amounts of data

Research has shown that edge AI applications that deal with video or image data typically have all of these requirements [2,3,35,36]. Smart city scenarios are also an illustrative example. Data from sensor and camera arrays distributed throughout the city can be used to create analytics models such as traffic risk assessment, crowd behavior, flooding or fire spreading models, or ecosystem/plant monitoring [2]. They could also serve as sensory input for cognitive assistance applications [2,3]. Serverless computing may be a good model for implementing such applications at scale on a distributed compute fabric [6].

We implemented a prototypical edge computing application that has the characteristics and requirements described above. Specifically, we found the most generalizable application to be a machine learning (ML) workflow with multiple steps, where each step has different computing requirements, and needs to make efficient use of a diverse set of edge resources. We consider a typical ML pipeline with three steps [37], where each step can be implemented as a serverless function: (1) **data preprocessing**, (2) **model training** (that can be accelerated by GPUs), and (3) **model serving**. In our concrete example we use the MNIST dataset to train an image classifier because of the dataset's availability allowing reproducibility.

A serverless function is essentially an event handler that reacts to some event. For example, in case of model training, the event would be triggered by the previous ML workflow step, i.e., the data preprocessing. An example of a serverless function written in Python that implements an ML training step is shown in Listing 1. In OpenFaaS, the platform packages function code and its dependencies into a container image, pushes it to a registry, from where the code is pulled by a compute node after scheduling.

```
import boto3
import numpy
# ... import ML libraries such as tensorflow or mxnet

def handle(req):
  s3 = boto3.client('s3')
  with open(tmpfile, 'wb') as f:
    s3.download_fileobj('bucket', req['train_data'], f)

    data = numpy.load(f)
    model = train_model(data, req['train_params'])

    s3.upload_fileobj(serialize(model), 'bucket',
     request['serialized_model'])
```

Listing 1: Example of a data-intensive serverless function.

The model training function involves: (1) connecting to an S3 server, downloading the training data from the file object encoded in the request (which was previously generated by the data preprocessing step), converting the data into some appropriate format for running a training algorithm, and then uploading the serialized model. As every data-intensive function has a similar format, i.e., fetching, processing, and then storing data, we previously developed a higher-level abstraction for these functions [6]. Specifically, we elevate fetching and storing data as platform features, which allows the platform to reason over metadata of the function, e.g., which specific data is pulled (encoded by a URN for example) to locate the closest data store that holds the data. We use this feature in the scheduler (see Section 4.3.1) for determining the trade-off between data and computation movement. Fig. 1 shows a comparison between the size of container images for each function of our application, and the total amount of data each function has to transfer during its execution. It also shows a back-of-the-envelope calculation on how much time the cloud or edge spends on transferring either container images or data for each function step. We consider a typical scenario, where an edge network has a 1 Gb/s internal bandwidth, 25 Mb/s uplink and 100 Mb/s downlink. Data is located at the edge, and the container registry is located in the cloud, which also has an internal bandwidth of 1 Gb/s. We can see that the difference in uplink and downlink bandwidth play a significant role in trading off data and computation movement.

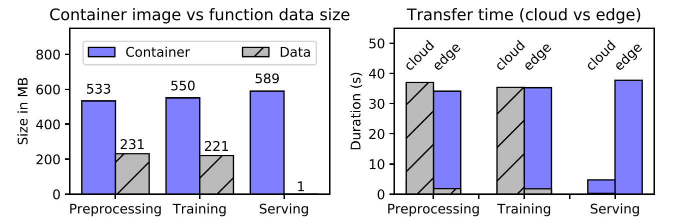

**Fig. 1.** Comparison of container image sizes and total data transferred by functions. The right figure shows the time spent on either container image or data transfer in either cloud or edge networks.

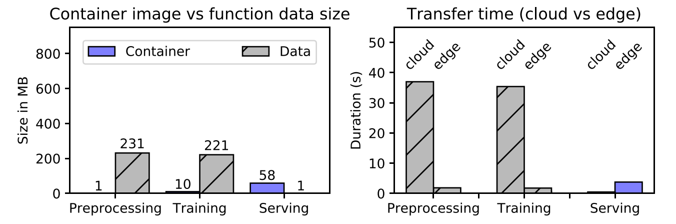

**Fig. 2.** The same calculation as Fig. 1 when subtracting shared layers between images and only considering unique image size.

Docker uses a layered file system, meaning that layers can be shared between container images. Because most containers build on similar base images, the unique image size when considering shared layers if often much smaller. For distributing container images this means that, if the base image has already been downloaded by some container, downloading a different container image that uses the same base image will also be much faster. When inspecting the images of our specific application, we found that almost 90% is shared across images. Fig. 2 shows the same calculation as above, illustrating that, now, most of a function's latency comes from pulling data.

### 3.2. Edge cloud compute continuum

A challenging aspect of edge computing are the extremes of the compute continuum [5]. We have built an edge computing testbed that reflects this, which we describe in more detail in Section 6.1. For the edge compute nodes, we consider the following computers and architectures. We have presented various application scenarios for each in [38]. (1) **Small-scale**

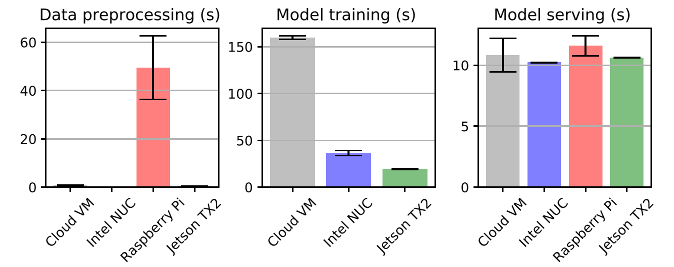

**Fig. 3.** Average execution time and standard deviation of ML workflow functions in seconds on different device types illustrating both workload and device heterogeneity.

**Table 1**
Cluster configurations of different scenarios.

| Scenario | Category | Nodes | % of compute device types | | | |
|---|---|---|---|---|---|---|
| | | | VMs | SBC | NUC | TX2 |
| Our testbed | | 7 | 14 | 57 | 14 | 14 |
| S1: Urban sensing | Edge | 1170 | 3 | 39 | 19 | 39 |
| S2: Industry 4.0 | Hybrid | 110 | 40 | 40 | 10 | 10 |
| S3: Cloud regions | Cloud | 450 | 100 | 0 | 0 | 0 |

**Table 2**
Device type specifications.

| Device | Arch | CPU | RAM |
|---|---|---|---|
| VM | x86 | 4 × Core 2 @ 3 GHz | 8 GB |
| SBC | arm32 | 4 × Cortex-A53 @ 1.4 GHz | 1 GB |
| NUC | x86 | 4 × i5 @ 2.2 GHz | 16 GB |
| TX2 | aarch64 | 2 × Cortex-A57 @ 2 GHz<br>256-core Pascal GPU | 8 GB |

**data centers** that use VM-based infrastructure and are placed at the edge of the network, often termed cloudlets [14]. (2) **Small form-factor computers**, such as Intel's Next Unit of Computing (NUC) platform with built-in CPUs are used in, e.g., Cannonical's Ubuntu Orange Box [39]. (3) **Single Board Computers (SBCs)** such as ARM-based Raspberry Pis used as IoT gateways or micro clusters [40]. (4) **Embedded AI hardware**, such as NVIDIA's Jetson TX2 that provide GPU acceleration and CUDA support [41].

We have profiled our ML workflow steps as OpenFaaS functions on these different devices. Table 2 lists the hardware specifications of the device instances we used. Fig. 3 shows the results of 156 warm function executions. The Raspberry Pis were not able to run the model training step as they ran out of memory. The results show the impact of extreme device heterogeneity. Also, we can see that the model training step benefits greatly from GPU acceleration, performing better on a Jetson TX2 compared to an Intel NUC despite the NUC's powerful i5 processor.

### 3.3. Cluster infrastructure scenarios

The lack of available reference architectures for edge systems and data on real-world deployments, make it challenging to evaluate edge computing systems in general [38]. Cloud computing architectures are fairly well understood, and traces such as the Borg cluster data [42], allow well grounded systems evaluations. To evaluate our approach under different conditions in a simulated environment, we use the *Edge Topology Synthesizer* framework [38] to synthesize plausible cluster configurations, which we draw from three different existing or emerging application scenarios. Table 1 summarizes the cluster configurations for our testbed, and each scenario which we describe below:

*S1: Urban sensing.* More and more cities are deploying sensor arrays and cameras to enable Smart City applications that require real-time data on urban environments [4]. These sensors are often attached to IoT gateways, and complemented by proximate compute resources such as cloudlets [14] to process sensor data. For this scenario, we assume a total of 200 sensor nodes, where each node is equipped with two SBCs (e.g., for data processing and communication). Furthermore, we assume that throughout the city, there are installations of cloudlets that comprise an Intel NUC, and two embedded GPU devices per sensor node camera for, e.g., video processing tasks. To meet peak demands, 30 VMs hosted at a regional cloud provider are added as fallback resources into to the cluster. In terms of network topology, we assume that each municipal district forms an edge network. Each edge network has an internal LAN bandwidth of 1 Gb/s and connected with 100/25 Mb/s down/uplink to the internet. Cloud nodes have an internal bandwidth of 10 Gb/s and a direct 1 Gb/s uplink to the internet. These data are plausible extensions of the urban sensing project Array of Things (AoT) [4], which operates a deployment in Chicago that currently consists of about 200 sensor nodes. Each AoT node contains two SBCs, and is connected via a mobile LTE network to the Internet.

*S2: Industry 4.0.* Edge computing is considered a key component in realizing Industry 4.0 concepts such as smart manufacturing or the Industrial IoT (IIoT) [43]. For this scenario, we assume that several factories at different locations are equipped with edge computing hardware, and each location has an on-premises but provider-managed cloud (e.g., a managed Microsoft Azure deployment, where on-premises cloud resource use is billed). We assume ten factory locations, each having 4 SBCs as IoT gateways, 1 Intel NUC, 1 Jetson TX2 board, and 4 VMs on the on-premises cloud. The numbers are plausible extensions to the prototypes presented in [43], and the general trend towards using embedded AI hardware for analyzing real-time sensor and video data in IIoT scenarios [44]. Each edge and on-premises cloud has a data store. The SBCs are connected via 300 Mb/s WiFi link to an AP that has a 10 Gb/s link to the edge resources, and a 1 Gb/s link to the on-prem cloud. Premises are connected via 500/250 Mb/s internet down/uplink.

*S3: Cloud federation.* To compare our system in non-edge computing scenarios, we also consider a cloud computing configuration where there are no edge devices and less heterogeneity than in edge scenarios [42]. We model a cloud federation scenario across three cloud regions, where each region has, on average, 150 VMs. All regions contain several nodes with data stores. Region one has slightly more VMs and more storage nodes than the others. The bandwidth is 10 Gb/s within a region, and 1 Gb/s cross-region. These data match the results of a recent benchmark on cross-region traffic of AWS [45]. We assume that each region has local access to a container registry, e.g., through a CDN.

### 3.4. Technical background: Kubernetes & OpenFaaS

Our system is designed to extend existing platforms that enable serverless computing and FaaS deployments, such as Kubernetes and OpenFaaS. Because our prototype was developed for these two systems, we present technical background on the interplay between the two. The core mechanisms, however, are found in similar systems.

#### 3.4.1. Kubernetes & container scheduling

Kubernetes is a container orchestration system used for automating application deployment and management in distributed environments. It is a popular runtime for serverless computing, micro-service-based application deployments, and, increasingly, Function-as-a-Service (FaaS) platforms [34]. Using Kubernetes, FaaS platforms can package function code into lightweight container images, and can then make use of all the features of the Kubernetes platform, such as scheduling, autoscaling, or request routing. The Kubernetes scheduler is one of the critical components of the platform. The task of the scheduler is to assign a *pod* (the atomic unit of deployment in Kubernetes) to a cluster node. A pod can consist of one or more containers, shared storage volumes, network configuration, and metadata through which the pod can, for example, communicate its resource requirements. The Kubernetes scheduler is an online scheduler, meaning it receives pods over time and generally has no knowledge of future arrivals. It is therefore different from many SPP solutions that schedule several services at once [24]. Similar to many resource schedulers of real-world systems, it employs a greedy MCDM procedure, which we formalize in Section 5. Hard and soft constraints are implemented as follows. First, the available cluster nodes are filtered by *predicate functions*. These functions evaluate to true or false, and represent hard constraints that eliminate nodes incapable of hosting a pod, e.g., because they are unable to provide the resources requested by a pod. Second, the remaining set of feasible nodes are ranked by *priority functions* that represent soft constraints to favor nodes based on their suitability for hosting the pod. Calculating the score for a pod–node tuple involves invoking each active priority function, normalizing the values of each function to a range between 0 and 10, and building a weighted sum. The highest scoring node is then selected for placement. Kubernetes provides a variety of predicate and priority functions that can be configured in the scheduler. However, as we describe in more detail in Section 4.3.1, the default priority functions do not perform well in edge scenarios. In particular, they do not consider the critical trade-off between data and computation movement which we have highlighted earlier. In the remainder of this paper, we use the terminology of Kubernetes (i.e., pod, node, priority function), to refer more generally to a unit of deployment, cluster resource, and soft constraint, respectively.

### 3.4.2. OpenFaaS

OpenFaaS is a serverless framework that uses Kubernetes as both execution runtime and deployment platform. Function code is packaged into Docker containers, and a small watchdog is added to the container that executes the function based on HTTP calls triggered by events or through invocations of the OpenFaaS API gateway. With OpenFaaS, the Kubernetes scheduler is triggered in two situations: an initial manual deployment of new functions, or automated function replica creation through autoscaling. If an OpenFaaS user runs the `faas-cli deploy` command to deploy function code, the code is packaged into a Kubernetes pod, and the pod is placed in the scheduling queue. Subsequent requests to the function triggered by events or HTTP endpoint calls are forwarded to Kubernetes, which takes care of load balancing requests among running replicas. This is the case if the function's autoscaling policy is set to at least one replica, and is useful to avoid cold starts for functions that are invoked frequently. A cold start refers to a function invocations where the container image has to be downloaded and the container is started for the first time, which incurs significant latency. For short-lived functions that are not invoked frequently, and would otherwise block a node's resources despite being idle, OpenFaaS allows a scale-to-zero policy, which removes such idle functions from nodes after a short time. This is useful for functions such as the data pre-processing or training step in our ML pipeline. In this case, a request to a function immediately triggers a replica creation and therefore scheduling.

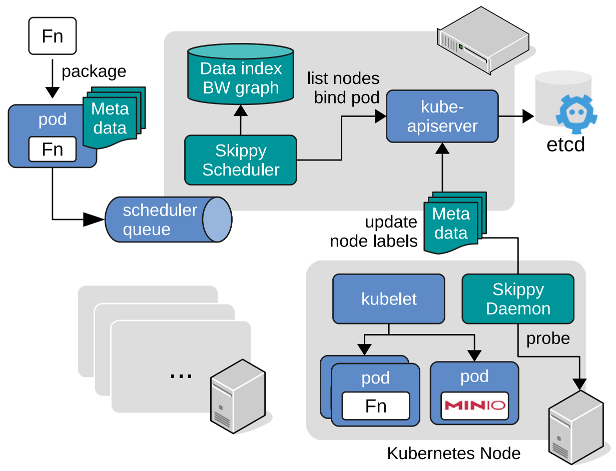

**Fig. 4.** Overview of Skippy's components and their interaction in a deployment with Kubernetes.

## 4. Skippy — design and prototype implementation

Skippy is designed to integrate with existing container orchestration systems, allowing them to satisfy the requirements of data-intensive edge computing applications described in Section 3. Skippy adds runtime components and domain concepts that make such systems sensitive to device capabilities; locality between nodes, data and container images; and cloud/edge network context. To do so, Skippy requires a minimal interface to the container orchestration system. We demonstrate this by building a prototype for Kubernetes. This section describes Skippy, its individual components, the scheduling logic, and the integration with Kubernetes and OpenFaaS.

### 4.1. System overview

We briefly outline the main components of Skippy. Fig. 4 shows the specific integration with Kubernetes.

- **metadata schema:** Skippy makes heavy use of container and node labels, which are supported by many container platforms, to communicate information about functions and compute nodes to the scheduler. Skippy uses various mechanisms to automate labeling, such as the skippy-daemon or annotation parsing as described in [6]. All metadata labels of Skippy have the prefix `*.skippy.io`.
- **skippy-daemon:** a daemon that runs on all cluster nodes alongside the primary node agent (e.g., kubelet in the case of Kubernetes). It scans a node's compute capabilities, such as the availability of a GPU, and then labels the node with the corresponding metadata. It can do this periodically to react to pluggable capabilities, such as USB attached accelerators.
- **skippy-scheduler:** the Skippy scheduler is an online scheduler that is sensitive to the characteristics of edge systems. It requires access to the cluster state and a programmatic way of binding containers to nodes. In the case of Kubernetes, the kube-apiserver provides these features via REST APIs.
- **data index & bandwidth graph:** Functions can access data via storage pods that host MinIO instances (an S3 compatible storage server), distributed across the cluster. Skippy currently does not automatically manage these storage nodes. Replication and data distribution is left to other system components. However, Skippy dynamically discovers storage nodes, keeps an index of the file tree, and is sensitive to the proximity between compute and storage nodes by using the bandwidth graph.

## 4.2. Node and function metadata collection

As any scheduler, Skippy depends on certain information about the cluster state and the job requirements. Skippy makes heavy use of metadata about functions and compute nodes in the cluster to communicate this information to the scheduler. We use the Skippy daemon to collect node metadata, and a high-level programming API to collect function metadata. The node metadata are stored and accessed via the cluster orchestration system. In the case of Kubernetes, this is stored in etcd, a distributed key–value store. If the orchestrator does not provide a storage system, Skippy can also store the metadata in memory.

### 4.2.1. Node metadata: Skippy daemon

The Skippy daemon is deployed as a container on all cluster nodes. It automatically probes a node's capabilities and maintains its Skippy-specific labels. Currently, the daemon probes if a node provides an NVIDIA GPU, the availability and version of a CUDA installation, and if the node is running a MinIO storage pod. The daemon code allows straight-forward addition of custom capability probes. When nodes are added to the cluster at runtime, the Skippy daemon labels the node with an appropriate locality label (edge/cloud). The system overhead of running the daemon is minimal given a fairly simple Python implementation. It requires roughly 120 MB of disk space and 25–40 MB of RAM depending on the CPU architecture, making it feasible even for resource constrained devices.

### 4.2.2. Function metadata: Annotation parsing

In previous research, we have proposed a programming model for data-intensive serverless edge computing applications [6] that allows developers to express operational requirements directly in their code, which is then translated into scheduler constraints. Examples include: execution deadlines, hardware requirements, or privacy rules. Listing 2 shows the example function from Listing 1 re-written with this high-level API.

```python
from skippy.data import DataArtifact, ModelArtifact,
    consumes, produces, policy
# ... import ML libraries such as tensorflow or mxnet

# can have multiple data annotations
@consumes.data(urns = 'my_bucket:train_data')
@produces.model(urn = 'my_bucket:model')
@policy.fn(capability = 'gpu')
def handle(req, data: DataArtifact) -> ModelArtifact:
    arr = data.to_ndarray()
    model = train_model(arr, req['train_params'])
    return model
```

Listing 2: Example of training function with metadata annotations.

By analyzing the function metadata, Skippy would label this function with the labels shown in Listing 3. These are then used as input for the priority functions described in Section 4.3.1. Metadata do not necessarily have to be specified in the code, but could be attached as, e.g., an additional YAML deployment file.

```
{
  'data.skippy.io/recv': ['my_bucket:train_data'],
  'data.skippy.io/send': ['my_bucket:model'],
  'capability.skippy.io/gpu': ''
}
```

Listing 3: Example function labels resulting from metadata parsing.

## 4.3. Skippy scheduler

The Skippy scheduler enables serveless platforms to schedule *edge functions* more efficiently. It is based on the default Kubernetes MCDM scheduling logic described in Section 3.4.1. We introduce two additional components that are commonly missing in state-of-the-art container schedulers: (1) a static bandwidth graph of the network holding the theoretical (or estimated) bandwidth between nodes, and (2) a storage index that maps data item identifiers to the storage nodes that hold the data. These two extra components facilitate our priority functions.

### 4.3.1. Edge-friendly priority functions

We introduce four priority functions that target requirements of edge computing applications and characteristics of edge systems, which complement common scheduling constraints found in, e.g., Kubernetes [46]. The additional functions are motivated by the following observations: First, in many data-intensive edge computing applications, data is stored at edge locations. Yet, container clusters typically rely on centralized cloud-based repositories such as Dockerhub for managing container images. When scheduling pods that operate on data, there is therefore an inherent trade-off between sending computation to the edge or sending data to the cloud, as we have highlighted in Section 3 and Fig. 1. The two priority functions *LatencyAwareImageLocalityPriority* and *DataLocalityPriority* help the scheduler make this trade-off at runtime. Second, the increasing diversity of specialized compute platforms for edge computing hardware provide new opportunities for accelerating the equally diverse workloads. The *CapabilityPriority* matches tasks and nodes based on their requirements and advertised capabilities, respectively. Third, it is often the case that functions should prioritize execution at the edge for a variety of reasons. The *LocalityTypePriority* enables the system to respect these placement preferences.

We explain each function in more detail and provide algorithmic descriptions. Note that the Kubernetes scheduler expected normalized values from priority functions, which are the result of mapping the range of scores to an integer range [0..10]. We omit the code for this step.

*LatencyAwareImageLocalityPriority.* Favors nodes where the necessary container image can be deployed more quickly. We use knowledge about the network topology to estimate how long it will take in an ideal case to download the image. Algorithm 1 shows pseudocode for the function. Because the bandwidth graph is static and does not consider actual available bandwidth during runtime, the calculation is only an approximation. Making a plausible estimate of actual network download speed would be too complicated for a priority function, which has minimal runtime knowledge and needs to execute quickly. However, together with the implementation of the DataLocalityPriority, the function allows us make a heuristic trade-off between fetching the container image, or fetching data from a data store.

*DataLocalityPriority.* Estimates how much data the function will transfer, and favors nodes where the data transfer happens more quickly. We leverage the high-level data API we have described in [6], to label functions that perform read or write operations on the data stores. Specifically, a function is labeled with the data item identifiers it reads or writes. We query the storage index to get all storage nodes that hold the specific data item. The data size can be queried through the MinIO S3 API. We then make the same network transfer time estimations as in LatencyAwareImageLocalityPriority using our bandwidth graph. Algorithm 2 shows pseudocode for the function.

**Algorithm 1:** LatencyAwareImageLocalityPriority

```
Result: Estimation of how long it will take to download a pod's images
1 Function score:
      Input: pod
      Input: node
      Input: bandwidthGraph
2     size ← 0;
3     for container in pod's list of containers do
4         if container's image is not present on node then
5             size $\overset{+}{\leftarrow}$ size of the container's image;
6         end
7     end
8     bandwidth ← bandwidthGraph[registry][node];
9     time ← $\frac{size}{bandwidth}$;
10    return time;
```

**Algorithm 2:** DataLocalityPriority

```
Result: Estimate how long it takes for a node to transfer the required runtime data
1 Function score:
      Input: pod
      Input: node
      Input: storageIndex, bandwidthGraph
2     time ← 0;
3     for urn in values of 'data.skippy.io/recv' do
4         storages ← storageIndex[urn];
5         bandwidth ← $\min_{s \in storages}$(bandwidthGraph[s][node]);
6         size ← of data item urn;
7         time $\overset{+}{\leftarrow}$ $\frac{size}{bandwidth}$;
8     end
9     for urn in values of 'data.skippy.io/send' do
10        ... analogous to 'recv'
11    end
12    return time;
```

**Algorithm 3:** CapabilityPriority

```
Result: Scores how many of a pod's requested capabilities are provided by a node
1 Function score:
      Input: pod
      Input: node
2     nodeCapabilities ← get all of node's labels starting with capability.skippy.io;
3     podCapabilities ← get all of pod's labels starting with capability.skippy.io;
4     score ← 0;
5     for podCapability in podCapabilities do
6         if nodeCapabilities contains podCapability ∧ values are equal then
7             score $\overset{+}{\leftarrow}$ 1
8         end
9     end
10    return score;
```

*CapabilityPriority.* Checks the compute platform requirements of a function, and favors nodes that have those capabilities (e.g., a GPU for a ML training function). The implementation uses node and function metadata gathered by the Skippy daemon and function annotation parsing. Algorithm 3 shows pseudocode for the function.

*LocalityTypePriority.* Favors nodes in a certain locality, e.g., nodes located at edge or in the cloud. Through the programming model we have described, developers can specify a high-level placement prioritization, e.g., for preferring nodes in a certain network context. It checks the presence of the same values of *locality.skippy.io/type* in pod and node labels. We omit the code for this function.

### 4.4. Integration with OpenFaaS

To enable the deployment of applications as serverless functions, our prototype makes use of OpenFaaS. It provides a framework for defining function deployments, an API gateway through which all function requests are routed, and several runtime components to manage monitoring, alerting, and autoscaling. OpenFaaS' runtime driver for Kubernetes is faas-netes, which deploys functions as Kubernetes pods, and then delegates scheduling decisions to the Kubernetes scheduler. We modified faas-netes to label pods resulting from OpenFaaS function deployments, to indicate that these pods should be scheduled by Skippy instead of the default Kubernetes scheduler. Otherwise Skippy integrates with OpenFaaS only via Kubernetes, in that Skippy schedules the pods created by faas-netes.

### 4.5. Serverless simulator

Part of our system is a discrete event simulator built with SimPy [47] to simulate the basic behavior of serverless function execution on container systems. It serves two purposes: (1) it allows experiments in different large-scale scenarios that we could not perform on our small-scale testbeds, and (2) it is used to estimate goal functions in our optimization technique described in Section 5. The simulator directly calls the Skippy scheduler code for scheduling functions, with the only difference that it does not call the Kubernetes API for requesting the cluster state and performing node bindings. The simulator is open source and can be found in our Git repositories [48].

The simulator uses *Ether* [38] to generate network topologies, and adds features for synthesizing function parameters, and generating random workload. We simulate the execution of functions on cluster nodes using the profiling data we have gathered from our testbed and a basic network simulation. Our network model is more high-level than packet-level simulators such as ns-3 or OPNET. Simulating data transfer involves opening a *flow* through several connected network *links*, i.e., a route. Each link has a certain amount of bandwidth, and we implement fair allocation of bandwidth across flows. We plan to add features for degradation functions to simulate the degrading TCP behavior with many flows [49]. For simulating container startups, we synthesize profiling data and our network simulation. A perfect simulation of a Docker pull command would consider the layers of an image, the availability of layers on a host, and the time it takes to decompress layers. This is out of scope for this paper. We make an assumption based on observations of our images we described in Section 3: around 90% of an image's size comes from layers that are shared with other images. Meaning that, if any one of the images has already been pulled before, only 10% of another image's unique data has to be pulled. For our evaluation this means that we are not biasing the simulation towards the estimation that the Skippy scheduler makes through the LatencyAwareImageLocalityPriority. Our simulator also implements the basic autoscaling behavior of OpenFaaS. In particular, it includes OpenFaaS' *faas-idler* component that enacts the scale-to-zero policy: when a function is idle for 5 min or more, the respective function replica is stopped and the underlying Kubernetes pod removed. A subsequent call to the function incurs a cold start. Our simulator currently only supports simulating platforms that deploy function code via containers, whereas some platforms like OpenWhisk deploy function code through platform-layer mechanisms.

## 5. Optimizing weights of priority functions

Some operational goals, such as minimizing overall function execution time or uplink usage, depend on too many (and possibly at runtime unknowable) factors that they could be calculated efficiently in priority functions. Schedulers that employ MCDM, such as the Kubernetes scheduler, often allow users to assign weights to each constraint to tune the scheduler towards certain behavior. This fine-tuning to meet specific operational goals can be difficult. Many factors need to be considered, such as the cluster topology, node heterogeneity, or workload characteristics. This leaves operators to either rely on their intuition, or use trial-and-error in production systems, to find weights that achieve the desired behavior.

We propose an approach to automatically find weights of priority functions that result in good placements that meet certain high-level operational goals. We consider a placement to be good if it: (1) leads to **low function execution time** during runtime, (2) **uses edge resource** efficiently, (3) **reduces traffic** leaving or entering a network, and (4) **reduces costs** associated with executing functions in the cloud. To that end, we use multi-objective optimization techniques, and use the simulator we have developed to evaluate the goodness of optimization solutions. We first formalize some key aspects.

### 5.1. Problem formulation

Let $\mathcal{S}$ be the set of priority functions $S \in \mathcal{S} : P \times N \rightarrow \mathbb{R}$ where $P$ is the domain of pods and $N$ is the domain of nodes (and all metadata attached to them). The function *schedule* : $P \rightarrow N$ maps a pod $p$ to a node $n$ by evaluating the scoring function *score* for each node, and selecting the highest scoring node. The scoring function is essentially a weighted sum model over all priority functions and feasible nodes. Formally, this can be expressed as

$$schedule(p) = \arg\max_{n \in N} score(p, n) : \sum_{i=0}^{|S|} w_i \cdot S_i(p, n). \quad (1)$$

The default Kubernetes scheduler sets every $w_i = 1$. Our goal is to find values for $\mathbf{w} = (w_1\ w_2\ \cdots\ w_{|\mathcal{S}|})$ that optimize towards the previously defined objectives.

We have explained the technical details of the simulator in Section 4.5, but formally a simulation run $sim(T, W, \mathbf{w})$ takes as input (1) the cluster topology $T$, (2) a workload profile $W$, (3) the vector of priority function weights $\mathbf{w}$, and simulates the function execution based on the profiling data we have gathered. The cluster topology $T$ is formally a graph $T = (V, E)$, $V = N \cup L$, where $N$ is the set of cluster nodes, $L$ is the set of links that have an associated bandwidth (e.g., several nodes can be connected to a common WiFi link), and $E$ are weighted edges that indicate the latency between nodes and links. A workload profile $W$ assigns each function (in our case, the ML workflow functions), an inter-arrival distribution, from which we sample at simulation time to generate workload. Our four goal functions $f_i(sim(T, W, \mathbf{w}))$ are calculated from the simulation traces as follows:

- $f_1$ : average function execution time over all functions
- $f_2$ : up/downlink usage, i.e., the number of bytes transferred between edge and cloud networks
- $f_3$ : edge resource utilization, i.e., the percentage of allocated resources on edge compared to cloud nodes
- $f_4$ : cloud execution costs, i.e. traffic and function execution time in the cloud, given a pricing model

We now want to find $\mathbf{w}$ s.t. $f_1, f_2, f_4$ are minimized, and $f_3$ is maximized.

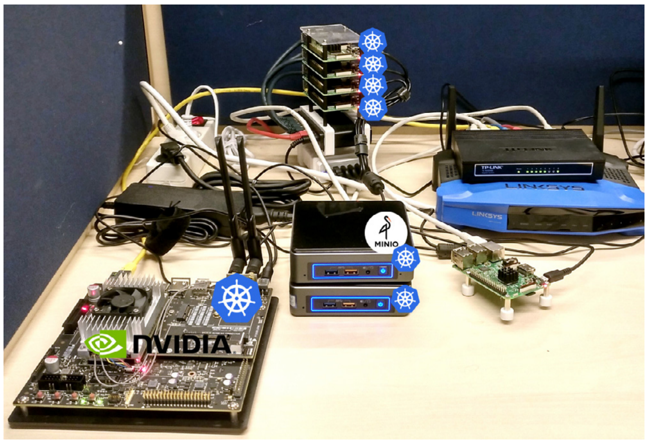

**Fig. 5.** Our edge cloud testbed comprising a Raspberry Pi cluster, an NVIDIA Jetson TX2 board, and two Intel NUCs, one acting as edge storage node by hosting a MinIO pod. A VM hosted on our on-premises cloudlet is also part of the cluster.

### 5.2. Implementation

We implement the optimization using our simulator and the Python Platypus [50] framework for multi-objective optimization. Platypus implements the well-known NSGA-II genetic algorithm [51], which has been found to be one of the best performing algorithms in the framework [52].

To find an optimized value of $\mathbf{w}$, we execute the Platypus framework's NSGA-II implementation with 10 000 generations. Each generation executes a single simulation run $sim(T, W, \mathbf{w})$ with a predefined $W$ and $T$, and the current evolution of $\mathbf{w}$. A run creates function deployments according to $W$ until the cluster is fully utilized, using our scheduler for placement decisions. We store execution traces into Pandas data frames, and then calculate the goal functions $f_i$ from the traces. The result is a set of 100 solutions that are at the Pareto frontier of the solution space. As input for the scheduler, we select from that set a single solution $\mathbf{w}$ that is balanced across all goals.

## 6. Evaluation

This section presents our experiment setup, results, and a discussion of limitations. We first present the testbed we have built that we used to test our prototype implementation, and generate traces for the simulator. The scenarios we have defined in Section 3, and the traces generated from our testbed, are then used as input for our serverless simulator. We investigate how the scheduling decisions and parameter optimization impact application and system performance. We discuss the scheduler's performance in terms of scheduling throughput and latency, and, finally, discuss the current limitation of our system.

### 6.1. Edge cloud testbed & profiling

The testbed we have built comprises several edge computing devices listed in Section 3.2. Fig. 5 shows the current setup. The nodes marked with a Kubernetes logo are part of the Kubernetes cluster used as runtime for OpenFaaS. The OpenFaaS gateway and Kubernetes master are hosted on a VM in our on-premises cloudlet.

We run the application we have described in Section 3.1 on the testbed using our system prototype. That is, we implement each task as an OpenFaaS function, and execute each task on each device in both cold and warm state using different bandwidth

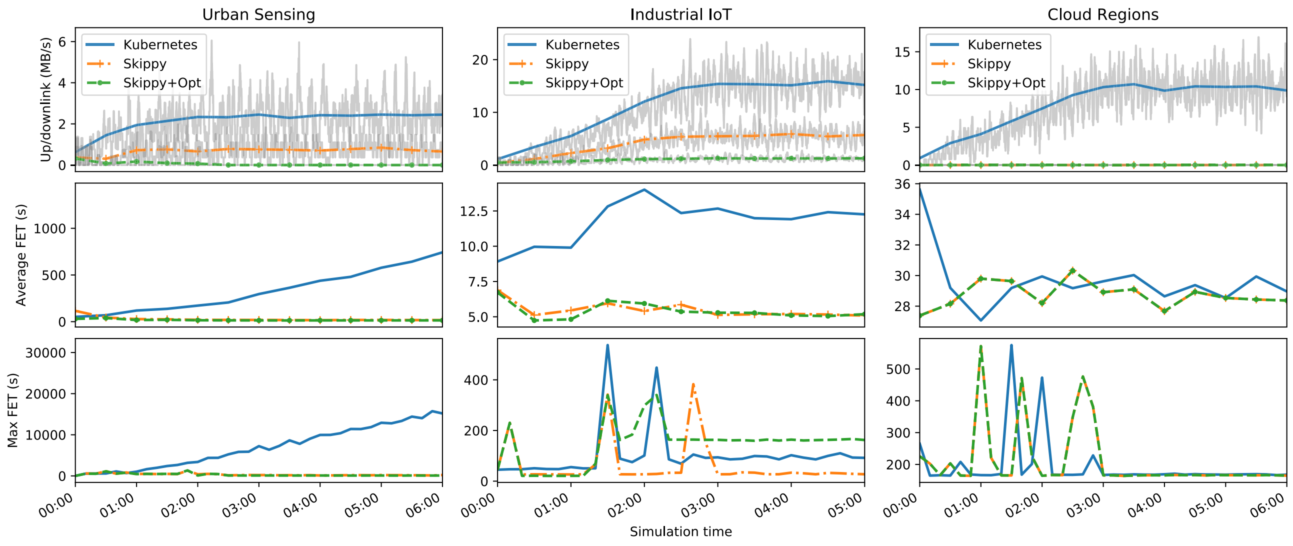

**Fig. 6.** Drill-down into timeseries data from simulated experiment runs. The first row shows the average data rate of traffic going over up/downlinks. The second row shows the average function execution time (FET) over time (10 min rolling window). The third row shows the maximum function execution time (FET) over time (10 min rolling window).

and latency configurations. The functions are implemented in Python and use the Apache MXNet machine learning framework. We measure various system and application metrics, such as the system resource utilization, task execution time, the bandwidth requirements, e.g., when the container image that holds the function has to be downloaded, as well as the traffic produced by function invocations.

### 6.2. *Experiment setup*

We perform the experiments with parameters drawn from the infrastructure scenarios described in Section 3.3, and compare three scheduler implementations: the default Kubernetes scheduler as baseline, Skippy with weights set to 1, and Skippy with optimized weights. The experiment process is as follows. We generate random application deployments, in our case ML workflow pipelines that comprise three ML functions, inject them into the scheduler queue, generate random requests given some workload profile, and then run the simulation for a certain number of invocations. Specifically, we deploy a new pipeline every few minutes and start generating requests to those functions a few seconds afterwards. After a specified number of function instances have been deployed, we generate another several thousand requests, until a request limit for that scenario has been reached. For example, in Scenario 2, each experiment ends after 30 000 invocations. Having the same amount of deployments and function invocations allows for a fair comparison of overall network traffic.

For synthesizing pipelines and requests we make the following assumptions. Each pipeline has three steps, where each step as an individual container image. However, as we have discussed in Section 3, we consider the commonalities across images. We synthesize both pipeline instances (i.e., functions deployed in Kubernetes pods) as well as container images, and assume a Pareto distribution of images. That is, not every function has a unique image. Instead we assume a Pareto-distributed relation of container images to pod instances, i.e., 80 percent of pods use the same 20 percent of images. For the workload profile $W$, we assume a typical [53] Poisson arrival process where inter-arrivals are described by an exponential distribution. We set distribution parameters s.t. model serving requests of an individual pipeline are triggered at 40 requests per second, and data-preprocessing requests happen every few minutes allowing the faas-idler to occasionally shut down a replica. For synthesizing data items (e.g., training data as input for training functions), we assume that data items are distributed uniformly across data stores and workflows.

Experiment runs that compare different scenarios and schedulers use the same random seed for distribution sampling to guarantee comparability between scenario runs.

### 6.3. *Experiment results*

This section presents the results of our experiments. The results show (1) how function placement affects system performance, (2) how function placement affects system scalability, and (3) which priority functions have the highest impact on optimization goals.

#### 6.3.1. *Runtime performance of placements*

Fig. 6 shows key performance indicators from simulation runs in each scenario for the schedulers: the default Kubernetes scheduler, the Skippy scheduler, and Skippy using optimized priority function weights. The first row shows the average data rate going over up and downlinks. Ideally, a placement keeps traffic within networks, resulting in a low up/downlink usage. As the deployments are injected in the first phase of the simulation, the data rate grows, but is overall significantly higher with the Kubernetes scheduler. The Cloud Regions scenario (S3) highlights the problem when there are many nodes within a network, and few up/downlinks between them. The second and third row show the function execution duration over time. In the Urban Sensing scenario (S1), the Kubernetes scheduler's placements run into queuing issues early on. Function time keeps increasing because the network cannot keep up transferring data necessary by the function executions. In the Industrial IoT scenario (S2), while there are no queuing issues, the Kubernetes scheduler's placements lead to overall higher function execution times. There is no significant difference in the Cloud Regions scenario, because the devices within the cluster (cloud VMs) are fairly homogeneous in terms of task execution performance. Overall, the second row shows the interplay between using resources effectively, and trading-off data movement costs.

Fig. 7 shows the aggregated results from several runs with different random seeds for the other two performance goals we have

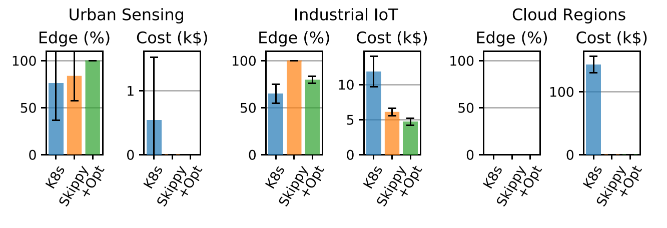

**Fig. 7.** Edge resources utilization and execution cost of placements in three scenarios. Bars show the average across ten runs, error bars show one $\sigma$.

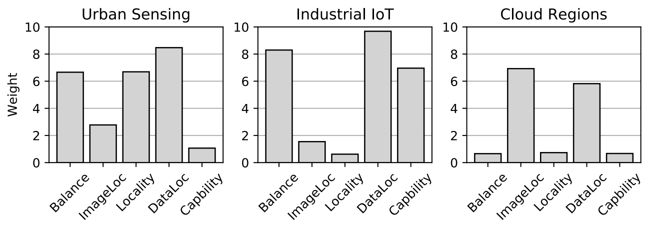

**Fig. 9.** Optimized priority function weights in each scenario.

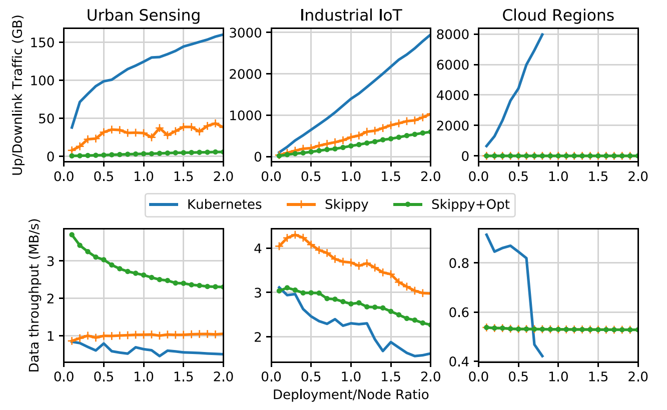

**Fig. 8.** Scalability analysis of placements with increasing number of deployments in each scenario. The first row shows the raw inter-network traffic in GB. The second row shows the data throughput of functions, i.e., the overall network traffic per compute second.

defined: edge resource utilization and execution cost. For calculating the cost we use the pricing model of AWS Lambda [54]. For S1 we observe the effect of a low amount of cloud resources: almost no cloud execution cost and generally high edge resource utilization. Skippy and the optimization perform slightly better. In S2, where data is also placed in on-premises managed cloud instances, we observe that optimized Skippy can make a useful trade-off between cost and edge utilization by preferring cloud resource in favor of moving data. S3 has no edge resources, but we can see that the Kubernetes scheduler's decision to place functions across regions leads to a high cost incurred by data movement. It also illustrates that most of the costs in our scenario comes from data movement (specifically data egress), rather than compute time, corroborating the results of a study about the unintuitive nature of serverless pricing models [55].

### 6.3.2. *Impact of placements on system scalability*

We investigate how function placements affect runtime scalability properties of the system. In ad-hoc experiments we found that network bottlenecks were the biggest challenge for guaranteeing low function execution times and high throughput. In our scenarios in particular, we were not able to saturate cluster resources before running into extreme network congestion (flows receiving less than 0.1 Mb/s link bandwidth). The most important metric of scalability in our scenarios is therefore network throughput, and whether the placement can maintain high data throughput in the face of an increasing amount of active deployments. To examine this, we run experiments that inject an increasingly larger number of deployments per node. As mentioned earlier, a deployment in our scenario is an instance of one ML pipeline with its three functions. We start at a ratio of 0.1 deployments per node up to 2 deployments per node. Fig. 8 shows the results of experiment runs without the scale-to-zero policy.

Two things in particular are noteworthy. First, in S2, while the optimized Skippy has a lower data throughput than Skippy, as we have seen in Fig. 7, it does this to trade off execution cost while maintaining similar function execution times (see Fig. 6). Second, in some situations, the Kubernetes scheduler produced infeasible placements even with very few deployments. In particular in S3, the inter-region bandwidth was quickly saturated and leading to infeasible placements. We consider a placement infeasible if it, during the course of a simulation, leads to bottlenecks in the network that degrade the bandwidth allocated to a flow to less than 0.1 Mb/s. Another finding was that, if the OpenFaaS scale-to-zero policy was used, the default scheduler produced no feasible placements in the first scenario. Functions would be rescheduled s.t. the network was quickly congested with inter-network traffic.

### 6.3.3. *Optimized priority function weights*

Fig. 9 shows the values of **w** assigned by the optimization as described in Section 5.2, i.e., the optimized weight of each priority function in the different evaluation scenarios. In S1, the capability priority is less relevant, as there is a high percentage of GPU nodes available, which are not saturated. Locality plays a much bigger role in avoiding using the scarce cloud resources. In S2, because there are few GPU nodes, and data is also distributed to on-premises cloud, the data locality and capability priorities are favored. In S3, the results confirm the intuition that resource balance, locality, and capabilities do not have much weight for scheduling in relatively homogeneous environments.

## 6.4. *Scheduling latency and throughput*

The main source of latency in greedy online MCDM schedulers comes from iterating over nodes and computing priority functions. Because Skippy requires a significant number of priority functions compared to the default Kubernetes scheduler, we think it is worth discussing the resulting impact on scheduling latency and throughput. Let $N$ be the set of all nodes in the cluster, $N^c$ be the set of feasible nodes for scheduling container $c$, and $\mathcal{S}$ be the set of priority functions. Scheduling requires the evaluation of every priority function $S \in \mathcal{S}$ for every feasible node $n \in N^c$. The algorithmic complexity of scheduling one container $c$ therefore depends on the complexity of the individual priority functions. If we neglect this, i.e., assume that invoking any $S$ is $O(1)$, the complexity of the scoring step is $O(|N^c| \cdot |\mathcal{S}|)$, where $N^c = N$ in the worst case. Because $|N|$ can reach several thousands in a production cluster, the Kubernetes scheduler employs a sampling heuristic to reduce $|N^c|$. The percentage of nodes that are sampled, progressively decreases with the number of nodes in the cluster. Once the cluster reaches $|N| \geq 6500$, the scheduler only considers 5% of available nodes for scoring. This heuristic works under the assumption that the cluster and the network is relatively homogeneous, and that aggressive sampling will not significantly impair placement quality. However, in the case of edge infrastructure, where these assumptions may not hold, this heuristic would introduce extreme variance in the placement quality, which is why we disable it and have to consider all nodes in the cluster. This leads to a general degradation in scheduling

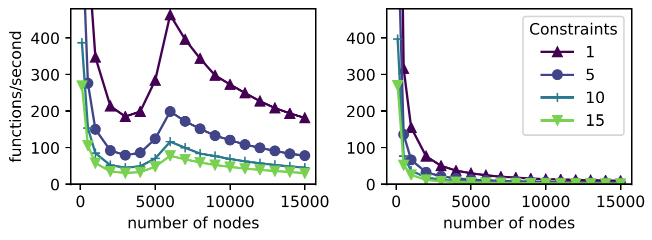

**Fig. 10.** Scheduler throughput in functions/second with sampling heuristic (left), and without (right).

throughput. We measured the throughput given different cluster sizes and number of priority functions. Our results in Fig. 10 roughly match those of a recent Kubernetes performance evaluation [56]. The default Kubernetes scheduler only uses two priority functions and the sampling heuristic, which allows it to process around 170 pods per second in a cluster of 10 000 nodes. Whereas Skippy uses by default five priority functions and scores all nodes, which, at 10 000 nodes, yields a throughput of around 15 pods per second. While in our scenarios this is not an issue because scheduling latency is only a small fraction of the overall round-trip time, it does negatively affect scheduling throughput. We are investigating alternative disaggregated scheduler architectures, such ones employed by Omega [57] or Firmament [29].

### 6.5. Challenges & limitations

Beyond scheduling performance, our system has several limitations that need to be discussed. We also identify several open challenges for serverless edge platforms.

Our system currently makes no particular considerations of the dynamic nature of edge systems. Reconciling deployment and runtime aspects of serverless edge computing applications is especially challenging. Generally, we can distinguish function deployment, function scaling, and function requests to already running functions. Finding good placements for long-living functions is challenging, especially when they are network bound. For functions such as those serving static content or simple image classification tasks, the request RTT perceived by clients will be dominated by link latency between the client and the node hosting the function. Therefore, to make better placement decision for such functions, the system would require knowledge about the location of clients with respect to nodes in the cluster [27]. In this case, an autoscaling strategy could, for example, spin up replicas that favor nodes in close proximity to clients. This falls into the category of dynamic service migration problems [58], and is a challenge that confronts edge computing systems in general. The centralized API gateway architecture of OpenFaaS, Kubernetes, and similar systems, presents a serious obstacle in solving this issue, as generally all traffic goes through a type of *ingress* to allow dynamic request routing and load balancing. A strategy could be to replicate API gateways across the network and using a localization mechanism to resolve an API gateway in proximity. This may not be fine-grained enough for scenarios such as the urban sensing infrastructure where the resolution would have to be on city neighborhood level. A solution or detailed analysis of this issue is out of scope of the paper.

Another issues of using state-of-the-art serverless platforms for edge infrastructure is the rudimentary way they model node resources and function requirements [13]. For example, in Kubernetes, a node has three capacities: CPU, memory and the maximum number of pods that can be scheduled onto the node. Modeling capabilities of edge resources is challenging, as their availability may not be known at design time, and whether they are shareable at runtime. This is particularly important for scarce, (potentially) non-shareable and discrete resources such as GPUs, where containers that use the resource may completely block other containers from execution, while not requiring them often. We therefore see resource modeling as an important aspect of future edge computing platforms.

Using container-based systems can have several drawbacks with respect to isolation and multitenancy. It is currently unclear how our system would behave in a multi-tenant scenario, where cluster resources are shared between multiple runtimes. Further research is necessary to investigate the effect of, e.g., workload interference.

Our system currently makes the assumption that function code is distributed in container images. Some FaaS platforms, such as OpenWhisk, have platform-level facilities for distributing function code, that may not benefit from the computation movement estimation made by the *LatencyAwareImageLocalityPriority*. Although we could conceive a more higher-level abstraction for a *code movement* soft-constraint, it would require additional facilities to allow the scheduler to query the runtime for function metadata (like it's code size), and whether a function's code has been deployed at a particular node.

## 7. Conclusion

Serverless computing helps platform providers to hide operational complexity from application developers, making it particularly attractive for edge computing systems. Analogously to serverless *cloud functions*, we believe that *edge functions* are a promising approach to manage applications that run on a distributed edge compute fabric. We have demonstrated several limitations of existing serverless platforms when they are used in such scenarios, leading to poor function placement on heterogeneous geo-distributed infrastructure that has limited up/downlink connections between edge networks.

We presented Skippy, a container scheduling system that enables existing container orchestrators, such as Kubernetes, to support serverless *edge functions*. Skippy does this by introducing scheduling constraints that leverage additional knowledge of node capabilities, the application's data flow, and the network topology. Overall our experiments show that (1) Skippy enables locality-sensitive function placement, resulting in higher data throughput and less traffic going over up/downlinks, (2) in scenarios where there is a fairly even distribution of cloud and edge resources, the optimization helps significantly in trading off execution cost and overall application latency, and (3) the improved placement quality comes at the cost of scheduler performance. We have shown that the most critical aspect of function placement in data-intensive serverless edge computing is the trade-off between data and computation movement. However, making this trade-off in a generalized way is challenging due to the wide range of edge infrastructure scenarios. By introducing higher-level operational goals, we can fine-tune the underlying scheduler parameters to consider infrastructure-specific aspects.

There are several open issues to fully realize the idea of edge functions on a distributed compute fabric. For example, the centralized API gateway architecture employed by most state-of-the-art serverless platforms may be impractical for edge computing, particularly with dispersed clients that consume network-bound functions. Moreover, the dynamic nature of edge systems requires the continuous re-evaluation of placement decisions, necessitating context-aware autoscaling and workload migration strategies. Finally, the automatic characterization of workloads and mapping to their preferred node capabilities could significantly improve function placement.

## CRediT authorship contribution statement

**Thomas Rausch:** Conceptualization, Methodology, Software, Validation, Formal analysis, Data curation, Writing - original draft, Writing - review & editing. **Alexander Rashed:** Conceptualization, Software, Validation, Formal analysis, Writing - original draft. **Schahram Dustdar:** Conceptualization, Supervision, Funding acquisition.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

This work was supported by TU Wien research funds, Austria, and the Austrian infrastructure program (HRSM 2016) as part of the CPS/IoT Ecosystem project.

## References

[1] W. Shi, S. Dustdar, The promise of edge computing, Computer 49 (5) (2016) 78–81.
[2] T. Rausch, S. Dustdar, Edge intelligence: The convergence of humans, things, and AI, in: 2019 IEEE International Conference on Cloud Engineering, in: IC2E '19, 2019.
[3] J. Wang, Z. Feng, S. George, R. Iyengar, P. Pillai, M. Satyanarayanan, Towards scalable edge-native applications, in: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, in: SEC '19, 2019, pp. 152–165.
[4] C.E. Catlett, P.H. Beckman, R. Sankaran, K.K. Galvin, Array of things: a scientific research instrument in the public way: platform design and early lessons learned, in: Proceedings of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering, 2017, pp. 26–33.
[5] M. Satyanarayanan, W. Gao, B. Lucia, The computing landscape of the 21st century, in: Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications, 2019, pp. 45–50.
[6] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, S. Dustdar, Towards a serverless platform for edge AI, in: 2nd USENIX Workshop on Hot Topics in Edge Computing, in: HotEdge '19, 2019.
[7] A. Glikson, S. Nastic, S. Dustdar, Deviceless edge computing: Extending serverless computing to the edge of the network, in: Proceedings of the 10th ACM International Systems and Storage Conference, in: SYSTOR '17, 2017.
[8] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, R. Prodan, A serverless real-time data analytics platform for edge computing, IEEE Internet Comput. 21 (4) (2017) 64–71.
[9] L. Baresi, D.F. Mendonça, Towards a serverless platform for edge computing, in: 2019 IEEE International Conference on Fog Computing, in: ICFC '19, 2019, pp. 1–10.
[10] D. Bermbach, S. Maghsudi, J. Hasenburg, T. Pfandzelter, Towards auction-based function placement in serverless fog platforms, 2019, arXiv preprint arXiv:1912.06096.
[11] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, Q. Li, Lavea: Latency-aware video analytics on edge computing platform, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, in: SEC '17, 2017, pp. 1–13.
[12] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al., Cloud programming simplified: a berkeley view on serverless computing, 2019, arXiv preprint arXiv:1902.03383.
[13] J.M. Hellerstein, J. Faleiro, J.E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu, Serverless computing: One step forward, two steps back, 2018, arXiv preprint arXiv:1812.03651.
[14] M. Satyanarayanan, V. Bahl, R. Caceres, N. Davies, The case for VM-based cloudlets in mobile computing, IEEE Pervasive Comput. (2009).
[15] V. Sreekanti, C.W.X.C. Lin, J.M. Faleiro, J.E. Gonzalez, J.M. Hellerstein, A. Tumanov, Cloudburst: Stateful functions-as-a-service, 2020, arXiv preprint arXiv:2001.04592.
[16] V. Ishakian, V. Muthusamy, A. Slominski, Serving deep learning models in a serverless platform, in: 2018 IEEE International Conference on Cloud Engineering, in: IC2E '18, 2018, pp. 257–262.
[17] A. Rashed, T. Rausch, Execution traces of an MNIST workflow on a serverless edge testbed, 2020, http://dx.doi.org/10.5281/zenodo.3628454.
[18] AWS IoT greengrass, 2020, https://aws.amazon.com/greengrass/, AWS. Online. Accessed 2020-06-11.
[19] I.E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, V. Hilt, SAND: Towards high-performance serverless computing, in: 2018 USENIX Annual Technical Conference, in: USENIX ATC '18, 2018, pp. 923–935.
[20] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, R. Arpaci-Dusseau, SOCK: Rapid task provisioning with serverless-optimized containers, in: 2018 USENIX Annual Technical Conference, in: USENIX ATC '18, USENIX Association, 2018, pp. 57–70.
[21] A. Hall, U. Ramachandran, An execution model for serverless functions at the edge, in: Proceedings of the International Conference on Internet of Things Design and Implementation, in: IoTDI '19, 2019, pp. 225–236.
[22] D. Bermbach, A.-S. Karakaya, S. Buchholz, Using application knowledge to reduce cold starts in faas services, in: Proceedings of the 35th ACM Symposium on Applied Computing, in: SAC '20, ACM, 2020.
[23] Y. Xiong, Y. Sun, L. Xing, Y. Huang, Extend cloud to edge with kubeedge, in: 2018 IEEE/ACM Symposium on Edge Computing, in: SEC '18, 2018, pp. 373–377.
[24] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, P. Leitner, Optimized IoT service placement in the fog, Serv. Orient. Comput. Appl. 11 (4) (2017) 427–443.
[25] A. Yousefpour, A. Patil, G. Ishigaki, I. Kim, X. Wang, H.C. Cankaya, Q. Zhang, W. Xie, J.P. Jue, Qos-aware dynamic fog service provisioning, 2018, arXiv preprint arXiv:1802.00800.
[26] Y. Sahni, J. Cao, L. Yang, Data-aware task allocation for achieving low latency in collaborative edge computing, IEEE Internet Things J. 6 (2) (2018) 3512–3524.
[27] T. He, H. Khamfroush, S. Wang, T. La Porta, S. Stein, It's hard to share: joint service placement and request scheduling in edge clouds with sharable and non-sharable resources, in: 2018 IEEE 38th International Conference on Distributed Computing Systems, in: ICDCS '18, 2018, pp. 365–375.
[28] F. Ait Salaht, F. Desprez, A. Lebre, An overview of service placement problem in Fog and Edge Computing, Research report RR-9295, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, LYON, France, 2019, pp. 1–43.
[29] I. Gog, M. Schwarzkopf, A. Gleave, R.N.M. Watson, S. Hand, Firmament: Fast, centralized cluster scheduling at scale, in: 12th USENIX Symposium on Operating Systems Design and Implementation, in: OSDI '16, USENIX Association, 2016, pp. 99–115.
[30] Y. Hu, H. Zhou, C. de Laat, Z. Zhao, Ecsched: Efficient container scheduling on heterogeneous clusters, in: European Conference on Parallel Processing, 2018, pp. 365–377.
[31] R.G. Aryal, J. Altmann, Dynamic application deployment in federations of clouds and edge resources using a multiobjective optimization AI algorithm, in: 3rd International Conference on Fog and Mobile Edge Computing, in: FMEC '18, 2018.
[32] H. Tan, Z. Han, X.-Y. Li, F.C. Lau, Online job dispatching and scheduling in edge-clouds, in: IEEE Conference on Computer Communications, in: INFOCOM '17, 2017, pp. 1–9.
[33] X.-Q. Pham, E.-N. Huh, Towards task scheduling in a cloud-fog computing system, in: 18th Asia-Pacific Network Operations and Management Symposium, in: APNOMS '16, 2016.
[34] J. Msv, How kubernetes is transforming into a universal scheduler, 2018, URL https://thenewstack.io/how-kubernetes-is-transforming-into-a-universal-scheduler The New Stack. Online. Posted 2018-09-07. Accessed 2019-03-14.
[35] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, et al., An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, in: SEC '17, 2017.
[36] S. Venugopal, M. Gazzetti, Y. Gkoufas, K. Katrinis, Shadow puppets: Cloud-level accurate AI inference at the speed and economy of edge, in: USENIX Workshop on Hot Topics in Edge Computing, in: HotEdge '18, 2018.
[37] W. Hummer, V. Muthusamy, T. Rausch, P. Dube, K. El Maghraoui, Modelops: Cloud-based lifecycle management for reliable and trusted AI, in: 2019 IEEE International Conference on Cloud Engineering, in: IC2E '19, 2019.
[38] T. Rausch, C. Lachner, P.A. Frangoudis, P. Raith, S. Dustdar, Synthesizing plausible infrastructure configurations for evaluating edge computing systems, in: 3rd USENIX Workshop on Hot Topics in Edge Computing, in: HotEdge '20, 2020.
[39] S.J. Vaughan-Nichols, Canonical's cloud-in-a-box: The ubuntu orange box, 2020, URL https://www.zdnet.com/article/canonicals-cloud-in-a-box-the-ubuntu-orange-box/ Online. Posted 2014-05-19. Accessed 2020-06-11.
[40] S.J. Johnston, P.J. Basford, C.S. Perkins, H. Herry, F.P. Tso, D. Pezaros, R.D. Mullins, E. Yoneki, S.J. Cox, J. Singer, Commodity single board computer clusters and their applications, Future Gener. Comput. Syst. 89 (2018) 201–212.
[41] NVIDIA, NVIDIA jetson - the AI platform for autonomous machines, 2020, URL https://developer.nvidia.com/embedded/develop/hardware Online. Accessed 2020-06-11.

[42] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, in: Proceedings of the Tenth European Conference on Computer Systems, 2015, pp. 18.
[43] B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas, Q. Zhang, Edge computing in iot-based manufacturing, IEEE Commun. Mag. 56 (9) (2018) 103–109.
[44] J. Yan, Y. Meng, L. Lu, L. Li, Industrial big data in an industry 4.0 environment: Challenges, schemes, and applications for predictive maintenance, IEEE Access 5 (2017) 23484–23491.
[45] B. Cutler, Examining cross-region communication speeds in AWS, 2018, URL https://medium.com/slalom-technology/examining-cross-region-communication-speeds-in-aws-9a0bee31984f Medium. Online. Posted 2018-05-30. Accessed 2020-02-15.
[46] K. Community, Kubernetes scheduler, kubernetes documentation, 2019, URL https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/ Accessed 2019-01-04.
[47] N. Matloff, Introduction to discrete-event simulation and the simpy language, Vol. 2, Dept of Computer Science. University of California at Davis, Davis, CA, 2009, pp. 1–33.
[48] Faas-sim: simulating serverless function execution on clusters, 2020, https://github.com/edgerun/faas-sim Online. Accessed 2020-06-11.
[49] R. Morris, TCP behavior with many flows, in: Proceedings 1997 International Conference on Network Protocols, 1997, pp. 205–211.
[50] D. Hadka, Platypus: A free and open source python library for multiobjective optimization, 2017, URL https://github.com/Project-Platypus/Platypus Online. Accessed 2020-06-11.
[51] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.
[52] D. Brockhoff, T. Tušar, Benchmarking Algorithms from the platypus Framework on the Biobjective bbob-biobj Testbed, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2019, pp. 1905–1911.
[53] A. Feldmann, Characteristics of TCP connection arrivals, in: Self-Similar Network Traffic and Performance Evaluation, 2000, pp. 367–399.
[54] Amazon, AWS lambda pricing, 2020, URL https://aws.amazon.com/lambda/pricing/ AWS. Online. Accessed 2020-02-19.
[55] A. Eivy, Be wary of the economics of serverless cloud computing, IEEE Cloud Comput. 4 (2) (2017) 6–12.
[56] H. Deng, Improving kubernetes scheduler performance, coreos blog, 2016, URL https://coreos.com/blog/improving-kubernetes-scheduler-performance.html Online. Posted 2016-02-22. Accessed 2019-03-14.
[57] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, J. Wilkes, Omega: flexible, scalable schedulers for large compute clusters, in: SIGOPS European Conference on Computer Systems, in: EuroSys '13, 2013, pp. 351–364.
[58] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. Chan, K.K. Leung, Dynamic service migration and workload scheduling in edge-clouds, Perform. Eval. 91 (2015) 205–228.

**Thomas Rausch** is a research assistant and Ph.D. candidate at the Distributed Systems Group of TU Wien, Vienna, Austria. He received his M.S. degree in Software Engineering & Internet Computing from TU Wien in 2016. His research interests include edge computing systems, with an emphasis on operations research, resource scheduling, and Internet of Things.

**Alexander Rashed** received his B.S. degree in Software and Information Engineering from TU Wien in 2016. He received his M.S. degree in Software Engineering & Internet Computing from TU Wien in 2020.

**Schahram Dustdar** is a full professor of computer science, and he heads the Distributed Systems Group at TU Wien. His work focuses on distributed systems. Dustdar is an IEEE Fellow, a member of the Academia Europaea, an ACM Distinguished Scientist, and recipient of the IBM Faculty Award. He is on the editorial boards of IEEE Internet Computing and Computer. He is an associate editor of IEEE Transactions on Services Computing, ACM Transactions on the Web, and ACM Transactions on Internet Technology. He is the editor-in-chief of Springer Computing.