Load Balancing for Heterogeneous Serverless Edge Computing: A Performance-Driven and Empirical Approach

Mohammad Sadegh Aslanpour[a,b], Adel N. Toosi[a], Muhammad Aamir Cheema[a], Mohan Baruwal Chhetri[b], Mohsen Amini Salehi[c]

[a]*Department of Software Systems and Cybersecurity, Monash University, Clayton, 3800, VIC, Australia*
[b]*CSIRO's Data61, Clayton, 3168, VIC, Australia*
[c]*Computer Science and Engineering, University of North Texas, Denton, 76201, Texas, USA*

Abstract

Serverless edge systems simplify the deployment of real-time AI-based Internet of Things (IoT) applications at the edge. However, the heterogeneity of edge computing nodes—in terms of both hardware and software—makes load balancing challenging in these systems. In this paper, we propose a performance-driven, empirical weight-tuning approach to achieve effective load balancing based on the characteristics and capabilities of the nodes. By extensively profiling the nodes, we gather knowledge on performance metrics such as throughput, energy efficiency, response time, AI accuracy, and cost. Using this acquired knowledge, we introduce a weighted round-robin strategy to optimize the performance metrics according to their observed significance. To address multiple objectives, we introduce a multi-objective method that aims to strike a balance between any arbitrary set of performance objectives simultaneously. Additionally, we explore a coordinated distributed approach to overcome the limitations of centralized load balancing. Next, we introduce *Hedgi*, a heterogeneous serverless edge architecture designed to efficiently configure and utilize the derived load balancing policies, validated empirically. To demonstrate the practicality of Hedgi, we containerize and serverlessize a real-time object detection application. Extensive empirical studies are conducted using *Hedgi* to evaluate the performance of the proposed load balancing approach. The results provide valuable insights into the design trade-offs of various load balancing policies and system designs in the heterogeneous serverless edge.

Keywords: heterogeneity, serverless, function-as-a-service, edge computing, load balancing, performance

1. Introduction

The rapid rise of serverless edge computing [1, 2] can be attributed to the increasing demand for real-time data processing and the ever-growing need for simplified application development and deployment in this context [3]. This is particularly relevant in domains such as the Internet of Things (IoT), Augmented Reality (AR) or Virtual Reality (VR), and Artificial Intelligence (AI) [4, 5]. The primary goal is minimizing latency and bandwidth usage via performing computations closer to the data source or user, eliminating the need to transmit large amounts of data to a centralized server. It also promotes a serverless architecture, simplifying application deployment and scalability, thereby, reducing operational costs and improving efficiency [6].

Thanks to advancements in hardware and machine learning methodologies, AI-based IoT applications are increasingly being deployed on edge nodes [7, 8, 9]. It has now become feasible to develop lightweight AI models that operate on single-board computers (SBCs) like Raspberry Pi. One compelling use case that exemplifies the progress in AI-based IoT applications on the edge is real-time video analytics. For instance, in industrial manufacturing, real-time image processing at the edge can be used to identify faulty products. Similarly, in a surveillance system, lightweight object detection models like YOLO (You Only Look Once) [10] or SSD (Single Shot MultiBox Detector) [11] can be trained on a powerful machine and then deployed on an SBC, taking advantage of hardware acceleration if available.

Heterogeneity is inevitable in edge computing due to its distributed nature and gradual evolution [7]. Such environments often consist of diverse devices, ranging from resource-constrained sensors and IoT devices to more powerful edge servers and gateways. Heterogeneity is not limited to the *hardware* alone, i.e., the processor type, e.g., ARM or X86, but also applies to the *platform*, including the OS, virtualization runtime, orchestration tools, and the *software*, i.e., the application requirements, programming languages, and runtime [12]. Furthermore, when considering AI-based IoT applications, it is also important to consider that hardware heterogeneity extends beyond CPU architecture and includes accelerators such as GPU and TPU.

Building serverless edge computing over a heterogeneous cluster of edge nodes presents formidable challenges due to the diverse characteristics of the nodes [13]. As an

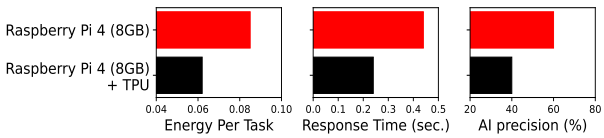

Figure 1: Comparison of Two Heterogeneous Edge Devices.

illustrative example, consider a smart city in which various IoT devices and sensors are deployed throughout the urban landscape to monitor and manage various aspects of city life, such as traffic flow, environmental conditions, waste management, public safety, and more. In this city-wide scenario, the edge computing infrastructure spans diverse devices and systems, ranging from low-power sensors for environmental monitoring to high-powered edge servers and gateways for data processing and analytics. Variations exist in computing power, memory, energy efficiency, and supported platforms across these devices.

Orchestrating and managing resources, configurations, and capabilities across these nodes necessitates specialized techniques to ensure efficient and reliable operation. One crucial challenge stemming from this heterogeneity is ensuring optimal application performance or QoS by effectively distributing the workload across serverless functions running on edge nodes [14]. Despite the importance of this issue, the current default load balancing methods in serverless domains are insufficient in effectively addressing this, as they are often designed without considering heterogeneity.

Fig. 1 demonstrates the significance of this problem by presenting a simplified scenario where a serverless AI-based IoT application – Single Shot MultiBox Detector (SSD [11]) – is evaluated on a single Raspberry Pi 4, both with and without the inclusion of a TPU accelerator. The results unequivocally indicate that neither setup can achieve optimal performance across all metrics, illustrating the presence of a trade-off. Note that while the same model is used, the TPU-enabled device consumes less energy and provides shorter response time, while the non-TPU device provides a considerably higher AI precision. TPU supports lower-precision floating-point formats, enabling faster processing with a lower AI precision in the SSD model compared to the CPU. The load balancing problem in heterogeneous serverless edge environments poses significant challenges, as it becomes impossible to optimize all performance metrics simultaneously [15]. Gaining a better understanding of the implications of heterogeneity in load balancing problems in serverless edge environments is crucial to achieving optimal or near-optimal solutions. The main objective of this paper is to develop such an understanding.

Users seek to optimize their systems based on various metrics, taking into account different edge node types that exhibit varying performance characteristics, such as throughput, energy efficiency, precision, and more. We hypothesize that by understanding and learning these traits unique to each node and incorporating this knowledge into the load balancer (LB), we can enhance the overall system performance. The primary objective is to assign weights effectively to individual nodes, allowing for a proportional distribution of the workload based on each node's specific performance capabilities. This allocation strategy plays a pivotal role in achieving optimal overall system performance.

This paper makes the following key contributions to the load balancing of serverless functions in heterogeneous environments:

- We devise a data-driven profiling strategy for performance characterization of edge nodes, focusing on metrics such as throughput, energy efficiency, responsiveness, AI inference precision, and cost efficiency, all in this work. Building upon this characterization and using a performance-driven weight-tuning approach, we develop a range of intuitive load balancing policies that enable the system to achieve diverse performance objectives for achieving effective load balancing among serverless functions in heterogeneous environments.

- We address the need for multi-objective and priority-based load balancing of serverless functions by employing a weighted sum method. This is achieved by the Pareto Optimality to determine the appropriate weights for multi-objective policies.

- We address the unreliability of the centralized load balancer, commonly designed for cloud environments that shows scalability constraints. To achieve so, we propose an effective distributed architectural design for the serverless edge, aiming to significantly improve the overall system performance.

- We address the limited knowledge in the design and implementation of a heterogeneous serverless edge computing architecture, by proposing *Hedgi*. As a framework for our experiments, *Hedgi* supports heterogeneity of serverless edge across hardware, platform, and software. Additionally, an AI-based data-intensive IoT application for object detection in images (video frames), a common task in many IoT applications, is developed for conducting experiments on *Hedgi*.

- We implement the load balancing policies on *Hedgi* and thoroughly analyze their practical behavior. Extensive empirical evaluations are performed using low-level resource metrics (energy use, throughput, energy efficiency, CPU use, memory use, and bandwidth use), high-level application metrics (AI precision, generated requests, success rate, latency, queuing time, and tail latency), and hybrid metrics (serverless cost).

Overall, this research contributes to the advancement of load balancing techniques for real-time AI-based IoT applications at the edge, addressing the challenges posed by heterogeneity and enhancing the efficiency and effectiveness of serverless edge systems.

2. Related Work

Our review of the literature reveals the gap in knowledge in the context of load balancing for heterogeneous serverless edge computing. Please refer to Table 1 for a summary of the related work.

To begin with, specialized system design for load balancing of serverless edge computing appears to be underinvestigated. Instead, there has been considerable research on system design and optimization for other components such as auto-scalers and function schedulers in serverless platforms. For example, in [16, 17, 18, 5], different system designs are proposed and experimented with to improve the performance of auto-scalers in serverless platforms. Function placement, i.e., container scheduling, in serverless has also received considerable attention in [12, 19, 20]. These studies investigate different heuristics to achieve a trade-off between different performance metrics in the presence of heterogeneous resources for data-intensive applications. However, a specific focus on load balancing for heterogeneous serverless edge computing remains relatively limited in the literature.

Another key distinction between our work and such proposals is our focus on the IoT domain, particularly with AI applications, where the performance objectives can expand beyond latency and throughput. In our approach, we consider the energy implications of the serverless at the edge, which is crucial for energy-constrained use cases in IoT deployments. Additionally, we take into account AI-related metrics, such as the precision of inferences, which provides valuable insights into understanding the implications of heterogeneity when AI applications are utilized at the edge. Furthermore, as serverless provides its own pricing model, we include the function cost as another indicator in our load balancing strategy. All of these inclusions can provide useful insights for system developers and practitioners.

Several works have attempted to address the load balancing of serverless edge systems, each focusing on different aspects of heterogeneity and performance optimization. In [4, 21], the focus is on latency-aware load balancing for IoT applications, particularly in real-time systems. In [22, 23], the emphasis is on augmenting latency awareness with network-level information and also considering user mobility. In [24], the authors focus on the architecture of serverless edge while dealing with load balancing. They argue that the central design of serverless systems needs to be upgraded to a decentralized system to better suit the requirements of edge environments. In [25], the central design of serverless is challenged even further, and the authors propose load balancing strategies for multi-agent systems using simulations. In [26], the opportunities of utilizing AI-based IoT applications, such as video processing at the edge, are explored. In contrast to these works, our approach attempts to emphasize the heterogeneity present in different layers of the serverless edge, ranging from hardware to platform and software. We aim to develop load balancing policies that not only consider latency-awareness, but also take into account several performance metrics such as cost, throughput, energy, and AI inference precision.

Our approach incorporates the heterogeneity in all the layers of the system, from hardware to platform and software. To the best of our knowledge, there are no other works that satisfy all such considerations at once, which is deemed a gap in the knowledge. For instance, in [27, 23, 28], hardware heterogeneity is explored to enhance load balancing, but their primary focus is on security and privacy objectives.

There are other attempts for realizing serverless at the edge, both theoretically [9] and empirically [29, 30, 31], that provide benchmark suites and conduct performance evaluations. Proposals for serverless platforms [32] and architectures [33, 34, 35] are other examples of such efforts. The possibility of utilizing AI-based applications at the edge by serverless is examined in [36] for image classifications.

While the merits of serverless at the edge have been sufficiently established, we believe that specific components such as the load balancer require special considerations to adapt to this new domain. As serverless edge systems embrace heterogeneity in hardware, platform, and software, the load balancing mechanism must be designed to effectively utilize the diverse capabilities of the nodes. Our focus is on developing a performance-driven, empirical weight-tuning approach that optimizes load balancing based on the characteristics and capabilities of individual nodes. By fine-tuning the load balancer to handle the unique challenges posed by the heterogeneity of the edge environment, we aim to enhance the overall performance and efficiency of serverless edge systems.

3. Load Balancing and Weight Tuning Problem

Here, we first elaborate on what load balancing looks like in the context of AI-based IoT applications. Then, we introduce the load balancing problem arising in this context that needs to be addressed.

3.1. Load Balancing AI-based IoT Applications

The proposed data pipeline [5] for AI-based IoT applications, as depicted in Fig. 2, includes eight steps. This pipeline is a common practice at the edge given the event-driven nature of IoT applications and it also eliminates the need for carrying the data along with requests over the network [8].

Table 1: Summary of Related Work and Their Limitations.

Ref.	Context					App.	Contributions			
	Heterogeneous									
	Hardware		Platform	Software	Serverless Edge	AI-based IoT	Architecture	Characterization	Load Balancing	All-levels Metrics
	CPU	Accel.								
[12]	✓	✓		✓	✓	✓				✓
[16, 37, 18]	✓				✓			✓		
[22, 24, 25, 4, 21, 38]	✓				✓				✓	
[27, 23, 28]	✓				✓					✓
[39]	✓	✓		✓	✓	✓		✓		
[32, 33, 34, 40]	✓		✓		✓		✓			
[5, 19, 20, 36, 30, 9, 35]	✓				✓					
[26]	✓			✓	✓	✓			✓	
[31]	✓		✓		✓		✓			✓
Ours	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

① The *Sensor* on the edge node generates sensor data, such as images taken by a camera, ② which is subsequently stored in the *Object Storage*. The storage process generates a reference identifier and triggers an event to the *Event Bus*. ③ The *Event Bus* looks up the corresponding function API endpoint for the processing and ④ invokes the function by sending an HTTP request to the gateway, where the load balancer logic is implemented. Next, ⑤ the gateway calls the function replica (each replica has a unique IP). ⑥ The function fetches the data (e.g., the image) from the data host's object storage, given the received data identifier in the HTTP call. ⑦ The function executes the processing task (e.g., the object detection) on the defined processor or accelerator. Lastly, ⑧, once the processing is completed, the output (e.g., detected object lists) is delivered to the data host for recording or actuation.

Remark: In a generic FaaS model [41], each function invocation triggers a new function replica, i.e., a container, so-called horizontal per request scaling. This causes a long, instantiation time for each container, e.g., a few seconds, to load the AI inference model and become ready for a request that will last only a few milliseconds. To overcome this, cloud providers practice reusing function replicas, i.e., warm functions [8], but horizontal scaling (i.e., adding replicas) and duplicating the container runtime per request appear resource inefficient. Therefore, we disable general-purpose auto-scaling on the serverless platform and allow a single-replica function to vertically scale out, in compliance with resource efficiency at the edge. This is achieved through leveraging the kubernetes built-in auto-scaling that allows functions to define the maximum resource it requests to be provisioned given the function's load increase. This imposes no overhead on our system since the enlargement of a function towards its maximum capacity is seamlessly handled by Kubernetes and no interruption, such as redeployment, in the function performance is imposed.

3.2. Weight Tuning Problem

In the context of the heterogeneous serverless edge, effectively balancing the workload across nodes poses a major challenge in load balancing. A reasonable strategy is to assign each node a subset of tasks, proportional to its performance. However, this demands the load balancer to know how performant each node is. The key objective is to allocate weights optimally among nodes, enabling proportional distribution of the workload based on each individual node's performance. This allocation strategy is crucial for achieving optimal overall performance. In this section, we provide a formal definition of the weight-tuning problem that load balancers must tackle in order to achieve the desired state of optimality.

Consider a set of edge nodes, which we refer to as D. Each node within this set is individually labeled with the index d_i. Within this context, the performance of any single node d_i in isolation is denoted as $P_{i,j}$, in relation to a specific performance metric m_j. Note that m_j is an integral part of a larger set M, which encapsulates a variety of performance metrics we might be interested in, ranging from energy consumption and throughput to AI precision, among other parameters.

Our primary objective is to determine the optimal weight $w_{i,j}$ for each edge node d_i for a given metric m_j to maximize overall performance with respect to m_j. Given m_j, this can be formally represented as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{|D|} w_{i,j} \cdot P_{i,j} \\ \text{subject to} \quad & \sum_{i=1}^{|D|} w_{i,j} = 1 \\ & w_{i,j} \geq 0, \quad \forall i \in D. \end{aligned}$$

Determining weights in this optimization problem is not a trivial task, as it requires understanding the performance characteristics of each node and how they impact the overall system's performance. Additionally, workloads in real-world IoT systems experience a varied rate of concurrency on each node, which increases the curse of dimensionality to the problem.

4. Proposed Performance-Driven Load Balancing

This section presents our approach to performance-driven weight tuning for addressing the load balancing problem. The approach involves first characterizing different performance metrics and quantifying the performance for each edge node. This provides insights into the design of load balancing policies. A weighted round-robin scheduling algorithm incorporates the obtained weights to achieve the performance objectives at runtime. Furthermore, our approach extends to the multi-objective domain, addressing the need for optimizing multiple performance

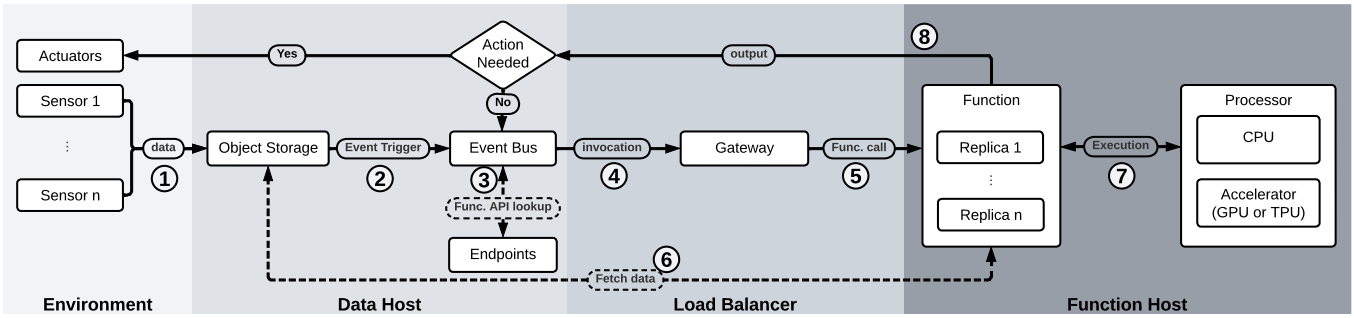

Figure 2: AI-based IoT Application Data Pipeline.

metrics. To enhance the overall efficiency of load balancing, we also introduce architectural adjustments that align with the specific requirements of the serverless edge environment. By combining these strategies, our load balancing approach aims to deliver a performance-driven solution, enabling seamless operation and resource optimization for serverless functions in the heterogeneous serverless edge environment.

4.1. Performance Characterization

Here, we elaborate on the proposed performance characterization procedure. Based on the literature [15], we study some of the most common metrics used for assessing the node's performance, listed in Table 2. The same approach can be applied to other metrics[1].

Let M be an ordered set of metrics. In our performance characterization, $M = \{$*throughput, energy per request, response time, AI-precision, function cost*$\}$. Each metric $m_j \in M$ has a unit of measurement. U denotes an ordered set of units where $u_j \in U$ is the unit of the metric $m_j \in M$. Here, $U = \{$*request per second, mWh per request, seconds, precision, memory-seconds*$\}$, e.g., the energy per request is measured in mWh per request. We use D to denote the set of edge nodes. We employ five sample heterogeneous commonly-used SBCs, whose specifications are listed in Table 3, as edge nodes for our investigations [12, 8]. So, $D = \{$*raspberry pi 4(4gb), raspberry pi 4(8gb), raspberry pi 4(8gb)+tpu, jetson nano, raspberry pi 3,*$\}$[2].

Our objective is to evaluate the performance characteristics of devices in D in terms of different metrics in M. To achieve this, we conduct profiling experiments for each individual device in isolation mode [23]. All nodes are tested individually, running the same application under heterogeneous conditions, particularly different runtime environments, as specified in Table 3. In this mode, a device generates and sends HTTP requests to a function deployed on the same device (node) for AI inferences. This allows us to observe the impact of heterogeneous nodes when dealing with the same inference task. The requests are sent synchronously, one after the other, with each request waiting for a response before sending the next one. Profiling is conducted for a sufficient amount of time and repeated multiple times to obtain reliable indicators.

The results obtained from the profiling experiments are denoted by R. Each $r_{i,j} \in R$ denotes the result for a specific metric $m_j \in M$ and a device $d_i \in D$, e.g., the throughput for the Jetson Nano. It is important to note that $|R| = |M| \times |D|$. We observe that for some metrics, larger values are preferable (e.g., throughput), whereas, for others, smaller values are more desirable (e.g., response time). Consequently, we first convert the result $r_{i,j}$ into a characteristic value $c_{i,j}$, ensuring that a larger $c_{i,j}$ is always better for any $m_j \in M$. We achieve this by utilizing an ordered set of objectives O that indicates whether a transformation is required. For our selected metrics, $O = \{0, 1, 1, 0, 1\}$. This implies that no transformation is necessary for throughput and AI precision, but for other metrics where lower values are better, a transformation is required.

Each metric corresponds to a characteristic, as stated earlier. For our selected set of metrics, the characteristics are named as $\{$*throughput, energy efficiency, responsiveness, AI-precision, cost efficiency*$\}$ after the transformation. We utilize R and O to derive $c_{i,j}$ a transformed value for each $r_{i,j} \in R$ as follows:

$$c_{i,j} = \begin{cases} \dfrac{1}{r_{i,j}} & \text{if } o_j = 1 \\ r_{i,j} & \text{otherwise} \end{cases} \tag{1}$$

4.2. Weights Tuning

After obtaining $c_{i,j}$'s, the subsequent step involves converting these values into load balancing weights for the load balancer. The goal is to assign a weight value to each

[1]Note that some aspects of the system can be disregarded as they remain constant or affect all nodes equally, despite their heterogeneity. For instance, the base power consumption of the nodes, which includes the power consumed in an idle state, energy usage by USB ports, fans, and load generators, remains constant throughout the lifetime of the system. Similarly, the cost associated with bandwidth use, storage upload/download, etc., affects all nodes equally.

[2]Note that the Raspberry Pi 4 (4GB) is revision 1.1 while the Raspberry Pi 4 (8GB) is revision 1.4 of the hardware with an improved power circuit and CPU overclocking increased from 1.5 to 1.8 GHz.

Table 2: Description of Characterization Metrics.

Metric	Description
Throughput	The number of requests executed per (total) time unit [42].
Energy per Task	The amount of energy used to process a request, excluding the base energy use.
Response Time	The duration from sending a request to receiving the response (end-to-end processing).
AI Precision	The precision of the AI task (e.g., image recognition) that the device provides.
Function Cost	A FaaS-based pricing model that measures memory usage multiplied by the function execution time per task.

node proportional to its performance for the specific characteristic relative to other nodes in the cluster. Let W be the set of determined weights: $W = \{w_{i,j}, \forall d_i \in D, \forall m_j \in M\}$. For a given m_j, weight $w_{i,j} \in W \forall d_i \in D$ is derived using the following formula:

$$w_{i,j} = \frac{c_{i,j}}{\sum_{i=1}^{|D|} c_{i,j}} \tag{2}$$

This equation ensures that each weight for a specific node d_i is determined by the ratio of a particular characteristic value to the sum of all characteristic indicator values for a given metric. It is worth noting that, for each characteristic, the weights of all devices for that characteristic sum up to 1. In other words, $\sum_{i=1}^{|D|} w_{i,j} = 1$.

4.2.1. Policies

Based on the performance characterizations and obtained weights, it is possible to achieve $|M|$ distinct single-objective load balancing policies. Accordingly, we consider five different load balancing policies as discussed below.

Throughput-aware policy assigns weights to edge nodes based on their observed *throughput*. This policy is suitable for high-throughput applications such as Smart Parking in Smart Cities, where a high rate of data processing is required.

Energy-aware policy assigns weights to edge nodes based on their observed *energy efficiency*. This policy is suitable for applications aim to minimize energy consumption, e.g., applications running over battery-powered edge nodes.

Latency-aware policy assigns weights to edge nodes based on their observed *response time*. This policy is suitable for many real-time applications that require a low latency response time such as Autonomous Vehicles or Emergency Department Health Monitoring applications.

Precision-aware policy assigns weights to edge nodes based on their observed *AI precision*. This policy is particularly well-suited for mission-critical IoT applications across various domains, such as Robotic Surgery and Harvesting Robots in Smart Agriculture, where precision is of paramount importance. It is essential to note that precision is chosen as a representative AI performance metric. However, other metrics such as accuracy, recall, F1 score, etc., can also be applied to our system.

Cost-aware policy assigns weights to edge nodes based on their observed *cost efficiency*. This policy is particularly beneficial for use cases with associated cost, such as scenarios involving serverless computing in a continuum of cloud to edge, where resources are charged by third-party edge providers. An illustrative example is Smart Traffic Lights in Smart Cities.

4.3. Multi-objective Load Balancing

The weight-tuning approach proposed in the previous subsection is designed to set weights in such a way as to maximize a single metric or characteristic of interest, e.g., throughput. However, in practice, one might need to optimize multiple objectives simultaneously, e.g., energy and throughput. One may also need to handle the interference of secondary factors to the target metric. For example, if response time is desired for the target application, but nodes are also running other applications that affect the energy, multi-objective policies can be the solution to handle this co-existence. Our proposed approach for multi-objective optimization is based on the well-established weighted sum method [43]. In this method, we transform the multi-objective problem into a single-objective problem by assigning weights to each objective to form a weighted sum of objectives, where the weights represent the relative importance of each objective. In this context, we introduce a new set of weights for the metrics or characteristics of interest. Subsequently, we calculate the weighted sum of $w_{i,j}$ values for each device d_i over all metrics m_j present in the multi-objective optimization.

Let N be a subset of M that includes the set of metrics of interest, $N \subseteq M$, e.g., throughput and AI-Precision, and let $\psi_k \in \Psi$ be the multi-objective weight associated with each metric $n_k \in N$. The weight ψ_k is set based on the user's preference for metric n_k. Assume that function $f(i,k)$ yields the weight $w_{i,j}$ calculated in Equation (2) for metric $n_k = m_j$ for device $d_i \in D$.

We calculate a new multi-objective weight s_i for device d_i as follows:

$$s_i = \sum_{k=1}^{|N|} \psi_k . f(i,k) \tag{3}$$

It is worth mentioning that $\psi_k \geq 0$, for all k, ensuring non-negativity of the multi-objective weights, and $\sum_{k=1}^{|N|} \psi_k = 1$, which ensures that the weights sum up to 1.

For our proposed weighted sum method, setting appropriate multi-objective weights for each metric/characteristic is crucial, as this heavily influences the outcome. Different weight combinations can lead to significantly different overall system performances. To guide the weight-setting process, we analyze Pareto Fronts [43]

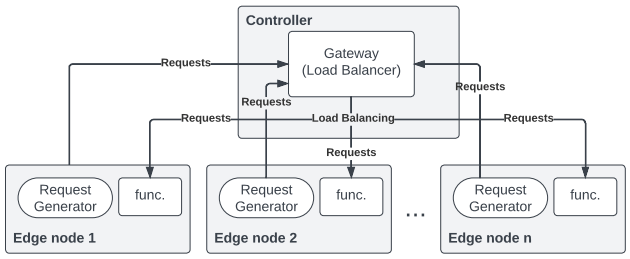

Figure 3: Centralized Load Balancing.

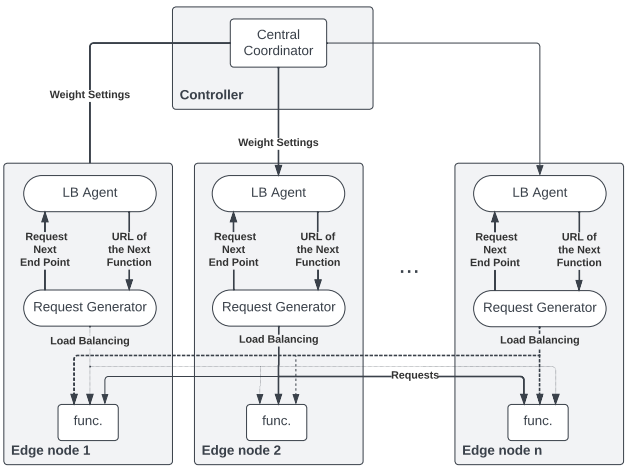

Figure 4: Coordinated Distributed Load Balancing.

in the performance evaluation section, which helps in optimizing the multi-objective method.

4.4. Coordinated Distributed Load Balancing

As shown in Figure 3, in the majority of current serverless computing frameworks including open-source ones such as OpenFaaS and OpenWhisk, a centralized load balancing design is adopted [44]. In this model, all incoming requests are directed to a central gateway, which subsequently distributes them to the appropriate functions deployed across various nodes.

However, centralized load balancing can suffer from issues such as Single Point of Failure, scalability, higher latency, and inefficiency in large networks. Therefore, we introduce an effective distributed architectural design pattern for serverless edge that addresses these issues to some extent and can improve the overall performance of the system.

We design a coordinated distributed load balancing system in which each node possesses a load balancer agent as shown in Figure 4. This agent's job is to redirect requests to functions hosted on the edge nodes in the cluster according to the policy (i.e., weight distribution) dictated by the central coordinator.

The expected benefit of this design is firstly the elimination of the need for sending all the traffic to a central gateway, thereby enhancing the overall efficiency and responsiveness of the system. Secondly, the central coordinator will handle the policy propagation to the edge nodes, facilitating even the bridge to a dynamic load balancer if desired.

5. Hedgi: The Heterogeneous Serverless Edge Computing

This section introduces the architectural design and implementation of heterogeneous serverless edge computing for AI-based data-intensive IoT applications, referred to as *Hedgi*. The illustration of *Hedgi* can be found in Fig. 5.

5.1. Manager

This component is responsible for managing the application including deployment, connectivity, and load balancing by its three main components: *software*, *platform*, and *hardware*.

Software: The *Hedgi-controller* resides in the software layer and is responsible for (a) launching a planned experiment, and, most importantly, (b) controlling the load balancer. To accomplish (a), the *Launcher* initiates an experiment based on the declarative JSON or Yaml manifests that describe the deployment of serverless functions. This means the user can describe the desired state of applications. For (b), the load balancer control plane is designed to reside in the software component for ease of use by the user or system admin. This allows both static and dynamic adjustments to the load balancer that effectively apply to its data plane, whether the data plane is central or distributed.

Platform: The operating system (*OS*), such as Linux or Windows, resides in this layer and supports *virtualization* technologies, such as Docker, containerd, microVMs, Unikernels, or Web Assembly [45]. Note that a heterogeneous system must allow for different variations of these technologies to realize platform-level heterogeneity [33, 32]. *Container Orchestration* tools such as Kubernetes run on top of the virtualization layer to facilitate the deployment and configuration of containerized applications [46]. *Serverless Platform*, is deployed to perform as a function deployment operator (*FaaS-Operator*), and as a gateway to receive the function invocations and distribute them to the corresponding function through the load balancer's data plane. In detail, the *FaaS-Operator* receives API calls, creates serverless function objects, delivers them to the orchestrator, and then notifies the hosting edge node through the orchestrator to run a container process for the function. For *Load Balancing*, a proxy server is adopted to distribute function invocations to the actual functions deployed on edge nodes, according to the policy implemented in its assignment component.

Hardware: Heterogeneity is further highlighted in this layer. The hardware used for a manager node is typically more powerful than for edge nodes, yet far less powerful than regular cloud servers [8].

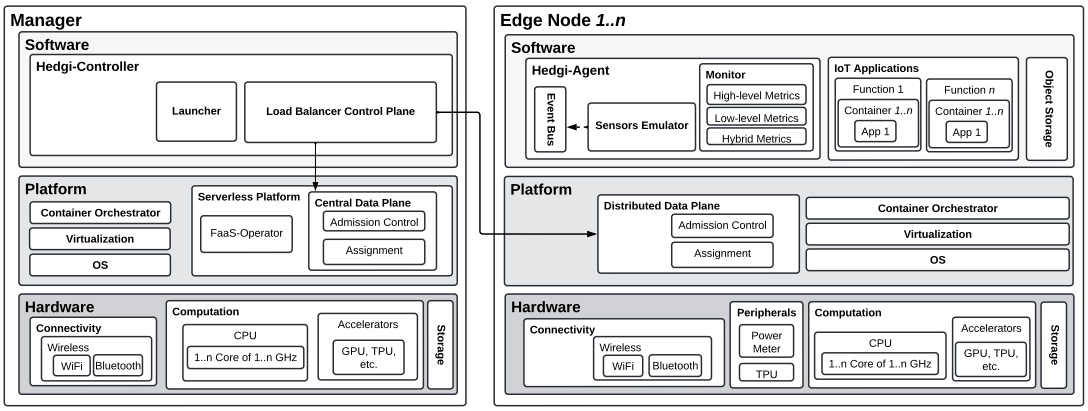

Figure 5: Architectural View of *Hedgi*: Heterogeneous Serverless Edge Computing for AI-based Data-intensive IoT Applications.

Connectivity between the manager and other nodes in the cluster is enabled through Local Area Networks (LANs) or wireless networks [47]. *Computation* refers to enabled computing units on a node, whether CPU or accelerators such as TPU and GPU. The heterogeneity of CPUs in terms of the number of cores, computing power in Gigahertz (GHz), and the inclusion of accelerators makes the system capable of executing both simple and complex tasks at the edge [48]. *Storage*, while presumably limited, is considered to have a larger capacity than storage used in edge nodes, and is used for storing the state of the orchestrator and *Hedgi* [8].

5.2. Edge Nodes

Edge nodes are responsible for generating and executing tasks, the former through the IoT sensors and the latter through the deployed serverless functions. We explain the three main components of the edge node: *software*, *platform*, and *hardware*.

Software: The software is completely different from that of the manager and is comprised of *Hedgi-Agent*, *Object Storage*, and *IoT Applications*. In *Hedgi-Agent*, the *Sensor Emulator* emulates the generation of sensor data; in the case of this work, images taken from a camera attached to an edge node. *Hedgi-Agent* can generate synchronous (waiting for a response before proceeding with the next request) or asynchronous (without waiting for an immediate response) sensor data. An *Event Bus* is triggered by the sensors that maintains the endpoints for sending requests and receiving responses from deployed functions. *Monitor* regularly collects *High-level*, *Low-level*, and *Hybrid* performance metrics such as request latency, CPU usage, and cost, respectively [15]. In *Object Storage*, the data produced by the sensors is maintained, e.g., images taken by the sensors, by using an object storage service. In the *IoT Applications* component, containerized IoT applications are deployed in the form of serverless functions, either on a single container or using multiple identical replicas, given the auto-scaling mechanism handled by the serverless platform [2]. The functions receive invocations from the load balancer and execute the given task, for which they require a particular design that allows them to run tasks on heterogeneous processors and accelerators.

Remark: The function runtime must comply with the heterogeneous edge resources as a requirement. In detail, the *Instruction Set Architecture (ISA)* of edge nodes, e.g., X86 or ARM, poses heterogeneity. Manager nodes are often associated with X86-based ISAs, whereas edge nodes are typically associated with energy-efficient ARM-based ISAs. Furthermore, the *processing units (PU)*, e.g., CPU or accelerators such as GPU or TPU, bring another heterogeneity dimension. For instance, monitoring functions such as fitness trackers and smartwatches can rely on CPUs to process the sensor data while computer vision and machine learning functions require accelerators.

Handling all the heterogeneity requirements in a function's runtime appears to be infeasible since container images are ISA-specific. A feasible solution is to utilize the multi-architecture feature of container registries like Docker Hub. That is, images for different ISAs can be built and then bundled under one single manifest. Then, upon deploying the function on the edge node, the container runtime of the node will automatically pull the corresponding image.

To handle the PU heterogeneity, a simple solution is to place all PUs' runtime dependencies inside the container image. Take an AI-based function as an example. The container image can have Tensorflow runtime to run tasks on CPUs and have CUDA and cuDNN for GPU use.

However, this approach results in a bulky container image that needs to handle all the runtimes, which may not be compliant with resource-constrained edge nodes [39]. To handle this limitation, we built different container images for different PUs, pre-cached them on the nodes, and renamed them to the same name. Ultimately, once the container image is used on the node, only the corresponding PUs' runtime will be loaded.

Platform: Virtualization is realized in this layer on

an edge node to enable the deployment of containerized serverless functions. The orchestrator handles the deployment of these functions. Also, when a distributed load balancer is desired, the data plane, as an agent introduced in Fig. 4, is deployed on the node that undertakes the load distribution task.

Hardware: This layer comprises of various heterogeneous SBCs [4] such as Raspberry Pis, Jetson Nano, etc., which serve as the computation units for IoT applications. *Connectivity* to nodes and manager is established through LAN or wireless. *Peripherals* include a Bluetooth-enabled power meter and other devices such as TPU accelerators and IoT sensors that are connected through ports or GPIO pins that are present on SBCs. *Storage* stores the state of the orchestrator and *Hedgi*.

6. Performance Evaluation

In this section, we empirically evaluate our proposed load balancing approach on the heterogeneous serverless edge architecture. We begin by providing an overview of the experimental setup. Subsequently, we present the results obtained from extensive investigations on load balancing (totally 255 experiments) that are carried out in the following order:

- First, we conduct performance characterization experiments as introduced in our proposed performance-driven load balancing approach.
- Next, we perform experiments using the obtained central single-objective load balancing policies. For comparison purposes, we include the conventional *round-robin* as the baseline policy, which assigns equal weights to all the nodes for load balancing. This allows meaningful comparisons with heterogeneous settings. Additionally, we compare against a *random* policy.
- Following this, we investigate the impact of multi-objective policies to evaluate the trade-offs upon applying preferences and priorities.
- We analyze the impact of distributed load balancing to demonstrate the limitations of a central controller.
- To evaluate the degree of expected improvement, we compare our approach against a dynamic load balancer.
- Lastly, we examine load balancing in a non-cooperative edge setting and highlight trade-offs.

6.1. Experimental Setup

We prototype a real-world implementation of *Hedgi*, a heterogeneous serverless edge computing environment as an experimental setup.

Hedgi: The infrastructure comprises a set of five different edge devices serving as edge nodes and a manager node [12], as specified in Table 3. The edge nodes utilize a UM25C USB power meter,[3] as a peripheral device for accurate hardware-based energy measurements. Virtualization is enabled on the devices in a considerably diverse mode, where the container runtime is either Docker or containerd. The orchestration of containers is handled by the edge-friendly distribution of Kubernetes, i.e., K3s [37]. The serverless platform atop it is OpenFaaS,[4] a widely used open-source implementation of the FaaS.

Application: In each experiment, we deploy the same serverless function on all edge nodes. A Single Shot Multi-Box Detector (SSD) [49] image annotation machine learning application is containerized and serverlessized for this purpose. The SSD uses MobileNet as the backbone and is trained on the COCO dataset in TensorFlow. SSD has achieved popularity in edge computing and is a model commonly deployed on resource-constraint nodes, thanks to its fast one-stage detector, as compared to slower two-stage detectors such as R-CNN (Regional Convolutional Neural Network) [49]. As the primary goal is to enable data-intensive AI-based IoT applications, the performance-wise difference and effect of using other machine learning applications, such as voice recognition or image classification, may not be significant from this perspective. To containerize and serverlessize the application, we bundle the business logic with a Flask micro-framework, backed by a watchdog agent provided by OpenFaaS. The business logic employs (a) TensorFlowLite as CPU runtime, (b) TensorRT v.8.2.1.8, CUDA v.10.2.300, cuDNN v.8.2.1.32, and JetPack v.4.6.2 (L4T v.32.7.2) as GPU runtime, and (c) EdgeTPU v.14 as TPU runtime. We set a threshold equal to 45% for inference confidences.

Workflow: We implement the workflow as depicted in Fig. 2. The IoT *environment* is simulated by a workload generator, which can generate synchronous requests at different concurrency rates, representing the number of request sender threads at each node. We experiment with the concurrency rates of 1, 3, 5, 7, and 9 for better observability unless specified otherwise. The concurrency rate can be understood as the arrival rate of requests as well. We repeat each experiment for a given rate at least three times to minimize statistical error and report the average as well as the standard error for each data point. Each experiment lasts for 10 minutes, as in [23], which is long enough to execute hundreds of tasks. The *data host* stores the sensor's data Minio, as in [5], a high-performance Kubernetes-native object storage service. A sensor's data here is a random image from a pool of 83 images with a size of between 22 and 131 KB. Minio then triggers the event bus, implemented by *Hedgi* that invokes a function. Next, in the *load balancer*, an endpoint is picked by Envoy to assign the request. Lastly, the image annotations function executes the request by fetching the image from Minio. The timeout for each request is set at 15

[3] https://tinyurl.com/um25c
[4] https://www.openfaas.com/

Table 3: Specification of Heterogeneous Edge Nodes.

Device		Hardware					Platform		Software			
Name	Role	Computation					OS	Container	IoT Application			Hedgi
		CPU			Accelerator	RAM			Function	Runtime	Precision	
		Arch	Core	GHz	GPU/TPU	GB						
Intel NUC	manager	X86	4	4.5	n/a	32	Ubuntu	docker	n/a	n/a	n/a	controller
Raspberry Pi 3	node	ARM	4	1.4	n/a	1	Raspberry Pi OS	containerd	SSD	TensorFlowLite	60%	agent
Raspberry Pi 4 Rev. 1.4	node	ARM	4	1.8	TPU	8	Raspberry Pi OS	containerd	SSD	EdgeTPU	40%	agent
Raspberry Pi 4 Rev. 1.4	node	ARM	4	1.8	n/a	8	Raspberry Pi OS	containerd	SSD	TensorFlowLite	60%	agent
Raspberry Pi 4 Rev. 1.1	node	ARM	4	1.5	n/a	4	Raspberry Pi OS	containerd	SSD	TensorFlowLite	60%	agent
Jetson Nano	node	ARM	4	1.43	GPU	4	Linux4Tegra	docker	SSD	TensorRT	76%	agent

seconds.

Load Balancing: We replace the OpenFaaS load balancer with Envoy proxy,[5] a load balancer that allows the realization of weighted round-robin policies, satisfying our approach's requirements. A central load balancer is adopted unless specified otherwise. Envoy utilizes the Earliest Deadline First (EDS) Scheduling[6] to realize the round-robin algorithm, as in Eq. (4). When a new request arrives, Envoy picks the endpoint (i.e., edge node d) with the earliest presumed deadline to which it sends the request.

$$endpoint \leftarrow \arg \min_{(endpoint \in Endpoints)} deadline(endpoint) \tag{4}$$

where the deadline for each endpoint is determined as in (5).

$$deadline \leftarrow \frac{endpoint.currentTime + 1.0}{endpoint.weight} \tag{5}$$

Such that the *endpoint.weight* is statically assigned to each endpoint according to the obtained values by performance characterization. The *endpoint.currentTime* starts at 0 and increments by 1 upon each request assignment to the endpoint.

Here is a simple example. Assume $Endpoints = [d_1, d_2]$, weighted as 1 and 2, respectively. The first request arrives and the load balancer needs to assign it to the earliest deadline endpoint. The deadline for d_1 is $1 \leftarrow \frac{0+1.0}{d_1.weight=1}$ and for d_2 is $0.5 \leftarrow \frac{0+1.0}{d_2.weight=2}$, so endpoint d_2 has the earliest deadline and receives the request. Accordingly, the d_2's $currentTime$ is incremented by 1 unit.

6.2. Experimental Results

6.2.1. Performance Characterization

Each node is individually tested and runs the same application but under its specific heterogeneous conditions, particularly different runtime environments, as specified in Table 3. This is to observe the impact of heterogeneous nodes dealing with the same inference task. The workload generator used during the experiment runs a four-thread spawner, equivalent to the maximum number of CPU cores on devices.

The weights obtained through experimentation are shown in Fig. 6. The key observation is that heterogeneous nodes exhibit significantly different behaviors from each other in various characteristics, which highlights the challenges associated with load balancing in the heterogeneous edge. For example, in terms of energy efficiency, the Raspberry Pi 4 (8GB) with TPU has the best performance, but it has the worst performance in terms of AI-Precision. Conversely, while the Jetson Nano with GPU has the best AI precision, it has the worst cost efficiency. In detail, for each characteristic, $\max(W) \mid w_{i,j} \in W$ represents the most performant device, while $\min(W) \mid w_{i.j} \in W$ represents otherwise.

Additionally, the standard deviation and skew measures represent the degree and direction of performance dispersion among nodes in every characteristic. The SD and skew measures are obtained as *throughput* (SD=0.10, Skew=1.15), *energy efficiency* (SD=0.51, Skew=0.26), *responsiveness* (SD=0.10, Skew=1.05), and *cost efficiency* (SD=0.15, Skew=0.86).

These weights form the basis for the performance-driven policies evaluated in the following section.

6.2.2. Central Single-objective Load Balancing Policies

Note that it is recommended to interpret the outcomes of a specific metric in alongside other relevant metrics to understand the reasons behind the differing behavior exhibited by a particular policy.

Low-level Metrics. Fig. 7a shows the total energy consumption in mWh of all nodes under different policies, labeled as ***energy consumption (mWh)***. The *throughput-aware* and *latency-aware* policies result in relatively higher energy consumption in contrast to the *energy-aware* policy which exhibits the lowest energy consumption. Energy consumption is directly related to the amount of work done, so the total executed requests per second labeled as ***throughput (request/sec.)*** in Fig. 7b shows that the *throughput-aware* and *latency-aware* policies led the load balancer to distribute and execute more requests over the experiments, which explains why they also consumed more energy on nodes. In general, policies that give a larger weight to the TPU-enabled node achieved higher throughput. ***energy efficiency***, which is the total en-

[5] https://www.envoyproxy.io/

[6] https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancers#arch-overview-load-balancing-types

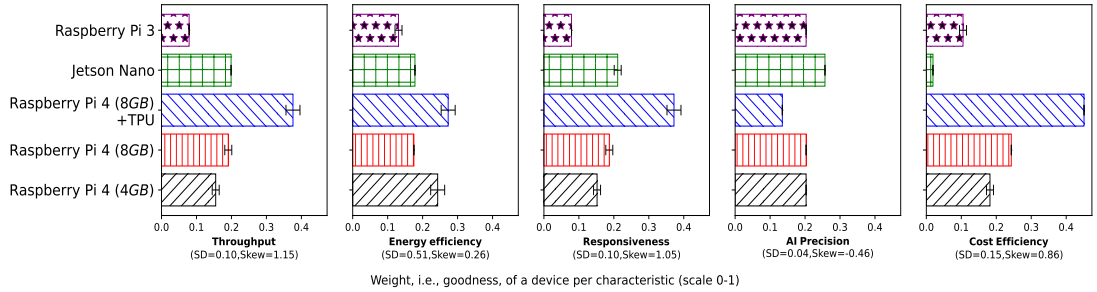

Figure 6: Performance Characterization Observations: Per-device Weights for Each Characteristic. Larger Values Indicate Better Performance.

ergy consumption divided by the throughput or energy per task, is shown in Fig. 7c. The lower the energy per task, the more energy efficient the policy is. High throughput policies, including *throughput-aware*, and *latency-aware*, respectively, also appear to be the most energy-efficient. This is mainly because these policies heavily rely on the TPU-enabled node, which has the lowest energy per task among the devices, as per the weights in Fig. 6.

The CPU-wise ***computation*** utilized by nodes in each policy is shown in Fig. 7d which is a measure of CPU usage multiplied by the CPU frequency. This is a more relevant measure than only CPU usage since the processor heterogeneity makes the significance of the CPU usage different on nodes. Not to mention that the ISA also plays a role, but we find it out of the scope and believe that even the current depth of CPU-wise evaluation can clearly showcase the importance of heterogeneity considerations. In Fig. 7d, the usage pattern implies that the *throughput-aware* and *latency-aware* imposed the highest CPU-wise computation usage. This pattern can be partially attributed to energy usage and throughput. This appears insightful since some nodes merely rely on accelerators to perform request executions, instead of CPUs, but the CPU-wise computation pattern appears representative of the throughput and energy implications.

Memory usage (%), as shown in Fig. 7e, exhibits a new pattern that warrants further examination, despite the differences being insignificant—averaging between 37% and 40% for nodes in all policies. The relatively low memory usage, i.e., below half the capacity, is due to the small memory footprint of the inference models used, which are 36 MB and 43 MB on CPU and TPU devices, respectively. Only the GPU-enabled inference model on Jetson Nano has a significantly larger memory footprint of 981 MB. One of the sources of differences could be the reliance of a device on multi-threading for the inference runtime which can increase the memory usage. Multi-threading refers to devices that allow concurrency, such as Raspberry Pi 4, where the inference model loads multiple copies of itself, equal to the number of cores on the node (4 in this case). Note that the TPU inference model operates sequentially on a single thread due to the TPU access lock and memory constraints; the CPU inference model on Raspberry Pi 3 also allows multi-threading, but the device cannot provide high throughput due to CPU limits; and the GPU inference model on Jetson Nano runs sequentially since loading several of the heavy models on this memory-constrained device is not practical. It is worth noting that, on the Jetson Nano device, the downstream Flask server of the serverless function allows concurrency, up to 4, equivalent to its CPU cores. Such multi-threading and memory constraints can be the key reasons for the position of each policy in Fig. 7e.

Upload (MB) in Fig. 7f and ***download (MB)*** in Fig. 7g show the amount of data uploaded and downloaded by nodes. These metrics are highly correlated with the throughput of the policy as more data transfer occurs to nodes handling more requests. Hence, the pattern of policy order follows the throughput order. This is mainly because all nodes run an identical, data-intensive application from the request generator perspective. Furthermore, there is a close relationship between the skew of load distribution weighting and the amount of increased downloaded data, i.e., images for inference. Theoretically, if there are x nodes weighted equally, as in the *round-robin* policy, the probability (p) that a node gets its own request and loads the image from its local object storage instead of downloading from peers, is $p = \frac{1}{x}$; however, if weights are skewed significantly, as in the *throughput-aware* (skew=1.15) and *latency-aware* (skew=1.05) policies, the probability (p) is $p < \frac{1}{x}$, resulting in more downloaded data and hence higher network traffic. Given this, the *round-robin* algorithm is expected to lower the download requests given its minimum skew of 0. This is, however, not confirmed in Fig. 7g since it is also important to note that the number of requests received by nodes is another factor to consider, and we already observed the varied throughput among nodes. Hence, the download ratio is a matter of both data distribution skew and throughput under such heterogeneity.

High-level Metrics. ***AI precision (%)***, the precision of annotating requested images by a serverless function in percentage in Fig. 7h, indicates how policies affect the obtained inference precision. Obviously, if a policy relies on nodes that are characterized as more precise, as identified in Fig. 6, it simply achieves an overall higher precision. Results in Fig. 7h confirm this, with the *precision-aware* policy providing the highest average precision. The *cost-*

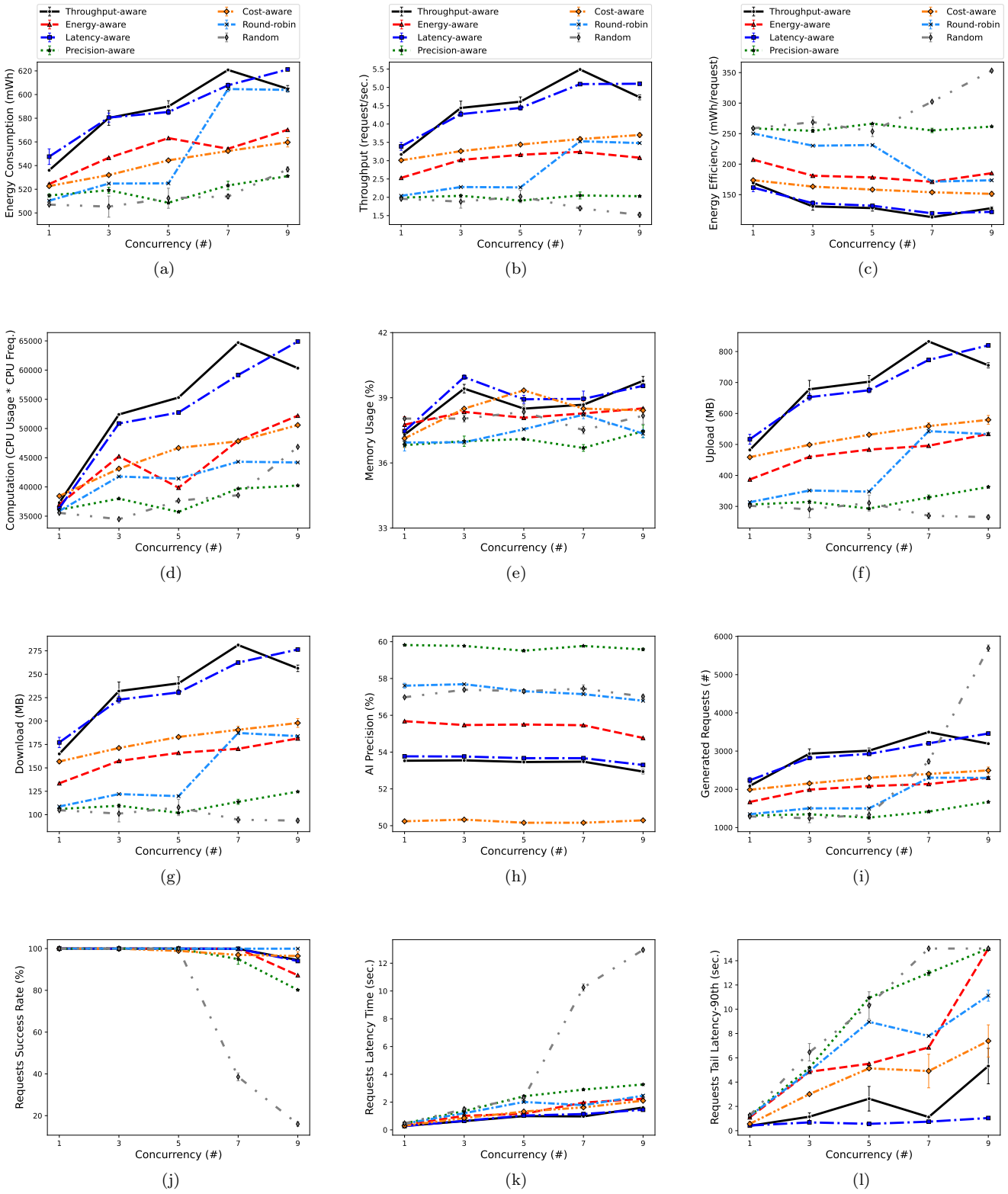

Figure 7: Comparison of Load Balancing Policies across Low-Level and High-Level Metrics under Varying Request Concurrency. The data points represent the mean of results and the error bar represents the deviation of repetitions from the average.

aware policy, on the other hand, provides the lowest average precision. In detail, highly precise policies rely more on GPU-based (precision = 76%) and CPU-based (precision = 60%) inferences, respectively, than on TPU-based (precision = 40%) inferences.

Diving into workload matters, ***generated requests (#)***, the total number of generated requests in Fig. 7i, closely follows the observed throughput shown in Fig. 7b. It is important to know the proportion of generated requests that are successfully executed by policies. Fig. 7j provides ***success rate (%)***, the number of successfully executed requests in percentage. Theoretically, an increased load in the system increases the chances of saturation and contention. However, *throughput-aware*, and *latency-aware* policies, which allow for the highest number of generated requests, did not exhibit the highest rate of failure. The reason for this is that high throughput policies tend to utilize efficient nodes with a lower response time, allowing them to handle a relatively higher number of requests before saturation. Therefore, if a policy, despite generating less workload, distributes the load on low throughput nodes, it can increase the number of failed requests, as observed in Fig. 7j by the *precision-aware* policy. Note that, the drops in the success rate of particular policies such as *precision-aware* in high concurrency suggest another evidence for the lower energy consumption and throughput of such policies (shown in Fig. 7a and 7b). That is, if requests are failed then the nodes skip the execution of such requests and hence consume less energy.

Requests latency (sec.), i.e., the response time or end-to-end processing time of all requests in seconds, is shown in Fig. 7k. It increases in line with the concurrency level, from 1 to 9. To compare policies, the *latency-aware* policy, which works based on the responsiveness characteristic of nodes (Fig. 6) exhibits the shortest response time along with the *throughput-aware* policy. This occurs despite the higher number of generated and executed requests by nodes under such policies, which once again evident that it matters where the requests are distributed. The superiority of the *throughput-aware* policy lies in the fact that it relatively follows the responsiveness weighting scheme, with the highest weight given to the superfast TPU-enabled device and appropriate weights to the weaker nodes. Therefore, the two policies share a common property. In general, a system with high throughput will have a lower response time because it can handle more requests in the same amount of time. To analyze the impact of long backlogs and latency at larger concurrency, the ***tail latency (sec.)*** at the 90th percentile can provide particular insights as shown in Fig. 7l. The well-performing policies in the average latency managed to maintain the tail latency as well while the *precision-aware* and *random* show the longest tail latency. The same behavior was observed in the tail latency at the 95th and 99th percentiles, so they are not reported.

Hybrid Metric. The ***function cost (GB-second)*** metric, shown in Fig. 8, is measured by multiplying the memory usage of the function in Gigabyte and the execution time of the function per request in second, akin to AWS Lambda services[7]. The results show that the *cost-aware* policy exhibits a relatively reasonable cost-efficiency. In contrast, the *precision-aware* policy presents the least cost-efficiency since it strives for precision satisfaction which is achievable through emphasizing the cost-inefficient GPU-enabled node. It is important to note that cloud providers' pricing models are mainly agnostic to the heterogeneity of resources such as CPU, GPU, and TPU.

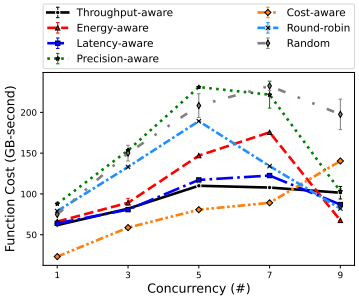

Figure 8: Comparison of Load Balancing Policies using a Hybrid Metric: *Function Cost*, Given Varied Requests Concurrency.

6.2.3. Single-objective vs. Multi-objective Load Balancing

Here, we demonstrate the realization of multi-objective policies using our proposed performance-driven weight-tuning approach. The *throughput-aware* and *precision-aware* policies are suitable examples for showcasing contradictory objectives with the highest trade-off. As shown previously in Fig. 6, the former exhibits the most positively-skewed distribution (1.15) while the latter exhibits the most negatively-skewed distribution (-0.46) of weights they give to nodes. Intuitively, to achieve both objectives equally, one may assign 50% weights to both policies. We conducted experiments with this simple solution, named as *Thr.-Prec. (50—50)*, and the results of throughput and precision are shown in Fig. 9 and 10. While the metrics' results are expected to stand in the middle compared to the single-objective version of each policy, it is clear that simply setting equal 50% weights does not yield the expected outcome, as the throughput tends to lean towards the single-objective *precision-aware* policy.

Our proposed solution for a better approximation, as explained in Section 4.3, is to determine the Pareto fronts of the combined policies. We conducted experiments for the same policies, shown in Fig. 11. We examine different weight combinations given to each policy at 20% increments, from 0 to 100. The trade-off curve showcases the best possible compromises between the different objectives. The Pareto front as provided in Fig. 11 for different concurrency levels allows decision-makers to explore different weight combinations of objectives and determine how they impact the overall performance.

[7]https://aws.amazon.com/lambda/pricing/

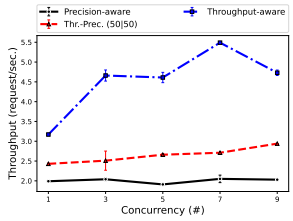

Figure 9: Throughput in Single-Objective vs. Multi-Objective Policies with Uniform Inclusions.

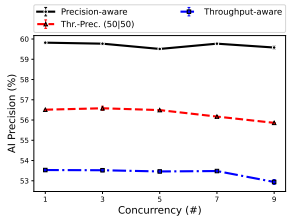

Figure 10: Precision in Single-Objective vs. Multi-Objective Policies with Uniform Inclusions.

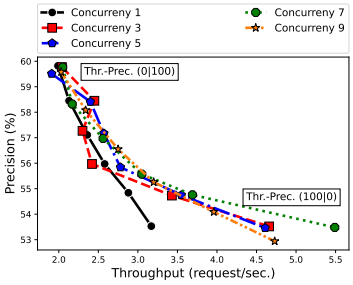

Figure 11: Pareto Fronts for Multi-objective Policies, i.e., Throughput and Precision, under Different Workload Concurrency Levels.

6.2.4. Central vs. Distributed Load Balancing

So far, a central load balancer has been employed as a single point of reference to implement the load distribution logic, specifically the performance-driven weighted round-robin policy. However, relying on a centralized approach can lead to scalability challenges due to the serviceability limitations associated with having a single central point of control. This limitation can be quantified by comparing the serviceability capacity of a central load balancer with that of the distributed version. Hence, we set an equal limit for the maximum requests the gateway can handle to evaluate the *throughput-aware* policy in both centralized and distributed modes. Technically, the implementation of this limit is based on the circuit-breaking[8] concept in load balancing, where a maximum allowed in-flight request is always maintained by the load balancer.

The maximum value for both the central and distributed modes must be set equally to ensure a fair comparison. To achieve this, we base our comparison on the maximum load of the central mode, which is nine requests—equivalent to the maximum workload concurrency. In all experiments within the central mode, the in-flight requests are maintained at, or below, the maximum workload concurrency, which is nine. Consequently, the same capacity is allotted to the distributed load balancer to ensure a fair comparison. While intuitively, the distributed load balancer appears to offer superior performance due to its capacity multiplied by the number of devices, understanding the degree of this improvement and the system's behavior under varying loads warrants further investigation.

Results in Fig. 12 show that high-rate workloads in the central mode not only fail to improve the throughput but cause a decrease in the throughput due to the overwhelming backlog on the load balancer. The other effect of this pressure is observable in Fig. 13, where the latency significantly increases in the central mode. Note that we allow the excess requests to remain pending on the central load balancer until a token is available to prevent an even worse performance scenario for the central mode.

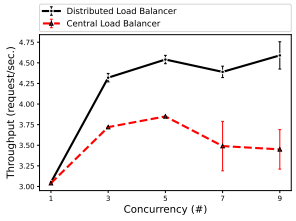

Figure 12: Throughput Comparison: Central vs. Distributed Load Balancing.

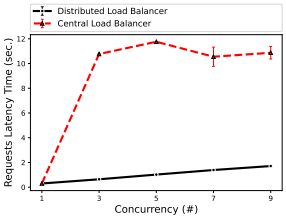

Figure 13: Latency Comparison: Central vs. Distributed Load Balancing.

6.2.5. Static vs. Dynamic Load balancing

The previously designed policies we used follow a static setting of weights at system setup time. Now, one may question the limitations of this approach, especially when other factors such as co-existing applications on nodes could affect the effectiveness of the statically set weights. A proper benchmark for such a comparison is the *least-request* policy. The *least-request* policy is a dynamic load balancing algorithm where requests are forwarded to the node with the least number of active/in-flight requests at the time the request is received. This policy attempts to dynamically adjust its preference to edge nodes with better throughput performance in an online mode. We compare this algorithm with our statically weighted round-robin *throughput-aware* policy in Fig. 14. An improvement is observed with the *least-request* load balancer. A key factor in its success is its awareness of the dynamics of the system, especially under heavy loads, whereas the static load balancer fails to maintain the high success rate, as shown in Fig. 15.

However, it must be acknowledged that dynamic load balancers such as the *least-request* approach need to maintain a list of active requests for all endpoints consistently, which may impose considerable overhead on the load balancer in large-scale scenarios. Furthermore, the *least-request* policy, in its original form, primarily emphasizes throughput. It lacks out-of-the-box support for other metrics such as energy, precision, or cost, although achieving such support, including multi-objective policies, is not impossible through this policy. Lastly, the dynamic load balancer is evaluated under the assumption that no limit is set on network bandwidth, energy consumption of node,

[8] https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/circuit_breaking

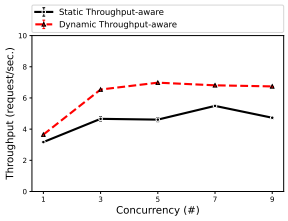

Figure 14: Throughput Comparison: Static vs. Dynamic *Throughput-Aware* Policies.

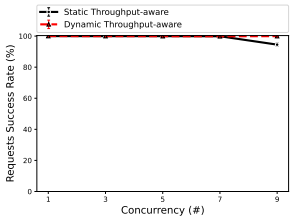

Figure 15: Request Success Rate Comparison: Static vs. Dynamic *Throughput-Aware* Policies.

and function cost to allow high throughput experiment observations representing intensive workloads.

6.2.6. Load Balancing in Self-contained vs. Cooperative Edges

All the previous experiments were conducted with the assumption that edge nodes should be enabled with resource sharing to utilize peers' resources for improving their performance needs. However, it is worth investigating the impact of locality and self-contained edge nodes with no resource sharing to see how the objectives are affected. We call the former strategy *cooperative edge* while the latter is called *self-contained edge*. In the self-contained mode, we run the same experiments on the *Hedgi* using its central load balancer, with the difference that the load balancer distributes the requests originating from each node to the same node to be executed by its own local function, which we call it the *local* policy. This way nodes are operating self-contained.

Results for all the previously introduced cooperative policies compared to the *local* policy are shown in Fig. 16 and 17. The *local* policy fails to guarantee a considerably shorter latency compared to the cooperative *latency-aware* and *throughput-aware* policies, despite its higher energy efficiency. Not to mention that no limit on the energy consumption of the nodes is considered in our experiments while otherwise, the self-contained nodes may be affected significantly by their limited energy resources such as batteries. Furthermore, note that the *local* policy fails to allow an under-performing node to improve its requests' precision, while in the cooperative mode, nodes can achieve that. For example, if a node can execute requests in isolation with a precision of 40%, it cannot achieve any improvement if the objective demands a higher precision. Overall, a significant performance trade-off between self-contained and cooperative edges exists when it comes to load balancing.

The improvements in some metrics such as throughput and latency for the *local* policy in the self-contained mode can be attributed to the elimination of data transfer over the network to access the object storage remotely, To assess such hypotheses, we examine the concurrency level of 3, where the *local* policy shows the highest throughput at 8.52, and compare it to its closest alternative, the *latency-aware* policy, which has a throughput of 5.13. If the high throughput is due to not having to download data remotely, we would expect to see a significant decline in the *local* policy's throughput if it is customized to download data remotely. We examined this by customizing the *local* policy so each function downloads its required data for a given request from remote nodes in a round-robin manner. The obtained throughput declined notably, from 8.52 to 6.02. In contrast, we modified the *latency-aware* policy to eliminate the need for data download from remote storage, by assuming that the required data for a given request is always available on the node of the function that is assigned to execute the tasks. Results show an increase in throughput from 5.13 to 6.35 for the *latency-aware* policy. The improvement from removing the burden of data download is substantial and confirms the hypothesis.

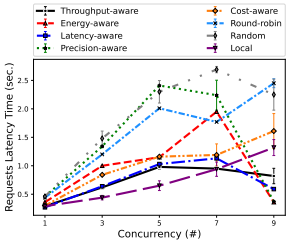

Figure 16: Latency Comparison: Self-contained (Local) vs. Cooperative (Others) Edge.

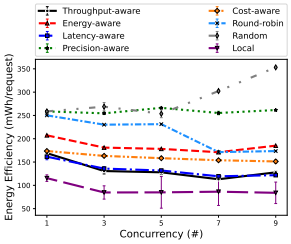

Figure 17: Energy Efficiency Comparison: Self-contained (Local) vs. Cooperative (Others) Edge.

7. Discussions

7.1. Edge Computing

Our investigation of the differences between a cluster of self-contained edge nodes and cooperative edge nodes implies that the self-contained edge design can offer additional resource efficiency benefits in certain cases, in particular data and bandwidth-intensive applications such as SSD.

Self-contained and cooperative edge

With the trade-off between designing self-contained or cooperative edge, a dynamic load balancer that can shift the load balancing scheme between isolation mode and resource sharing or consider data locality opens up intriguing future directions for further exploration.

The evaluation of load balancing policies was conducted with certain assumptions, such as an unlimited energy supply to nodes and a constant request rate over time, to ensure fairness and an isolated analysis of the respective performance metrics, as suggested by [50]. However, these assumptions may not be appropriate for use cases that involve battery-powered edge nodes with variable energy input (e.g., solar panels) in Smart Cities and Smart Farming,

as studied in [47, 2, 8], or for use cases with variable workload that follow random patterns, as studied in [12, 8, 7]. Therefore, care must be taken when applying the findings to other contexts in order to achieve the desired results.

Variable energy and load in real-world

Evaluating load balancing strategies under varying arrival rates of users/IoT requests and varying energy inputs demands special considerations to ensure the resiliency of the strategy in real-world scenarios.

7.2. Serverless Computing

Auto-scaling for serverless computing originally occurs at the granularity of individual requests. With the increasing use of AI-based applications in the serverless landscape, this work has uncovered additional limitations. For example, initializing a new replica can cause long cold starts, typically a few seconds, due to the loading time of an AI inference model. This is unacceptable for per-request scaling. Furthermore, our designed system revealed that functions that rely on accelerators such as TPU cannot be replicated easily over the same node by the serverless platform due to access control limitations. To overcome these shortcomings, we investigated the possibility of vertical scaling of serverless functions, rather than employing horizontal scaling. Our experiments confirm the feasibility and practicality of this approach. It is noteworthy that horizontal scaling is not completely discouraged since it brings benefits such as isolation, multi-tenancy, and resiliency opportunities.

Vertical and hybrid scaling of serverless functions

Due to the deteriorated cold-start problem of serverless functions caused by heavy AI models, implementing vertical scaling of serverless functions can allow a more smooth adaptation of AI-based applications in the serverless domain, as examined in this work. The next ambitious improvement could be hybrid scaling, which combines the benefits of both vertical and horizontal scaling approaches.

Resource sharing between functions on hosts with GPUs and accelerators is not trivial, leading us to design the functions in a way that a single function will exclusively use the accelerator. Additionally, it is beneficial for an accelerator-enabled device to be allowed to switch between its processing resources, i.e., CPU and accelerator to satisfy varied performance metrics. For example, a node can switch to CPU when running out of battery and then switch to its high throughput accelerator when shorter latency is desired.

Dynamic CPU and accelerator allocation

A dynamic resource allocation that shifts between processors, i.e., CPU and accelerators, can allow a higher degree of resource efficiency, especially when multi-objective optimization is desired.

The *cost model* of serverless computing is currently only applicable to CPU-based computation and is not yet mature for accelerator-enabled environments. To the best of our knowledge, there is currently no practical accelerator-based pricing model in place. Hence, our measurements also follow CPU-driven pricing models, although companies such as DataRobot (https://www.datarobot.com/) have started to fill this gap.

Heterogeneity-aware serverless pricing model

The original CPU-based pricing model of serverless appears outdated with the emergence of GPU and accelerator-demanding applications.

Furthermore, we designed a novel multi-purpose serverless function that handles all the pre-processing, processing, and post-processing of a task. This allowed us to simplify the workflow and remove inter-dependencies. Function decomposition is also a common practice in serverless.

Single- and multi-purpose functions' trade-offs

Contrary to the initial idea of FaaS, which focused on single-purpose short-lived applications, our study demonstrates that multi-purpose and long-lived functions are also feasible and can be suited for certain applications.

7.3. AI Models

To characterize heterogeneous edge nodes in terms of AI-related performance, we measured the precision metric. However, it is important to note that other metrics such as recall and F1 score can be similarly employed.

Application-specific evaluation of AI models

To address heterogeneity from an AI application's perspective, an important line of research will be to conduct a comprehensive application-specific comparison of devices and AI models.

Furthermore, we chose the TensorFlow framework for deploying the function since the TPU device is specifically designed to use TensorFlowLite and also a pre-trained model, i.e., SSD MobileNet, is publicly available for CPU-, TPU-, and GPU-based inferences for a relatively fair comparison. However, if resource efficiency of the AI framework is concerned, alternatives such as Pytorch or MXNet are also worthy of exploration, as initial studies evidence

their desirable performance in memory footprint, throughput, and loading time of an inference model [42].

8. Conclusions and Future Directions

In this paper, we proposed a performance-driven load balancing approach based on empirical weight-tuning, supporting the realization of several single-objective load balancing policies for serverless edge computing, such as energy-aware and cost-aware. We further extended our approach to encompass multi-objective and distributed load balancing. Additionally, we presented *Hedgi*, a heterogeneous edge computing architecture facilitating the deployment of AI-based IoT applications in a serverless manner. Through extensive experiments, we explored the performance implications of our load balancing approach in a heterogeneous serverless edge computing setup by the *Hedgi*. Our results provide valuable insights into the design trade-offs of various load balancing policies and trigger an informative discussion on the challenges posed by load balancing in heterogeneous environments.

Future research can be conducted to extend those insights in the following aspects: (a) self-adaptive load balancing policies for variable objectives, (b) self-contained and cooperative edge switching of edge nodes, (c) variable energy input and request arrival in IoT, (d) vertical and hybrid scaling of AI functions, (e) dynamic CPU and accelerator harvesting for multi-objective applications, (f) heterogeneity-aware serverless pricing models to support accelerators, (g) the trade-offs of using the single- and multi-purpose design of functions, and (h), application-specific evaluation of AI models for load balancing.

References

[1] B. Wang, A. Ali-Eldin, P. Shenoy, LaSS: Running Latency Sensitive Serverless Computations at the Edge, HPDC 2021 - Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (2021) 239–251arXiv:2104.14087, doi:10.1145/3431379.3460646.

[2] M. Aslanpour, A. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. Gill, R. Gaire, S. Dustdar, Serverless Edge Computing: Vision and Challenges, in: Australasian Computer Science Week Multiconference, 2021. doi:10.1145/3437378.3444367.

[3] S. S. Tuli, R. Mahmud, S. S. Tuli, R. Buyya, FogBus: A Blockchain-based Lightweight Framework for Edge and Fog Computing, Journal of Systems and Software 154 (2019) 22–36. arXiv:1811.11978, doi:10.1016/j.jss.2019.04.050.
URL `https://doi.org/10.1016/j.jss.2019.04.050`

[4] S. Sarkar, R. Wankar, S. N. Srirama, N. K. Suryadevara, Serverless Management of Sensing Systems for Fog Computing Framework, IEEE Sensors Journal 20 (3) (2020) 1564–1572. doi:10.1109/JSEN.2019.2939182.

[5] S. R. Poojara, C. K. Dehury, P. Jakovits, S. N. Srirama, Serverless data pipeline approaches for IoT data in fog and cloud computing, Future Generation Computer Systems 130 (2022) 91–105. arXiv:2112.09974, doi:10.1016/j.future.2021.12.012.
URL `https://doi.org/10.1016/j.future.2021.12.012`

[6] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, A. Iosup, The State of Serverless Applications: Collection, Characterization, and Community Consensus, IEEE Transactions on Software Engineering 5589 (c) (2021) 1–1. doi:10.1109/tse.2021.3113940.

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

[8] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, 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.

[9] S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, A. Y. Zomaya, Dependent Function Embedding for Distributed Serverless Edge Computing, IEEE Transactions on Parallel and Distributed Systems 33 (10) (2022) 2346–2357. doi:10.1109/TPDS.2021.3137380.

[10] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You Only Look Once: Unified, Real-Time Object Detection, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[11] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A. C. Berg, SSD: Single Shot MultiBox Detector, in: B. Leibe, J. Matas, N. Sebe, M. Welling (Eds.), Computer Vision – ECCV 2016, Springer International Publishing, Cham, 2016, pp. 21–37.

[12] T. Rausch, A. Rashed, S. Dustdar, Optimized container scheduling for data-intensive serverless edge computing, Future Generation Computer Systems 114 (2021) 259–271. doi:https://doi.org/10.1016/j.future.2020.07.017.

[13] I. Martinez, A. S. Hafid, A. Jarray, Design, Resource Management, and Evaluation of Fog Computing Systems: A Survey, IEEE Internet of Things Journal 8 (4) (2021) 2494–2516. doi:10.1109/JIOT.2020.3022699.

[14] Y. Dai, D. Xu, S. Maharjan, Y. Zhang, Joint load balancing and offloading in vehicular edge computing and networks, IEEE Internet of Things Journal 6 (3) (2019) 4377–4387. doi:10.1109/JIOT.2018.2876298.

[15] M. S. Aslanpour, S. S. Gill, A. N. Toosi, Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research, Internet of Things 12 (2020) 100273. doi:10.1016/j.iot.2020.100273.

[16] A. Tzenetopoulos, E. Apostolakis, A. Tzomaka, C. Papakostopoulos, K. Stavrakakis, M. Katsaragakis, I. Oroutzoglou, D. Masouros, S. Xydis, D. Soudris, FaaS and Curious: Performance Implications of Serverless Functions on Edge Computing Platforms, in: H. Jagode, H. Anzt, H. Ltaief, P. Luszczek (Eds.), High Performance Computing, Springer International Publishing, Cham, 2021, pp. 428–438.

[17] F. Carpio, M. Delgado, A. Jukan, Engineering and Experimentally Benchmarking a Container-based Edge Computing System, IEEE International Conference on Communications 2020-June (2020). arXiv:2002.03805, doi:10.1109/ICC40277.2020.9148636.

[18] A. Luckow, S. Jha, Performance Characterization and Modeling of Serverless and HPC Streaming Applications, Proceedings - 2019 IEEE International Conference on Big Data, Big Data 2019 (2019) 5688–5696arXiv:1909.06055, doi:10.1109/BigData47090.2019.9006530.

[19] H. Sedghani, F. Filippini, D. Ardagna, A Randomized Greedy Method for AI Applications Component Placement and Resource Selection in Computing Continua, Proceedings - 2021 IEEE International Conference on Joint Cloud Computing, JCC 2021 and 2021 9th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2021 (2021) 65–70doi:10.1109/JCC53141.2021.00022.

[20] H. Sedghani, F. Filippini, D. Ardagna, A Random Greedy based Design Time Tool for AI Applications Component Placement and Resource Selection in Computing Continua, in: 2021 IEEE International Conference on Edge Computing (EDGE), IEEE, 2021, pp. 32–40. doi:10.1109/edge53862.2021.00014.

[21] M. Szalay, P. Matray, L. Toka, Real-Time task scheduling in a FaaS cloud, IEEE International Conference on

Cloud Computing, CLOUD 2021-Septe (2021) 497–507. doi:10.1109/CLOUD53861.2021.00065.

[22] C. Cicconetti, M. Conti, A. Passarella, A Decentralized Framework for Serverless Edge Computing in the Internet of Things, IEEE Transactions on Network and Service Management (2020) 1doi:10.1109/TNSM.2020.3023305.

[23] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, P. Chen, Function delivery network: Extending serverless computing for heterogeneous platforms, Software - Practice and Experience 51 (9) (2021) 1936–1963. arXiv:2102.02330, doi:10.1002/spe.2966.

[24] M. Ciavotta, D. Motterlini, M. Savi, A. Tundo, DFaaS: Decentralized Function-as-a-Service for Federated Edge Computing, 2021 IEEE 10th International Conference on Cloud Networking, CloudNet 2021 (2021) 1–4doi:10.1109/CloudNet53349.2021.9657141.

[25] Q. Tang, R. Xie, F. R. Yu, T. Chen, R. Zhang, T. Huang, Y. Liu, Distributed Task Scheduling in Serverless Edge Computing Networks for the Internet of Things: A Learning Approach, IEEE Internet of Things Journal 9 (20) (2022) 19634–19648. doi:10.1109/JIOT.2022.3167417.

[26] M. Salehe, Z. Hu, S. H. Mortazavi, T. Capes, I. Mohomed, VideoPipe: Building video stream processing pipelines at the edge, Middleware Industry 2019 - Proceedings of the 2019 20th International Middleware Conference Industrial Track, Part of Middleware 2019 (2019) 43–49doi:10.1145/3366626.3368131.

[27] F. Al-Doghman, N. Moustafa, I. Khalil, Z. Tari, A. Zomaya, AI-enabled Secure Microservices in Edge Computing: Opportunities and Challenges, IEEE Transactions on Services Computing 1374 (c) (2022) 1–20. doi:10.1109/TSC.2022.3155447.

[28] A. Grafberger, M. Chadha, A. Jindal, J. Gu, M. Gerndt, FedLess: Secure and Scalable Federated Learning Using Serverless Computing, Proceedings - 2021 IEEE International Conference on Big Data, Big Data 2021 (2021) 164–173arXiv:2111.03396, doi:10.1109/BigData52589.2021.9672067.

[29] A. Das, S. Patterson, M. Wittie, EdgeBench: Benchmarking Edge Computing Platforms, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 175–180. doi:10.1109/UCC-Companion.2018.00053.

[30] C. Cicconetti, M. Conti, A. Passarella, Architecture and performance evaluation of distributed computation offloading in edge computing, Simulation Modelling Practice and Theory 101 (July 2019) (2020) 102007. arXiv:2109.09415, doi:10.1016/j.simpat.2019.102007.
URL https://doi.org/10.1016/j.simpat.2019.102007

[31] T. J. Skluzacek, Dredging a Data Lake: Decentralized Metadata Extraction, in: Proceedings of the 20th International Middleware Conference Doctoral Symposium, Middleware '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 51–53. doi:10.1145/3366624.3368170.
URL https://doi.org/10.1145/3366624.3368170

[32] L. Baresi, D. F. Mendonca, D. F. Mendonça, Towards a serverless platform for edge computing, Proceedings - 2019 IEEE International Conference on Fog Computing, ICFC 2019 (2019) 1–10doi:10.1109/ICFC.2019.00008.

[33] N. P. Shah, Design of a Reference Architecture for Serverless IoT Systems, 2021 IEEE International Conference on Omni-Layer Intelligent Systems, COINS 2021 (2021). doi:10.1109/COINS51742.2021.9524180.

[34] A. Garbugli, A. Sabbioni, A. Corradi, P. Bellavista, TEMPOS: QoS Management Middleware for Edge Cloud Computing FaaS in the Internet of Things, IEEE Access 10 (2022) 49114–49127. doi:10.1109/ACCESS.2022.3173434.

[35] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, T. Huang, When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues, IEEE Wireless Communications 28 (5) (2021) 126–133. doi:10.1109/MWC.001.2000466.

[36] M. Chadha, A. Jindal, M. Gerndt, Towards Federated Learning using FaaS Fabric, WOSC 2020 - Proceedings of the 2020 6th International Workshop on Serverless Computing, Part of Middleware 2020 (2020) 49–54doi:10.1145/3429880.3430100.

[37] F. Carpio, M. Michalke, A. Jukan, Engineering and Experimentally Benchmarking a Serverless Edge Computing System, 2021 IEEE Global Communications Conference, GLOBECOM 2021 - Proceedings (952644) (2021). arXiv:2105.04995, doi:10.1109/GLOBECOM46510.2021.9685235.

[38] F. Martella, G. Parrino, G. Ciulla, R. D. Bernardo, A. Celesti, M. Fazio, M. Villari, Virtual device model extending NGSI-LD for FaaS at the Edge, Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021 (2021) 660–667doi:10.1109/CCGrid51090.2021.00079.

[39] K. R. Rajput, C. D. Kulkarni, B. Cho, W. Wang, I. K. Kim, EdgeFaaSBench: Benchmarking Edge Devices Using Serverless Computing, in: C. A. Ardagna, H. Bian, C. K. Chang, R. N. Chang, E. Damiani, G. Elia, Q. He, V. Puig, R. Ward, F. Xhafa, J. Zhang (Eds.), 2022 IEEE International Conference on Edge Computing and Communications (EDGE), IEEE; IEEE Comp Soc, 2022, pp. 93–103. doi:10.1109/EDGE55608.2022.00024.

[40] A. Mehta, R. Baddour, F. Svensson, H. Gustafsson, E. Elmroth, Calvin Constrained - A Framework for IoT Applications in Heterogeneous Environments, Proceedings - International Conference on Distributed Computing Systems (2017) 1063–1073doi:10.1109/ICDCS.2017.181.

[41] H. Ko, S. Pack, S. Member, Function-Aware Resource Management Framework for Serverless Edge Computing (2022) 1–10doi:10.1109/JIOT.2022.3205166.

[42] J. Hao, P. Subedi, I. K. Kim, L. Ramaswamy, Characterizing Resource Heterogeneity in Edge Devices for Deep Learning Inferences, Vol. 1, Association for Computing Machinery, 2021. doi:10.1145/3452411.3464446.

[43] I. Y. Kim, O. L. de Weck, Adaptive weighted-sum method for bi-objective optimization: Pareto front generation, Structural and Multidisciplinary Optimization 29 (2) (2005) 149–158. doi:10.1007/s00158-004-0465-1.
URL https://doi.org/10.1007/s00158-004-0465-1

[44] Y. Li, Y. Lin, Y. Wang, K. Ye, C.-Z. Xu, Serverless Computing: State-of-the-Art, Challenges and Opportunities, IEEE Transactions on Services Computing 1374 (c) (2022) 1–1. doi:10.1109/tsc.2022.3166553.

[45] V. Cozzolino, O. Flum, A. Y. Ding, J. Ott, MirageManager: enabling stateful migration for unikernels, in: Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems, 2020, pp. 13–19.

[46] G. Casale, M. Artač, W. J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, S. N. Srirama, D. A. Tamburri, M. Wurster, L. Zhu, RADON: rational decomposition and orchestration for serverless computing, Software-Intensive Cyber-Physical Systems 35 (1-2) (2020) 77–87. doi:10.1007/s00450-019-00413-w.

[47] H. Hacid, O. Kao, M. Mecella, N. Moha, H.-y. P. Eds, B. Steffen, G. Woeginger, M. S. Aslanpour, A. N. Toosi, R. Gaire, M. A. Cheema, WattEdge: A Holistic Approach for Empirical Energy Measurements in Edge Computing, in: International Conference on Service-Oriented Computing, Vol. 2, Springer, 2021, pp. 531–547. doi:10.1007/978-3-030-91431-8.
URL http://dx.doi.org/10.1007/978-3-030-91431-8_33

[48] R. B. Roy, T. Patel, D. Tiwari, IceBreaker: warming serverless functions better with heterogeneity (2022) 753–767doi:10.1145/3503222.3507750.

[49] R. Mahmud, A. N. Toosi, Con-Pi: A Distributed Container-Based Edge and Fog Computing Framework, IEEE Internet of Things Journal 9 (6) (2022) 4125–4138. arXiv:2101.03533, doi:10.1109/JIOT.2021.3103053.

[50] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, P. Cao, How to Build a Benchmark, in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ICPE '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 333–336. doi:10.1145/2668930.2688819.
URL https://doi.org/10.1145/2668930.2688819