Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Microservices and serverless functions—lifecycle, performance, and resource utilisation of edge based real-time IoT analytics

Francesco Tusa [a,b,*], Stuart Clayman [b], Alina Buzachis [c], Maria Fazio [c,d]

[a] University of Westminster, London W1B 2HW, UK
[b] University College London, London WC1E 6BT, UK
[c] University of Messina, 98166, Messina, Italy
[d] Gruppo Nazionale per il Calcolo Scientifico (GNCS) - INdAM, Rome, Italy

ARTICLE INFO

Keywords:
Edge analytics
Function as a service
IoT
Microservices
Real-time
Serverless computing

ABSTRACT

Edge Computing harnesses resources close to the data sources to reduce end-to-end latency and allow *real-time* process automation for verticals such as Smart City, Healthcare and Industry 4.0. *Edge* resources are limited when compared to traditional *Cloud* data centres; hence the choice of proper resource management strategies in this context becomes paramount. *Microservice* and *Function as a Service* architectures support modular and agile patterns, compared to a monolithic design, through lightweight containerisation, continuous integration / deployment and scaling. The advantages brought about by these technologies may initially seem obvious, but we argue that their usage at the *Edge* deserves a more in-depth evaluation. By analysing both the software development and deployment lifecycle, along with performance and resource utilisation, this paper explores *microservices* and two alternative types of *serverless functions* to build edge real-time IoT analytics. In the experiments comparing these technologies, microservices generally exhibit slightly better end-to-end processing latency and resource utilisation than serverless functions. One of the serverless functions and the microservices excel at handling larger data streams with auto-scaling. Whilst serverless functions natively offer this feature, the choice of container orchestration framework may determine its availability for microservices. The other serverless function, while supporting a simpler lifecycle, is more suitable for low-invocation scenarios and faces challenges with parallel requests and inherent overhead, making it less suitable for real-time processing in demanding IoT settings.

1. Introduction

The Internet of Things (IoT) envisions large-scale deployments of embedded devices that interact with each other to pursue a common goal [1]. Gathering and harnessing in *real-time* the massive amount of data generated by all those interconnected entities can help organisations to develop new business models and streamline operational processes, hence supporting the creation of more innovative products and services across various industries [2]. However, this strategy also introduces new requirements for both the distributed network and the computing infrastructures involved in the data processing tasks. New demands for mobility support and geo-distribution, as well as for location awareness and low latency, have to be taken into account when setting up the required end-to-end application layer.

Using *Cloud Computing* would not be ideal in this context, as large amounts of data needs to be transferred—from the data producers to the centralised data centres—to perform the required computation. As a result, considerable round-trip delays would incur during this communication and could ultimately affect the experienced Quality of Service (QoS) [3]. Therefore, since IoT and data acquisition devices are usually deployed at the *edge* of the network, a more effective solution to this problem may use computing resources that are as close as possible to the location where the data are generated [4].

The *Edge Computing* paradigm realises the above idea by shifting the computation from the core of the network to its *edge*, i.e., to resources located close to the data acquisition devices, thereby dramatically reducing the latency associated with the data transfer [4]. Thanks to its distributed nature, *Edge Computing* introduces advantages in terms of reduced communication time, which makes it a viable approach for IoT scenarios where *real-time* process automation is executed based on the data collected from the *"things"*.

Edge Computing requires an efficient architectural design where both the *computation-constrained* and *energy-constrained* nature of the *edge nodes* is taken into account [5]. Consequently, a hybrid distributed

* Corresponding author at: University of Westminster, London W1B 2HW, UK.
E-mail address: f.tusa@westminster.ac.uk (F. Tusa).

https://doi.org/10.1016/j.future.2024.02.006
Received 16 July 2023; Received in revised form 1 February 2024; Accepted 10 February 2024
Available online 12 February 2024

infrastructure has emerged based on the combination of different resource types distributed across the *edge* and the *core* of the network. It can be seen as a spectrum of computing options where the trade-offs between latency, bandwidth, availability, reliability, and scalability vary depending on the location and characteristics of the computing nodes, and is commonly referred to as the *Edge–Cloud Continuum* [6,7].

A well-engineered use of the resource continuum, according to an application's requirements, can improve the data processing latency and reduce the volume of data needed to be transferred from the end devices to the network's core [8]. Therefore, unlike traditional data analytics, *real-time edge analytics* typically performs a first processing stage on the limited edge resources near the data sources. Long-term processing tasks with flexible real-time demands, requiring additional resources, are delegated to the other layers of the computing continuum, extending up to the cloud data centres.

Meanwhile, emerging modular and scalable application design patterns are replacing monolithic systems with cooperating *microservices* or with even smaller building components known as *functions* [9]. This trend has been nurtured by the timely growth of more lightweight virtualisation technologies, such as containers and microVMs (micro Virtual Machines) in place of traditional VMs [10], as well as by the adoption of *agile* software continuous integration and development workflows. These, coupled with *event-driven programming*, have led to the rise of *Serverless Computing*—a paradigm whereby third-party providers allocate resources dynamically to support the on-demand execution of computing services.

The *Serverless Function as a Service (FaaS)* model embraces this approach by allowing application logic, written as *stateless functions*, to be executed on-demand by containerised runtime environments without pre-allocating resources [11]. This aims to overcome some of the limitations of the Infrastructure as a Service (IaaS) Clouds by delivering a third-party managed resource infrastructure, accessible via a pure *pay-per-use* model and supporting effortless scalability [12].

1.1. Contributions of this work

Microservice and *FaaS* architectures leverage lightweight containerisation technologies to allocate their components in a significantly shorter time frame than traditional VMs. Moreover, deploying applications based on these patterns can align more effectively with the users' computing demands, resulting in improved resource utilisation compared to monolithic systems. While seemingly ideal for the *Edge Computing* context, transitioning to these new technologies imposes constraints on the design of application components and the *interfaces* through which they can be accessed externally.

In our previous work [13], we demonstrated how the interface exposed by *edge analytics* applications can considerably affect both the number of bytes transmitted over the network towards the *edge* and the amount of CPU and memory resources required by an *edge node* to perform even the simple decoding of the received IoT data. We identified that those metrics are significantly impacted by the messaging protocol, network technology, data format, and the programming language and frameworks utilised for the implementation.

Based on the above results, the research question this paper aims to answer is: **are *Microservice* and *FaaS* architectures viable technologies for executing *edge* applications that perform real-time IoT analytics tasks?** It is essential to note that the advantages of using these models at the *edge* may not be immediately apparent, and it is crucial to consider various impacting factors when trying to answer this question [14]. Therefore, in addition to an investigation of the mechanisms involved in the development and deployment of real-time IoT edge analytics applications, this paper evaluates the resource allocation aspects associated with the execution of those applications on large-scale IoT scenarios: a *Smart Factory* with a thousand sensors and a maximum cumulative data rate of 25k messages/sec; a *Smart City* with ten thousand sensors and ten times the maximum data rate of the previous scenario.

The analysis is conducted from the perspective of infrastructure providers or service providers delivering IoT services to their customers through resources distributed across the Edge–Cloud continuum. The primary goal is to identify the most suitable approach, choosing between *Serverless FaaS* and *Microservice* architectures, to enhance the *resource utilisation* of these providers' infrastructures—focussing on the *resource-constrained edge*. The evaluation considers the fulfilment of the performance requirements of customers' applications in the IoT context of this paper—the *end-to-end processing latency* of real-time analytics.

Experiments involving microservices and *two types of serverless functions* reveal that, while microservices entail a less abstract lifecycle with hands-on tasks like container configuration, they consistently demonstrate slightly better *end-to-end processing latency* and *resource utilisation* compared to serverless functions, when the data volume remains below 25k messages/sec. In contrast, the two types of serverless functions we examined present a more abstract lifecycle, but their performance varies based on the frequency of invocation and data batch sizes. Moreover, the microservices and one type of serverless function excel at handling large data streams of up to 250k messages/sec with auto-scaling. This feature is inherently provided by FaaS architectures, but its availability for microservices may depend on the framework used for orchestrating the associated containers. Despite supporting a simpler and more abstract lifecycle, the other type of serverless function is suitable for low-invocation scenarios only. It struggles with serving parallel requests due to its intrinsic overhead, making it inappropriate for real-time processing in high-demand IoT settings.

The paper is organised as follows. Section 2 discusses the background concepts and existing research related to this work; Section 3 describes the Edge Computing and IoT analytics scenarios considered in this paper, while Section 4 provides the details of the testbed; Section 5 describes the criteria whereby the experiments of Section 6 were performed; finally, Section 7 concludes the work and sheds lights on potential future work.

2. Background

2.1. IoT real-time analytics and edge computing

Sensors within machines, devices, or other sources—in areas such as Smart Factories, Smart Cities, and Healthcare applications—produce a massive amount of data that can be processed to gain valuable insights and possibly create proactive and predictive business models [2]. Due to their location in machinery at the network's border, those data producers are commonly referred to as the *edge*. Many of the above areas require that the data produced at the *edge* is processed in real-time. As such, when considering *real-time IoT analytics*, the latency between the data generation and the availability of the processing results should be as small as possible and ideally close to zero [3].

While the volume of data generated at the edge continuously increases, and the real-time processing requirements are becoming more stringent for certain applications, the sole use of centralised cloud resources is showing its limitations. Hence, computing and network resources located along the path between the data source and the Cloud might be utilised to reduce the intrinsic communication latency between the edge and the core of the network. These objects, usually available at the edge of the network, are named *edge nodes* [4].

The *Edge Computing* paradigm is an extension of the traditional *Cloud Computing* model wherein additional computational, data handling and networking resources (nodes) are placed closer to the end devices. The consequence of this extension is that all the tasks requiring data management and processing, along with storage and networking communication, can now not only occur on centralised cloud servers but also on the *continuum* of nodes available between those servers and the end devices [8]. Thereby, *Edge Computing* becomes extremely useful for low-latency applications, as well as for applications that generate an enormous amount of data that, due to bandwidth constraints, cannot practically be transferred to cloud servers in real-time [15].

2.2. Microservices, functions and serverless computing

During the last few years, monolithic applications have evolved towards service-oriented architectures and, more recently, to *Microservice* architectures. These are based on small and loosely coupled components, where each piece can be executed autonomously for a specific task. The *Function as a Service (FaaS)* model considers even smaller components at the granularity of *stateless functions*. These are created just in time, following the users' requests, and combined to build more complex, highly modular services [14]. This evolutionary path has been sustained by the affirmation of event-driven programming and the usage of continuous integration and continuous development technologies. Furthermore, the rise and diffusion of containerisation, led by Docker [16], played a significant role in the above process, as it facilitated the execution of the above service components within lightweight, isolated environments [17].

Although the traditional IaaS clouds allow for the allocation of services on-demand, they fall short of a pure pay-per-use billing model and of scalability mechanisms transparent to the users, who would have to implement their auto-scaling methods [14]. Serverless Computing builds on the advances brought about by the usage of Microservice and FaaS architectures, event-driven programming and containerisation, and tries to overcome the above limitation of IaaS clouds by introducing a pure pay-per-use model along with effortless scalability [12]. Serverless is one step forward in the abstraction staircase from IaaS to *Platform as a Service (PaaS)*. It provides customers with a platform that supports the execution of software without exposing any details of the underlying Operating System (OS) and virtualisation technologies [18]. Therefore, the life-cycle management of the VMs/containers, their images, and the burden of monitoring the servers will be entirely delegated to the Serverless provider [19].

Available Serverless Providers include Amazon AWS Lambda, Azure Functions, Google Cloud Functions, IBM Cloud Functions and Oracle Fn [20]. The last two have released their tools as open-source projects (respectively, OpenWhisk and Fn project); Amazon has also recently open-sourced Firecracker, the platform on which AWS Lambda is based. Meanwhile, the research community has been actively developing easy-to-use Serverless solutions such as OpenFaaS [21] (used for the experiments presented in this paper), Fission, Kubeless, etc. All those serverless platforms can be analysed and compared based on the following list of provided features: (i) functions composition and communication patterns; (ii) functions invocation/triggering methods (e.g., request-reply, pub/sub, etc.); (iii) resource provisioning and scaling specifications (e.g., CPU, memory or both); (iv) resource abstractions or virtualisation (e.g., containers or microVMs [22]); (v) supported programming languages.

2.3. Related work

Our previous work [13] considered utilising microservices for the execution of real-time edge analytics and solely investigated the impact of various encoding and transmission protocols on the edge resource utilisation, leaving the analysis of the actual computing tasks as future work. This paper complements our previous findings and provides further insights on the resource utilisation of the actual data analysis when both Microservice and FaaS frameworks are considered. Earlier results have shown that using JSON over HTTP/REST, widely adopted by many of those frameworks, requires more CPU resources than other types of data encoding (e.g., XDR) and transmission mechanisms (e.g., WebSockets). Nonetheless, these technologies are utilised by the tools chosen for the evaluation presented in this paper, as they are the de-facto solution adopted in distributed cloud scenarios and by most of the existing Microservice and FaaS solutions.

The authors of [3] present an analysis of the state-of-the-art on using Edge Computing in Smart Factory to execute analytical calculations in real-time. In paper [4], the usage of microservices running at the edge is investigated for the execution of IoT analytics. In paper [17], a modular and scalable architecture based on lightweight Docker containers is presented, and its suitability to process IoT data at the edge is evaluated. Finally, an IoT platform that combines microservices and Serverless Computing to process data in a smart farming scenario is discussed in [23].

The work presented in [14] provides an in-depth qualitative evaluation of the suitability of Serverless Computing for the edge. In [24], the authors analyse the resource utilisation of applications designed according to the FaaS model and deployed on inexpensive Single Board Computers to process data generated by an IoT service platform. The evaluation of a serverless edge platform that supports real-time data-intensive applications is presented in [25]. The performance of four alternative open-source serverless frameworks is assessed in [26], but the evaluation is not performed at the large scale targeted by our paper and does not include a comparison with the deployment of traditional preallocated containers.

Motivated by concerns about power consumption in data centres, brought about by the rise of cloud technologies, the energy efficiency of serverless computing is evaluated in [27]. The results reveal OpenFaaS's [21] enhanced power efficiency relative to Docker [16], especially when subjected to high CPU and memory demands.

A study on the dynamics of the costs when using serverless computing compared to IaaS deployments is presented in [28]. It reveals that serverless may not consistently save user costs, urging providers to diversify use scenarios. Current pricing models pose challenges, prompting the proposal of an auction-based pricing system for serverless, reducing function costs for users without compromising provider revenue, as demonstrated by experimental results.

From the above state-of-the-art analysis, it emerges that some work done in this area evaluates the usage of edge applications to minimise the processing latency of real-time analytics considering Microservice or FaaS models. Other work analyses the energy and cost implications of using these architectures. However, there is currently no comprehensive quantitative comparison between these two approaches for processing massive IoT data in real-time at the *edge*—a direct and detailed analysis conducted on an actual testbed under real conditions is currently unavailable. Our paper aims to fill this gap by providing an extensive evaluation and comparison of the lifecycle, performance and resource utilisation of edge real-time IoT analytics, implemented as microservices deployed within preallocated containers versus functions executed via the Serverless Computing paradigm.

3. Scenario

This paper considers the reference scenario in Fig. 1, a distributed and multi-layered computing infrastructure for processing massive IoT data, referred to as the *Edge–Cloud Continuum* [7,8]. At the bottom layer, a large-scale deployment of IoT devices generates a vast amount of data to be processed. In general, these are small, low-maintenance, low-power consumption devices that are deployed in the field for extended durations, sometimes up to a decade. Although some IoT devices may offer embedded computation capabilities, those that do not provide such a feature are mainly considered in this work.

As a consequence, data from the devices is offloaded and processed at various layers within the *Edge–Cloud Continuum*, depending on the nature of the required tasks and the available resources at each layer. Initially, IoT data undergoes local pre-processing at accessible local devices before being transmitted to the subsequent layers in the computational hierarchy, encompassing the *edge* and ultimately the *cloud* [6]. As shown in Fig. 1, our reference scenario includes components deployed in these three separate layers: the *IoT Adaptation* layer and the *Real-time Edge Processing* layer, both in the Edge Domain, and the *Long-term Cloud Processing* layer, in the Cloud Domain.

While using edge resources can reduce the data transmission latency compared to a centralised cloud model, it introduces inherent resource

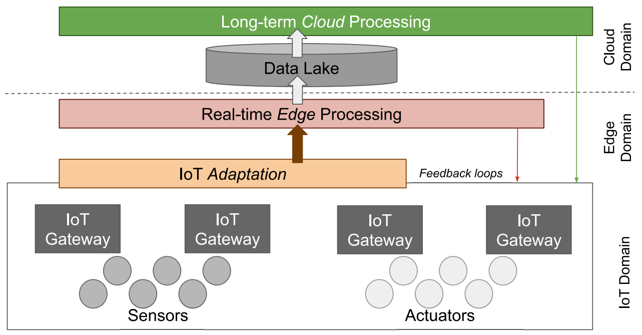


Fig. 1. Reference computing scenario.

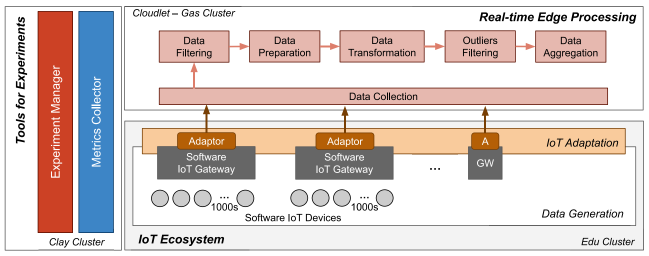


Fig. 2. Hardware and Software Components of the Testbed.

and energy constraints that need to be considered. Therefore, further long-term processing tasks with non-stringent real-time requirements, demanding more resources, cannot be executed at the edge and have to be offloaded to the *Long-term Cloud Processing* in the cloud. Rather than evaluating the unified orchestration of the above data processing on the whole spectrum of the resources of the *Edge–Cloud Continuum*, this work assesses the execution of real-time IoT analytics tasks inside the *Real-time Edge Processing* layer.

To better understand the context of the proposed reference scenario, some possible application domains are now briefly introduced. A *Smart Factory* uses connected devices, machinery, and sensors to create flexible and self-adapting production systems. In particular, massive amounts of data generated by IoT devices deployed in a smart production line are processed in real-time to collect valuable insights that can help reduce downtimes, improve production efficiency, achieve lower energy consumption, etc. Likewise, in a *Smart City* scenario, CCTV cameras and analytics functions would be utilised to process data for detecting and preventing specific situations in real-time, e.g., accidents, crimes, potential threats, or to recognise particular features (face recognition, demographics, etc.). The details of those three layers are discussed below.

3.1. IoT adaptation

The *IoT adaptation* layer is characterised by limited computing capacity but minimal data transmission latency. Its primary role is collecting data from diverse types of IoT devices deployed within an IoT domain. This layer acts as an abstraction, effectively concealing potential heterogeneity in data formats and presenting a harmonised IoT dataset through processes such as re-encoding and re-aggregation. The resultant dataset adheres to a specific encoding/serialisation approach and is subsequently relayed to the upper processing layer, denoted as the *Real-time Edge Processing*. As the above adaptation operations do not require substantial computing power, they can be executed on nodes close to the IoT devices, such as the IoT Gateways, with minimum data transmission latency. Such an adaptation layer is often used in orchestrated distributed cloud environments to normalise data [29].

3.2. Real-time edge processing

Resources in the proximity of the IoT domain, such as cloudlets and edge servers, nodes of the mobile access network, and other heterogeneous network nodes, can become part of a distributed pool that provides enough computational capabilities for executing real-time analytics with acceptable data transmission latency [30]. The *Real-time Edge Processing* layer consists of a dynamic number of processing elements that can be orchestrated [8], i.e., created and destroyed on-demand—according to the characteristics of the received data. This computation is expected to happen without intermediate long-term storage to minimise the resulting end-to-end processing latency [31].

Two alternative implementations of this layer have been developed as part of this work—a first one based on *microservices* and *containers*, and a second one built via *serverless functions*. They have both been used to support the main evaluation targeted by this paper, which was already briefly introduced in the previous sections. Relevant experiments have been carried out based on those implementations to identify which technology, among microservices and serverless functions, is more suitable—from a resource utilisation and latency perspective—for the execution of real-time IoT analytics at the edge.

3.3. Long-term cloud processing

The *Long-term Cloud Processing* layer deals with complex processing tasks, possibly involving vast amounts of data, that are not expected to be performed with strict real-time guarantees. As data is usually processed offline, this layer is decoupled from the data producers via the introduction of *Data Lakes* (as shown in Fig. 1). This type of operation usually requires massive resource capabilities compared to the ones available at the edge. As this work is focussed on the real-time processing part of the above reference scenario, this layer will not be considered in the rest of the paper.

4. Testbed implementation

To implement the reference scenario presented in Fig. 1, and to carry out the performance evaluation of this work, a distributed testbed, consisting of various hardware and software components, was used. The details are described in this section. The evaluation focuses on the two bottom layers of the reference scenario. The testbed representation of Fig. 2 shows that the *Edge Domain* is implemented as a *cloudlet*, on which the *Real-time Edge Processing* functionalities are executed; the *IoT Domain* is comprised of a distributed network of software IoT devices, referred to as the *IoT Ecosystem*.

From a hardware point of view, the testbed consists of three relatively small clusters of compute and network resources—*Clay*, *Gas* and *Edu*—hosted at University College London (UCL). The *Clay* cluster comprises five servers with 2x *AMD Quad-Core Opteron 2347HE* @1.9 GHz and 32 GB of memory. The *Gas* cluster comprises four servers with 4x *Intel Quad-Core Xeon E5520* @2.27 GHz and 32 GB of memory. Finally, the *Edu* cluster includes three servers with 4x *Intel 12-Core Xeon E5-2650 v4* @2.20 GHz and 192 GB of memory. All these nodes are inter-connected via a 1 Gbps Ethernet Local Area Network (LAN) and run the Linux Rocky 9 OS with kernel version 5.14.

As Fig. 2 shows, two of the above clusters—*Edu* and *Gas*—were specifically utilised to run the software components of the *IoT Ecosystem* and the *Real-time Edge Processing*, respectively. Additionally, the *Clay* cluster was dedicated to hosting the necessary tools for managing and executing experiments involving these components, which from now on in the paper will be referred to as subsystems.

The *IoT Ecosystem* is a software IoT platform that generates and pushes streams of data towards the *Real-time Edge Processing* subsystem;

the *Real-time Edge Processing* implements the functionalities to perform real-time computation on those IoT data, using either microservices or serverless functions; finally, the *Tools for Experiments* is the subsystem that automates the configuration of experiments and the collection of measurements for the performance evaluation.

Using separate clusters to deploy the above subsystems made it possible to replicate on our testbed the same working conditions of an actual IoT scenario. Specifically, the *Gas* cluster acted as a *cloudlet*—a resource-constrained edge data centre—on which the two alternative implementations of the *Real-time Edge Processing* were executed during the performance evaluation. The generation of the large volume of data required for the experiments via the *IoT Ecosystem* demanded relatively high resource capabilities. Therefore, *Edu* was selected for running that subsystem, as it is the cluster exhibiting the highest capacity in the testbed. The *Edu* and the *Gas* clusters were interconnected via an existing 1 Gbps Ethernet network, which allowed reproducing the communication layer whereby, in an actual IoT scenario, the data generated by the *IoT Domain* is transferred to the *Edge Domain*.

Whilst the *IoT Ecosystem* and the *Tools for Experiments* (running on the *Clay* cluster) subsystems are both critical for the setup of this experimentation environment, it should be noted that the core of our quantitative performance analysis revolves around the number of resources consumed by the *Real-time Edge Processing* on the *Gas* cluster. The usage of a software *IoT Ecosystem* does not affect the general meaning of the results presented in this paper, as these mainly depend on how the components deployed in the *Edge Domain* behave when they receive and process IoT data; however, how those data are generated by the *IoT Domain* is not relevant to the rest of the system and can be abstracted away.

4.1. IoT ecosystem

This subsystem can reproduce via software the actual behaviour and scale of a real-world IoT domain with thousands of physical *"things"* [13]. It was deployed on the *Edu Cluster* to provide a fully programmable software *IoT Domain* to generate the custom streams of data required for our performance evaluation. The *IoT Ecosystem*'s implementation is based on the dynamic features and composable software elements of the Lattice monitoring framework [32,33], which were used as the foundation for building both its *Data Generation* and *IoT Adaptation* components, as represented in Fig. 2.

The *Data Generation* allows the definition of topologies of *Software IoT Devices* of various sizes, where each device can be programmed to generate data at a given rate. Although both the format and the encoding of those data can be fully customised, *sensor-like measurements* were considered during our experiments. A topology is described by a set of the above *Software IoT Devices* attached to various *Software IoT Gateway*s. While acting as multiplexers, the gateways also support the execution of simple data manipulation tasks. Fig. 2 shows that basic data formatting operations can be performed via different *Adaptor*s connected to the *Software IoT Gateway*s. In effect, these *Adaptor*s form the distributed *IoT Adaptation* component of this subsystem, which is responsible for (i) receiving the multiplexed streams of data from the *Software IoT Gateway*s; (ii) grouping those data in batches of a given size; (iii) encoding those batches using a specific data interchange format; and (iv) sending each batch of data to the *Real-time Edge Processing*.

The *IoT Ecosystem* can be customised to implement a specific experimental scenario by adjusting the *number* of *Software IoT Devices* and the *rate* at which they *generate data*. Likewise, the *number* of *Software IoT Gateway*s, and the *type* of *Adaptor*s attached to them, can be selected; finally, for each *Adaptor*, the *ingress* point of the *Real-time Edge Processing*, and the *size* of the *batches* of data to be transmitted, can both be configured. Further details can be found in [13].

4.2. Real-time edge processing

This subsystem plays a key role in the performance evaluation of this paper. Consistently with the idea of *real-time analytics*, it implements a data processing workflow that needs to be executed on resource-constrained edge nodes in real-time. The individual tasks of this workflow are represented by the interconnected set of components shown in the top part of Fig. 2. The *Data Collection* acts as the ingress point of the workflow, as it gathers batches of data generated by the *IoT Ecosystem*. At each processing step, the output of one component is provided as input to the next one in the workflow until the *Data Aggregation* task is eventually performed, and a result is produced as the final output. As discussed, in this work, we developed and evaluated two alternative versions of the *Real-time Edge Processing* workflow of Fig. 2. The first one consists of *microservices* deployed via *containers*, each related to one of the tasks of the above workflow; the second version also implements the same tasks, where each task is built via *serverless functions* instead.

Currently, JSON and HTTP/REST are the de-facto solutions for implementing the communication layer of a distributed system, thanks to the loose coupling and high degree of interoperability they deliver [34]. Many *Microservice* and *FaaS* frameworks use these technologies to implement the external interfaces of their components. This practice extends to the tools used in this work, specifically *Flask* [35] microservices deployed on *Docker Swarm* [36] and the *serverless functions* managed by *OpenFaaS* [21].

This choice was motivated by the broad support these tools receive from the open-source community and by the relatively lower demand for computational resources they exhibit compared to similar solutions [37], which is relevant for resource-constrained edge nodes. Nonetheless, the tools should only be considered possible examples of systems based on the above technologies. Therefore, our choice of using them in our testbed to support the experiments is not to be deemed a factor directly impacting the collected measurements and the general validity of the subsequent results.

We had previously determined that using JSON over HTTP/REST might introduce transmission/decoding overhead into a distributed system [13]. Hence, our approach to mitigating this issue consisted of aggregating IoT data as batches of different sizes before they are sent to the *Real-time Edge Processing* for computation. *Python* was also chosen as the preferred programming language for both the *Microservice* and *FaaS* implementations of this subsystem. *Python* is fully supported by the chosen frameworks, it is well integrated into their development workflow, and it provides data analysis functionalities through the *Pandas* library [38]. Moreover, it also shows lower memory requirements than other languages, such as *Java*, thus becoming the obvious choice for developing applications that are expected to run on resource-constrained edge nodes [13].

4.2.1. Microservice implementation

This version of the *Real-time Edge Processing* subsystem adopts a modular architecture comprised of microservices, wherein each microservice corresponds to a specific task in the workflow illustrated in Fig. 2. These microservices were instantiated as separate *Docker* containers deployed within the *Gas* cluster of the testbed. The *Gas* cluster served as *cloudlet*, i.e., a resource-constrained edge data centre managed via *Docker Swarm*—Docker's lightweight native container orchestration engine for clusters.

Each microservice represents a RESTful *Flask* web application that receives data POSTed via HTTP, performs the required processing tasks, and returns the result in the HTTP response. JSON was the data format exchanged over HTTP/REST. Once running, the whole processing application receives batches (i.e., JSON arrays) of IoT measurements POSTed to the REST endpoint of the (ingress) *Data Collection* microservice.

The cloudlet utilised for allocating the above microservices encompasses three nodes, consisting of one master and two workers, that provide the computational resources required to support the operation of the *Real-time Edge processing* workflow. To facilitate the instantiation of the containers, a *docker-compose* descriptor was employed to define the individual microservices and outline their intercommunication approach. More specifically, following the Docker Swarm's default *spread* scheduling strategy [39], the microservices were distributed across the available hosts of the cloudlet so that an even utilisation of the cluster's resources was ensured. Moreover, they could seamlessly communicate through an overlay network, built on top of the existing Ethernet connection of the testbed, and established at deployment time by *Docker Swarm*.

Lifecycle considerations. The microservices were developed in *Python 3* and built using the *Flask 1.1 microframework* [35]. This was used to implement the required external web-based interface; the features provided by *Pandas 1.1.3* [38] were used for the data analysis/manipulation. The default *Flask* web server was replaced by a production-ready *Waitress 1.4* [40], running with one worker thread. This implementation of the *Real-time Edge Processing* subsystem implied direct exposure to the low-level setup and deployment aspects of the *Docker* containers. Moreover, different *Docker* images had to be created in the *Docker Registry* repository by hand, based on the already available *alpine-python3* image; additional package dependencies (i.e., numpy and pandas) had to be explicitly specified in the relevant *Dockerfiles*. Finally, references to the created images were explicitly added to the *docker-compose* that described the whole application, together with the details of how the containers communicate (i.e., endpoints, ports, overlay network).

4.2.2. FaaS implementation

This version of the *Real-time Edge Processing* subsystem uses *serverless functions*, managed by the *OpenFaas* framework [21], to build each of the tasks of the workflow illustrated in Fig. 2. Similar to the Microservice implementation described earlier, this version was also developed in *Python 3* and includes features offered by *Pandas 1.1.3*. However, it did not require the additional development of the functions' external interface since OpenFaaS provides a collection of *predefined templates* for various programming languages that enable developers to create functions easily. These templates include a web interface that is automatically generated for each function. Developers only need to add their specific function logic to the template's designated *handle* method. Docker images and containers are still generated and utilised for function instantiation when a new function is created using a template. However, OpenFaaS manages these operations transparently, abstracting them from the software development workflow and system administration.

Functions cold start. *Cold start* of functions is one of the well-known concerns associated with FaaS architectures. It stems from using containers created on-demand—per function invocation—and removed when that function receives no further requests within a given time frame. Even though creating a container is much faster than creating a VM, the required time is still not negligible, mainly when the container is allocated for the first time on a given host, as its image has to be fetched from a remote registry. Additional delays can then be potentially introduced by the container orchestration system, such as Kubernetes [41], due to the health checks performed after a new container is created [42].

This inherent delay would not be practical for the specific scenario of this work, which necessitates minimal processing latency to support real-time computation of IoT data. Various FaaS frameworks offer alternative approaches to mitigate the cold start problem [43], involving trade-offs between function response time, resource utilisation, and process isolation. A widely adopted strategy aims to keep functions *warm* based on specific criteria [44] so that there will always be at least one function instance immediately available to serve an incoming request. OpenFaas' solution to the cold start issue consists of creating a persistent container for each type of function during deployment. Notably, this long-lived container can handle multiple invocation requests by leveraging process-level isolation [42] and an initialisation process known as the *Watchdog*, which actively monitors the invocations a function receives and launches these processes as needed.

OpenFaas watchdog. In addition to the Microservice implementation discussed earlier, to carry out the performance evaluation of this work, two more variants of the *Real-time Processing* subsystem were developed based on different OpenFaaS templates. These templates address the above *cold start* problem by employing two alternative versions of the OpenFaas *Watchdog* process. The first version is built on the *classic watchdog* template, while the second version utilises the *of-watchdog* template [45].

The *classic watchdog* provides an unmanaged and generic interface between a given function and the outside world. It is responsible for marshalling an HTTP request, accepted on the OpenFaaS *API Gateway* [21], and invoking the requested function by creating a new process. Every function embeds this binary and uses it as its entry point. Once a new process is forked, the *Watchdog* passes in the HTTP request via *stdin* and reads an HTTP response via *stdout*. Because the above abstraction is in place, the process does not need to know anything about the web or HTTP. Therefore, the *classic watchdog* simplifies the development of the application and provides the highest level of portability. However, it does have the drawback of forking one process per request that, as we will demonstrate later, may lead to poor performance.

Conversely, the *of-watchdog* enables an HTTP mode whereby a process is first created and reused repeatedly between multiple function invocations to offset the above latency of forking. However, this introduces the additional complexity of writing functions that have to be HTTP-aware, i.e., able to parse data from the received HTTP request and return the result in the HTTP response. In effect, this approach embeds a web application within a container using *Flask* like the Microservice implementation discussed earlier. To support our experiments and comparative evaluation, the software components' versions of the *of-watchdog* function template and the Microservice implementation were aligned to use *Flask 1.1* and *Waitress 1.4* with one worker thread.

Lifecycle considerations. As discussed earlier in the paper, the IoT data produced by the *IoT Ecosystem* were encoded in JSON format and transmitted over the network via HTTP/REST. This also applies to the two implementations based on serverless functions, regardless of the chosen OpenFaaS *Watchdog* template. However, the endpoints of the allocated components (functions) were not directly exposed to the rest of the system; instead, the execution of the data processing workflow was initiated through interaction with the (ingress) *Data Collection* function via the respective endpoint created on the OpenFaaS *API Gateway*.

An OpenFaaS application can be defined through a YAML file that outlines the composition of its various functions and their interactions. When functions are defined according to either the *classic watchdog* or the *of-watchdog* template, Docker containers are automatically created. Unlike the *docker-compose* descriptor used in the Microservice implementation, the deployment workflow did not require the specification of low-level container details such as the Dockerfile definition, image management and run-time instantiation. OpenFaaS dealt with the transparent creation of the functions by automating the deployment and interconnection of the required Docker containers on the underlying *Docker Swarm*.

4.3. Tools for experiments

This subsystem includes the software tools supporting the management and the execution of the experiments performed during the evaluation targeted by this paper, i.e., the *Metrics Collector* and the *Experiment Manager* components described hereafter.

4.3.1. Metrics collector

This component collected metrics related to the performance and testbed resource utilisation of the *Real-time Edge Processing* subsystem during the execution of a given experiment. A purposely developed Python application, deployed on the *Clay Cluster*, aggregated values collected from three different sources: (i) physical resource utilisation and containers-related metrics are gathered via querying an instance of the *Swarmprom* open source tool [46]; (ii) metrics values related to the execution time and the request-reply time of the processing applications are collected from the *Docker Swarm* and the OpenFaaS log files, as well as from the log files of the *Software IoT Gateway*s; finally, (iii) additional metrics associated with the sole execution of the serverless functions were gathered from the internal OpenFaaS monitoring system.

4.3.2. Experiment manager

This subsystem facilitated the setup and execution of diverse experiments in this work by automating the activation of customised instances of the necessary software components on the testbed. The process involved two steps, i.e., generating the appropriate configuration settings for a specific experimental scenario and deploying bespoke instances of the *IoT Ecosystem* and *Real-time Edge Processing* subsystems automatically on the designated hosts of the testbed.

Specifically, the *Experiment Manager* was responsible for the execution of the following tasks: (i) activation of a particular implementation of the *Real-time Edge Processing* on the testbed; (ii) instantiation of the *IoT Ecosystem* to generate streams of data according to the experiment's settings; (iii) deployment of an instance of the *Metric Collector* so that various metrics related to the system's performance were gathered. This subsystem [47] was implemented in Python, and for each experiment to be carried out, it performed the activation of the above components by interacting with the relevant hosts of the testbed over SSH via the Fabric library [48].

5. Evaluation approach

This section presents the methodology whereby different experiments were devised to answer the initial research question of this paper—the viability of *Microservice* and *FaaS* architectures for building edge applications that perform real-time IoT analytics tasks.

A typical stream of data produced by an IoT domain that needs to be processed in real-time is a continuous flow of events or messages from devices that capture various aspects of the physical world, such as temperature, humidity, motion, location, etc. These data streams need to be processed quickly and efficiently to enable real-time decision-making, analytics and actions based on the insights derived from the data [2]. The scale of the streams depends on the type and number of devices and the size and frequency of the messages.

To investigate how *Microservice* and *FaaS* architectures deal with the above IoT scenario, a comprehensive performance evaluation was carried out using the different implementations of the *Real-time Edge Processing* subsystem presented earlier. Experiments that reproduced the type of data and the scale characterising typical IoT systems [49] were devised and executed on our testbed to collect and analyse measurements related to two *performance indicators*, selected according to the specific target of our analysis. As mentioned, this work is centred on exploring resource allocation and performance of user real-time analytics from a provider perspective. Therefore, based on the study already performed in paper [50] and the metrics used in our previous work [13], the performance indicators deemed more relevant for this work were the *end-to-end processing latency* of the analytics and the related edge *resource utilisation*. The first performance indicator measures the ability of the *Real-time Edge Processing* to execute computations with minimal processing latency, which motivates the choice of using edge resources for the computation but also brings about additional constraints on the resource infrastructure. Hence, the second performance indicator measures the amount of *edge* resources required to perform real-time computation.

Table 1
Summary of the IoT settings used for the experiments.

	IoT scenario A	IoT scenario B
Software IoT Gateways (g)	1	10
Software IoT Devices (d)	1000	10,000
Sensor Rate (r_m)	4, 12, 25, 25	4, 12, 25, 25
Data batch size (b_s)	4000, 16,000, 32,000, 64,000	4000, 16,000, 32,000, 64,000
Experiments' duration	120 min	120 min

The evaluation approach used in our previous paper [13] focussed solely on the effects that transport and encoding facets have on *Massive Real-time IoT Data systems*, leaving out the analysis of the impact of the data processing tasks. In this paper, we extend that investigation and focus specifically on the resource capabilities that edge nodes should have to process IoT data streams in real-time via Microservice or FaaS applications. To facilitate the measurement and direct comparison of the *resource utilisation* and *end-to-end processing latency* of these technologies, alternative implementations of the *Real-time Edge Processing*—where all of the processing logic is encapsulated into a single subsystem—have been considered. This approach can mitigate the run-time complexities of multi-component configurations and setup and the associated inter-component interactions; moreover, it has provided valuable insights into the behaviour of microservices and serverless functions and their possible utilisation as the building blocks for real-time IoT analytics running at the *edge*.

The experiments of our performance evaluation, based on the above approach, were explicitly devised to model the scale—in terms of the type of devices and volume of generated data—of two alternative IoT scenarios, which will be referred to as *Scenario A* and *Scenario B*. To reproduce in our testbed different IoT settings related to these scenarios, bespoke instances of the *IoT Ecosystem* needed to be adequately configured and instantiated during the execution of the experiments. In particular, the configuration allowed the selection of the number of *Software IoT Devices* (d) and *Software IoT Gateway*s (g), as well as the rate (r_m) at which each IoT device generated the IoT events. Table 1 shows that *Scenario A* included a single *Software IoT Gateway* and 1000 *Software IoT Devices*; *Scenario B* was based on ten *Software IoT Gateway*s and 10,000 *Software IoT Device*s.

The streams of data were modelled as sensor measurements (*msgs*) encoded in JSON format and including the fields *Sensor ID*, *Name*, *Type*, *Unit*, and *Value*. When an experiment was executed, the software sensors continuously generated IoT *msgs* at a given rate r_m. Different experiments considered growing values of r_m to model an increasing cumulative number of IoT measurements generated per second. To mitigate the intrinsic transmission overhead of HTTP/REST, those data were buffered on the gateway(s) of the *IoT Ecosystem*—the *IoT Adaptation Layer*—and a single batch of a given size (b_s) was sent to the *Real-time Edge Processing* during each workflow's invocation. This allowed us to evaluate how the frequency of the workflow's invocation and the size of the elements to be processed affected the performance indicators.

As Table 1 shows, the rate r_m varied during the experiments in the range $4-25$ *msgs/sec*, while the size of the batches b_s was chosen among the values $\{4000, 16,000, 32,000, 64,000\}$. Each IoT gateway sent a single batch of data to the *Real-time Edge Processing* during each workflow's invocation, but multiple invocations occurred during the 120-minute timespan of an experiment. Averaged values of the *end-to-end processing latency* and *resource utilisation* metrics were calculated throughout each experiment considering multiple executions of the same workflow on different input data.

The IoT scenarios presented in Table 1 serve as possible models of real-life use cases that are highly relevant in today's IoT landscape. In particular, *Scenario A* reproduces the data generation rate that may occur in a *Smart Factory*. In such a setting, a range of IoT devices and sensors are strategically placed throughout the production floor to capture data related to machine performance, inventory levels, and product quality. The generated data, consisting of thousands of events per second, can provide valuable insights for optimising production processes, minimising downtime, and ensuring a secure working environment. On the other hand, the settings of *Scenario B* may resemble those found in a *Smart City* environment, where numerous sensors are deployed across an urban area to collect data on air quality, traffic flow, noise levels, and energy consumption. These sensors continuously generate a significant volume of events per second, with higher frequency than *Scenario A*, delivering real-time information essential for urban planning, resource management, and environmental monitoring [51].

Further details on the performance indicators utilised during the experiments and how they were calculated are described in the remainder of this section before some considerations on the auto-scaling features are presented.

5.1. End-to-end processing latency

The effective execution of real-time IoT analytics relies on minimum latency between data generation, invocation of the processing workflow, and availability of the results. During the considered experiments, these aspects were evaluated by analysing the metrics of type *(ii)* and *(iii)* discussed in Section 4.3.1. These include the *execution time* t_{ex} and the *request-reply* time t_{rr} associated with a single invocation of the *Real-time Edge Processing* workflow. Specifically, t_{ex} is the actual time required to execute the code that performs the requested processing tasks; t_{rr} indicates the overall latency detected by a *Software IoT Gateway*, i.e., the time elapsed between the submission of a processing request to the edge domain and the receipt of the related results. While the execution time t_{ex} is included within t_{rr}, the value $t_{rr} - t_{ex}$ indicates any overhead due to the invocation handling, such as process forking, the additional delay introduced by any intermediate components of the system (e.g., the OpenFaaS *API Gateway*, the *Watchdog*, etc.) and the network communication time.

5.2. Resource utilisation

The deployment of processing functions in close proximity to the IoT devices can reduce the *end-to-end processing latency* thanks to a smaller network communication latency. However, the amount of edge resources available for the execution of those applications is often limited—determining the *resource utilisation* associated with either the Microservice or the FaaS implementation of the *Real-time Edge Processing* is therefore strategic to evaluate the usability of edge nodes. This performance indicator was measured during the experiments via collecting several parameters associated with the metrics of types *(i)* and *(ii)* discussed in Section 4.3.1. These include $cpu_{\%}$ and mem, where $cpu_{\%}$ represents the average percentage of system CPU utilisation associated with the execution of the processing application during the lifetime of an experiment; mem indicates the average amount of RAM (in megabytes) utilised by the processing application throughout an experiment. While there were no restrictions on the memory amount that both functions and microservices could utilise, CPU limits were implemented to ensure the execution of each of their individual instances on a single core to facilitate the analysis of the results.

5.3. Auto-scaling

A key aspect of Microservice and FaaS architectures, and of lightwe ight containerisation, is the ability of these technologies to scale out the number of instances of a service component (or task) promptly and dynamically. Serverless FaaS frameworks inherently provide *auto-scaling* capabilities to their users. In OpenFaaS, for instance, when the number of invocations of a given function exceeds a specified threshold (of five requests per second, by default), an additional (configurable) number of instances of that function is spawned automatically by the API Gateway until a maximum value (20 by default) is reached.

The microservice architectural style was first described by Lewis and Fowler [52] as “an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API”. This architecture allows individual microservices to scale out, but this mechanism is not always enforced automatically. Some frameworks, such as the *Flask microframework* [35], only deal with the development aspects of the microservices. The microservice run-time management is performed by container orchestration engines that do not always support auto-scaling features. The *Docker Swarm* engine, for instance, does provide mechanisms for scaling the number of allocated containers at runtime; however, this is *not done automatically* and has to be triggered by an infrastructure administrator, e.g., via the command line interface [53]. *Auto-scaling* could still be built within those frameworks upon the existing manual scaling functionalities but this would require developing and integrating additional features in their vanilla versions. Other container orchestration engines have built-in container auto-scaling capabilities that can be applied to microservices. Yet, they often require external components and APIs to be configured and launched separately, such as in Kubernetes [54] and OpenShift [55].

For completeness, the performance evaluation that will be presented in Section 6 covers all the above-mentioned scaling alternatives. Specifically, it will investigate a serverless scenario with inherent auto-scaling capabilities provided by OpenFaaS. The results will be compared with experiments performed on microservices deployed via statically pre-allocated containers (which could be scaled manually) and via a Docker Swarm container orchestration engine that supports auto-scaling via an external component.

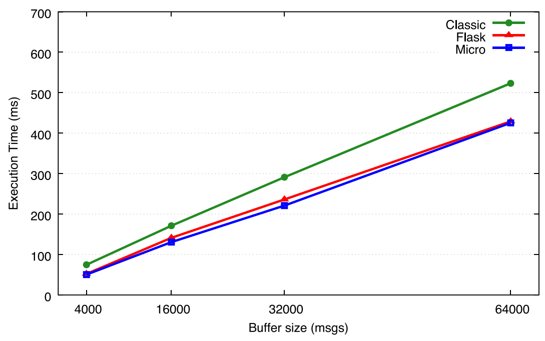
5.4. Frameworks resource utilisation

To ensure a fair comparison of the frameworks utilised for the deployment of the two versions of the *Real-time Edge Processing* application in the experimentation environment, a brief analysis of the resource utilisation associated with the execution of the main OpenFaaS components was compared with the resource required for the execution of the Docker Swarm container orchestration engine. This comparison is also included in the performance evaluation presented in the next section.

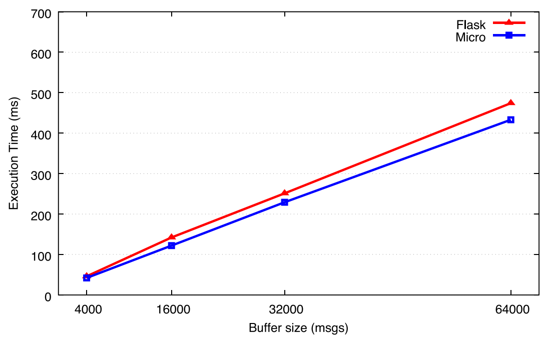
6. Performance evaluation results

This section presents the results of the experiments executed on the testbed to evaluate the performance of the *Real-time Edge Processing* subsystem, according to the methodology discussed earlier. The two FaaS implementations presented in Section 4.2.2, based on the *classic watchdog* and *of-watchdog* templates, were compared to the Microservice implementation discussed in Section 4.2.1. To simplify the description of the results, in the remainder of the paper, these implementations will be referred to as *Classic*, *Flask*, *Micro* and *Micro_s* (microservices with auto-scaling), respectively.

The average *execution time (t_{ex})* and the Cumulative Distribution Function (CDF) of the *request-reply time (t_{rr})* were utilised for quantifying the *end-to-end processing latency*, whereas the *CPU utilisation* and *Memory utilisation* shown in the graphs are related to the *resource utilisation* performance indicator of the edge nodes. All the graphs, whose data points have been calculated as the average of multiple values, show the associated 95% confidence interval as a shadowed area. However, it should be noted that such intervals are minimal in some of the graphs and, therefore, may not be visible.



(a) IoT Scenario A (1 gateway, 1000 sensors)



(b) IoT Scenario B (10 gateways, 10000 sensors)

Fig. 3. Execution Time (t_{ex}) of the *Real-time Edge Processing* in the considered IoT settings.

Auto-scaling considerations. Our experimentation setup relies on the auto-scaling capabilities natively provided by OpenFaaS. To ensure a comprehensive evaluation covering various scaling scenarios, we also developed an auto-scaler component deployed on the testbed during some of the experiments with Docker Swarm to provide auto-scaling capabilities for the microservices. This auto-scaler implements a feedback loop that monitors CPU core utilisation. Suppose a container utilisation of the assigned CPU core exceeds a pre-configured threshold (set at 85% in our tests) for more than one minute. In that case, the auto-scaler triggers a scaling-out action by instantiating additional containers of the respective microservice (in our tests, we used four in addition to the existing one).

Therefore, the microservice evaluation consisted of a group of experiments based on static container allocation (*Micro*) and another group executed via Docker Swarm and the above auto-scaler component (*Micro_s*). The auto-scaling was only triggered in OpenFaaS under the settings of IoT Scenario B, involving the *Flask* and the experiments with $b_s = 4000$ and $b_s = 16,000$. With identical IoT settings and function invocation rate, the *Classic* did not trigger the auto-scaling because the performed invocations reached a timeout before receiving a reply, as discussed later. The scaling-out action was triggered for the microservices under the settings of IoT Scenario B with $b_s = 16,000$, $b_s = 32,000$ and $b_s = 64,000$. Comparing the effectiveness of various auto-scaling mechanisms is a well-known research area [56] that we deem out of the main scope of this paper. However, it represents an interesting topic we plan to consider as potential future work.

6.1. End-to-end processing latency

In this subsection, Fig. 3 presents the measured *execution time* of the various implementations of the *Real-time Edge Processing*, whereas the associated distribution of the *request-reply time* is reported in Figs. 4 and 5.

6.1.1. Execution time

The graphs of Fig. 3 show on the y-axis the execution time t_{ex} of the experiments related to *IoT Scenario A* and *IoT Scenario B*, respectively. The different sizes of the batches of measurements b_s are reported on the x-axis. It can be observed that t_{ex} grows with the size of the data processed during each invocation. Looking at IoT Scenario A, this growth is faster for the *Classic* when compared to the *Flask* (Fig. 3(a)). This behaviour can be explained by considering how a function invocation is handled. The *Classic* is prone to an additional overhead, as a new process is forked per invocation; conversely, the *Flask* is devised to offset this latency by reusing the same process across different invocations. Even though t_{ex} does not measure this overhead directly, the collected values highlight a slight performance slowdown—the actual processing task took more time for the *Classic* because a given percentage of CPU resource was used for the handling of such process allocation. Hence, the t_{ex} measured for the *Classic* became on average 25% higher than the t_{ex} of the *Micro*.

When the *Flask* is considered, its behaviour in terms of t_{ex} is similar to that of the *Micro*. This result is expected as they are both based on similar software components. Moreover, the presence of the *of-watchdog* in the function implementation did not impact the t_{ex} in this experimental scenario. The overhead associated with a *Flask* invocation was minimal because, unlike the *Classic*, the same process could be reused between independent requests and no additional forking operations had to be executed.

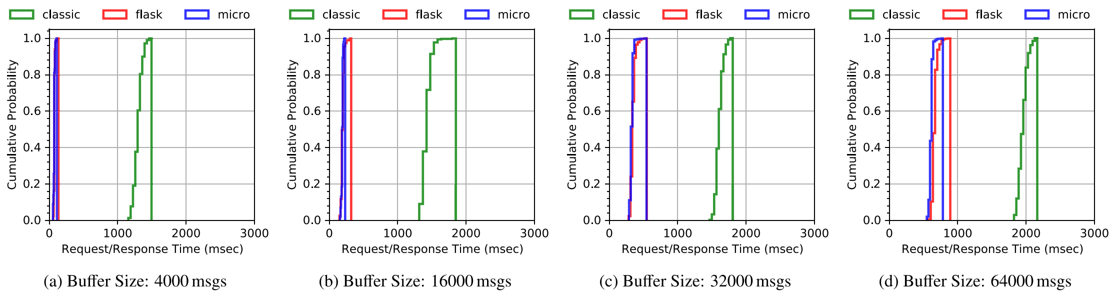
A different behaviour of the processing applications can be noticed in the experiments of IoT Scenario B. In this case, concurrent function invocations significantly impacted the performance of the *Classic*. The overhead due to the process forking was too high and required considerable CPU time. This prevented the *Classic* from working as expected, and the received invocations consistently reached a timeout in all the different experiments. Therefore, the measured t_{ex} values were not meaningful and have not been reported in Fig. 3(b).

Under the same IoT settings of Scenario B, the t_{ex} of the *Flask* was, on average, 15% higher than the one of the *Micro*. Since multiple parallel invocations were submitted to the processing applications during these experiments, the additional CPU time demanded by the presence of the *of-watchdog* became more substantial. This had a higher impact on the t_{ex} than the previous IoT scenario. The difference between the t_{ex} of the *Flask* and the t_{ex} of the *Micro* is 10% higher than the same average difference measured during the experiments of IoT Scenario A. The results related to this IoT scenario are similar to the previous one, even though the OpenFaaS Gateway triggered the autoscale for the *Flask* when $b_s = 4000$ and $b_s = 16,000$. Therefore, the t_{ex} measured for each invocation does not seem to be related to the total number of active function instances because these are *spread* by *Docker Swarm* on the available CPUs of the cloudlet. Likewise, the t_{ex} of the *Micro_s* did not change with the number of (container) instances allocated for that microservice when auto-scaling was triggered and, as a result, it is not shown in the graphs.

6.1.2. Request-reply time

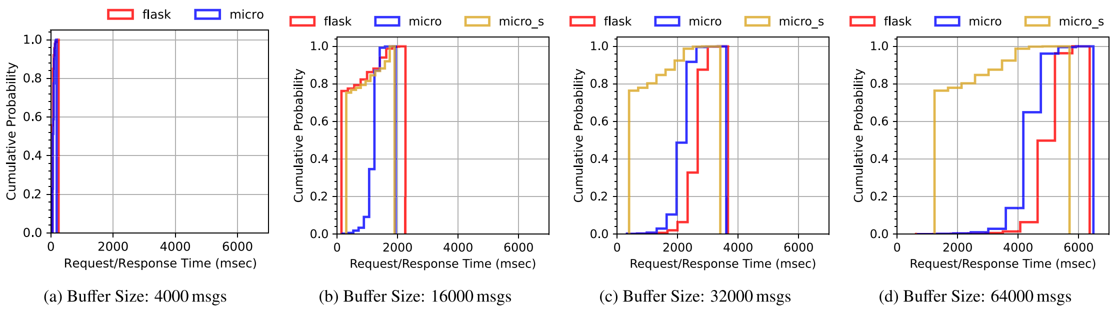
Figs. 4 and 5 show the CDF of the t_{rr} for the two considered IoT scenarios. As explained, during the experiments related to IoT Scenario B, the *Classic* failed to achieve real-time processing; in fact, the associated function invocations consistently reached a timeout in all the different experiments. Hence, the measured t_{rr} values have not been represented in Fig. 5.

As seen in Fig. 4, when IoT Scenario A is considered, both the *Flask* and the *Micro* implementations show a steadier and faster response time compared to the *Classic*. With $b_s = 4000$ (see Fig. 4(a)), the measured t_{rr} values are narrowly distributed around 100 ms for both the *Flask* and the *Micro*. On the same graph, the distribution of values of the *Classic* indicates a slower and less deterministic t_{rr}, with an average value of 1200 ms. The t_{rr} value increases with the buffer size b_s, due



(a) Buffer Size: 4000 msgs (b) Buffer Size: 16000 msgs (c) Buffer Size: 32000 msgs (d) Buffer Size: 64000 msgs

Fig. 4. CDF of Request-reply Time (t_{rr}) in IoT Scenario A (1 gateway, 1000 sensors).



(a) Buffer Size: 4000 msgs (b) Buffer Size: 16000 msgs (c) Buffer Size: 32000 msgs (d) Buffer Size: 64000 msgs

Fig. 5. CDF of Request-reply Time (t_{rr}) in IoT Scenario B (10 gateways, 10,000 sensors).

to the larger number of IoT messages to be transmitted and processed during a single invocation. The remaining graphs of Fig. 4 show that the above trend is not critical for both the *Flask* and the *Micro*, for which comparable t_{rr} were measured. However, one can notice a slightly slower and wider distribution of values associated with the *Flask*. This is due to the invocations routed through the OpenFaaS API Gateway and the *of-watchdog*. The same consideration also applies to the *Classic*, with the additional latency due to a new process being forked for each function invocation.

The CDFs of the t_{rr} for the experiments of IoT Scenario B are presented in Fig. 5. The *Classic* (not shown in the graphs) could not cope with the parallel function invocations due to the additional latency introduced by the process forking—the requests submitted by the software IoT Gateways reached a timeout before receiving the expected HTTP reply.

The graph of Fig. 5(a), related to $b_s = 4000$, highlights that the *Micro* has the best performance in terms of t_{rr}. The *Flask* shows a similar behaviour, although the measured t_{rr} is slightly more variable. The auto-scaling feature did not improve the t_{rr} in this case, and the overall measured t_{rr} is not impacted by the number of running function instances. We believe this is due to the small t_{ex} of each invocation when $b_s = 4000$. As such, the available instances were not effectively used in parallel, and the resulting t_{rr} did not change significantly. The *Micro_s* implementation did not exhibit auto-scaling due to the relatively low CPU utilisation. Hence, the resulting t_{rr} is similar to the one of the *Micro*, and it is not shown in the graph.

Fig. 5(b) shows the CDF of the t_{rr} when $b_s = 16,000$. It can be noticed that the *Flask* exhibits better performance than the *Micro*. Moreover, the distribution of the t_{rr} before the creation of the additional functions was around the value of 1500 ms, but after the instantiation of the additional function replicas, the t_{rr} went down to 150 ms. Hence, the auto-scaling effectively reduced the initial t_{rr} of a factor of approximately 1/10th. A similar behaviour can be observed for the *Micro_s*, where the allocation of additional replicas resulted in a performance improvement, with the measured t_{rr} decreasing to about 250 ms.

The CDFs of the t_{rr} when $b_s = 32,000$ and $b_s = 64,000$ are shown in Fig. 5(c) and Fig. 5(d), respectively. The rate of incoming requests was not as high as required to trigger auto-scaling in OpenFaaS. As for $b_s = 4000$, both the *Micro* and the *Flask* show similar CDFs for the t_{rr}, although the latter seems to perform slightly worse than the former. This is again due to the additional delay introduced by the OpenFaas API Gateway and the *of-watchdog*. The auto-scaler based on CPU load we used with the *Micro_s* did trigger the scale-out during these experiments, leading to a notable improvement of the t_{rr} thanks to the distribution of requests among all the available replicas.

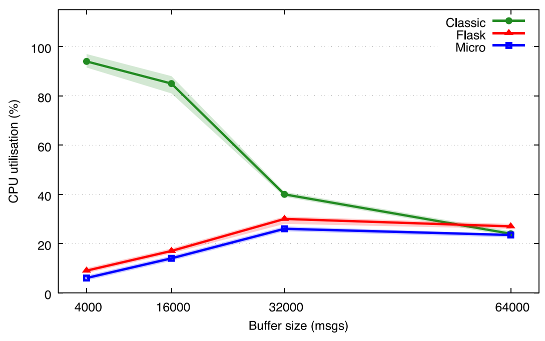
Although OpenFaaS provides auto-scaling features out-of-the-box, the scale-out was not automatically triggered under these circumstances. Therefore, further investigations may be required to identify the best metrics an auto-scaler should use to instantiate additional container or function instances. The effectiveness of using the number of received requests per second or the measured CPU load depends on the measurement generation rate and the size of the batch processing.

6.2. Resource utilisation

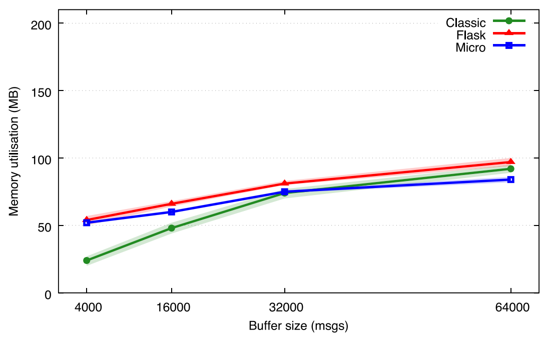
In this subsection, Fig. 6 shows the $cpu_{\%}$ and mem usage of the *Classic*, *Flask* and *Micro* implementations of the *Real-time Edge Processing* in IoT Scenario A, whereas the same performance metrics related to IoT Scenario B are reported on Fig. 7.

6.2.1. CPU utilisation

When IoT Scenario A is considered in Fig. 6(a), it can be noticed that the $cpu_{\%}$ usage of the *Classic* is significantly higher than the one of the other implementations. The highest $cpu_{\%}$ usage value (95%) is reached when $b_s = 4000$, and it is almost 90% higher than the one measured for the *Micro*. As b_s increases, the $cpu_{\%}$ metric of the *Classic* decreases; conversely, the *Micro* $cpu_{\%}$ metric increases. When $b_s = 64,000$, both implementations consume comparable CPU resources. As seen earlier, the overhead related to the process forking for the *Classic* overweights the actual processing tasks with small values of b_s (i.e., 4000 msgs). Therefore, with a higher frequency of function invocation, most CPU time is spent on the process forking. When $b_s = 64,000$, the number of invocations decreases and, as a result, more time is required for the actual computation; therefore, the measured CPU utilisation becomes closer to the one of the *Micro*.

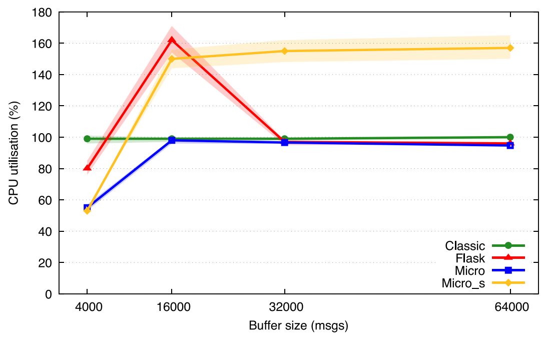


(a) CPU

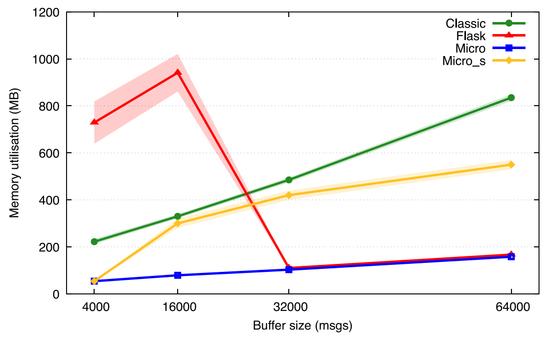


(b) Memory

Fig. 6. Resource Utilisation in IoT Scenario A (1 gateway, 1000 sensors).



(a) CPU



(b) Memory

Fig. 7. Resource Utilisation in IoT Scenario B (10 gateways, 10,000 sensors).

The opposite behaviour can be noticed for the *Flask*. The measured CPU usage increases with the selected size of the buffer. This result differs from the one of the *Classic*, as the related invocation overhead is considerably smaller. The *Flask* brings in a decrease of the CPU usage on an average of 40% with respect to the *Classic*. However, the *Flask* CPU usage is slightly higher than the *Micro* for a given size of the received input (on average 25% more). Again, this happens because of the overhead associated with the *of-watchdog* component.

The CPU usage related to IoT Scenario B is depicted in Fig. 7(a) (on a wider y-range than Fig. 6(a)). Again, it should be noted that the *Classic* failed to achieve the throughput required to cope with the rate of incoming concurrent requests. In fact, the *Classic* used on average around 100% of the CPU time just to perform process forking. The $cpu_{\%}$ usage metric of the *Flask*, presented on the same figure, deserves a more in-depth discussion. As mentioned earlier, the auto-scaling was triggered during the experiments with $b_s = 4000$ and $b_s = 16,000$. This is reflected by a higher average CPU utilisation than the *Micro* implementation because this metric is calculated as the sum of the CPU utilisation of all the allocated function instances of that type.

The graphs show that with $b_s = 4000$ and $b_s = 16,000$, the overall *Flask* CPU utilisation is higher than the one of the *Micro* due to the multiple function instances allocated during the related experiments. Moreover, since the number of instances changed dynamically during the experiment (from one to 20), the metrics' distribution associated with those batches is bimodal. This is reflected by the larger confidence intervals shown in the graphs. The number of function instances did not change during the experiments performed with $b_s = 32,000$ and $b_s = 64,000$, and the $cpu_{\%}$ metrics of the *Flask* and *Micro* implementations are similar.

In the experiment with $b_s = 4000$, the behaviour of the *Micro* and *Micro_s* is identical. The measured container CPU core utilisation never exceeded the pre-configured threshold (85%) required to trigger the auto-scaling. On the other hand, in the experiments with $b_s = 16,000$, $b_s = 32,000$ and $b_s = 64,000$, the same metric was above that threshold and approx 100%. As a result, with the *Micro_s*, the number of containers increased from one to five at run-time as the auto-scaler was triggered. This is reflected by higher CPU core utilisation than the *Micro* because this metric is calculated as an aggregated value of all the allocated containers, and each of the five containers had an average of 30% CPU core usage. Like with *Flask*, the number of instances changed dynamically during the experiment, leading to a bimodal distribution and the larger confidence intervals shown on the graph.

6.2.2. *Memory utilisation*

The average memory utilisation related to IoT Scenario A, shown in Fig. 6(b), indicates that the *Classic* requires slightly less memory than the *Micro*. This is justified by the on-demand instantiation mechanism on which the *Classic* implementation is based, which creates and destroys a process during each function invocation. Conversely, the *Flask* uses more memory than the *Micro*. Although those implementations are based on similar technologies, slightly higher memory consumption was measured for the *Flask* due to the *of-watchdog* component.

The memory consumption of the processing applications in IoT Scenario B (Fig. 7(b)) shows that the *Classic* required a substantial amount of memory, even though it failed to achieve the desired throughput (note a wider y-range than Fig. 6(b)). Again, this behaviour stems from the overhead of instantiating a separate process per function invocation, which resulted in most memory resources being used for this task rather than for the actual data computation. Like the CPU utilisation, the distribution of values of memory utilisation of the *Flask* when $b_s = 4000$ and $b_s = 16,000$ is bimodal since it reflects the dynamic allocation of function instances during these experiments. Therefore, in this case, the confidence intervals for $b_s = 4000$ and $b_s = 16,000$ are larger than those shown for $b_s = 32,000$ and $b_s = 64,000$. Moreover, as for the CPU utilisation, this metric is higher than that of the *Micro* when $b_s = 4000$ and $b_s = 16,000$, as it represents the sum of the memory

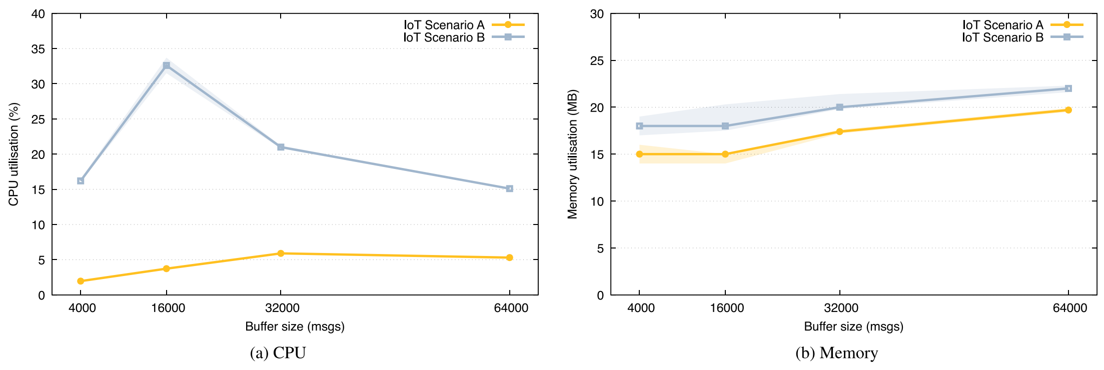


Fig. 8. Resource Utilisation of the OpenFaaS Gateway in the considered IoT settings (with *Flask*).

consumed by all the allocated function instances. Similarly, the *Micro_s* memory consumption is higher than the *Micro* when $b_s = 16,000$, $b_s = 32,000$ and $b_s = 64,000$. This is because it is calculated by adding up the memory consumption of individual containers allocated for the microservice after the auto-scaling has been performed.

6.3. Serverless computing platform resource overhead

This section focuses on analysing the resource utilisation related to the execution of the *OpenFaaS* platform. The graphs of Fig. 8 represent the CPU and Memory utilisation observed during the execution of the *Flask* experiments in the two IoT scenarios under consideration. It is worth noting that the resource utilisation of *OpenFaaS* components, apart from the API Gateway, was found to be negligible, and hence, it is not depicted in the presented graphs.

The $cpu_{\%}$ was mainly affected by the concurrent requests received by the *API Gateway* and the selected b_s. When in IoT Scenario A, a single software IoT Gateway was up and running, the resulting CPU utilisation was minimal and, on average, nearly 5 times smaller than the values observed for IoT Scenario B. It can be noticed that, in IoT Scenario A, the CPU utilisation presents a relative maximum when $b_s = 32,000$. The $cpu_{\%}$ values of IoT Scenario B show a relative maximum at $b_s = 16,000$. This trend highlights that the number of received invocations per second impacts CPU utilisation more than the number of messages sent with each invocation. Moreover, the measured value above 30%, reported for IoT Scenario B, indicates that the execution of the auto-scaling operations requires additional CPU resources.

The memory usage, shown in Fig. 8(b), is barely affected by the number of concurrent functions invocations, although the measured memory requirements for IoT Scenario B are slightly higher than the ones related to IoT Scenario A. In general, the number of messages b_s, sent with each function invocation seems to impact the memory usage, although the growth highlighted in the graph is not substantial. Finally, it can be noticed that the memory consumption is not strictly related to the execution of the auto-scaling mechanisms; this is different from what is reported on the $cpu_{\%}$ graph of Fig. 8(a).

6.4. Analysis of the results

A comprehensive analysis of the results of the previous experiments is discussed here. Since several tests have been executed under different settings, it may be hard to provide a definitive short answer to the initial research question asked in the paper. In fact, the pros and cons of using microservices and serverless functions for real-time IoT analytics at the edge depend on the specific IoT scenario where these technologies are used and the associated volume of the data streams they need to process. Therefore, an overview of what we learned from the experiments with regard to the *lifecycle*, *performance* and *resource utilisation* of *Microservice* ad *FaaS* architectures is provided below.

From a lifecycle perspective, building **Microservice** applications involves creating the necessary components using a framework like Flask, along with the direct engagement of developers in configuring and deploying the required containers. Consequently, the lifecycle of Microservice applications is less abstract compared to the FaaS technologies used in this work. As indicated by our prior findings [13], using JSON over HTTP/REST web interfaces in Flask microservices introduces performance and resource utilisation overheads. Nevertheless, compared to the FaaS technologies examined in this performance evaluation, microservices generally exhibited better *end-to-end processing latency* and *resource utilisation*.

This advantage is noticeable in IoT use cases that generate data volumes similar to IoT Scenario A, where microservices consistently demonstrated the most favourable *request-reply time* and exhibited the lowest *resource utilisation*. For example, a single instance of a microservice could process a buffered stream of data generated by 1000 sensors at a rate of 25 messages per second and provide a response to the IoT Gateway within 500 ms. Such processing might be accomplished on resource-constrained edge nodes, as evidenced by the CPU utilisation in this scenario, which slightly exceeds 20% of a single core, and memory consumption nearing 70MB.

In contrast to microservices, the lifecycle of **Flask** functions does not require the involvement of developers in low-level tasks associated with container configuration, creation, and instantiation, as the OpenFaaS framework abstracts these aspects. The performance of this technology demonstrates stability and consistency throughout the conducted tests, similar to the behaviour observed for microservices. Due to the ability to reuse the same process across separate function invocations, *Flask* exhibited acceptable response time and resource utilisation during the experiments. Consequently, this technology proves suitable for IoT analytics scenarios characterised by frequent invocations and varying data batch sizes. The *end-to-end processing latency* and *resource utilisation* indicators for *Flask* were only marginally inferior to those of microservices.

The advantages of employing *Flask* functions become particularly evident when very large data streams need to be processed, such as in the IoT use cases exemplified by Scenario B. In this scenario, OpenFaaS could dynamically scale the number of function instances on the available resources, responding to specific runtime conditions that impacted the workload. While auto-scaling necessitates additional computational resources, it effectively improved the *end-to-end processing latency* compared to static container allocation.

Specifically, by processing multiple parallel requests on the available replicas, *Flask* was able to achieve a minimum *end-to-end processing latency* of 150 ms when handling buffered requests generated by 10,000 sensors at a rate of 12 messages per second. This demonstrates a noteworthy improvement over statically allocated microservices, which proved nearly ten times slower under identical conditions. Allocating multiple function replicas naturally requires additional resources compared to a single microservice instance. However, based on the collected *resource utilisation* metric, such auto-scaling capabilities required two nearly fully utilised CPU cores and less than 1 GB of RAM in the above-mentioned IoT settings, aligning with the capabilities offered by typical edge devices. Furthermore, the advantage of this approach stems from the on-demand creation of replicas, eliminating the need for preallocation. This can result in more efficient resource utilisation than statically preallocated microservices, which is paramount from a provider viewpoint.

The **Microservice** application with **auto-scaling** also exhibited notable improvement compared to the static version and performance similar to or, in some cases, better than the *Flask* version. Specifically, it achieved a minimum *end-to-end processing latency* of roughly 1/5 of the static version using up to five containers. The improvement brought about by the additional auto-scaler component was notable when handling buffered requests generated by 10,000 sensors at a rate of 12 and 25 messages per second, buffered in batches of 16,000, 32,000 and 64,000 measurements. This was achieved thanks to the CPU core utilisation as the metric for triggering the scaling mechanism, which proved more effective than the number of received requests per second on which the OpenFaaS built-in auto-scaler is based (by default). Comments similar to those already made for the *Flask* also apply to this implementation, where the allocation of additional containers required further CPU core resources and memory, like in the scenario when up to 20 functions were allocated as a result of the auto-scaling enforced by OpenFaaS.

The **Classic** function offers the highest level of abstraction in terms of development and deployment lifecycle when compared to *Flask* and *Micro*. Similar to *Flask*, developers are relieved from container creation and management concerns, and the function logic does not require HTTP awareness. This flexibility allows for the inclusion of various process types, including binary or shell processes, within a function. However, it is important to note that this approach introduces a noticeable latency, as demonstrated by the experimental results. Consequently, the *Classic* function is not well-suited for processing scenarios resembling IoT Scenario B, as it cannot cope with the associated volume of generated data. Even in the settings of IoT Scenario A, where the data stream is continuous but less demanding, the experimental results indicate that the *Classic* implementation achieved acceptable performance only when dealing with less frequent invocations and larger batches of measurements.

Compared to the other implementations, the *Classic* exhibited significantly higher *end-to-end processing latency* and CPU utilisation while demonstrating slightly less demanding memory utilisation. This is due to the approach this technology uses, which optimises resource allocation avoiding the instantiation of processes for idle functions. Such resource management proved inadequate in the considered IoT settings, where the *Classic* function failed to handle multiple parallel incoming requests due to the high overhead associated with creating and destroying a process per invocation. Since this overhead consumed a significant portion of available CPU time and memory, we can conclude that the *Classic* function is better suited for scenarios where the invocation rate is low, and the amount of data to be processed per invocation is large.

7. Conclusions and future work

As *Edge Computing* aims to reduce the latency associated with data transfer dramatically, it represents a viable approach for IoT scenarios requiring latency-sensitive networks and computing resources for *real-time* process automation. Lightweight containers and continuous integration workflows have been sustaining the rise of modular and scalable software design patterns, such as those based on *microservices* and *functions*. *Serverless Computing* builds on these technologies and on event-driven programming to overcome the limitations of IaaS Clouds through a pure pay-per-use model with effortless scalability [12]. Through an extensive analysis of the software development and deployment *lifecycle*, as well as of the *performance* and *resource utilisation* of the above technologies, this paper attempted to answer the research question: *are Microservice and FaaS architectures viable technologies for building edge applications that perform real-time IoT analytics tasks?*

An analysis was conducted from the perspective of providers leveraging resources distributed across the Edge–Cloud continuum, with the primary goal of identifying what approach may enhance the *resource utilisation* of these providers' infrastructures specifically focusing on the *resource-constrained edge*. *Microservice* applications, built using frameworks like *Flask*, involve direct developer engagement in configuring and deploying containers. Compared to FaaS technologies, they generally exhibit better *end-to-end processing latency* and *resource utilisation*, despite the inherent overhead introduced by JSON over HTTP/REST interfaces. We found that a single microservice instance could be deployed on a resource-constrained edge node and process IoT data streams generated in Scenario A (Smart Factory) in real time.

Serverless functions, deployed via *OpenFaaS*, can effectively simplify the software development, configuration and deployment process. The *Classic* is the simplest approach that abstracts some of the implementation details and decouples the external HTTP interface of the function from the process performing the computation. However, allocating a separate process per invocation is also very slow, especially in the presence of multiple parallel requests, due to the resulting overhead. Compared to the other implementations, this approach showed the highest *resource utilisation* with the poorest *end-to-end processing latency*; hence, it could still be used on resource-constrained edge nodes but possibly only in those scenarios characterised by occasional bursts of data.

The *Flask* exhibited better performance than the *Classic*, and *resource utilisation* similar to the *Micro*, because of its internal process being reused among multiple invocations. However, this approach implies a slightly more complicated software development workflow, as a function needs to deal with HTTP requests and responses explicitly. During the experiments related to IoT Scenario A, with a single continuous stream of data sent to the processing applications, the *Flask end-to-end processing latency* and *resource utilisation* performance indicators were just slightly worse than the ones of the *Micro*. However, when considering parallel data streams to be processed, such as in IoT Scenario B (Smart City), OpenFaaS could automatically scale out the number of *Flask* function instances. This outperformed the simple usage of a fixed number of microservices deployed via *Docker Swarm* in terms of real-time response.

The *Micro_s* with custom *auto-scaler* also proved to be more effective than the static *Micro* in the same scenario, thanks to the additional containers allocated dynamically at run-time according to the CPU core utilisation. Additional resources were required in both cases to enable the auto-scaling; however, the experiments showed that such a mechanism could be performed on an edge cloudlet with relatively low resource capabilities, and these additional resources would only be consumed when requested by the workload.

Finally, the *OpenFaaS* components and the custom auto-scaler we developed exhibited acceptable resource utilisation—auto-scaling only had a minor impact on the CPU resources consumed by the API Gateway, and the resources required to run our auto-scaler on top of Docker Swarm were negligible. This slight overhead does not hinder OpenFaaS utilisation on resource-limited edge environments, and the advantages of simplified development, deployment, and effortless scalability might offset this overhead in some IoT use cases. On the other hand, the choice of the container orchestration engine for the microservices can determine the availability of built-in auto-scaling capabilities and the need to (develop and) deploy additional components.

From the above discussion, we can conclude that infrastructure or service providers might use *Microservice* and *FaaS* architectures to manage their resource infrastructures and enable customers to perform real-time IoT analytics at the edge; however, each of those technologies should be carefully selected according to the features—in terms of number of devices and volume of the generated streams of data—characterising an IoT scenario. The approach used by *OpenFaaS* with *Flask* functions, in particular, represents a promising solution for the effective extension of *Serverless Computing* towards the edge. This work demonstrated how this technology can ensure a good trade-off between the complexity of the lifecycle versus the performance achievable by using a limited number of edge resources, thanks also to the inherent auto-scaling mechanisms.

The outcome of this paper might be used as the basis for *future work* that explores interactions among multiple microservices or functions, and where alternative implementations of *Serverless Computing* based, for instance, on microVMs [22] could also be assessed. Further evaluations might be carried out to understand how these technologies perform compared to the *Flask* approach, to uncover what metrics should be used to trigger function and container auto-scaling consistently and identify the trade-off between the frequency of the invocation, the size of the data to be processed, and the resource utilisation. Furthermore, the results obtained could be used for the design and development of an energy-aware and resource-efficient *Serverless FaaS Framework*—using the full spectrum of resources of the *computing continuum*—to support the demanding real-time requirements of next-generation IoT applications in a wide range of domains.

Following the results of this work, further studies can investigate the details of resource and cost allocation within an organisation's infrastructure, which may help evaluate the need for leveraging third-party providers through a cost analysis model. The reason is that many new pricing models are now emerging in the IaaS landscape, in addition to the one where users are billed according to the CPU and memory size of the virtual machines they lease [28]. These cost models include, e.g., Pay-As-You-Go, Prepaid, Reserved Instances, AWS Saving Plan, and Spot Instances, or rely on actual execution time and actual memory consumption, as seen for serverless functions. Therefore, given the complexity introduced by these varied models, an exhaustive exploration of this subject is imperative for attaining definitive conclusions, and as such, it is earmarked for future investigation.

CRediT authorship contribution statement

Francesco Tusa: Writing – original draft, Writing – review & editing, Visualization, Software, Validation, Investigation, Conceptualization. **Stuart Clayman:** Writing – review & editing, Visualization, Validation, Investigation. **Alina Buzachis:** Visualization – original draft, Software. **Maria Fazio:** Methodology, Conceptualization.

Declaration of competing interest

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

Data availability

No data was used for the research described in the article.

Acknowledgments

Francesco Tusa is partially supported by the European Commission's H2020 project DIGITbrain (No. 952071) and Horizon Europe project Swarmchestrate (No. 101135012). Stuart Clayman is funded by the TUDOR project (UK DSIT).

References

[1] J.H. Nord, A. Koohang, J. Paliszkiewicz, The internet of things: Review and theoretical framework, Expert Syst. Appl. 133 (2019) 97–108.

[2] S. Verma, Y. Kawamoto, Z.M. Fadlullah, H. Nishiyama, N. Kato, A survey on network methodologies for real-time analytics of massive IoT data and open research issues, IEEE Commun. Surv. Tutor. 19 (3) (2017) 1457–1477.

[3] S. Trinks, C. Felden, Edge computing architecture to support real time analytic applications: A state-of-the-art within the application area of smart factory and industry 4.0, in: 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 2930–2939.

[4] P. Patel, M.I. Ali, A. Sheth, On using the intelligent edge for IoT analytics, IEEE Intell. Syst. 32 (5) (2017) 64–69.

[5] R. Mahmud, K. Ramamohanarao, R. Buyya, Application management in fog computing environments: A taxonomy, review and future directions, ACM Comput. Surv. 53 (4) (2020).

[6] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, O. Rana, The internet of things, fog and cloud continuum: Integration and challenges, Internet Things 3–4 (2018) 134–155, URL https://www.sciencedirect.com/science/article/pii/S2542660518300635.

[7] A. Ullah, T. Kiss, J. Kovács, F. Tusa, J. Deslauriers, H. Dagdeviren, R. Arjun, H. Hamzeh, Orchestration in the cloud-to-things compute continuum: taxonomy, survey and future directions, J. Cloud Comput. 12 (1) (2023) 135.

[8] F. Tusa, S. Clayman, End-to-end slices to orchestrate resources and services in the cloud-to-edge continuum, Future Gener. Comput. Syst. 141 (2023) 473–488.

[9] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, S. Pallickara, Serverless computing: An investigation of factors influencing microservice performance, in: 2018 IEEE International Conference on Cloud Engineering, IC2E, 2018, pp. 159–169.

[10] A. Celesti, D. Mulfari, A. Galletta, M. Fazio, L. Carnevale, M. Villari, A study on container virtualization for guarantee quality of service in cloud-of-things, Future Gener. Comput. Syst. 99 (2019) 356–364.

[11] J. Li, S.G. Kulkarni, K.K. Ramakrishnan, D. Li, Understanding open source serverless platforms: Design considerations and performance, in: Proceedings of the 5th International Workshop on Serverless Computing, WOSC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 37–42.

[12] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The rise of serverless computing, Commun. ACM 62 (12) (2019) 44–54.

[13] F. Tusa, S. Clayman, The impact of encoding and transport for massive real-time IoT data on edge resource consumption, J. Grid Comput. 19 (3) (2021) 32.

[14] M.S. Aslanpour, A.N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S.S. Gill, R. Gaire, S. Dustdar, Serverless edge computing: Vision and challenges, in: 2021 Australasian Computer Science Week Multiconference, ACSW '21, Association for Computing Machinery, New York, NY, USA, 2021.

[15] K. Cao, Y. Liu, G. Meng, Q. Sun, An overview on edge computing research, IEEE Access 8 (2020) 85714–85728.

[16] Docker: Accelerated, containerised, application development, docker, 2023, https://www.docker.com. (Accessed: June 2023).

[17] M. Alam, J. Rufino, J. Ferreira, S.H. Ahmed, N. Shah, Y. Chen, Orchestration of microservices for IoT using docker and edge computing, IEEE Commun. Mag. 56 (9) (2018) 118–123.

[18] F. Martella, G. Parrino, G. Ciulla, R. Bernardo, A. Celesti, M. Fazio, M. Villari, Virtual device model extending NGSI-LD for faas at the edge, in: Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021, 2021, pp. 660–667.

[19] J. Nupponen, D. Taibi, Serverless: What it is, what to do and what not to do, in: 2020 IEEE International Conference on Software Architecture Companion, ICSA-C, 2020, pp. 49–50.

[20] Compare serverless tools and services in the public cloud, 2023, https://www.techtarget.com/searchcloudcomputing/feature/Compare-serverless-tools-and-services-in-the-public-cloud. (Accessed: June 2023).

[21] OpenFaas: Serverless functions, made simple, 2023, https://www.openfaas.com. (Accessed: June 2023).

[22] M. Jain, Study of firecracker microvm, 2020, arXiv:2005.12821.

[23] S. Trilles, A. González-Pérez, J. Huerta, An IoT platform based on microservices and serverless paradigms for smart farming purposes, Sensors 20 (8) (2020).

[24] M. Großmann, C. Ioannidis, D.T. Le, Applicability of serverless computing in fog computing environments for IoT scenarios, in: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC '19 Companion, Association for Computing Machinery, New York, NY, USA, 2019, pp. 29–34.

[25] L. Baresi, D. Filgueira Mendonça, Towards a serverless platform for edge computing, in: 2019 IEEE International Conference on Fog Computing, ICFC, 2019, pp. 1–10.

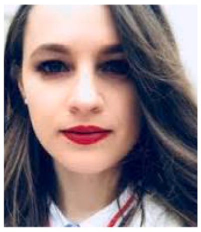
[26] A. Palade, A. Kazmi, S. Clarke, An evaluation of open source serverless computing frameworks support at the edge, in: 2019 IEEE World Congress on Services, Vol. 2642-939X, SERVICES, 2019, pp. 206–211.

[27] A. Alhindi, K. Djemame, F.B. Heravan, On the power consumption of serverless functions: An evaluation of OpenFaaS, in: 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing, UCC, 2022, pp. 366–371.

[28] F. Liu, Y. Niu, Demystifying the cost of serverless computing: Towards a win-win deal, IEEE Trans. Parallel Distrib. Syst. (2023) 1–15, http://dx.doi.org/10.1109/TPDS.2023.3330849.
[29] F. Tusa, S. Clayman, D. Valocchi, A. Galis, Multi-domain orchestration for the deployment and management of services on a slice enabled NFVI, in: IEEE Mobility Support in Slice-Based Network Control for Heterogeneous Environments Co-Hosted with Conference on Network Function Virtualization and Software Defined Networks, Verona, 2018.
[30] L. Mendiboure, M.-A. Chalouf, F. Krief, Edge computing based applications in vehicular environments: Comparative study and main issues, J. Comput. Sci. Tech. 34 (2019) 869–886.
[31] C.Y. Chen, J.H. Fu, T. Sung, P. Wang, E. Jou, M. Feng, Complex event processing for the internet of things and its applications, in: 2014 IEEE International Conference on Automation Science and Engineering (CASE), 2014, pp. 1144–1149.
[32] F. Tusa, S. Clayman, A. Galis, Real-time management and control of monitoring elements in dynamic cloud network systems, in: 2018 IEEE 7th International Conference on Cloud Networking, CloudNet, 2018, pp. 1–7.
[33] Lattice: Dynamic and programmable monitoring framework, 2023, https://github.com/UCL/lattice-monitoring-framework. (Accessed: June 2023).
[34] X.J. Hong, H. Sik Yang, Y.H. Kim, Performance analysis of restful API and rabbitmq for microservice web application, in: 2018 International Conference on Information and Communication Technology Convergence (ICTC), 2018, pp. 257–259.
[35] Flask web development, one drop at a time, 2022, https://flask.palletsprojects.com/en/1.1.x/. (Accessed: May 2022).
[36] Docker swarm: Cluster management and orchestration feature, 2022, https://docs.docker.com/engine/swarm/. (Accessed: May 2022).
[37] Docker swarm vs. Kubernetes: A comparison, 2023, https://www.ibm.com/cloud/blog/docker-swarm-vs-kubernetes-a-comparison. (Accessed: June 2023).
[38] Pandas open source data analysis and manipulation tool, 2022, https://pandas.pydata.org. (Accessed: May 2022).
[39] Docker —- deploy services to a swarm: placement preferences, 2023, https://docs.docker.com/engine/swarm/services/#placement-preferences. (Accessed: June 2023).
[40] Waitress pure-python WSGI server, 2022, https://docs.pylonsproject.org/projects/waitress/. (Accessed: May 2022).
[41] Kubernetes: Production-grade container orchestration, 2022, https://kubernetes.io. (Accessed: May 2022).
[42] OpenFaas – dude where's my coldstart?, 2023, https://www.openfaas.com/blog/what-serverless-coldstart/. (Accessed: June 2023).
[43] D. Bermbach, A.-S. Karakaya, S. Buchholz, Using application knowledge to reduce cold starts in faas services, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, pp. 134–143.
[44] P. Silva, D. Fireman, T.E. Pereira, Prebaking functions to warm the serverless cold start, in: Proceedings of the 21st International Middleware Conference, 2020, pp. 1–13.
[45] OpenFaas watchdog implementations, 2023, https://docs.openfaas.com/architecture/watchdog/. (Accessed: June 2023).
[46] Swarmprom: a starter kit for docker swarm monitoring, 2022, https://github.com/stefanprodan/swarmprom. (Accessed: May 2022).
[47] IoT experiment orchestrator, 2023, https://github.com/francesco-tusa/iot-orchestrator. (Accessed: June 2023).
[48] Fabric: Pythonic remote execution, 2022, http://www.fabfile.org. (Accessed: May 2022).
[49] K. Shafique, B.A. Khawaja, F. Sabir, S. Qazi, M. Mustaqim, Internet of things (IoT) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5G-IoT scenarios, IEEE Access 8 (2020) 23022–23040.
[50] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, M. Guo, The serverless computing survey: A technical primer for design architecture, ACM Comput. Surv. 54 (10s) (2022).
[51] K. Yasumoto, H. Yamaguchi, H. Shigeno, Survey of real-time processing technologies of iot data streams, J. Inf. Process. 24 (2) (2016) 195–202.
[52] J. Lewis, M. Fowler, Microservices, 2024, https://martinfowler.com/articles/microservices.html. (Accessed: January 2024).
[53] Scale the service in the swarm, 2024, https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/. (Accessed: January 2024).
[54] Kubernetes: Horizontal pod autoscaling, 2023, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. (Accessed: October 2023).
[55] RedHat OpenShift: Pod autoscaling, 2024, https://docs.openshift.com/container-platform/3.11/dev_guide/pod_autoscaling.html. (Accessed: January 2024).
[56] J.P.K.S. Nunes, T. Bianchi, Y. Iwasaki, E.Y. Nakagawa, State of the art on microservices autoscaling: An overview, in: Anais do XLVIII Seminário Integrado de Software e Hardware (SEMISH 2021), 2021, URL https://api.semanticscholar.org/CorpusID:237675969.

Francesco Tusa is an Assistant Professor in the School of Computer Science and Engineering at the University of Westminster. He has a PhD in Advanced Technologies for Information Engineering from the University of Messina. With expertise in distributed systems, virtualisation, cloud and fog computing, grid and high-performance computing, federated management, information security and the Internet of Things, he co-edited a book and co-authored over 50 conference and journal papers. He has also contributed to several EU-funded research projects and is a TPC member (and co-chair) of several conferences and a reviewer for various prestigious journals. His research currently focuses on exploring new fog computing paradigms for IoT, based on serverless computing and function as a service, while also applying homomorphic encryption techniques to the cloud and network domains.

Stuart Clayman received his Ph.D. in Computer Science from University College London in 1994. He is currently a Principal Research Fellow at UCL EEE department, and he has worked as a Research Lecturer at Kingston University and at UCL. He co-authored over 70 conference and journal papers. His research interests and expertise lie in the areas of software engineering and programming paradigms; distributed systems; virtualised compute and network systems, network and systems management; sensor systems and smart city platforms, and artificial intelligence systems. He is looking at enhanced mechanisms for end-to-end delivery of digital video, as well as new techniques for large-scale sensor systems in Industry 4.0. He also has extensive experience in the commercial arena undertaking architecture and development for software engineering, distributed systems and networking systems. He has run his own technology start-up in the area of NoSQL databases, sensor data, and digital media.



Alina Buzachis holds a Ph.D in Distributed Systems at the Mediterranean University of Reggio Calabria in April 2021. She received her master's degree cum laude in Engineering and Computer Science at the University of Messina, in October 2017. She primarily works with cloud technologies and is also interested in deep learning and data mining.

Maria Fazio is Associate Professor in Computer Science at the University of Messina (Italy). She was involved in several national and international projects, such as the EU-FP7 RESERVOIR, EU-FP7 VISION Cloud, EU-FP7 CloudWave, EU-H2020 BEACON, EU-H2020 URBANITE. She is member of the Editorial Review Board of several international journals and has published more than 150 papers. Currently, she is responsible for the University of Messina of the CINI "HPC: Key Technologies and Tools" (HPC-KTT) National Laboratory. Her research interests are focused on Computing continuum, with particular reference to intelligent microservice orchestration, auto-configuring mesh networks at the Edge, security in IoT-Edge ecosystems.