IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 July 2022, accepted 16 August 2022, date of publication 22 August 2022, date of current version 31 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3200666

RESEARCH ARTICLE

Utilizing Microservices Architecture for Enhanced Service Sharing in IoT Edge Environments

KHALED ALANEZI[1] AND SHIVAKANT MISHRA[2]
[1]Department of Computing, College of Basic Education, PAAET, Safat 12064, Kuwait
[2]Computer Science Department, University of Colorado, Boulder, CO 80309, USA

Corresponding author: Khaled Alanezi (kaa.alanezi@paaet.edu.kw)

ABSTRACT Latency sensitive IoT (Internet of Things) applications at the edge are designed using a microservice-based architecture. This architecture is comprised of a set of microservices, each implementing a simple functionality with clearly-defined interfaces, and applications are constructed by selecting and interconnecting appropriate microservices. To understand the performance implications of using a microservice-based architecture for constructing IoT applications at the edge, this paper provides a detailed evaluation based on an actual prototpye implementation and performance measurement. In our setup, an edge server fulfills dual roles of being an administrative controller of the IoT infrastructure and satisfying application's latency and privacy constraints. We demonstrate the utility of this architecture by isolated and independent implementation of different microservices, constructing an IoT application by interconnecting these microservices, and potential sharing of microservices between different IoT applications running simultaneously to enhance interoperability. Finally, we provide an extensive performance evaluation focusing on application latency as well as CPU and memory consumption.

INDEX TERMS Containers, Docker, edge computing, Internet of Things (IoT), microservice, virtualization.

I. INTRODUCTION

The Internet of Things (IoT) promise to interconnect billions of geographically-distributed heterogeneous devices has unleashed myriad of applications in the fields of smart cities, smart health monitoring, industrial IoT (IIoT), smart agriculture, and several other areas. By immersing an environment with sensors and actuators, businesses gather enormous volume of multimodal, live-streaming data that is then fed to smart decision systems to enable efficient and smart operations. Cloud computing is utilized by IoT solutions to aid in storage and analytics for such a large volume of data. In particular, the elasticity of cloud service provisions, an ability to automatically scale up or down based on IoT workloads allows IoT applications to tap into the abundant storage and computation resources that cloud services provide in a cost-effective way.

However, utilizing cloud resources in IoT applications suffers from two limitations: high latency and high network bandwidth consumption. Indeed, it is expected that the Internet backbone networks will be overwhelmed by the amount of data streamed to cloud servers as IoT applications become mainstream. As a result, edge computing (also known as fog computing) has been proposed to overcome the two limitations of utilizing cloud resources [1]. Fog nodes (also known as edge servers) [2] are defined as virtualized platforms placed at the edge of the network to bring cloud services closer to IoT nodes. The local processing introduced by the resultant Cloud-Fog-IoT architecture provides great benefits such as supporting latency-sensitive applications, enhanced privacy, and reduced workloads reaching the backbone of the network.

Despite these benefits, building and integrating IoT applications on top of a Cloud-Fog-IoT infrastructure is inherently challenging due to its dispersed nature. Consequently, IoT vendors opt for implementing IoT solutions as end-to-end silos to fully control the application flow. Indeed, a major challenge hindering wide adoption of IoT is the lack of a standard to enable interoperability among different solutions and components of IoT applications. Popular IoT applications that are commercially available at present, such as intelligent door locks, camera-based home surveillance systems,

The associate editor coordinating the review of this manuscript and approving it for publication was Yassine Maleh.

smart home temperature control systems, and even smart city applications and services are implemented as islands. They follow an end-to-end design spanning smart sensors, fog nodes and cloud providers without any possibility of component/code reuse. For example, a generic temperature sensor used in a smart home application cannot be shared with another application that serves the purpose of a fire alarm. Similarly, a face detection component in a home surveillance application cannot be reused by another application, e.g. a smart home temperature control system.

To address this limitation, a *microservice-based architecture* has been proposed [3]. In this architecture, an application is built from a selection of individually isolated microservices, where each microservice is a self-contained piece of software with clearly-defined interfaces implementing a simple and well-defined functionality. Compared to a traditional monolithic model, such an architecture promotes modularity, code reuse, scalability and resiliency [4]. Figure 1 illustrates a motivating example of this architecture, comprised of six microservices.

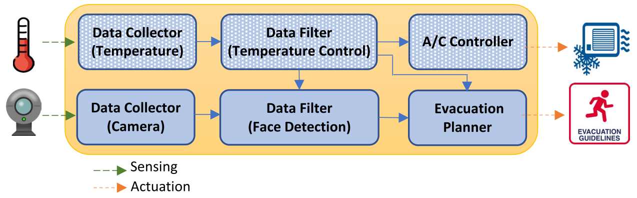


FIGURE 1. An example of a microservices-based architecture.

In this example, the six microservices are used to compose two applications, one to operate A/C control settings and the other to detect if there is a fire and develop an evacuation plan. The **Data Filter (Temperature Control)** microservice detects changes in temperature to control AC as well as abrupt changes in temperature indicating possible fire. The **A/C Controller** microservice controls the A/C settings based on data received from the **Data Filter (Temperature Control)** microservice while the **Evacuation Planner** microservice develops an evacuation plan based on the data received from the **Data Filter (Temperature Control)** and **Data Filter (Face Detection)** microservices. Of course, the fire-alarm functionality of the architecture is oversimplified for the sake of example clarity. A full-fledged fire-alarm/evacuation system would have so much more details into it beyond our scope.

The above described example architecture illustrates three important characteristics of a microservices-based architecture. First, each microservice has been designed and implemented independent of other mocroservices. Second, different applications can be constructed by interconnecting these microservices appropriately. Finally, a microservice can be reused and shared by multiple applications running simultaneously.

Indeed, there have been several research efforts in recent years that address different aspects of building a microservice-based architecture ranging from efficient microservice implementation, architecture scalability, distribution and load balancing, orchestration and other system level services [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. An overview of this work is provided in Section II. Despite a plethora of work in this area recently, there is a lack of evaluation of a microservice-based architecture based on an actual prototpye implementation and performance measurement. This paper addresses this issue.

In particular, this paper evaluates the design, implementation and performance of a microservice-based architecture of an edge server to implement end-to-end IoT applications. In particular, we provide a microservice-based design and implementation of an edge server architecture and demonstrate its utility by constructing multiple IoT applications and evaluate their performance. The paper makes the following important contributions:

1) We provide a detailed design and implementation of an edge server architecture that fulfills dual roles of (1) being an administrative controller of an IoT infrastructure, and (2) satisfying application's latency and privacy constraints by acting as computing and communication continuum between low end sensor devices and the high end cloud resources.
2) We demonstrate the utility of a microservice-based edge server architecture illustrating isolated and independent implementation of different microservices, constructing an IoT application by interconnecting these microservices, and potential sharing of microservices between different IoT applications running simultaneously. This design leads to enhanced interoperability.
3) We provide an extensive performance evaluation of the edge server architecture focusing on application latency as well as CPU and memory consumption.

II. RELATED WORK

A. GENERAL LITERATURE REVIEW

This paper utilizes a mixture of techniques and concepts discussed in literature to tackle the issue of fragmentation in IoT environments thereby enabling resource sharing and enhancing interoperability among IoT solutions. We begin by discussing research works tackling the lack of interoperability in IoT environments, which is the main issue hindering wide adoption of IoT technologies and our main goal to tackle in this paper. The survey paper by Noura *et al.* [15] provides a summary of research works in this area and discusses open issues. It lists adopting the edge computing paradigm as a contributor for enhanced interoparbility in IoT. We share the same vision in our architecture by employing an edge server for coordination and orchestration of service sharing in the IoT environment. The BIG IoT API [16] provides an architecture for tackling the problem of lack of standards for IoT interoperability based on lessons learned from the IoT EPI project [17]. The presented architecture provides standard

APIs, resource descriptors and a marketplace to monetize access to IoT resources as a means for building a holistic IoT ecosystem. Our architecture similarly employs APIs with the focus on resource sharing in the IoT environment but mainly focuses on utilizing the edge server for this means. Another architecture presented by Aloi *et al.* [18] envisions the smartphone to play a central role in enabling IoT interoperability. A smartphone gateway application is provided to handle various communication protocols and manage IoT devices. We also adopted the smartphone for representing user interests in the IoT environment in our previous work [19] but our focus in this work is on the IoT deployment owner interests represented by the edge server in our architecture. Finally, the Hypercat [20] and SGS [21] are also solutions directed at enhanced IoT interoparability. However, the focus of these middlewares is building standard semantics for gathering information about the IoT environment while we focus on the systems aspects of utilizing containers to support service sharing.

Another research area related to our work is about utilizing microservices that run on edge servers or clouds to facilitate running IoT applications. Morabito *et al.* [22] proposed a gateway running on single-board computers SBCs to manage heterogeneity in IoT. An evaluation for the performance of this architecture is then presented. Our architecture has similar approach but we employ our solution on a high-end edge server and implement the temperature pipeline as an example application to utilize it. Alam *et al.* [23] also proposed a microservices architecture utilizing Docker to support IoT applications. The presented architecture runs on the Cloud-Fog-Edge continuum with the aim of enhanced modularity and providing fault tolerance. We share the same techincal vision of using Docker to decompose the IoT application into microservices for better modularity. But, we utilize an edge server to run the architecture with the aim of bringing cloud-like hardware performance near the vicinity of the typically low-end edge devices in resemblance to cloudlets [24]. More recently, Con-Pi [25] was introduced to enable resource sharing in edge/fog computing environments by running IoT applications as microservices, which is similar to our objective. However, Con-Pi is targeted towards running containers on SBCs while our presented architecture runs on a high-end edge server (i.e. cloudlet paradigm). Finally, a work by Ahmed and Pierre [26] looked onto improving the performance of Docker to better run microservices on microcontrollers such as the Raspberry Pi.

Cloud platforms have also been utilized to host software infrastructure to support IoT applications [27]. In this solution, cloud resources were utilized for better processing and storage of IoT stream data. It is true that cloud infrastructure should ultimately provide the final destination for processing and storage of IoT data. Nonetheless, the edge server in our work provides staged processing before reaching to the cloud. Our objective is to utilize this unique position to address heterogeneity in IoT environments. Last, IoTDoc [28] utilizes IoT devices to promptly form a mobile cloud for running a docker-based architecture to support IoT applications. The provided solution is more cost-efficient when compared to cloud usage. Indeed, cost is an important factor when designing systems for the Cloud-Fog-Edge continuum. The business viability of our approach of installing cloudlets is discussed in a survey paper [29] by Shaukat *et al.*

TABLE 1. Comparison of our work to generic microservice-based edge frameworks for IoT.

	Main Focus									
	Objective(s)							Evaluation		
	Service Orchestration	Interoperability	Scalability	Trust, Privacy and Security	Fault Tolerance	Energy/Resource Optimization	Quality of Service	Prototype Implementation	Performance Evaluation	Simulation
Taherizadeh *et al.* [5]	X						X	X	X	
Filip *et al.* [6]	X					X	X			X
Gaur *et al.* [7]	X						X	X	X	
Samanta and Tang [8]	X					X	X			X
Fernandez *et al.* [9]	X					X		X	X	
Jin *et al.* [10]	X			X		X			X	
Alam *et al.* [23]	X		X		X				X	
Morabito *et al.* [22]	X	X						X	X	
Mahmud *et al.* [25]	X		X			X		X	X	
Pallewatta *et al.* [11]	X		X				X			X
Islam *et. al.* [12]	X					X			X	
Ioini and Pahl [13]	X			X						
Javed *et al.* [14]	X				X			X		
Our work	X	X						X	X	

B. COMPARISON TO OTHER FRAMEWORKS

This section illustrates our contributions by juxtaposing our work to relevant literature as shown in Table 1. We included in this survey generic microservice-based edge frameworks supporting IoT. Frameworks targeting special IoT use cases such as industrial IoT (IIoT) or mobile health (mHealth) were excluded. As seen in the table, we listed service orchestration as one of the objectives to reflect to readers that all works utilize it to achieve a single or combination of goals to tackle well know IoT challenges. Service orchestration is the automated management of service lifecycle including service starting, deployment, scaling up and termination. As expected, acheiving energy/resource optimization and providing QoS (i.e. better time performance and responsiveness) are the most prevalent goals for the various frameworks, which is due to the resource-limited nature of edge devices. Other objectives include scalability, security and fault tolerance. We notice that only our work and the work by Morabito *et al.* [22] focused on interoperability. To evaluate their proposed framework, most works focused on measuring the performance (time, CPU utilization and memory footprint) of implemented microservices over limited-resource edge devices. This is done by deploying containers on single board computers SBCs such as the raspberry pi. Using pure simulation was also another option to evaluate the proposed ideas. Similar to our experimental work, a prototype implementation involving an actual IoT application utilizing sensors and edge devices was also explored by other researchers. The contribution we provide over similar works presented in Table 1 stems from providing a step-by-step detailed prototype implementation and evaluation for the proposed architecture. The provided details inform other researchers while architecting implementations specific to their use case.

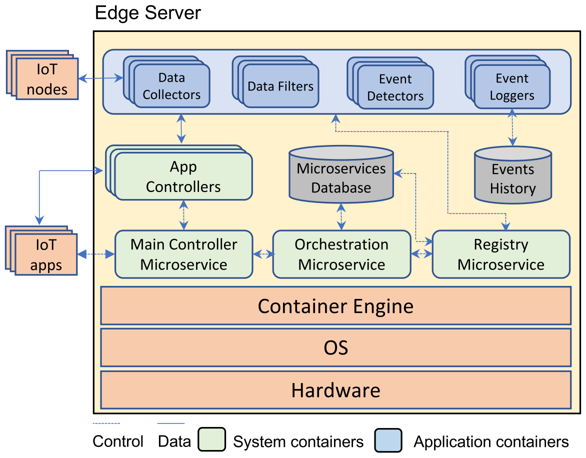


FIGURE 2. **Solution architecture.**

III. DESIGN

We adopt a microservice-based architecture for edge servers to tackle the complexity of developing dispersed IoT applications to run on the Cloud-Fog-IoT continuum. In essence, microservice-based architectures [30] have been suggested in software engineering industry as a means for replacing monolithic architectures by decomposing any application into a set of independent services. This proposal brings in great flexibility since microservices can be written using different programming languages and by different development teams. A microserivce implements a clearly defined unit of functionality and defines a communication channel based on a lightweight mechanism, typically HTTP, to integrate with other microservices and to interact with the outside world.

In our design, an edge server serves dual purposes. First, it has a low-latency connection of one wireless hop to all sensing and actuating devices in an IoT environment (e.g. a home, a building, an agricultural field, etc.), and is typically located close to these devices. Thereby, it has a birds-eye-view of the entire IoT environment and can coordinate access, sharing and operation of all these devices among multiple IoT applications running concurrently. In this role, the edge server is a part of the administrative domain [31] of the IoT environment and represents the administrators by enforcing their admission control and security policies. Second, the edge server provides an intermediate computing and communication platform that IoT applications may use for computing or pre-processing in accordance with their latency, privacy, storage and/or cost constraints.

A. ARCHITECTURE

We employ container technology [32] to design and implement individual microservices and facilitate communication between them. Containers provide a very convenient way to implement a microservice-based architecture, wherein each microservice is implemented as a separate container and applications are constructed by interconnecting these containers as needed. Here we adopt the technique [33] of decomposing the application into a Directed Acyclic Graph DAG, or simply chained services, to enable distributed execution across the edge and the cloud. Our proposed edge server architecture is shown in Figure 2. The edge server [1], also known as a cloudlet [24] or a fog node [2] is a trusted, resource-rich compute box located at the edge of network. It is installed and managed by the owner/administrator of an IoT environment and has one-hop wireless access with physically co-located IoT sensor and actuator devices. The edge server in our design runs a **Container Engine** installed on top of the local hardware and operating system to provide lightweight container-based virtualization.

As shown in the figure, our edge server architecture is comprised of a set of microservices (containers) that can be divided into system and application microserivces. **IoT applications** are the clients to be served by this architecture. Essentially, the main purpose of the architecture is to allow IoT applications to discover and share services available in the environment. We believe that, if engineered carefully, applications will share large number of components. For example, the camera sensor can be shared by two applications one requiring face recognition and another requiring intruder alert only. To discover and consume the services provided by the framework, **IoT applications** first contact the **Main Controller Microservice**. This contact happens only for the first time to learn the application requirements and devise an execution plan according to the currently available resources.The **Main Controller Microservice** directs new IoT client requests to the **Orchestration Microservice**, which is the focal point for devising execution plans. In order to come up with the best execution plan, this latter microserivce requires information about current running microservices and applications in the system. This information can be acquired by contacting the **Registry Microservice**

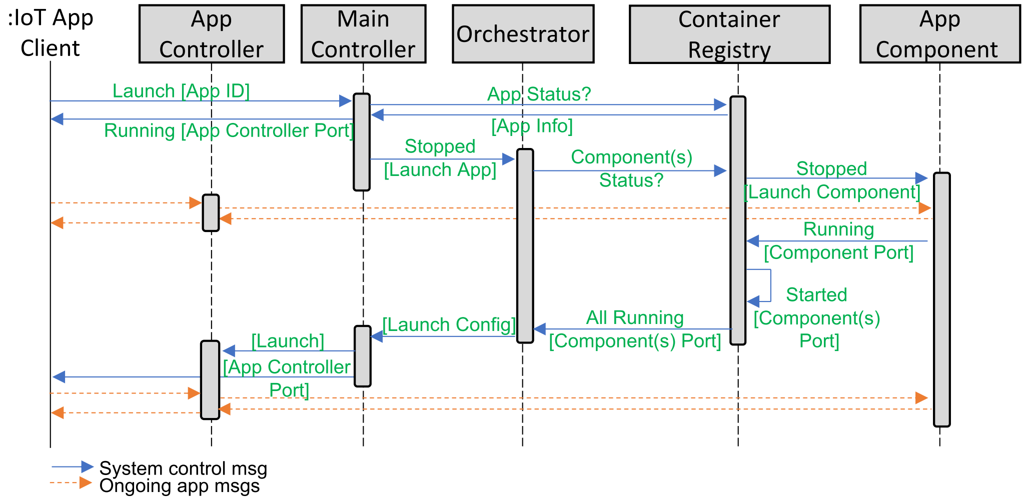


FIGURE 3. Sequence diagram for serving IoT App client using the architecture.

which gathers the needed information from the system by contacting the **Microservices Database**. Upon devising a new execution plan, the responsibility for executing it is then delegated to an **App Controller**. Any future requests for the same **IoT application** will be directed to this **App Controller**, to avoid any redundancy in service execution.

So far, we have described the functions of system containers that are illustrated in green color and the corresponding control signals by dashed arrows in Figure 2. In Section III-B we discuss an example scenario where we draw a sequence diagram depicting the interactions between the components to serve a specific **IoT application** request. In addition to the system containers, applications are decomposed onto application containers or microservices (illustrated in blue color). IoT applications require generic microservices that can be either chained to run sequentially or in parallel depending on the nature of the task. We envision four types of generic application-specific microservices. First, the **Data Collectors** microservices interface directly with the **IoT nodes** (sensors and/or actuators) in the environment. These microservices serve as a gateway for any future request to share the IoT node resources. For example, a data collector microserivce could represent a camera sensor and expose APIs for getting pictures and/or videos from the camera whenever required by an application. Second, the **Data Filter** microservices store any data retrieved from the sensor in a data store and provide any needed logic relevant to the sensor data. Using the same camera example, the data filter stores all pictures and video clips retrieved from the camera and exposes APIs for any advanced processing based on this information (e.g. an API for face detection). Third, the **Event Detectors** microservices are responsible for detecting any triggers requested by the user which could be based on raw sensor data or filters implemented by the **Data Filters**. Finally, the **Event Logger** microservices keep a history for earlier triggers in case they are needed by any application in the future.

B. IoT APP EXECUTION FLOW

To explain how this proposed architecture is used, we describe in detail the interactions among various subcomponents to execute an IoT application using the sequence diagram shown in Figure 3. First, an **IoT App Client** sends a request to the **Main Controller** service requesting to run a particular application that is uniquely identified by an App ID. The **Main Controller** then contacts the **Registry** to discover whether this request is to start a new application, or if the requested application is already running and the client could simply join this ongoing application. As discussed in Section III-A, the execution of each IoT application is handled by a corresponding **App Controller**. Hence, in case the requested IoT application is found to be already running, the request needs to be routed to the existing **App Controller** serving the application. Hence, the **Registry** will reply with the application information to the **Main Controller** which forwards the **App Controller** port number to the requesting client. After that, the **IoT App Client** has an ongoing communication (orange dashed arrows) with the **App Controller** which communicates with the **application Components** implementing the application logic.

Now we turn to describing the interactions that occur when the requested application is not running currently, which requires negotiating an execution plan and starting a new **App Controller**. In this case, the application information returned by the **Registry** indicates that the application status is stopped. The **Main Controller** then forwards the request to the **Orchestrator** who is responsible for starting the necessary components needed to start the new application. Here, the application has alternative DAG execution graphs [33] and the goal of the **Orchestrator** is to start a DAG graph with the most components that are already started by currently running applications to maximize sharing of resources. DAG information for popular applications are known before hand while for other applications they can be simply provided to the framework by the requesting client. The **Orchestrator** starts contacting the **Registry** to discover if some of the needed components are already running. If this was true, an opportunity for sharing these microservices across IoT applications is exploited to bring the benefit of conserving resources and enabling more IoT applications. For those needed microservices that are not running, the **Registry** launches those microservices. The information of those microservices along with the already running microservices is then returned to the **Orchestrator**. This information is used by the **Orchestrator** to devise a launch configuration and sending it to the main controller. The main controller launches a corresponding **App Controller** to serve this new application and returns the **App Controller** port number to the client.

IV. IMPLEMENTATION AND EVALUATION

The proposed architecture described in Section III depends heavily on containerization technology for virtualization. We utilized the docker engine [32] to implement a prototype for the proposed architecture where each docker container is used to implement an application or system microservice. This section provides details for this prototype along with extensive evaluation of its performance.

A. INTER-MICROSERVICE COMMUNICATION

As the performance of the architecture is governed by the overhead resulting from inter-docker containers communication, we started the evaluation by measuring this overhead. When looking at the architecture shown in Figure 2, one can identify three types of communication schemes among microservices. Those three schemes are depicted in Figure 4. First, the **Host Inquiry** scheme represents a consumer microservice sending a request to a provider microservice running on the same host. Here the communication happens by means of a docker network bridge [34], which is a software bridge that permits group of containers attached to it to communicate. It provides isolation since other containers running on the same host cannot communicate with the group of containers as they are not connected to the same bridge. Note here that the provider microservice exposes APIs for other microservices to consume its services. For example, a face detection microservice will expose RESTful APIs where an image can be posted using HTTP POST and the number of faces and their locations in the picture is returned. In order to expose the APIs, the provider microservices typically run a web server to establish various HTTP API types of GET, POST, PUT, UPDATE and DELETE. Those APIs will also carry some sort of computational task provided by libraries as needed such as the face recognition library needed to perform face detection in the aforementioned example.

Second, the **Database Inquiry** communication scheme represents a scenario where the provider microservice requires accessing a data store before replying to the consumer inquiry. This situation represents retrieving a stored piece of data and performing some kind of computation or just returning the data itself. Last, the **Network Inquiry** represents a situation where the consumer communicates with the provider service running on another host via wireless link (typically Wi-Fi). This scenario is akin to the IoT application acting as the consumer and calling the provider service running on the edge server.

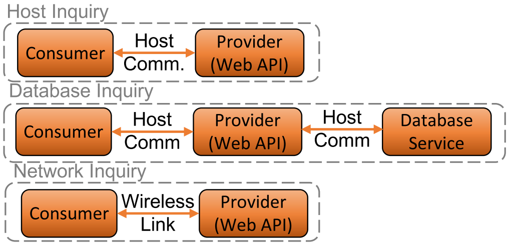


FIGURE 4. Implemented inter-container communication types.

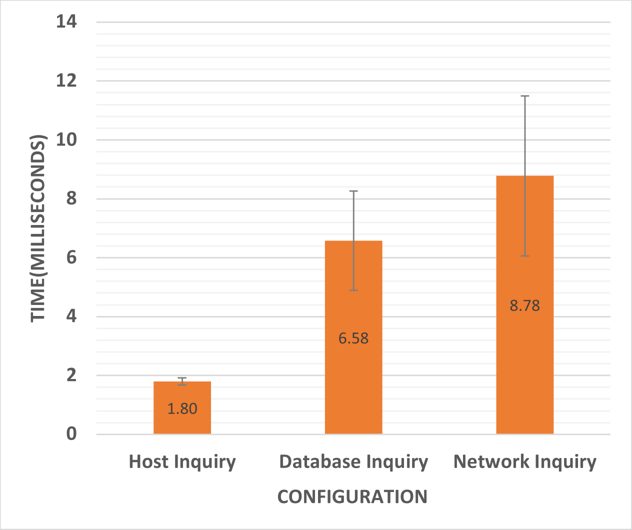


FIGURE 5. Time performance for the three inter-container communication schemes.

We begin by depicting the time performance of these three communication schemes in Figure 5. Note that we have performed each experiment five times and report the standard error for each experiment. For stress testing the schemes, we launch tens of threads bombarding the service with hundreds of requests as will be described later. From the figure we see that the host inquiry takes an average of 1.8 milliseconds of round trip time. This is the time it takes to send an HTTP get request to the API exposed by the web server implemented in the provider microservice and getting the response back. We deliberately did not do any processing at the provider side so as to exclude this factor as it hugely varies depending on the required computation type. The reported number simply reflects the communication cost. Note here that to implement the provider microservice, we containerized a web application that was implemented using Java Spring Boot [35]. This application exposed simple APIs to resemble a provider service. Now when looking at the

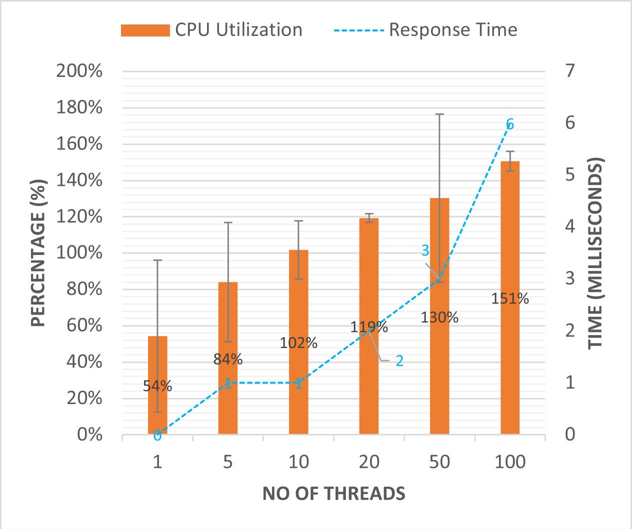


FIGURE 6. CPU utilization/response time for the Host Inquiry configuration under varying workloads.

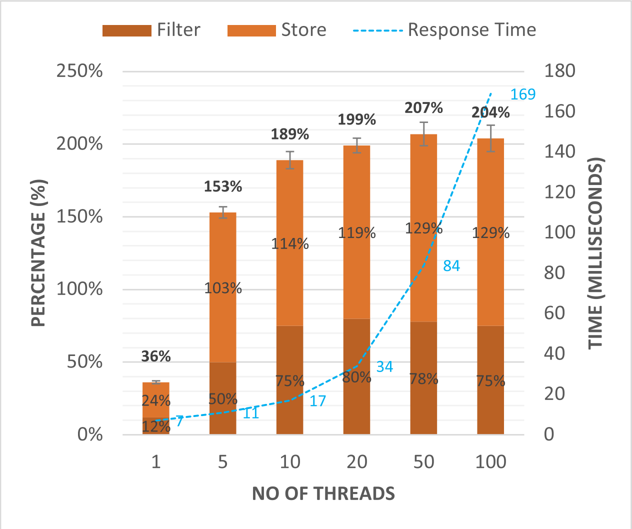


FIGURE 7. CPU utilization/response time for the database inquiry configuration under varying workloads.

database inquiry scheme we can see that the average response time is increased to 6.58 milliseconds. In this situation, when receiving an inquiry, the provider needed to consult a data store microservice, which we implemented as a MySQL database [36] deployed inside a docker container. The result is then returned back to the provider microservice, which can perform some processing on the data before returning the results to calling consumer. Last, we describe the situation where the consumer service is calling the provider via the network. The reported average time is 8.78 milliseconds. This time also varied considerably between the five runs which we attribute to the variability in network performance at the time of calling the provider service. This experiment contrasted the various time costs associated with inter-container communication schemes. Tolerating these costs depends on the type of the application or the use case at hand. Nonetheless, utilizing service sharing, a key design principle in our architecture, promotes scalability as IoT applications will share similar generic microservices and only create/reuse containers' links to create the DAG processing graph specific to their needs.

The inherent mobility of IoT environments means that this architecture will be subject to varying degrees of workloads as users joining and leaving the IoT environment send concurrent requests to access the edge server services. To study the impact of this factor, we used JMeter [37] to perform load testing for the three aforementioned scenarios. Results for these experiments are depicted in Figures 6, 7 and 8. We report in each figure the CPU utilization for the involved containers along with the response time. This will help us understand the behavior of the architecture when handling different loads. In this experiment JMeter was used to simulate 1, 5, 10, 20, 50 & 100 threads sending HTTP requests simultaneously to the provider microservice. This process is continued for 30 seconds while measuring the CPU utilization every three seconds. The average of the resultant 10 readings, retrieved using docker stats [38], is reported in the figures with the standard error shown on the bars. The response time here is the average time for all HTTP calls to return a result, which is measured by JMeter. Note that logging the performance for 30 seconds was sufficient as reported performance averages got saturated after that when the same number of thread/requests is used. Figure 6 shows the results of load testing the **Host Inquiry** configuration. We see from the figure that the provider container was able to handle the increase in number of requests by increasing the CPU utilization by a factor of around 20% each time. Similarly, a liner increase in the response time is also observed. For example, when the number of threads got doubled from 5 to 10, the provider container handled this increase smoothly without any increase in response time. However, the situation starts to worsen when the number of threads sending concurrent requests goes from 50 to 100 where the response time doubles from three to six milliseconds with the CPU utilization reaching 151%. Note here that CPU utilization exceeding 100% means that the container is expanded by the docker engine to run on more than one CPU core. Finally, for this experiment, we conclude that the provider container is stable in handling about 50 threads sending concurrent requests. Solutions for automatic scaling such as docker swarm mode [39] or Kubernetes [40] can be utilized when receiving requests more than the container can handle.

Now we look into the results for load testing the **Database Inquiry** configuration. To measure the performance here, we logged the CPU utilization of the data filter and the data store containers and reported the sum of the two. We observe from Figure 7 that the CPU utilization of this scenario is maxed out at 20 threads at 199% utilization. This saturation caused a dramatic increase in average response time of more than 100% when increasing the threads after that to 50 & 100 threads. We also note from the figure that the

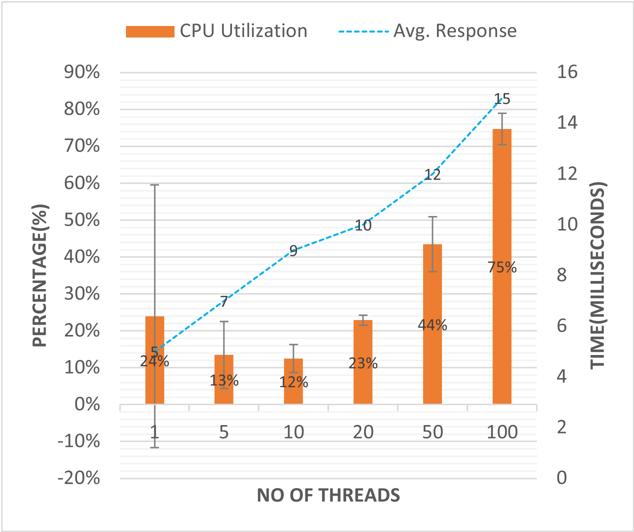


FIGURE 8. CPU utilization/response time for the network inquiry configuration under varying workloads.

data store, which consists of a container running MySQL image, constituted more that 50% of the CPU utilization of the scenario. It was also the quicker microservice to saturate and reach 100% CPU utilization at 5 threads only. This quick saturation caused the data filter component to also saturate beginning at 10 threads at 70% to 80% as we expect the data store component to start queuing received requests thereby delaying them. When comparing the performance of the data filter component to the same component in Figure 6 when the data store component was not introduced we see that the CPU utilization did not exceed 100%, which means that the component did not expand to other CPU cores. It is clear that the data store component caused the bottleneck in this scenario leaving no space for the filter component to increase its performance.

Finally, we turn into inspecting the performance of the **Network Inquiry** scenario shown in Figure 8. We see from the figure that the introduction of the network factor in this scenario throttled the rate of requests received at the service provider side. This in turn caused an increased average response time when comparing the performance with the host inquiry scenario depicted in Figure 6. It also caused the CPU utilization to increase in smaller factors. The CPU utilization here never exceeded 100% as we have seen in the host inquiry scenario due to the fact that the requests were delayed by the network.

A general note from some of the figures is that a high degree of variability is observed in CPU utilization which is reflected in the error bars. This high variability is attributed to the spontaneous logging of the CPU utilization number every three seconds. However, we have not seen this behavior to happen very frequently and hence decided to stick with our choice of three seconds logging frequency.

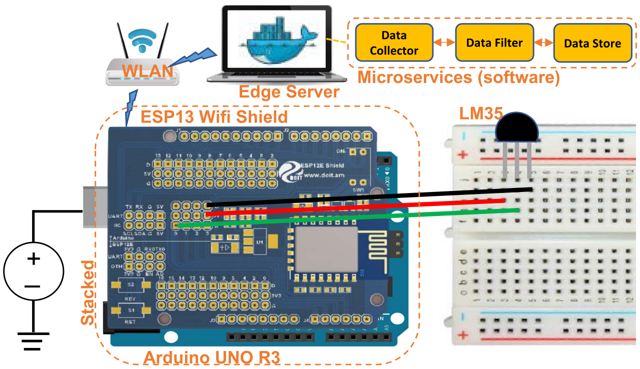


FIGURE 9. Prototype testbed.

V. PROTOTYPE APPLICATION

We have focused in the previous section on evaluating the cost of inter-container communication as it plays a major part in the performance of the architecture. Besides that, the cost stemming from the container (i.e the code running inside it) can not be neglected. This cost consists of the time needed to start any microservice before being able to call it and receive a response. We also looked into the memory footprint cost of loading and keeping a microservice in the memory. To look into these aspects we needed to choose a particular application as an example. Therefore, we implemented a prototype consisting of the temperature pipeline of the A/C Controller - Fire evacuation planner application described in the introduction and shown in Figure 1.

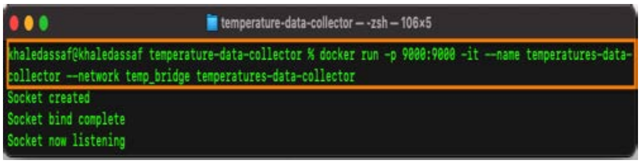
The testbed we used for the prototype implementation is shown in Figure 9. Also, Table 2 lists the specifications of used hardware. We utilized a MacBook Air laptop to act as the edge server in the setup. An Arduino UNO R3 board was used as the micro-controller. As seen in the figure, we stacked an ESP13 WiFi shield [41] on top of the Arduino to provide it with WiFi access. An LM35 temperature sensor is chosen as the sensory interface. The temperature sensor was wired to the Arduino micro-controller by means of the electronic breadboard also shown in the figure.

A. TEMPERATURE PIPELINE IMPLEMENTATION

The software part consisted of the minimal number of chained microservices to implement the temperature pipeline. The goal of this pipeline is to collect temperatures periodically and push them to a database where there is an API exposing them to other microservices who can request a temperature reading at any time. We created three microservices as shown in Figure 9 to achieve this goal. In the beginning, the **Data-Collector** microservice is responsible for interfacing with the temperature sensor and representing it in the architecture. The implementation of this container consisted of a python TCP server listening to port 9000 inside the container. The docker python image was used as the parent image when building this container in order to run the python server code at the time of starting it.

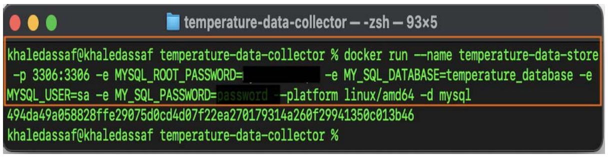
TABLE 2. Summary of testbed hardware.

Name	Description
Edge server	MacBook Air, CPU Apple M1 (8 cores), 8GB RAM, 256GB SSD
Micro-controller	Arduino UNO R3, ATmega328P, 16MHz Clock Speed
Ethernet Shield	ESP13 Wifi Shield from doit.com
Temperature Sensor	M35 Precision Centigrade from Texas Instruments



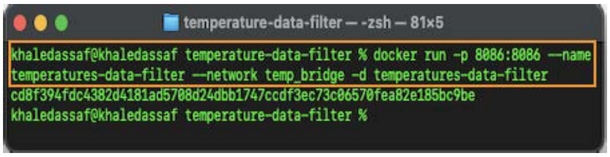
```
khaledassaf@khaledassaf temperature-data-collector % docker run -p 9000:9000 -it --name temperatures-data-
collector --network temp_bridge temperatures-data-collector
Socket created
Socket bind complete
Socket now listening
```

FIGURE 10. Command for starting the Data-Collector microservice container.



```
khaledassaf@khaledassaf temperature-data-collector % docker run --name temperature-data-store
 -p 3306:3306 -e MYSQL_ROOT_PASSWORD=[redacted] -e MY_SQL_DATABASE=temperature_database -e
MYSQL_USER=sa -e MY_SQL_PASSWORD=[redacted]-platform linux/amd64 -d mysql
494da49a058828ffe29075d0cd4d07f22ea270179314a260f29941350c013b46
khaledassaf@khaledassaf temperature-data-collector %
```

FIGURE 11. Command for starting the Data-Store microservice container.



```
khaledassaf@khaledassaf temperature-data-filter % docker run -p 8086:8086 --name
temperatures-data-filter --network temp_bridge -d temperatures-data-filter
cd8f394fdc4382d4181ad5708d24dbb1747ccdf3ec73c06570fea82e185bc9be
khaledassaf@khaledassaf temperature-data-filter %
```

FIGURE 12. Command for starting the Data-Filter microservice container.

Before starting the **Data Collector** microservice, it is required to create a Docker bridge which would act as the communication channel between this microservice and the **Data-Filter** microservice. Docker bridge [34] is a software bridge that allows containers attached to it to communicate while isolating them from other containers running on the same host. The command we used to create the bridge that we named **temp_bridge** is as follows:

docker network create temp_bridge –driver bridge

Now after building the **Data-Collector** microservice container and creating the network bridge we are finally ready to run the container while attaching it to the bridge. The command we used to run the **Data-Collector** microservice container is shown in Figure 10.

Note from the figure that the *p* flag was used to indicate forwarding traffic from port 9000 at the local host to port 9000 inside the container, which is the port that the python TCP server script is listening to. Port forwarding allows the microservice to receive communication requests from temperature sensors through WiFi as mentioned when describing the testbed in Figure 9. Also, notice that the microservice container was attached to the created network bridge at the time of starting it by setting the network to the created **temp_bridge**.

Before we describe the steps needed to run the **Docker-Filter** microservice, we need to work on launching the **Data-Store** microservice. The dependency here comes from the fact that the former relies on the latter for data persistence. We choose to implement the **Data-Store** in a separate container running MySQL database. To launch this container, we need to pull it first from Docker hub using the following command:

docker pull mysql

After that, we run the command shown in Figure 11 to launch the **Data-Store** microservice container. This command requires setting many environment variables and configuration parameters. We only describe the important ones. First, we also set the *p* flag to forward network traffic to port 3306 in the container. This port will be used inside the **Data-Filter** to configure the Object-Relational Mapping ORM, which performs data-management tasks with the use of the **Data-Store**. The command also creates a new schema named **temperature_database** and sets it as the default schema. Other parameters are concerned with creating database administration accounts inside the container.

The **Data-Filter** component provides RESTful APIs for data retrieval and processing tasks where the **Data-Store** acts as the backend. These APIs will be used by the **Data-Collector** microservice to persist temperature readings received from the temperature sensor over WiFi. The **Data-Filter** microservice is implemented as a spring boot Java web application [35]. The JAR file of this application was then containerized using Java docker container as the parent image. The resultant container was then started using the command depicted in Figure 12.

As seen in the figure, this command runs the **Data-Filter** microservice container while attaching it to the network bridge, which enables communication between the **Data-Collector** and this microservice using the container name. We can also see from the figure that this microservice uses port forwarding to be able to communicate with the WiFi network via port 8086. This is essential so as to allow any client in the domain to access temperatures sensor data through the edge server, thereby allowing the edge server to play the intermediary role planned in the design.

B. TEMPERATURE PIPELINE EVALUATION

In this section, we evaluate the performance of the implemented temperatures microservices pipeline described in

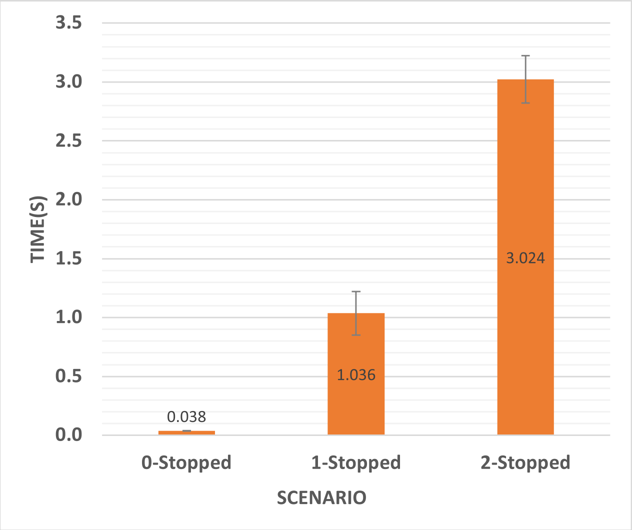


FIGURE 13. Time performance for the temperature application under different container readiness scenarios.

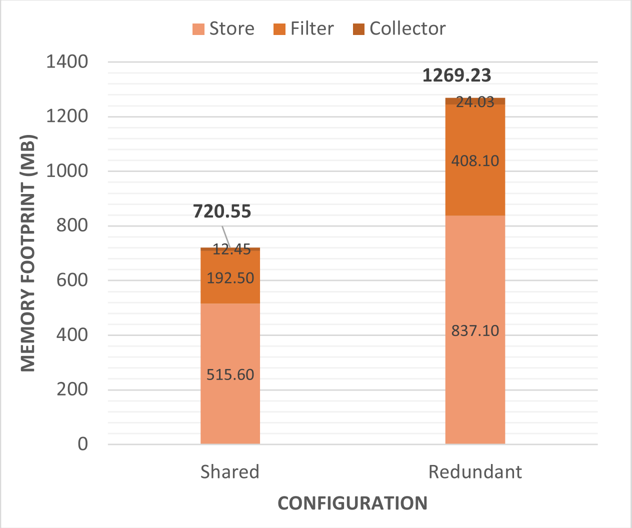


FIGURE 14. Memory footprint for shared temperature application microservices vs. redundant services.

Section V-A. The essence of this evaluation stems from the design principle that enabling discovery and sharing of microservices across IoT applications improves interoperablity as well as saves system resources. However, this comes at the cost of ensuring that the needed microservices are readily available for applications to invoke them whenever needed. We covered three scenarios to measure the time cost associated with microservices readiness as shown in Figure 13. The first scenario, **0-stopped**, covers the best case scenario where all three microservices are up when a client in the environment requests a temperature reading. In this case, the time needed to call the **Data-Filter** component APIs and receive the latest temperature is 38 milliseconds. The second scenario, **1-stopped**, resembles a situation where the **Data-Collector** is offline. In this case, we need to start the **Data-Collector** microservice before being able to call the **Data-Filter** and receive the temperature after it is posted by the **Data-Collector**. The time measured to achieve this was 1.036 seconds. Finally, in the last scenario, **2-stopped**, both the **Data-Collector** and the **Data-Filter** were stopped. We requested docker through Docker APIs to start both services. After that, we measured the time needed to be able to call the **Data-Filter** as it is the one with the start time dominating the performance. The total time for this scenario is 3.024 seconds. From this experiment, we conclude that the time needed to start a stopped microservice depends heavily on the type of implementation it provides. Hence, when designing architectures where microservices are not expected to be pre-started, system designers must choose components with acceptable launch times so as not to negatively impact the performance of the IoT application.

Lastly, we turn into evaluating the memory savings achieved from sharing components. In Figure 14 the memory footprint of a pipeline sharing scenario is juxtaposed with a redundant scenario. For the former, we only launch one copy from each microservice in the temperature pipeline whereas for the latter, two copies from each microserivce are launched. Predictably, the redundant configuration consumed almost double the memory needed by the shared scenario with each component doubling its memory usage contributing to this result. We also notice from this figure the large memory cost associated with the **Data Store** and the **Data Filter** components as opposed to the **Data Collector** component. The reason behind this big difference is that the **Data Collector** component is a lightweight python code that we wrote from scratch to receive temperatures via WiFi. On the other hand, the other two components rely on ready made software packages of MySQL database and the Java Spring Boot web framework thereby requiring the large memory footprint.

VI. CONCLUSION

This paper provides the design, implementation and performance evaluation of a microservice-based architecture of an edge server and demonstrates its utility by constructing end-to-end IoT applications. The proposed design categorizes the architectures into sets of two types of containers, systems containers and application containers. Systems containers provide support for interacting with the clients, keeping track of what containers and applications are running, starting and stopping containers and applications, and sharing containers whenever feasible. The application containers implement various functionality based on specific sensors and actuators. The paper demonstrates the architecture's utility by isolated and independent implementation of different microservices, constructing an IoT application by interconnecting these microservices, and potential sharing of microservices between different IoT applications running simultaneously. The performance evaluation shows significant savings when microservices are shared.

REFERENCES

[1] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, ''Fog computing and its role in the Internet of Things,'' in *Proc. 1st, Ed., MCC Workshop Mobile Cloud Comput. (MCC)*, 2012, pp. 13–16.

[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, ''Microservices: Yesterday, today, and tomorrow,'' in *Present and Ulterior Software Engineering*. Switzerland: Springer, 2017, pp. 195–216.

[4] P. D. Francesco, I. Malavolta, and P. Lago, ''Research on architecting microservices: Trends, focus, and potential for industrial adoption,'' in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Apr. 2017, pp. 21–30.

[5] S. Taherizadeh, V. Stankovski, and M. Grobelnik, ''A capillary computing architecture for dynamic Internet of Things: Orchestration of microservices from edge devices to fog and cloud providers,'' *Sensors*, vol. 18, no. 9, p. 2938, Sep. 2018.

[6] I.-D. Filip, F. Pop, C. Serbanescu, and C. Choi, ''Microservices scheduling model over heterogeneous cloud-edge environments as support for IoT applications,'' *IEEE Internet Things J.*, vol. 5, no. 4, pp. 2672–2681, Aug. 2018.

[7] A. S. Gaur, J. Budakoti, and C.-H. Lung, ''Design and performance evaluation of containerized microservices on edge gateway in mobile IoT,'' in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Phys. Social Comput. (CPSCom) IEEE Smart Data (SmartData)*, Jul. 2018, pp. 138–145.

[8] A. Samanta and J. Tang, ''Dyme: Dynamic microservice scheduling in edge computing enabled IoT,'' *IEEE Internet Things J.*, vol. 7, no. 7, pp. 6164–6174, Jul. 2020.

[9] Fernandez, Vidal, and Valera, ''Enabling the orchestration of IoT slices through edge and cloud microservice platforms,'' *Sensors*, vol. 19, no. 13, p. 2980, Jul. 2019.

[10] W. Jin, R. Xu, T. You, Y.-G. Hong, and D. Kim, ''Secure edge computing management based on independent microservices providers for gateway-centric IoT networks,'' *IEEE Access*, vol. 8, pp. 187975–187990, 2020.

[11] S. Pallewatta, V. Kostakos, and R. Buyya, ''Microservices-based IoT application placement within heterogeneous and resource constrained fog computing environments,'' in *Proc. 12th IEEE/ACM Int. Conf. Utility Cloud Comput.*, Dec. 2019, pp. 71–81.

[12] J. Islam, E. Harjula, T. Kumar, P. Karhula, and M. Ylianttila, ''Docker enabled virtualized nanoservices for local IoT edge networks,'' in *Proc. IEEE Conf. Standards Commun. Netw. (CSCN)*, Oct. 2019, pp. 1–7.

[13] N. E. Ioini and C. Pahl, ''Trustworthy orchestration of container based edge computing using permissioned blockchain,'' in *Proc. 5th Int. Conf. Internet Things, Syst., Manage. Secur.*, Oct. 2018, pp. 147–154.

[14] A. Javed, K. Heljanko, A. Buda, and K. Framling, ''CEFIoT: A fault-tolerant IoT architecture for edge and cloud,'' in *Proc. IEEE 4th World Forum Internet Things (WF-IoT)*, Feb. 2018, pp. 813–818.

[15] M. Noura, M. Atiquzzaman, and M. Gaedke, ''Interoperability in Internet of Things: Taxonomies and open challenges,'' *Mobile Netw. Appl.*, vol. 24, no. 3, pp. 796–809, 2019.

[16] A. Bröring, S. Schmid, C.-K. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. Le Phuoc, J. Mitic, D. Anicic, and E. Teniente, ''Enabling IoT ecosystems through platform interoperability,'' *IEEE Softw.*, vol. 34, no. 1, pp. 54–61, Jan./Feb. 2017.

[17] The Horizon 2020 Programme of the EU. *The IoT-European Platforms Initiative*. Accessed: Aug. 25, 2022. [Online]. Available: https://iot-epi.eu

[18] G. Aloi, G. Caliciuri, G. Fortino, R. Fortino, P. Pace, W. Russo, and C. Savaglio, ''Enabling IoT interoperability through opportunistic smartphone-based mobile gateways,'' *J. Netw. Comput. Appl.*, vol. 81, pp. 74–84, Mar. 2017.

[19] K. Alanezi and S. Mishra, ''Incorporating individual and group privacy preferences in the Internet of Things,'' *J. Ambient Intell. Hum. Comput.*, vol. 13, no. 4, pp. 1–16, 2021.

[20] M. Blackstock and R. Lea, ''IoT interoperability: A hub-based approach,'' in *Proc. Int. Conf. Internet Things (IoT)*, Oct. 2014, pp. 79–84.

[21] P. Desai, A. Sheth, and P. Anantharam, ''Semantic gateway as a service architecture for IoT interoperability,'' in *Proc. IEEE Int. Conf. Mobile Services*, Jun. 2015, pp. 313–319.

[22] R. Morabito, R. Petrolo, V. Loscri, and N. Mitton, ''Enabling a lightweight edge gateway-as—A-service for the Internet of Things,'' in *Proc. 7th Int. Conf. Netw. Future (NOF)*, Nov. 2016, pp. 1–5.

[23] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, ''Orchestration of microservices for IoT using Docker and edge computing,'' *IEEE Commun. Mag.*, vol. 56, no. 9, pp. 118–123, Sep. 2018.

[24] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, ''The case for VM-based cloudlets in mobile computing,'' *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct./Dec. 2009.

[25] R. Mahmud and A. N. Toosi, ''Con-Pi: A distributed container-based edge and fog computing framework,'' *IEEE Internet Things J.*, vol. 9, no. 6, pp. 4125–4138, Mar. 2021.

[26] A. Ahmed and G. Pierre, ''Docker container deployment in fog computing infrastructures,'' in *Proc. IEEE Int. Conf. Edge Comput. (EDGE)*, Jul. 2018, pp. 1–8.

[27] C. K. Dehury and P. K. Sahoo, ''Design and implementation of a novel service management framework for IoT devices in cloud,'' *J. Syst. Softw.*, vol. 119, pp. 149–161, Sep. 2016.

[28] S. Noor, B. Koehler, A. Steenson, J. Caballero, D. Ellenberger, and L. Heilman, ''IoTDoc: A Docker-container based architecture of IoT-enabled cloud system,'' in *Proc. 3rd IEEE/ACIS Int. Conf. Big Data, Cloud Comput., Data Sci. Eng.* Switzerland: Springer, 2019, pp. 51–68.

[29] U. Shaukat, E. Ahmed, Z. Anwar, and F. Xia, ''Cloudlet deployment in local wireless networks: Motivation, architectures, applications, and open challenges,'' *J. Netw. Comput. Appl.*, vol. 62, pp. 18–40, Feb. 2016.

[30] M. Flower and J. Lewis. (Mar. 25, 2014). *Microservices: A Definition of This New Architectural Term*. [Online]. Available: https://Martinfowler.com/articles/microservices.html

[31] N. Davies, N. Taft, M. Satyanarayanan, S. Clinch, and B. Amos, ''Privacy mediators: Helping IoT cross the chasm,'' in *Proc. 17th Int. Workshop Mobile Comput. Syst. Appl.*, Feb. 2016, pp. 39–44.

[32] J. Turnbull, *The Docker Book: Containerization is the New Virtualization*. 2014.

[33] M. Taneja and A. Davy, ''Resource aware placement of IoT application modules in fog-cloud computing paradigm,'' in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 1222–1228.

[34] Docker. *Use Bridge Networks, Docker Documentation*. Accessed: Feb. 19, 2022. [Online]. Available: https://docs.docker.com/network/bridge/

[35] VMware. *Spring Boot*. Accessed: Aug. 25, 2022. [Online]. Available: https://spring.io/projects/spring-boot

[36] Oracle Corporation. *MySQL Open Source Database*. Accessed: Aug. 25, 2022. [Online]. Available: https://www.mysql.com

[37] Apache Software Foundation. *Apache Jmeter*. Accessed: Aug. 25, 2022. [Online]. Available: https://jmeter.apache.org/

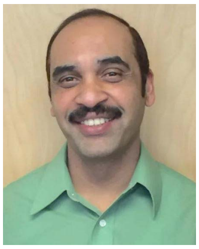
[38] Docker. *Display a Live Stream of Container(s) Resource Usage Statistics*. Accessed: Feb. 19, 2022. [Online]. Available: https://docs.docker.com/engine/reference/commandline/stats/

[39] Docker. *Swarm Mode Overview*. Accessed: Aug. 25, 2022. [Online]. Available: https://docs.docker.com/engine/swarm/

[40] The Linux Foundation. *Production-Grade Container Orchestration*. Accessed: Aug. 25, 2022. [Online]. Available: https://kubernetes.io/

[41] LTD Shenzhen Primus Technology CO. *ESP8266-Based Serial Wifi Shield for Arduino*. Accessed: Feb. 18, 2022. [Online]. Available: https://pdf.direnc.net/upload/esp8266-esp13-serial-wifi-shield-datasheet.pdf

KHALED ALANEZI received the B.Sc. degree in computer engineering from Kuwait University, in 2003, and the M.Sc. and Ph.D. degrees in computer science from the Department of Computer Science, University of Colorado, Boulder, CO, USA, in 2012 and 2016, respectively. He is currently an Assistant Professor of computer science at the College of Basic Education, Kuwait. His current research interests include edge computing, the Internet of Things, and the IoT privacy.



SHIVAKANT MISHRA is currently a Professor with the Department of Computer Science, University of Colorado, Boulder, CO, USA. His research interests include edge computing, cybersafety, mobile and pervasive computing, and large scale distributed computing.