# faasHouse: Sustainable Serverless Edge Computing through Energy-aware Resource Scheduling

Mohammad Sadegh Aslanpour, *Student Member*, *IEEE*, Adel N. Toosi, *Member*, *IEEE*, Muhammad Aamir Cheema, *Senior Member*, *IEEE*, Mohan Baruwal Chhetri, *Member*, *IEEE*

**Abstract**—Serverless edge computing is a specialized system design tailored for Internet of Things (IoT) applications. It leverages serverless computing to minimize operational management and enhance resource efficiency, and utilizes the concept of edge computing to allow code execution near the data sources. However, edge devices powered by renewable energy face challenges due to energy input variability, resulting in imbalances in their operational availability. As a result, high-powered nodes may waste excess energy, while low-powered nodes may frequently experience unavailability, impacting system sustainability. Addressing this issue requires energy-aware resource schedulers, but existing cloud-native serverless frameworks are energy-agnostic. To overcome this, we propose an energy-aware scheduler for sustainable serverless edge systems. We introduce a reference architecture for such systems and formally model energy-aware resource scheduling, treating the function-to-node assignment as an imbalanced energy-minimizing assignment problem. We then design an optimal offline algorithm and propose faasHouse, an online energy-aware scheduling algorithm that utilizes resource sharing through computation offloading. Lastly, we evaluate faasHouse against benchmark algorithms using real-world renewable energy traces and a practical cluster of single-board computers managed by Kubernetes. Our experimental results demonstrate significant improvements in balanced operational availability (by 46%) and throughput (by 44%) compared to the Kubernetes scheduler.

**Index Terms**—edge computing, serverless, function-as-a-service, energy awareness, scheduling, sustainability

## 1 Introduction

Serverless edge computing [10], [11] is an emerging technological innovation that combines the flexibility and scalability of serverless computing with the proximity and low latency of edge devices. This approach has the potential to revolutionize edge computing and drive real-time Internet of Things (IoT) applications. By distributing computational workloads to the edge of the network, serverless edge computing enables real-time processing, reduces data transfer costs, and enhances user experience. It effectively bridges the gap between cloud computing and localized processing, empowering developers to leverage the advantages of both.

Serverless computing, implemented through Function-as-a-Service (FaaS), is a paradigm initially designed for cloud computing to relieve developers from the burdens of managing resources and backend components [9]. Similarly, serverless at the edge extends this concept to IoT software development, allowing developers to focus on the core business logic of their applications instead of complex operational practices [6]. By leveraging serverless at the edge, developers can write independent stateless functions using high-level programming languages. The scalability and portability offered by serverless computing enable applications to dynamically scale functions based on real-time demand [9]. This flexibility optimizes resource utilization and efficiently accommodates varying workloads. The statelessness and lightweight virtualization used to deploy serverless functions enable efficient execution, making it conceivable to have an application running with as little as 5 MB of memory.

Extreme edge computing refers to the practice of performing computing tasks and data processing at the extreme edge of a network often directly on the devices or sensors themselves. Extreme edge network configuration is characterized by a cluster of small-scale computers called edge nodes, interconnected with a central, more powerful computer functioning as the gateway at the network's edge [1]. Extreme edge computing finds applications in various fields, particularly in remote/inaccessible areas such as smart farming, smart forestry, and the Oil & Gas Industry's Industrial IoT [10]. While efforts have been made to adopt the serverless computing model in edge environments to leverage the advantages of reduced operational complexity and latency [5], [13], [32]–[34], its adoption in extreme edge computing poses unique challenges.

One of the significant obstacles in adopting serverless computing models for extreme edge computing lies in the reliance on energy harvesting methods and battery-powered edge devices, such as a cluster of single board computers (SBCs) powered by solar panels [2]. The use of renewable energy sources such as solar introduces energy supply variability due to factors such as instability,


- *Mohammad Sadegh Aslanpour is with Monash University and CSIRO's DATA61, Email: mohammad.aslanpour@monash.edu*
- *Adel N. Toosi and Muhammad Aamir Cheema are with Monash University*
- *Mohan Baruwal Chhetri is with CSIRO's DATA61*

intermittency, and unpredictability [2]–[4]. For instance, the availability of solar irradiation for edge nodes varies based on factors such as the location of the edge node in the field and the time of day. Consequently, this creates an imbalance in operational availability, leading to resource unavailability, node failure, and degradation of Quality of Service (QoS) [3].

The situation becomes even more complex due to the highly variable workload generated by IoT applications on the edge nodes. This variability further exacerbates the challenges, as the edge system strives to maintain operational availability, throughput, and reliability, thereby reducing the frequency of node failures and improving overall performance.

Such consequences pose challenges to the sustainability of edge computing, as they undermine key aspects such as efficient utilization of renewable energy inputs, uniform operational availability, reduced failure rates, and uninterrupted operation over extended periods of time.

While several well-established serverless frameworks such as OpenFaaS, KubeEdge, AWS Lambda@Edge, and Azure Functions on IoT Edge are designed to meet the demands of edge computing and address its unique challenges, it is worth noting that these frameworks are primarily designed to be energy-agnostic. They are focused on leveraging the advantages of the serverless edge computing paradigm without specifically addressing the energy-related considerations. To address this challenge, we propose *faasHouse*, a dynamic resource scheduling algorithm designed specifically to tackle the energy and load variability issues in serverless edge computing. In this paper, we make the following **key contributions**:

- We propose a reference architecture based on the serverless model for battery- and renewable energy-operated edge computing environments.
- We propose *faasHouse*, a dynamic, extensible, and energy-aware function scheduling algorithm that exploits computation offloading to address the imbalanced energy availability and the system's throughput challenges. *faasHouse*—inspired by Kubernetes design—follows a rigorous scoring scheme to rank edge nodes for function placements and then incorporates an assignment algorithm modeled as a House Allocation problem to decide on the placements.
- We provide a practical implementation of *faasHouse* using a real testbed and prototype, extending PiAgent [12]. We empirically evaluate *faasHouse*'s performance where experimental results demonstrate that *faasHouse* significantly improves the balanced operational availability (by 46%) and throughput (by 44%) of edge nodes.

## 2 Sustainable Serverless Edge Computing

### 2.1 System Overview

Fig. 1 depicts an overall system overview in which the edge nodes are wirelessly connected to each other and to a controller node, forming a cluster. Power is unevenly supplied to the edge nodes by renewable energy sources and rechargeable storage devices (e.g., batteries). Unique to the extreme edge, a node is both a *task generator*—directly connected to IoT sensors, generating data at variable and irregular rates—and a *computation resource* to execute tasks. A controller plays the role of the gateway and scheduler simultaneously. As a gateway, it distributes tasks/requests to their corresponding functions for execution. The scheduler monitors the edge nodes' State of Charge (SoC), which is the battery level of charge relative to its capacity, and utilizes offloading opportunities to migrate/place functions of low-powered nodes to well-powered nodes. Our earlier research shows that computation offloading allows a node to save a considerable amount of energy without imposing significant performance overhead [2].

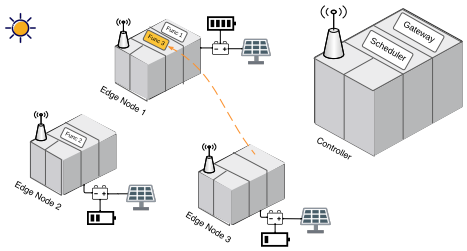

Fig. 1: An overview of the system

The challenge in this system is that while low-powered and/or overloaded nodes are prone to run out of energy and become operationally unavailable, well-powered and/or underutilized nodes are likely to waste their excess energy. The scheduler is responsible for handling this imbalance to achieve a more balanced energy distribution and improved throughput. A node is deemed operationally available if it has enough battery charge; otherwise, it is unavailable and unable to run functions and generate requests.

#### 2.1.1 A motivational exemplary application

We consider Smart Farming as a motivational example for this research. Assume a farming field enabled with edge computing for crop monitoring and precision agriculture to improve crop productivity, as demonstrated in [3], [16]. Edge nodes, like SBCs [16], are positioned across the farm and are connected through WiFi or LoraWAN. Sensors such as camera, temperature, and humidity are attached to each node, so that each node can collect data from the surrounding area, e.g., estimating the bee population. Edge nodes are equipped with actuators to enable automated actions such as pest deterrence or water disconnection.

In use cases like these, edge devices are meant to run without or with limited internet connection and are powered by solar panels and batteries in remote and vast farming areas [3]. Such constraints give rise to particular challenges. For instance, an edge node may have to perform recurring computations for object detection due to the high presence of pest birds, leading to the draining of the node's energy. Alternatively, a node may be positioned in a shaded area, resulting in inadequate energy input from its connected solar panel. Conversely, a node may perform minimal computations and thus waste its excess input energy. All of these events have the potential to result in a severe energy imbalance among the edge nodes, leading to operational unavailability. In such scenarios, some

nodes may be forced to turn off due to overloading/under-powering, while others may remain up but idle due to under-loading. This situation is particularly undesirable for farmers as it leaves certain areas unprotected from pests, posing a significant risk of crop damage.

## 2.2 System Architecture

Fig. 2 depicts the software architecture of our proposed sustainable serverless edge computing platform as well as the application workflow. A cluster of edge devices constituted of SBCs are enabled with container virtualization such as *Docker* or *containerd* for resource efficiency and portability [14]. An orchestrator tool such as *Kubernetes* resides on edge nodes to enable container deployment, container/node failure handling, high availability, etc [14]. Orchestration tools generally operate in a centralized master/worker architecture, where the controller node within the cluster is assumes the master role and resides on the master node. Edge nodes responsible for executing serverless functions act as worker nodes.

The controller hosts three main components: (a) database, (b) serverless platform, and (c) scheduler. A key-value database such as Redis serves data and state persistence so that services freely migrate and scale. A serverless platform is deployed on the controller to enable FaaS executions. That is, serverless functions are deployed and requests to the functions are admitted by a serverless gateway that uses a queue component for asynchronous executions. A load-balancing component distributes the requests between the function's replicas. A replica means an identical copy of a function. A function may have multiple replicas which are launched and scaled automatically by the auto-scaler in response to the demand. The queue holds exclusive lines per function. A scheduler—designed based on the IBM MAPE-K loop [17]—dynamically decides on the placement of functions. The MAPE-K loop will be explained later. The placement decisions are interpreted into the orchestrator language using an integration component.

On the workers' side, each node hosts a number of functions (i.e., services), depending on the scheduler's decision. Nodes are equipped with sensors to communicate with the IoT environment. Once the node receives sensor data from the environment, it triggers an event. The trigger calls the API related to a function through the serverless gateway for function invocation. The function performs processing and the output can be either an action or data to store in database/storage.

## 2.3 System Model and Problem Formulation

### 2.3.1 System Model

We discretize the time into equal timeslots denoted by $\mathbb{T} = \{t \mid t \in [0, T], (t+1) - t = \Delta t \text{ seconds}\}$. The scheduler rearranges the placements at the beginning of each timeslot. A summary of the table of notations is in Appendix A.

**Edge Nodes**: A set $D = \{d^i \mid i \in [1, n]\}$ defines wirelessly connected edge nodes in a cluster in a star-like network. A controller, excluded from $D$, performs the dynamic scheduling primarily based on the SoC and resource (CPU) capacity of each $d^i$. The SoC shows the battery charge of nodes at each timeslot and is defined by $S = \{s_t^i \mid i \in [1, n], t \in \mathbb{T}, 0 \leq s_t^i \leq \vartheta\}$, where $\vartheta$ denotes the maximum battery charge. The SoC is determined by energy input and energy consumption. Let $R = \{r_t^i \mid i \in [1, n], t \in \mathbb{T}, r_t^i \geq 0\}$ denote the renewable energy input to nodes over $t$. Also, let $E = \{e_t^i \mid i \in [1, n], t \in \mathbb{T}, e_t^i \in [0, \vartheta]\}$ denote the energy consumed on nodes over $t$, which depends on the node's base energy usage, i.e., $b^i$ where $B = \{b^i \mid i \in [1, n], t \in \mathbb{T}, b \in [0, \vartheta]\}$, and executed workload. $e_t^i$ is capped at the battery capacity so that the excess will be wasted.

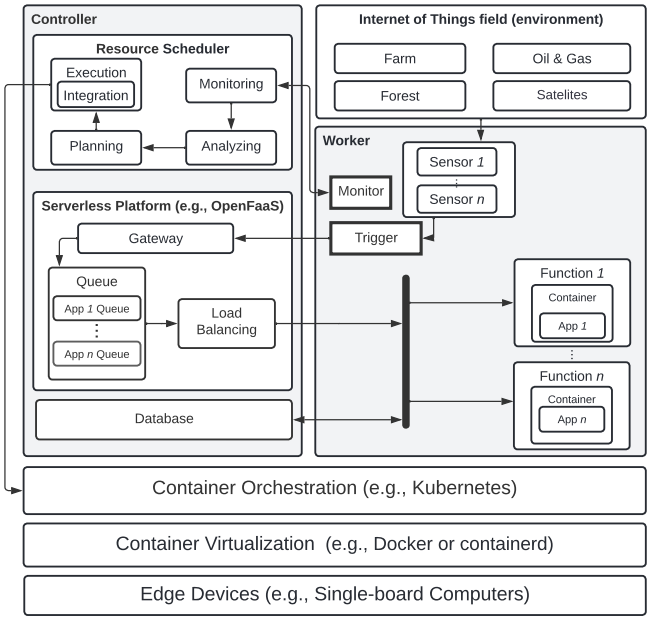


Fig. 2: Architectural view of the system

Let nodes' available resource capacity at time $t$ be denoted in Million Instruction Per Second (MIPS), as $C = \{c_t^i \mid i \in [1, n], t \in \mathbb{T}, c_t^i \in [0, \omega]\}$ where $\omega$ denotes the maximum capacity. $\omega$ is determined by the number of CPU cores and adopted CPU governor of a node. The latter in Linux systems can be `powersave`, `conservative`, `ondemand` (default governor), `performance`, etc.[1] While CPU frequency can vary between a minimum and maximum value, *powersave* always keeps it at a minimum; *conservative* varies the value slowly according to the demand; *ondemand* does the same as conservative, but more aggressively; and *performance* always keeps the frequency at a maximum. Modeling resource capacity allows heterogeneity considerations. Note that for a GPU-operated node, $c_t^i$ can be considered as the GPU capacity where $\omega$ denotes the maximum GPU capacity. In this work, we focus on CPU-only devices, and GPU and heterogeneity aspects thereof are left as future work. Note that $S$, $R$, $E$, and $C$ represent values per $d^i$ at timeslot $t \in \mathbb{T}$.

**Application & Workload**: An IoT application normally includes multiple microservices. A set of microservices is denoted by $A = \{a^j \mid j \in [1, m]\}$. An edge node owns, hosts, and runs a full set of microservices in $A$ that are identical across all nodes. The rate at which workload (sensor data) is generated for microservices at different nodes at different timeslots is modeled as $\Lambda = \{\lambda_t^{i,j} \mid i \in [1, n], j \in$

1. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

$[1,m], t \in \mathbb{T}, \lambda_t^{i,j} \in [0, \varpi^{i,j}]\}$, where $\varpi^{i,j} \in \pi = \{\varpi^{i,j} \mid i \in [1,n], j \in [1,m], \varpi > 0\}$ represents the node's bandwidth capacity in terms of the number of tasks per microservice it can push to the network at $t$. The amount of processing required for a task in Million Instructions (MI) is defined as $M = \{\mu^j \mid j \in [1,m], 0 \leq \mu^j \in \mathbb{R}\}$. The workload modeling is employed later to determine the number of required function replicas.

**Serverless Functions**: The unit of resource scheduling is a function in serverless [9]. Let $F = \{f^{i,j} \mid i \in [1,n], j \in [1,m]\}$ denote a set of functions, where function $f^{i,j}$ executes tasks of microservice $a^j$ owned by node $d^i$. Function $f^{i,j}$ requires a certain amount of resources to be provisioned on a node, defined as $V = \{v^j \mid j \in [1,m], v^j \in [0,\omega]\}$, where $v^j$ is capped at the node's maximum capacity $\omega$.

**Function Replicas (instances)**: For the sake of horizontal scaling of functions (i.e., adjusting the number of function replicas) in serverless, $f^{i,j}$ may have a varied number of function replicas at $t$, capped at $\aleph$, depending on the incoming workload. Hence, set $\Gamma = \{\gamma_t^{i,j} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, \gamma_t^{i,j} \in [0,\aleph]\}$ defines the required number of replicas per function by determining the associated microservice's workload over the computation capacity. The number of replicas of $a^j$ on $d^i$ at $t$ is measured as $\gamma_t^{i,j} = \min\left(\aleph, \lceil \frac{\lambda_t^{i,j} \times \mu^j}{v^j} \rceil\right)$ that is between 0–$\aleph$ depending on the workload. In an optimal offline model with future knowledge, $\Gamma$ is assumed to be known. If node $d^i$ is down at $t$, then $\gamma_t^{i,j} = 0$ replicas are deployed in the cluster. In serverless, zero replicas, or so-called scaled-to-zero functions, consume zero capacity.

For replica-level modeling, we upgrade $F$ to include replica indexes, denoted by $F = \{f_t^{i,j,k} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, k \in [1, \gamma_t^{i,j}]\}$ where $k$ indicates the k-th replica.

**Function Placement**: A function's replica is either placed on its local node, or offloaded to peers, by the scheduler decision [8]. Considering SoC $s_t^i$ on $d^i$ at timeslot $t$, the scheduler determines local and offloaded function placements per node. Let $\Psi = \{\psi_t^{i,j} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, \psi_t^{i,j} \geq 0\}$ denote local placements and $\Theta = \{\theta_t^{i,j} \mid i \in [1,n], j \in [1,m], t \in \mathbb{T}, \theta_t^{i,j} \geq 0\}$ denote offloaded placements. This clarification is particularly essential, as communication overheads are involved, in the case of offloading functions, for both sender and receiver. That is, if $f_t^{i,j,k}$ is offloaded to $d^q$, a send data energy cost for $d^i$ and a receive data energy cost for $d^q$ are involved. A cost rate per microservice for either one is defined as $O = \{o^{j,h} \mid j \in [1,m], h \in \{send, recv\}, o^{j,h} \in [0,1]\}$, where $o^{j,h}$, $h = send$ or $h = recv$, specify the overhead energy cost imposed by offloading function $f_t^{i,j,k}$ and hosting peers function $f_t^{i,j,k}$, respectively. The determination of $O$ depends on the function's replica, which itself is determined by the workload it has to execute. Hence, the communication cost incorporates the data volume to transfer over the network, uniformly for requests of a function [2]. This cost is mutated into energy usage. Note that local placement $\psi_t^{i,j}$ of replicas of a function is bound to $\gamma_t^{i,j}$, so the constraint $\psi_t^{i,j} \leq \gamma_t^{i,j}$ must be maintained by the scheduler.

**Node Availability (Up or Down)**: Nodes generate requests and run functions at the same time only if they are up, i.e., if they satisfy low energy threshold $s_t^i \geq \varphi$. Otherwise, they are excluded from hosting any local or offloaded function. Node availability (status) is denoted by $X = \{x_t^i \mid i \in [1,n], t \in \mathbb{T}, x_t^i \in \{0,1\}\}$ where $x = 0$ means the node is unavailable, or down, during timeslot $t$, and vice versa. Given that, the nodes' binary status $x_t^i$ at $t$ are as follows:

$$\forall d^i \in D : x_t^i = \begin{cases} 1 & \text{if } s_t^i \geq \varphi \\ 0 & \text{else} \end{cases}, \tag{1}$$

where a node's SoC $s_t^i$ is determined by the SoC at the previous timeslot $s_{t-1}^i$, renewable energy input $r_t^i$, and consumed energy $e_t^i$ at timeslot $t$.

$$\forall d^i \in D : s_t^i = \min\left(\vartheta, \max(0, s_{t-1}^i + r_t^i - e_t^i)\right), \tag{2}$$

where SoC can vary between 0 and $\vartheta$. Assuming renewable input is known, the energy consumed $e_t^i$ depends on the hosted functions and is computed for all $d^i \in D$ as follows:

$$e_t^i = \Bigg(b^i + \left(\frac{c_t^i}{\omega} \times \rho\right) + \Bigg(\sum_{j=1}^{m} \left(o^{j,send} \times (\gamma_t^{i,j} - \psi_t^{i,j})\right) + \sum_{j=1}^{m} (o^{j,recv} \times \theta_t^{i,j})\Bigg)\Bigg) \times \Delta t \tag{3}$$

where $e_t^i \leq s_{t-1}^i + r_t^i$ and $e_t^i$ takes into account: (a) *base usage*, (b) *direct usage* and (c) *offloading overhead*. For the *base usage*, the $b^i$ determines a static rate of energy use for the node's hardware and OS to operate. For the *direct usage*, occupied resources are measured by $\frac{c_t^i}{\omega}$ and multiplied by a power consumption rate $\rho$. The occupied resources of a node, as a key element in $e_t^i$ measurements, is obtained by $c_t^i = \sum_{j=1}^{m} \left((\psi_t^{i,j} + \theta_t^{i,j}) \times v^i\right)$. For the *offloading overhead*, energy consumption is calculated by the send ($o^{j,send}$) and receive ($o^{j,recv}$) overhead multiplied by the offloaded functions ($\gamma_t^{i,j} - \psi_t^{i,j}$) and by the received offloaded functions ($\theta_t^{i,j}$), respectively. This measurement is made for every microservice $j$. The power consumed by (a), (b), and (c) is multiplied by the length of the timeslot, $\Delta t$, to obtain energy.

### 2.3.2 Problem Formulation

We aim to optimize the placement of functions across the cluster by dynamically adjusting it to minimize availability variance among nodes and enhance system performance. This adjustment is demanded due to the skew in power and load distribution resulting in uneven operational availability, i.e., up-time, of different nodes which hinders seamless serviceability across the entire cluster [7]. More importantly, this adjustment aims to satisfy sustainability requirements such as minimizing nodes' failure and renewable energy input wastage. Here we focus on an Imbalanced Energy Minimizing Assignment Problem, and the objective function aims at maximizing the availability of the least available nodes, by technically relying on well-powered nodes for improving low-powered ones, as in (4):

$$\underbrace{\max \min_{i \in [1,n]} \sum_{t=0}^{T} x_t^i}_{\mathbb{T},D,X,S,R,E,B,A,\Lambda,M,F,\Gamma,V,\Psi,\Theta,O,C} \tag{4}$$

subject to:

$$\Big(\sum_{i=1}^{n} \psi_t^{i,j} + \theta_t^{i,j}\Big) = \Big(\sum_{i=1}^{n} \gamma_t^{i,j}\Big) \forall t \in \mathbb{T}, \forall a^j \in A \quad (5)$$

$$\gamma_t^{i,j} = x_t^i \times \min\Big(\aleph, \Big\lceil \frac{\lambda_t^{i,j} \times \mu^j}{v^j} \Big\rceil\Big) \forall t \in \mathbb{T}, \forall d^i \in D, \forall a^j \in A \quad (6)$$

$$\Big(c^i = \sum_{j=1}^{m} \big((\psi_t^{i,j} + \theta_t^{i,j}) \times v^j\big)\Big) \leq (x_t^i \times \omega) \forall t \in \mathbb{T}, \forall d^i \in D \quad (7)$$

At any given $t$, constraint (5) specifies that the scheduler must not leave any function replica unscheduled; constraint (6) defines that if a node is down, scheduling of its microservices is not considered, i.e., scale to zero; this also implies that if a node is down, its generated tasks are discarded, i.e., $\lambda_t^{i,j} = 0$; constraint (7) defines that the sum of the capacity required for placed functions must not exceed the node maximum capacity $\omega$, either local or offloaded, and inclusion of $x_t^i$ forces no placement on down nodes.

The optimal algorithm is used as a baseline offline algorithm. It requires the renewable energy input and incoming workload at each $t \in \mathbb{T}$ to be known to the scheduler in advance, which is difficult to achieve in practice. It is also computationally unaffordable for constrained devices to use. In the following section, we propose an online energy-aware scheduler, named *faasHouse* that does not require such knowledge and presents acceptable computational complexity.

## 3 faasHouse: Resource Scheduling

In this section, we introduce *faasHouse*, a scheduling algorithm for the sustainable serverless edge.

***Remark:*** In serverless edge computing, effectively managing IoT application functions, that dynamically scale up or down through the backend auto-scaler, requires a dynamic solution with reasonable computational complexity.

Therefore, the design principle of *faasHouse* is to dynamically operate in every timeslot, according to IBM MAPE loop [17]—Monitoring, Analysis, Planning, and Execution—to address the imbalance in nodes' availability. Algorithm 1 shows *faasHouse* design which is discussed in the following:

**Monitoring:** In each timeslot, the latest SoC of nodes is read through HTTP-based communication (Lines 2–3).

**Analysis:** The SoC value of each node can undergo a rigorous analysis. Here, we simply use the actual observation of the SoC of a node. Predictive analysis is out of the scope of this work, but it can expand this phase further.

**Planning:** As the decision-making phase in the context of serverless edge computing for IoT applications, it requires special considerations such as practicality and extensibility. We have developed a novel planner inspired by the principles of the Kubernetes scheduler, the first-class scheduling model in containerized edge computing environments [2], [5], [12]–[14]. Our scheduler follows a two-step process consisting of a ***scoring*** phase followed by an ***assignment*** phase. The detailed design of the scoring and assignment phases (Lines 6–8) is explained in the subsequent subsections.

**Execution:** It applies the action of function placement by Kubernetes so that functions are scheduled on the assigned nodes according to the planned assignments.

**Algorithm 1:** faasHouse Scheduler

**Data:** edge nodes and serverless functions
**Result:** energy-aware function placement

1 **while** $t \leq T$ **do**
`/* monitor: get latest SoC of nodes */`
2 **do in parallel**
3 **foreach** $d^i | d^i \in D$ **do** $s_t^i \leftarrow$ getSoC$(d^i)$
`/* analyze: improve the monitored data */`
4 **do in parallel**
5 **foreach** $d^i | d^i \in D$ **do** $z_t^i \leftarrow$ Instant$(d^i)$
`/* plan: make placement decisions */`
6 **do in parallel**
7 **for** $f_t^{i,j,k} | f_t^{i,j,k} \in F$ **do**
$\mathfrak{S} \leftarrow$ Scoring$(f_t^{i,j,k}, D, P)$ `// Algorithm 2`
8 $F \leftarrow$ Assignment$(D, F, \mathfrak{S}, C, V)$ `// Algorithm 3`
`/* execute: make assignments effective */`
9 Execute$(F)$
10 Sleep$(\Delta t)$

**Algorithm 2:** Scoring

**Data:** $f_t^{i,j,k}, D, P$
**Result:** $\mathfrak{S}$, updated preferences of given functions

1 **foreach** $q, q | d^q \in D, in\ [1, n]$ **do** $\mathfrak{s}_q^{i,j,k} \leftarrow 0$
`/* calculate nodes' scores per plugins */`
2 **for** $q, q | d^q \in D, in\ [1, n]$ **do**
3 **for** $p | p \in P, p \leftarrow (plugin, weight)$ **do**
4 $\mathfrak{s}_q^{i,j,k}$ += Plugin$(p, f_t^{i,j,k}, q)$
5 **return** $\mathfrak{S}$

### 3.1 Scoring

The scoring algorithm, Algorithm 2, is invoked per function's replica to determine its preference for nodes. The preference is quantified using scores, where a higher score indicates a stronger preference. Algorithm 2 first initializes the score $\mathfrak{s}_q^{i,j,k}$ of the function $f_t^{i,j,k}$ for each node $d^q$ to zero (Line 1). Then, it determines the desirability of placing the function on each node (Lines 2–4) according to constraints. Without the loss of generality, this determination is made extensible by designing a `Plugin` module that allows scoring using customized requirements. A `Plugin` gets an implemented plugin $p$ name and its weight as a tuple $(< name >, < weight >)$, a function, and a candidate node. Next, it fetches the corresponding plugin and determines the node's score for the given function.

***Remark:*** Extensibility is essential for a scheduler as it allows the inclusion of non-energy factors such as soft QoS constraints like data locality and customization to meet diverse IoT application requirements.

In this work, we have implemented the following plugins: *energy*, *locality*, and *stickiness*. The *energy* plugin, defined as $p(energy, weight) \mid p \in P, weight \geq 0$, determines the function's preference on nodes based on the energy benefit as:

$$(z_t^q - z_t^i) \times weight \times x_t^q x_t^i, \quad (8)$$

where the desirability of a candidate node $z_t^q \mid q \in [1,n]$ than the function's local node $z_t^i$ is measured and weighted. Measuring the difference instead of using actual $z_t^q$ not only sets higher preferences for better-powered nodes but judiciously deters lower-powered nodes. Multiplying by $x_t^q$ and $x_t^i$ excludes down nodes and functions of down nodes from scoring as they are not meant for placements.

The *locality* plugin, defined as $p(locally, weight)|p \in P, weight \geq 0$, is measured by (9).

$$\begin{cases} z_t^i \times weight \times x_t^i & \text{if } d^q = d^i \\ 0 & \text{else} \end{cases}, \tag{9}$$

where the desirability of placing a function on its local node, if operationally available, is measured and weighted proportional to that of the node's $z_t^i$. This is important to avoid excess offloading and to preserve QoS by letting functions stay closer to the data source.

The third plugin is named *stickiness*, defined as $p(sticky, weight) \mid p \in P, weight \geq 0$, to prevent recurring replacements of a function that deteriorate QoS [12]. The *stickiness* plugin determines a weighted preference for a function on a node hosted the function in the previous scheduling round as:

$$\begin{cases} weight \times x_t^q x_t^i & \text{if } f_t^{i,j,k} \neq d^i \wedge f_t^{i,j,k} = d^q \\ 0 & \text{else} \end{cases}, \tag{10}$$

where the *stickiness* plugin favors the previous location of a function indicated by $f_t^{i,j,k}$, only if both the function's local and remote nodes are still operationally available.

The implemented plugins are highly relevant to the system under test. However, it is important to note that other systems and use cases may require support for other QoS requirements such as latency (nodes proposing lower latency), bandwidth (nodes requiring lower bandwidth consumption), privacy (nodes providing a higher level of privacy satisfaction), etc.

### 3.2 Assignment

The assignment problem is a generalized assignment problems (GAP) [19], such that it attempts to find an optimal assignment of tasks (functions) to agents (nodes), subject to the agent's capacity and task size constraints. A linear programming model of the problem is as follows.

$$\max \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{\gamma_t^{i,j}} \left[ \sum_{q=1}^{n} \mathfrak{s}_q^{i,j,k} \times y_q^{i,j,k} \times x_t^i \times x_t^q \right] \tag{11}$$

subject to:

$$\sum_{q=1}^{n} y_q^{i,j,k} \times x_t^q = 1 \times x_t^i \quad \forall i \in [1,n], j \in [1,m], k \in [1, \lambda_t^{i,j}] \tag{12}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{\lambda_t^{i,j}} v^j \times y_q^{i,j,k} \leq c_t^q \qquad \forall d^i \in [1,n] \tag{13}$$

$$y_q^{i,j,k} \in \{0,1\} \tag{14}$$

where $y_q^{i,j,k}$ is an integer decision variable that returns 1 if function $f_t^{i,j,k}$ is assigned to $d^q$; otherwise returns 0. Objective (11) is to maximize the total score of assigning functions to nodes. The assignment is subject to the following constraints. All functions are required to be assigned and each to only one node, as in (12), excluding down nodes and their functions from each side. Nodes' capacity constraints are enforced by (13), where $v^j$, $y_q^{i,j,k}$, and $c_t^q$, denote the function size, function assignment, and node capacity, respectively. Given integral constraints, the problem can be solved using Integer Programming techniques which are not scalable for large systems.

***Solution***: To address that, we incorporate an assignment algorithm that is inspired by the House Allocation Problem [18], a practical model in economics and computer science. House allocation is the problem of assigning houses (nodes) to people (functions) considering people's preferences. Examples of this problem are school allocation for Boston school [20] and office allocation for Harvard University professors [18]. The mechanism to solve the House Allocation problem is that all functions request their most preferred nodes and the allocation occurs, if no conflict exists. Conflicts are resolved by a supplementary mechanism.

A challenge in solving the assignment at hand is the need to prioritize the local functions of a node over offloading functions. This is to avoid situations where a function of a node hosting other nodes' functions will be left without a host. This also helps reduce the offloading effects. Thus we employ the special case of *House Allocation with Existing Tenants*, which accepts such challenges out of the box. That is, nodes are considered to own their functions whereas, upon receiving remote functions for placements, local functions are prioritized over the newcomers. This solution mechanism is called *"You Request My House, I Get Your Turn"* [18].

The assignment procedure is demonstrated in Algorithm 3, as follows. Firstly, functions and their preferences for nodes are given. The functions' list is reconstructed and functions are considered unassigned (Line 1).

***Remark:*** The list reconstruction process allows the inclusion of newly scaled-up functions and recognition of the scaled-down functions performed by the serverless platform.

A `cycle` list is used to resolve conflicts (Line 2). Functions are sorted in descending order, based on the sum of scores they have given to nodes. This brings functions with higher preferences to the front of the queue (Line 3). The assignment procedure continues until there exists no unassigned function (Lines 4–14). A function from the front of the queue is selected (Line 5). Note that if the function's most preferred node (with sufficient capacity) is requested by another function, a conflict occurs and the function in hand joins the `cycle` list. This list holds functions and preferences until a function appears for the second time which triggers a cycle clearance. This means the assignment accepts all requests and every function receives the requested node. Hence, a cycle clearance verification is always conducted before searching nodes for a function (Line 6). If the function is not involved in the cycle list, the assignment proceeds (Line 8–14); otherwise, the cycle clearance is triggered that performs as mentioned earlier (Line 7). To assign (Line 8–14), the maximum scored node of the function is selected (Line 9). Priority Mechanism verifies if the selected node owns any unassigned function (Line 10). If not, the assign-

**Algorithm 3:** Assignment

**Data:** $D, F, \mathfrak{S}, C, V$
**Result:** $F$, updated functions' assignment
1 **foreach** $f_t^{i,j,k} | f_t^{i,j,k} \in F$ **do** $f_t^{i,j,k} \leftarrow \emptyset$
2 **cycle** $\leftarrow \emptyset$
/* sort funcs. by scores descendingly */
3 sort($\mathfrak{S}$)
4 **while** $\exists f_t^{i,j,k} \in F | f_t^{i,j,k} = null$ **do**
/* get highest scored unassigned func. */
5 $f_t^{i,j,k} \leftarrow \mathfrak{S}$.**peek**()
/* verify existence of func. in cycle */
6 **if** $f_t^{i,j,k} \in$ **cycle then**
/* clear cycle by accepting requests */
7 $f_t^{i,j,k} \leftarrow d^q \quad \forall f_t^{i,j,k} \in$ **cycle**
8 **else**
/* get max. scored node with capacity */
9 $d^q \leftarrow \max($all scored $d^q$ for $f_t^{i,j,k} | c_t^q \geq v^j)$
10 **if** $f_t^{q,j,k} \neq \emptyset \quad \forall f_t^{q,j,k} \in F | q \in [1,n], j \in [1,m]$ **then**
/* assign func. to selected node */
11 $f_t^{i,j,k} \leftarrow d^q$
12 **else**
/* prioritize funcs. of the node */
13 $\mathfrak{S}$.**enqueueFront**$(f_t^{q,j,k} \quad \forall f_t^{q,j,k} \in F)$
14 **cycle**.**enqueue**$(f_t^{i,j,k}, d^q)$
15 **return** $F$

ment is allowed (Line 11); otherwise, the local functions of the selected node are queued to the front, respecting their priority (Line 13) and the function under investigation is added to `cycle` list (Line 14). A sample scenario is provided in Appendix B for further information.

**Computational Complexity:** We analyze the computational complexity of the assignment algorithm which is the main computation required for the scheduler. We use $R$ to denote the total number of functions to be assigned. Note that $R = |F|$ and is bounded by $O(nm\aleph)$. The computation complexity is bound by $O(R^2)$. Please, refer to Appendix C for a detailed analysis.

## 4 Performance Evaluation

### 4.1 Experimental Setup

We implement a highly practical and real-world distributed software system, extending `PiAgent` [12], written in $\sim 5000$ lines of Python 3.9 code to control nodes, generate workload, perform scheduling, etc. We conduct a 24-hour experiment and repeat each experiment five times as per scheduler. The implemented system and key settings, extensively following [12], are as follows.

∘ ***Edge Nodes***: A cluster of 10 Raspberry Pi Model 3 B+ ($n = 10$) is used. Although we expect our system to work well in heterogeneous settings, we have deliberately opted for a homogeneous setup to better analyze the performance evaluation results, given the system is tested under heterogeneous energy input and workload. This could be more challenging to comprehend in heterogeneous settings, as devices have varying energy consumption. It is crucial to emphasize that while the homogeneity aspect strictly pertains to the hardware configuration, the workload and energy input of devices retain their inherent heterogeneity.

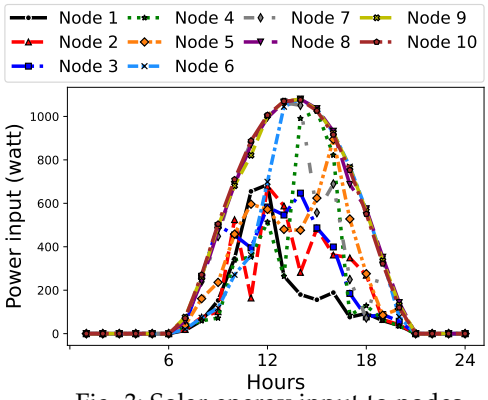

Fig. 3: Solar energy input to nodes

The energy input by solar panels is emulated using real traces [21] to set $R$, shown in Fig. 3. Each node receives a considerably different rate of energy input. The battery capacity is modeled from the original-size PiJuice battery of Pi 3 B+ [2] and is set to $\vartheta = 1250$ mWh, and the low energy threshold $\varphi$ is set to $\frac{\vartheta}{10} = 125$ mWh, where $b^i = 0.2$. The model for battery is obtained by empirical studies on the same devices in [2], [12]. Furthermore, we study the effect of varying the battery size in the sensitivity analysis experiments. A hardware-based measurement is adopted to calculate the actual energy consumption of edge nodes, using UM25C[2] USB power meters which are highly precise [2]. A 4-core Pi gives $\omega = 4000$, but we keep a safety net of 10% to avoid overheating. We empirically set the send ($o^{j,send}$) and receive ($o^{j,recv}$) overhead for function offloading at 0.02 and 0.01, according to benchmarks in [2]. *containerd* is enabled on Pis for containerization, Kubernetes K3s (version v1.212) is deployed for orchestration, and OpenFaaS (version 8.0.4) enables serverless computing. Given their edge-friendly design, they impose a negligible amount of computing overhead, i.e., 1.5% CPU use on Raspberry Pis, and around 50MB of memory footprint, which will not affect the execution of functions [2]. A containerized Redis server, a high-speed in-memory data storage, is deployed on the controller node to store function states, a technology selection highly encouraged for the lightweight realization of edge computing [2], [13], [14].

∘ ***Application and Workload***: A generic *workflow* that builds a commonly-used pipeline [2], [3], [22], i.e., sensing data, generating requests, invoking an execution unit, processing, storing results, and triggering actuators, is developed for evaluations. This pipeline applies to a wide range of event-driven IoT applications [22], from Smart City, and Autonomous Vehicles to Industrial IoT and Smart Farming. We acknowledge that, although our system design can accommodate a broad spectrum of applications, attempting to host an excessive number of applications on a single edge node may prove impractical due to the inherent resource limitations of these devices. For instance, a Raspberry Pi 3, equipped with only 1 GB of RAM, has its capabilities

2. https://tinyurl.com/um25c

constrained. Therefore, we have carefully selected three microservices ($m = 3$) with distinct characteristics of CPU-, data-, and I/O-intensive tasks - as motivated in Section 2.1. This selection ensures a diverse and representative sample for our comprehensive evaluations. It is worth noting that adhering to this range aligns with practical use cases as supported by existing literature [22].

The CPU-, I/O-, data-intensive microservices include (1) soil moisture controlling—it constantly reads the soil sensor data, calculates the moisture level, and stores results in Redis; (2) irrigation management—it monitors the soil moisture level and applies the irrigation policies to connect/disconnect water through the attached actuators; and (3) a pest-repellent AI-enabled application—it captures an image using an embedded camera, triggers an HTTP request, the image is fetched from the generating node by the executing function, and a Single Shot Detector (SSD) [23] machine learning model performs object detection on the image to find pest birds in the area [12]. All microservices are developed in Python and are packaged as serverless functions. We set the CPU requirements of these three functions as $V = \{50, 100, 650\}$, acquired through profiling, and allow maximum replicas of $\aleph = 3 \equiv k \leq 3$ per function. This limited range of replica numbers leaves sufficient room for the serverless platform to experiment with frequent scale-up/down of functions.

The *workload* is generated at a different and random $\lambda$ rate per node per function [2], [13], [24], where a deterministic uniform distribution of rates is adopted to achieve diversity. The request generation is synchronous, as once a request is finished, the next one is triggered. During the entire experiment, such workloads are being generated, but we also vary the intensity of the workload in the sensitivity analysis. Given the observation on a local network in extreme edge and its high available bandwidth [2], we let $\varpi^{i,j}$ be large enough, i.e., bound to the computation capacity.

∘ ***Scheduler***: We perform the scheduling every 30 minutes, i.e., $\Delta t = 1800$s. Further, we also study the effect of varying this value between 15 minutes to 60 and 180 minutes in Appendix G. For *faasHouse*, in the scoring phase, the plugins' weights are set at 100, 60, and 40 for energy, locality, and stickiness plugins, respectively. These values are achieved by resource profiling and represent the best performance upon experiments on the demonstrated setup. To evaluate the effectiveness of our assignment algorithm (i.e., Algorithm 3), we also conducted a comparative study against three other state-of-the-art GAP solutions. The results of this comparison are detailed in Appendix D.

∘ ***Execution***: The placement determination is translated to Kubernetes scheduling terminologies described in Yaml language [14]. Dynamic scheduling demands CI/CD (continuous integration and continuous deployment). Maintaining a vast number of Yaml files for $n \times m$ deployments is highly cumbersome. We thus automate the entire execution process using Helm,[3] an automation tool for Kubernetes deployments, which allows easy and fast (re-)deployments [12].

#### 4.1.1 Benchmark algorithms

We evaluate *faasHouse* against the following benchmarks:

∘ ***Optimal***: It is a baseline offline algorithm that requires in-advance knowledge of renewable energy input and incoming workload for all future timeslots under consideration. It solves the constrained optimisation problem described in Section 2.3.1 modeled by MiniZinc[4] (version 2.5.5) exploiting Gurobi[5] solver (version 9.1.12).

∘ ***Kubernetes***: This is the built-in default scheduler of Kubernetes.

∘ ***Local***: It is a baseline algorithm that always deploys functions locally, i.e., $f_t^{i,j,k} \leftarrow d^i$. This is worth evaluating to understand the impact of computation offloading.

∘ ***Random***: It is another baseline that randomly places functions across the cluster. It demonstrates the worst-case scenario where no intelligence is considered in the scheduler's decisions.

∘ ***Zonal***: It is a dynamic energy-aware scheduler, as detailed in [12], that forms zones of nodes, having different ranges of SoC. The scheduler aims to assign functions within a zone to equally- or better-powered zones, under special considerations such as stickiness and warm scheduling.

#### 4.1.2 Metrics

∘ ***Operational Availability***: The operational availability of a node is defined as the percentage of time it is available and operational, indicating that it has sufficient battery charge to generate tasks and execute functions [12]. A longer availability of the minimum available node quantifies the effect of the scheduler on the key objective outlined in Eq. (4). A smaller standard deviation (SD) and range of operational availability among nodes in the cluster reflects an improved variation/balance in their availability.

∘ ***Nodes' Failure***: It quantifies the average of the number of times at which nodes in the cluster experience operational failures caused by insufficient energy levels. The lower the value, the more desirable the outcome.

∘ ***Longest Availability***: It measures the average of the longest time nodes in the cluster could remain operational without any failure, representing the reliability improvements. The interruption can occur by a node running out of battery and turning off temporarily, for instance. The longer the better.

∘ ***Throughput***: The average of the number of tasks executed per time unit (i.e., one second) by nodes in the system [12].

∘ ***Nodes' Wasted Energy***: It quantifies the average energy input waste in mWh across the nodes in the cluster. This occurs when a node's battery is full, and it fails to efficiently utilize the excess energy input. A smaller value indicates better efficiency in minimizing energy waste.

∘ ***Response Time***: The round trip time of a request from being generated to being executed and completed by a function [12].

### 4.2 Experimental Results

We conducted five sets of experiments, which are detailed below, and analyzed the results obtained. Each experiment was repeated five times, and the averages and standard deviations are reported.

3. https://helm.sh/

4. https://www.minizinc.org/
5. https://www.gurobi.com/

- First set of experiments, using the setting in Section 4.1, gives ***overall results*** obtained by schedulers.
- Second set evaluates the ***impact of varying the battery size*** of edge nodes ($\vartheta$) from 1000 to 1250, 1500, and 1750 mWh.
- Third set evaluates the ***impact of varying workload intensity*** from 25 to 50, and 75 to 100%.
- Forth set evaluates the ***impact of varying the scheduling interval*** times ($\Delta t$) from 15 to 30, and 60 to 180 minutes. The results are in Appendix G.
- Fifth set evaluates the ***impact of varying CPU governor, influencing resource capacity*** ($\omega$), from *powersave* to *conservative*, *ondemand*, and *performance*. The results are in Appendix H.

#### 4.2.1 Overall Results

**Operational Availability:** As shown in Fig. 4, we observe that the maximum operational availability of the ***minimum available node*** is achieved by the *optimal* (17.1%), *faasHouse* (15.9%), and *zonal* (12.2%), respectively, over the *Kubernetes* (11.7%) scheduler, demonstrating a 46%, 36%, and 4% improvement. In contrast, the *local* and *random* impair the value to 8.6% and 9.4%.

In Fig. 4, ***the range***, i.e., max-min difference, of operational availability shows that the *optimal*, *faasHouse*, and *zonal* improve it to 36.63%, 42.5%, and 48.23%, respectively, over the *Kubernetes* scheduler. This minimization represents a more balanced utilization of energy by nodes despite skewed energy supply and load generation.

Another indication of achieving a balanced energy usage is the reduction in the ***SD*** of the cluster-wide operational availability, where the *optimal* (11.04) followed by *faasHouse* (13.4) minimize it up to 45% and 33%, respectively, over the *Kubernetes* scheduler (20.02).

The key factor for such differences between schedulers is that well-performed schedulers utilize the well-powered nodes, particularly their wasted energy, to help low-powered nodes. In detail, *faasHouse* exploits the enabled resource sharing to offload functions of low-powered nodes to well-powered nodes. This intelligence is augmented with future knowledge in the *optimal* scheduler.

On the weak side, the *local* scheduler exhausts the low-powered nodes' energy in isolation and fails to exploit resource sharing and offloading. The *Kubernetes* scheduler is enabled with resource sharing, but its performance-driven scheduler fails to satisfy sustainability requirements. The *random* scheduler is enabled with resource sharing, but there is no rationale behind the re-scheduling decisions; hence, not only no improvement is observed for low-powered nodes by *random*, but also the experiments' logs report that it is highly likely that it offloads well-powered functions to low power nodes. Lastly, while the *zonal* algorithm respects the energy status of nodes and exhibits better performance than the *Kubernetes*, *local*, and *random* schedulers, it fails to perform as efficiently as the *faasHouse*, mainly due to its dependence on sticky offloading and warm scheduling features that can make it perform over-cautiously.

Note that the average operational availability of nodes is observed to be approximately identical for all schedulers: *optimal* (41.8%), *faasHouse* (42.09%), *Kubernetes* (42.29%), *random* (40.61%), and *zonal* (39.98%), with the exception of the *local* scheduler (35.55%). This finding confirms that the improvement in minimum availability through computation offloading has not resulted in a significant energy loss or energy overhead. The similarity in operational availability can also be attributed to the significant improvement in throughput for schedulers such as *optimal* and *faasHouse*, which incur higher energy consumptions.

**Failure:** Lowering failure is crucial for sustainability, and it is important for the scheduler to actively prevent nodes from reaching a low battery state, minimizing the chances of failure. Fig. 5 shows the number of times on average nodes have experienced a failure (off state) as per scheduler. Compared to the *Kubernetes* scheduler (#73), the *optimal* (53), *faasHouse* (59), and *zonal* (68) reduced the failures by 27%, 19%, and 7%, respectively. The *local* scheduler performs considerably poorly in this case, as it disregards the low-energy state of the node and excessively pushes the node back into the failed and booting-up state. This behavior appears so extreme that the *local* scheduler is less practical than the *random* scheduler, in this metric.

**Longest Availability:** Fig. 6 shows that the *optimal* (520 minutes) and *faasHouse* (470 minutes), followed by the *zonal* (450 minutes) schedulers succeed in prolonging the operational time of nodes on average, by up to 24%, 11%, and 7% against the *Kubernetes* scheduler, respectively. The *local* scheduler performed poorly since it fails to recognize the low-power state on the nodes and keeps utilizing them even when they have just been turned on. This on-and-off situation can continue for a long period of time in some cases, especially if the energy input and consumption of a low-power node remain relatively at the same rate.

**Throughput:** Fig. 7 illustrates the system's overall throughput for different schedulers. The *optimal* (1.02 tasks/sec) and *faasHouse* (0.92) schedulers enhance the throughput by up to 44% and 30%, respectively, compared to the *Kubernetes* scheduler (0.71). This improved throughput confirms that nodes, particularly low-powered ones, can execute more tasks when they remain operationally available for a longer duration. Conversely, the reduced throughput observed in the *random* scheduler indicates that nodes experience interruptions in their serviceability due to energy insufficiency. It is worth noting that since requests are sent synchronously, meaning a new request is sent only after the response for the previous one is received, nodes that are available for a longer duration generate more requests, ultimately increasing the overall throughput.

**Wasted Energy:** Energy waste occurs when a node's battery is fully charged, resulting in the underutilization of the excess energy input. Fig. 8 shows that the *optimal* (14 mWh) and *faasHouse* (103 mWh) managed to minimize the energy wastage significantly compared to the *Kubernetes* scheduler (433 mWh), demonstrating a 97% and 76% improvement, respectively. This is justified by the fact that energy awareness, coupled with proper placement of functions, allows *optimal* and *faasHouse* to utilize well-powered nodes more frequently, thereby reducing the likelihood of full battery and mitigating energy wastage. This intelligence also benefits the low-powered nodes to survive longer, as shown before. However, the *local* scheduler, while continuously utilizing all nodes, fails to achieve such performance, since it fails to minimize the wastage of the energy of a node having

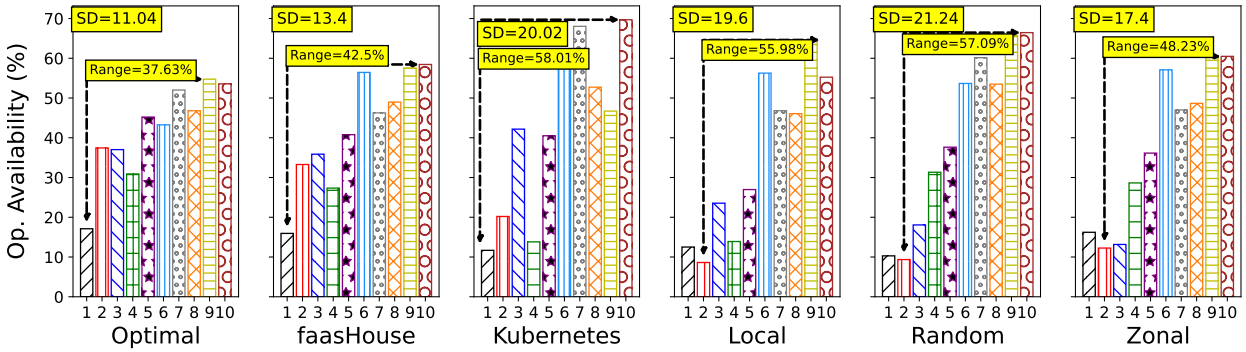


Fig. 4: Operational availability of nodes per scheduler

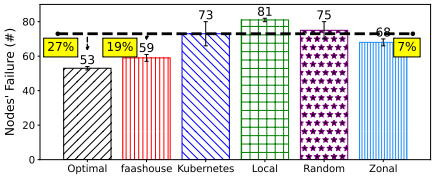


Fig. 5: # of nodes' failure per scheduler

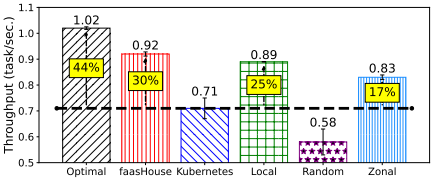


Fig. 7: Throughout per scheduler

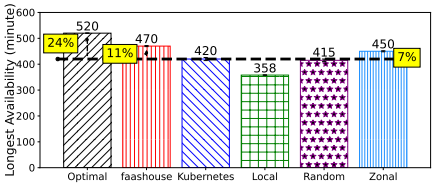


Fig. 6: Longest availability of nodes per scheduler

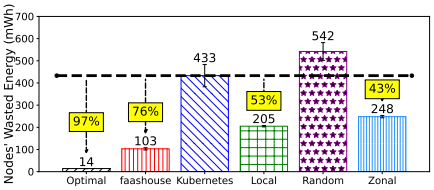


Fig. 8: Nodes' wasted energy per scheduler

a low rate of workload.

By keeping a low-powered node available longer, as in the *optimal* and *faasHouse*, more tasks are generated in the cluster. More generated tasks means more task execution as well, thereby increasing the throughput. This behavior arises from the dual role of nodes: being both load generators and executors. This factor is also a major reason for the *optimal* and *faasHouse* to not increase the average availability of the cluster when utilizing computation offloading. That is, the increased load in the system also escalates the overall energy consumption, which decelerates the increase in average operational availability. The advantage of this increased load generation is that the IoT application serves more tasks and experiences a more desirable Quality of Experience (QoE).

**Response Time:** Fig. 9 shows the CDF of response time per scheduler. All schedulers present a relatively similar trend, mainly due to the local-like network in the extreme edge. The minor difference in obtained response time of schedulers confirms that the balanced energy usage achieved by *optimal* and *faasHouse* is not at the expense of significant QoS degradation (see Fig. 9). The main source of the difference lies in how much a scheduler promotes the local placement of functions. Given this, the *local* scheduler appears to benefit more. The *faasHouse* can encourage local placements by its *locality* plugin if a shorter response time is desired. Furthermore, for offloading-enabled schedulers, another potential source of difference can be the trade-off between throughput and response time. In other words, having fewer hosted functions on under-loaded nodes can lead to a relatively shorter response time. This approach, however, leads to lower throughput, which is in line with the resource-aware nature of the *Kubernetes* scheduler.

### 4.2.2 *Impact of Battery Size*

Figure 10 shows the gradual increase in availability for the node with ***minimum availability*** as the battery size transitions from the smallest to the largest. This observation is evident in the increasing trends of both the *optimal* and *faasHouse* measurements The larger battery storage capacity effectively minimizes energy input wastage, providing a justification for the observed trend. Note that the largest battery size, i.e., 1750 mWh, while consistent in improving the minimum available node, fails to improve the average availability of nodes significantly since the energy input to batteries is limited and not scaled proportionally. The ***nodes' failure*** metric in Figure 11 demonstrates improvement across all schedulers as the battery size increases. However, it is worth noting that the energy-aware schedulers, such as *optimal*, *faasHouse*, and *zonal*, outperform the other schedulers in this regard.

The ***longest availability*** metric depicted in Figure 12 demonstrates an initial substantial increase followed by a more gradual improvement for the energy-aware schedulers., upon increasing the battery size. This slow-down in the increase can be associated with the fact that at some point onward the size of the battery does not matter since

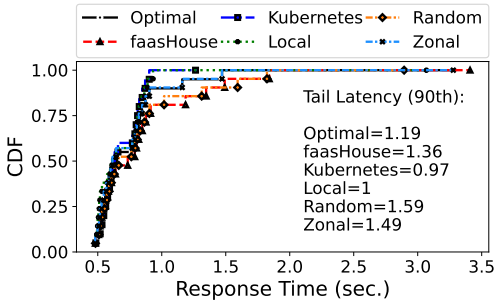


Fig. 9: Response time per scheduler

the energy intake remains unchanged due to the unchanged renewable energy inputs.

The ***throughput*** measure, shown in Fig. 13, demonstrates a linear increase with the enlarged batteries since low-powered nodes can remain available longer and tasks are continuously executed, where the *random* and *local* present the least improvement. This can be associated with the increased opportunity to utilize peers' excess energy. That is, with larger batteries, not only well-powered (and/or under-utilized), but also moderately-powered, nodes are enabled to store more energy at peak time; hence, other nodes have more opportunity to benefit.

For further analysis in terms of other metrics and observations, please refer to Appendix E.

#### 4.2.3 Impact of Workload Intensity

In the main series of experiments, the workload is generated by nodes in the entire experiment time, i.e., intensity=100%, although the request rate is different per node and per application. It is important, however, to observe the performance under varying workload intensity, i.e., 25, 50, 75, and 100%. To achieve intensity=50%, for example, a node's workload generator triggers the function during half the experiment time, in different timeslots that last for a certain time. The trigger times and timeslots are generated using Poisson and Exponential distributions. The overall decreasing trend in Fig. 14 is attributed to the increase in both the workload and consequently overall energy consumption. Maximizing the availability of the ***minimum available node*** shown in Fig. 14 demonstrates the persistent dominance of the *optimal* and *faasHouse*, followed by the *zonal*, schedulers. According to Fig. 14, the minimum available node is observed by the *optimal* scheduler at 22.1%, although the nodes experienced the workload only 25% of the time. This is because the energy input limitation is a key factor and also the base energy usage of nodes appears inevitable. Also, with the increase in workload intensity, the importance of a dynamic energy-aware scheduler is highlighted further. For example, with 25% intensity, the *Kubernetes* scheduler performs only 39% less efficiently than the *optimal*, but this gap increases to 46% when the intensity is 100%.

The ***nodes' failure*** results in Fig. 15 show that by increasing the workload intensity, the number of failures increases for all schedulers. This confirms a more challenging situation for schedulers when experiencing intensive workloads. It is observable that despite such challenges the *optimal* and *faasHouse* retain their dominance by handling intensive workloads more desirably.

The ***longest availability*** measure in Fig. 16 shows that, by increasing the workload intensity, the nodes experience shorter availability at its longest possible period. The *faasHouse* remains performant, with an 11% distance from the *optimal*. The *local* scheduler is the most affected scheduler when the workload is intense since bombarding a recently booted-up node using intensive workloads can more quickly cause the node to fail and restart the timer for measuring the longest availability, which happens to be a recurring accident for nodes when using the *local* scheduler.

The ***throughput*** observations in Fig. 17 show an upward direction by the increase in the workload intensity, as expected. The key observation here is that the variance between different schedulers' throughput is increased as the workload becomes more intense. This once again dictates the necessity of a dynamic energy-aware scheduler for challenging workloads. While the *local* scheduler consistently maintains a reasonable throughput, it falls short in meeting sustainability metrics such as nodes' failure and longest availability.

For further analysis in terms of other metrics and observations, please refer to Appendix F.

#### 4.2.4 Impact of Other Factors

Additionally, we analyzed the impact of varying scheduling intervals and resource capacity (i.e., CPU Governor) on *faasHouse*. For a detailed analysis of such sensitivity analysis, please refer to Appendix G & H. In terms of sustainability metrics, those types of CPU governors that allocate fewer resources to functions, e.g., *powersave*, provide better performance.

## 5 Discussion

In this paper, we introduced an energy-aware scheduler called *faasHouse* to enhance the sustainability of serverless edge computing in extreme edge environments. It is worth noting the following key innovations:

- The problem of imbalanced energy in edge is handled by scheduler software design, which appears much more affordable than dealing with hardware upgrades.
- *faasHouse*, unlike Kubernetes, adopts collective placements—the decision and placement for all nodes and functions are made at once, as opposed to the one-by-one strategy that ignores the requirements of other functions in the scheduling queue, resulting in major conflicts.
- *faasHouse* supports both hard and soft constraints and features an extensible design inspired by the Kubernetes scheduler. This adaptability allows it to meet the evolving requirements of edge applications. Unlike *faasHouse*, the state-of-the-art, e.g., the *zonal* scheduler, generally requires a re-design to either accommodate a different hard/soft constraint than energy or apply to a different application. For example, the *zonal* requires the system admin to empirically determine zones' ranges in advance, which is a non-trivial task, making it challenging to reuse for diverse IoT applications.

***Key Insights:*** While the scheduler proves the applicability of software-based adjustments to improve the environment- and hardware-related challenges, which is

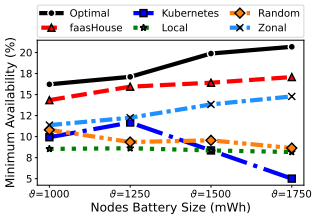


Fig. 10: Battery size vs. minimum available nodes

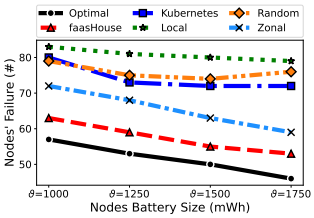


Fig. 11: Battery size vs. nodes' failure

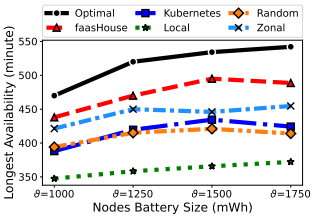


Fig. 12: Battery size vs. longest availability

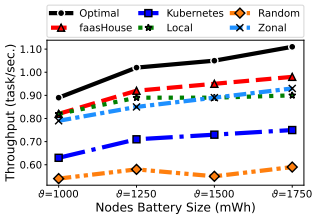


Fig. 13: Battery size vs. throughput

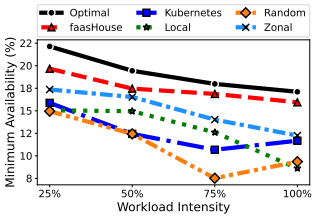


Fig. 14: Workload intensity vs. minimum available nodes

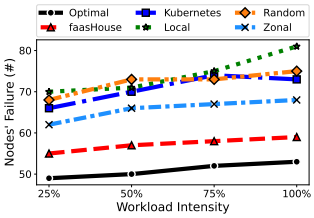


Fig. 15: Workload intensity vs. nodes' failure

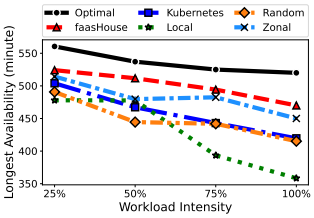


Fig. 16: Workload intensity vs. longest availability

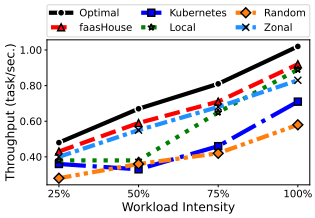


Fig. 17: Workload intensity vs. throughput

much more affordable than hardware-based solutions, there are certain interesting observations worthy of consideration.

- The computation offloading allowed using well-powered nodes to host peers' functions. However, we observed that the operational availability of well-powered nodes is not reduced proportionally to the increased available energy of low-powered nodes. This insight shows that the offloaded functions tend to utilize the excess energy of well-powered nodes which could have been wasted if not used.
- Letting low-powered nodes to remain available for longer durations does not always lead to a proportional increase in the overall operational availability of the cluster. The rationale behind this lies in the improved overall throughput of the cluster and the generation of more tasks. In extreme edge computing, where edge nodes have a dual-role of task generation and execution, prolonging the availability of a node results in an influx of additional tasks into the cluster for execution. This, in turn, leads to an overall enhancement of performance and quality of service (QoS).

***Scalability:*** The *faasHouse* is meant for IoT applications that may scale up to hundreds of nodes, e.g., Smart Farming [16] and Smart Manufacturing [3]. In terms of the design, the computational complexity of the assignment algorithm is $O(R \log R + R(n + m\aleph))$ which is tractable even when the number of functions to be assigned is in the scale of thousands. In our experiments, *faasHouse* imposed only around 1% computation overhead on the demonstrated cluster. In terms of the implementation, *faasHouse* requires a container orchestrator, Kubernetes, that at the time of writing this paper, allows scaling the cluster to 5000 nodes, with each of them running up to 110 pods (a pod can hold multiple containers). Please note that scalability experiments are not conducted in this paper, as we are limited by the real testbed implementation cost and space.

Technically, the controller node might be a potential bottleneck of the system leading to failure or saturation. For failure concerns, the controller is backed with failure handling and high availability mechanisms in Kubernetes such that the leadership can be shared among multiple controllers. For saturation concerns that can occur due to heavy communication between nodes or lack of coverage for the extended areas, leveraging multiple clusters of Kubernetes appears a reasonable solution, as supported by Liqo.[6] Lastly, a decentralization of the scheduler appears worthy of consideration as a potential avenue for future work.

## 6 Related Work

Since there is limited work focused on resource scheduling of sustainable serverless edge, we broadly discuss the literature on the resource scheduling of sustainable edge.

### 6.1 Resource Scheduling in Sustainable Edge

Jiang et al. [7] survey a considerable number of works on energy awareness for edge computing, pinpointing the following aspects: (a) hardware choice, (b) computing architecture, (c) operating system, (d) middleware, (e) microservices or functions, and (f) computation offloading.

(a) The importance of hardware choice, including attached devices such as batteries, is of concern where we, similar to [4], consider the support for alternative energy supplies such as renewable solar irradiation [7].

(b) Our proposed architecture is supported by energy benchmarks and measurements provided by *WattEdge* [2] which are found critical for practical designs [7] that incorporate energy usage by CPU as the dominant and energy storage by a battery [2].

(c) The operating system and (d) middleware are the most relevant considerations in our proposal. In [25], scheduling improvements are practiced by energy-aware tuning of CPU frequency by machine learning models, which appear computation-hungry for the resource-constrained extreme edge, whereas our proposal solves the problem with a reasonable computation complexity.

6. https://liqo.io/

(e) Microservices are treated differently in our work than in the literature by adopting the FaaS model. This facilitates the migration process (i.e., moving from one node to another), encouraged by [7]. Also, to minimize the interruptions over migrations, our solution practices graceful terminations—launch the new container and then terminate the old one, which is not typically practiced in the literature [4], [26].

Finally, (f) computation offloading, a key contribution of our work, is an ideal opportunity to achieve energy awareness at the edge [7]. A simulation-based solution is practiced in [27], but energy harvesting by mobile devices from peers makes the problem distinctly different than ours which attempts to exploit renewable energy sources.

### 6.2 Scheduling in Serverless Cloud Computing

Studies on resource scheduling of serverless functions in clouds are primarily focused on non-energy matters [28]–[31]. For instance, memory and cache improvements have been implemented by a scheduler to enhance the cache hit ratio [28], [29]. Contrary to these approaches, we focus on energy challenges at the extreme edge. For instance, we simply eliminate container caching effects by pre-caching. Lastly, Xtract [31] is a scalable high-performance platform, which is an extension of funcX [30], and adopts a random placement of functions.

### 6.3 Scheduling in Serverless Edge Computing

Studies on scheduling problems in serverless edge computing have primarily focused on resource allocation, QoS, and cost efficiency [5], [13], [32]–[34]. However, energy considerations have been overlooked to a large extent. LaSS [34] takes care of latency-sensitive workflows by vertically adjusting the functions' allocated resources to the demand. Similarly, LETO [32], a theoretical solution, considers the horizontal scaling of resources. In [33], the QoS and cost efficiency is approached through task offloading in a cloud-to-edge continuum, where they adopt Pi 3 B+ as edge nodes, as we did. With cost efficiency in mind, ActionWhisk [5] exploits computation offloading. In [13], migrating both task and function in a cluster of heterogeneous nodes, including Pis, is studied to improve the QoS.

### 6.4 Energy-aware Scheduling in Serverless Edge

There is a significant gap in the existing literature regarding the energy awareness of serverless platforms. Discussions on sustainable serverless at the edge were commenced by [6], and we initiated practical investigations in [12]. The present work builds upon our previous study in [12] and introduces significant advancements in several key aspects. Respectively, we introduce a practical architecture for the serverless edge. The problem context is defined on top of [12], where we here incorporate new factors into (a) the edge nodes, i.e., the inclusion of CPU governors, (b) application & workload, i.e., the inclusion of bandwidth and data transfer, and (c) node availability, i.e., the inclusion of standby energy considerations. The algorithm design presents a completely novel and distinctive approach, prioritizing practicality and extensibility, inspired by Kubernetes scheduler's design, a first-class scheduling tool at the edge [2], [5], [12]–[14]. The prototype platform of [12] is extended to incorporate our new algorithms. In addition, this work encompasses a completely new set of experiments, focusing on sustainability-related metrics and parameters for evaluation. Furthermore, Spillner et al. [6] suggested the need for energy-aware serverless edge computing. They theoretically analyzed viable techniques such as resource scheduling, server consolidation, dynamic resource scaling, and utilizing artificial intelligence. However, the suggestions lack an implementation or simulation.

## 7 Conclusions and Future Works

In this paper, we investigated sustainable edge computing augmented with serverless through resource scheduling, in the presence of variable renewable energy input and workload. A realistic architecture and system model for sustainable edge was presented where an energy-aware scheduler called *faasHouse* was proposed to address the challenge of imbalanced energy supply for renewable energy and battery-powered edge clusters. The experimental results demonstrate notable improvements in various aspects: (a) a 46% increase in the utilization of the least available nodes, (b) a 27% reduction in node failure rates, (c) a 24% improvement in the reliability of edge nodes, and (d) a 97% minimization of wasted renewable energy. These advancements were achieved through the implementation of a software-based solution for dynamic energy-aware scheduling. This approach eliminates the need for hardware adjustments while yielding substantial benefits. Moreover, it promotes the environmentally friendly deployment of modern computing models, such as serverless computing, at the edge. Future work includes extending the system model to support a continuum of cloud-to-edge nodes for a broader range of IoT applications, including cost-aware ones. Additionally, exploring other adaptation techniques like load balancing and resource consolidation, as well as incorporating heterogeneous resources (e.g., GPU, TPU), can enhance the scheduler's capabilities.

## References


[1] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[2] M. S. Aslanpour, A. N. Toosi, and R. Gaire, "WattEdge : A Holistic Approach for Empirical Energy Measurements in Edge Computing," vol. 2. ICSOC, Springer, 2021, pp. 531–547.

[3] T. Ojha, S. Misra, and N. S. Raghuwanshi, "Internet of Things for Agricultural Applications: The State-of-the-art," *IEEE Internet Things J.*, p. 1, 2021.

[4] A. Karimiafshar, M. R. Hashemi, M. R. Heidarpour, and A. N. Toosi, "Effective Utilization of Renewable Energy Sources in Fog Computing Environment via Frequency and Modulation Level Scaling," *IEEE Internet Things J.*, vol. 7, no. 11, pp. 10 912–10 921, 2020.

[5] D. Bermbach, J. Bader, J. Hasenburg, T. Pfandzelter, and L. Thamsen, "AuctionWhisk: Using an Auction-Inspired Approach for Function Placement in Serverless Fog Platforms," pp. 1–48, 2021.

[6] Panos, J. Spillner, A. V. Papadopoulos, O. Rana, P. Patros, J. Spillner, A. V. Papadopoulos, B. Varghese, O. Rana, and S. Dustdar, "Toward Sustainable Serverless Computing," *IEEE Internet Computing*, vol. 25, no. 6, pp. 42–50, 2021.

[7] C. Jiang, T. Fan, H. Gao, W. Shi, L. Liu, C. Cérin, and J. Wan, "Energy aware edge computing: A survey," *Computer Communications*, vol. 151, no. 2018, pp. 556–580, 2020.

[8] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource Scheduling in Edge Computing: A Survey," *IEEE Communications Surveys and Tutorials*, vol. 48202, no. c, pp. 1–36, 2021.

[9] Y. Li, Y. Lin, Y. Wang, K. Ye, and C.-Z. Xu, "Serverless Computing: State-of-the-Art, Challenges and Opportunities," *IEEE Trans. Services Comput.*, vol. 1374, no. c, pp. 1–1, 2022.

[10] M. Aslanpour, A. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. Gill, R. Gaire, and S. Dustdar, "Serverless Edge Computing: Vision and Challenges," in *Australasian Computer Science Week Multiconference*, 2021.

[11] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[12] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and R. Gaire, "Energy-Aware Resource Scheduling for Serverless Edge Computing," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 190-199, doi: 10.1109/CCGrid54584.2022.00028.

[13] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[14] L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "NetMARKS: Network metrics-AwaRe kubernetes scheduler powered by service mesh," *Proceedings - IEEE INFOCOM*, vol. 2021-May, 2021.

[15] David W. Pentico, "Assignment problems: A golden anniversary survey," *European Journal of Operational Research*, pp. 774–793, 2007

[16] R. Mahmud and A. N. Toosi, "Con-Pi: A Distributed Container-Based Edge and Fog Computing Framework," *IEEE Internet Things J.*, vol. 9, no. 6, pp. 4125–4138, 2022.

[17] M. S. Aslanpour, A. N. Toosi, R. Gaire, and M. A. Cheema, "Auto-scaling of Web Applications in Clouds: A Tail Latency Evaluation," in *2020 IEEE/ACM 13th UCC*. IEEE, dec 2020, pp. 186–195.

[18] T. Sönmez and M. U. Ünver, "House allocation with existing tenants: A characterization," *Games and Economic Behavior*, vol. 69, no. 2, pp. 425–445, 2010.

[19] T. Öncan, "A survey of the generalized assignment problem and its applications," *Infor*, vol. 45, no. 3, pp. 123–141, 2007.

[20] T. Sönmez and M. U. Ünver, "Matching, allocation, and exchange of discrete resources," *Handbook of Social Economics*, vol. 1, no. 1 B, pp. 781–852, 2011.

[21] "Bureau of Meteorology." [Online]. Available: http://www.bom.gov.au/climate/data-services/solar-information.shtml.

[22] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, "The State of Serverless Applications: Collection, Characterization, and Community Consensus," *IEEE Trans. Softw. Eng.*, vol. 5589, no. c, pp. 1–1, 2021.

[23] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. & Berg, A. SSD: Single Shot MultiBox Detector. *Computer Vision – ECCV 2016*. pp. 21-37 (2016)

[24] S. Tuli, G. Casale, and N. R. Jennings, "PreGAN: Preemptive Migration Prediction Network for Proactive Fault-Tolerant Edge Computing," in *Proceedings - IEEE INFOCOM*, 2022.

[25] Q. Zhang, M. Lin, L. T. Yang, Z. Chen, S. U. Khan, and P. Li, "A Double Deep Q-Learning Model for Energy-Efficient Edge Scheduling," *IEEE Trans. Services Comput.*, vol. 12, no. 5, pp. 739–749, 2019.

[26] S. Ghanavati, J. H. Abawajy, and D. Izadi, "An Energy Aware Task Scheduling Model Using Ant-Mating Optimization in Fog Computing Environment," *IEEE Trans. Services Comput.*, pp. 1–1, 2020.

[27] W. Chen, D. Wang, and K. Li, "Multi-User Multi-Task Computation Offloading in Green Mobile Edge Cloud Computing," *IEEE Trans. Services Comput.*, vol. 12, no. 5, pp. 726–738, 2019.

[28] P. Andreades, K. Clark, P. M. Watts, and G. Zervas, "Experimental demonstration of an ultra-low latency control plane for optical packet switching in data center networks," *Optical Switching and Networking*, vol. 32, pp. 51–60, 2019.

[29] G. Aumala, E. Boza, L. Ortiz-Aviles, G. Totoy, and C. Abad, "Beyond load balancing: Package-aware scheduling for serverless platforms," *Proceedings - 19th IEEE/ACM CCGrid 2019*, pp. 282–291, 2019.

[30] R. Chard, A. Woodard, I. Foster, and K. Chard, "f unc X : A Federated Function Serving Fabric for Science," pp. 65–76, 2020.

[31] T. J. Skluzacek, R. Wong, Z. Li, R. Chard, K. Chard, and I. Foster, "A Serverless Framework for Distributed Bulk Metadata Extraction," *HPDC 2021 - Proceedings of the 30th HPDC* , pp. 7–18, 2021.

[32] H. Ko, S. Pack, and V. C. M. Leung, "Performance Optimization of Serverless Computing for Latency-Guaranteed and Energy-Efficient Task Offloading in Energy Harvesting Industrial IoT," *IEEE Internet Things J.*, vol. 4662, no. c, pp. 1–1, 2021.

[33] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance Optimization for Edge-Cloud Serverless Platforms via Dynamic Task Placement," *Proceedings - 20th IEEE/ACM CCGRID 2020*, no. 1, pp. 41–50, 2020.

[34] B. Wang, A. Ali-Eldin, and P. Shenoy, "LaSS: Running Latency Sensitive Serverless Computations at the Edge," *HPDC 2021 - Proceedings of the 30th HPDC* , pp. 239–251, 2021.


**Mohammad Sadegh Aslanpour** is a PhD student at Monash University and CSIRO's DATA61, Australia. His research interests include Cloud, Edge, and Serverless Computing—where self-adaptive, efficient, and green solutions are desired. He is a recipient of the best paper award/nomination from AusPDC & ICSOC conferences; as well as research funding, grant, and scholarships from Monash University, CSIRO, and Building 4.0 CRC.

**Adel N. Toosi** (Member, IEEE) received his PhD degree from University of Melbourne, in 2015. He is currently a senior lecturer with the Department of Software Systems and Cybersecurity, Monash University, Australia. From 2015 to 2018, he was a postdoctoral research fellow with the University of Melbourne. Dr Toosi has published over 70 peer-reviewed publications in top-tier venues such as IEEE TCC, IEEE TSC, and IEEE TSUSC. His publications have received over 3,500 citations with a current h-index of 28 (Google Scholar). His research interests include cloud, fog, edge computing, software-defined networking, and sustainable computing. For further information, visit his homepage: http://adelnadjarantoosi.info.

**Muhammad Aamir Cheema** is an ARC Future Fellow and Associate Professor at the Faculty of Information Technology, Monash University, Australia. He obtained his PhD from UNSW Australia in 2011. He is the recipient of 2012 Malcolm Chaikin Prize for Research Excellence in Engineering, 2013 Discovery Early Career Researcher Award, 2014 Dean's Award for Excellence in Research by an Early Career Researcher, 2018 Future Fellowship, 2018 Monash Student Association Teaching Award and 2019 Young Tall Poppy Science Award. He has also won two CiSRA best research paper of the year awards, two invited papers in the special issue of IEEE TKDE on the best papers of ICDE, and three best paper awards at ICAPS 2020, WISE 2013 and ADC 2010, respectively.

**Mohan Baruwal Chhetri** is a Senior Research Scientist with CSIRO's Data61, Australia, where he leads the Human-Machine Collaboration for Cybersecurity research theme. He received his PhD degree from Swinburne University of Technology, Australia. His research focus is on developing technologies to facilitate decision support, automation and optimization in cyber-physical-social ecosystems.