Contemporary Mathematics
https://ojs.wiserpub.com/index.php/CM/

Research Article

Joint SDN Controller and Microservice Placement for Scalable Deployments in Computing Continuum Networks

Juan Luis Herrera[1*], Sergio Laso[2], Javier Berrocal[2], Jaime Galán-Jiménez[2]

[1] Distributed Systems Group, TU Wien, Vienna, Austria
[2] Department of Computer Systems and Telematics Engineering, University of Extremadura, Cáceres, Spain
E-mail: j.gonzalez@dsg.tuwien.ac.at

Received: 7 July 2025; **Revised:** 19 August 2025; **Accepted:** 20 August 2025

Abstract: The Internet of Things (IoT) paradigm is a crucial enabler for the computerization of real-world processes, and is becoming especially attractive in intensive domains. However, the critical IoT applications used in such environments require very high Quality of Service (QoS) and low economic deployment costs to be practical. To achieve the necessary QoS and costs, it is paramount to deploy the application in a distributed Cloud Continuum environment, by disaggregating it into a Microservices Architecture, and automating service discovery exploiting the Software-Defined Networking (SDN) paradigm. In this context, the placement and replication of both SDN controllers and microservices greatly impacts the network QoS and, by extension, the overall application QoS, as well as the deployment costs. Managing the placement and replication of microservices and SDN controllers is especially complex to do manually, especially considering that microservice placement affects the QoS and cost provided by a given SDN controller placement and vice-versa. However, while there are initial approaches to address these problems, they are highly time and resource-consuming, limiting the support to large scenarios. To tackle these issues, this work proposes the Microservice and SDN Controller Workflow Heuristic (MCWH), a multi-objective algorithmic heuristic to replicate and place SDN controllers and microservices in the Cloud Continuum efficiently and supporting large scenarios. The main novelty of MCWH is considering both microservice and SDN controller replication and placement not as separate problems, but as a single optimization effort that considers the effects across the problems, which so far only optimal methods could consider. The evaluation in a realistic healthcare use case shows that MCWH can achieve up to 70.48 $\times$ speed-up w.r.t. optimal methods, with an optimality gap as low as 8.2%, allowing it to improve the QoS and cost in large scenarios, which need automated solutions the most.

Keywords: Software-Defined Networking (SDN), microservice architecture, cloud continuum, Quality of Service (QoS), cost

MSC: 68N30, 68W99

Abbreviation

Capital Expenditures	CAPEX
Electrocardiogram	ECG
Internet of Things	IoT

Internet of Medical Things	IoMT
Microservice and SDN Controller Workflow Heuristic	MCWH
Microservices Architecture	MSA
Operational Expenditures	OPEX
Quality of Service	QoS
Software-Defined Networking	SDN

1. Introduction

The Internet of Things (IoT) paradigm is a crucial enabler in the computerization of real-world processes [1]. Through sensors and actuators, IoT devices can interact with the real world, following the instructions of IoT applications, which process the data obtained and define how to react to real-world events. This potential is especially attractive in intensive domains, such as healthcare [2] or smart cities [1], as IoT can handle critical processes from the real world leveraging computer applications and networks. Thus, these IoT applications from intensive domains become critical, as the processes they handle are critical themselves. On the application side, this criticality is expressed as strict Quality of Service (QoS) requirements, such as very low response times for application requests [1, 2]. Hence, for IoT applications to be leveraged in intensive domains, it is paramount to ensure a high enough QoS can be achieved. Furthermore, QoS is not the only factor to consider: the economic cost of deploying the application is also crucial, because deployments that achieve high QoS but are not cost-effective may not be applied by the application developers and operators.

One of the main proposals to achieve high QoS in the IoT space is the Cloud Continuum [3], a paradigm in which IoT applications are executed in a distributed manner throughout the network. The Cloud Continuum allows critical components to run very close to IoT devices, thus improving their response time due to shorter network latencies. Moreover, components that are non-critical or require extensive resources can be executed in appropriate devices, including the traditional cloud. However, exploiting the benefits of distribution that the Cloud Continuum provides requires applications to be designed for distributed execution. In this regard, the Microservices Architecture (MSA) design pattern is fit for the Cloud Continuum. MSAs allow applications to be architected as a set of independently deployable and replicable modules named microservices [4]. Each microservice is capable of performing a limited subset of the application's functionality. Complex functionalities can be performed by microservice collaboration, by pipelining multiple microservices in workflows to combine their functionality.

As microservices are distributed and replicated, IoT devices may not know a priori the location of microservices in the Continuum. Hence, service discovery is crucial [5]. To make this process transparent to IoT devices and MSA-based applications, as well as to maximize QoS, the Software-Defined Networking (SDN) paradigm can be used [6]. SDN allows network programmability by decoupling the control plane in an entity named SDN controller, while SDN switches exclusively handle the data plane [7]. By programming the SDN controller, communication functionalities can be provided transparently and QoS-efficiently at the network level, including service discovery [5]. Using the Cloud Continuum, MSAs and SDN, it is possible to achieve the QoS required by critical IoT applications [2].

However, leveraging these three paradigms can be complex due to the multitude of factors that affect QoS. As microservices are replicable, it is necessary to decide how many replicas are needed for each microservice, based on the volume of requests for each microservice, the topology of the underlying infrastructure, the distribution of such requests, or the hardware capacity of the computing devices. Moreover, it is necessary to decide in which device each replica will be deployed, i.e., the microservice placement. Microservice replication and placement are both impactful on QoS. Microservices provided far away from IoT devices may have poor QoS, and placing microservices close to multiple IoT devices that are physically distant requires replication [3, 8]. This phenomenon also holds true for the network: the number and placement affects the latency of the network, as SDN switches need to communicate with controllers to function [9]. Furthermore, the QoS obtained by microservices is governed by the routing of their traffic in the network, as well as the control latency [9]. This latency is, in turn, determined by the routing of control traffic [5]. As these factors are related, optimizing the QoS of an IoT application calls for a joint optimization for microservice replication and placement, and SDN controller number and placement [5]. The involved routing of data and control traffic must also be considered in this effort [5]. Moreover, solving this optimization problem does not require only high QoS, but also cost-effectiveness. These two are contradictory objectives, as using less capable nodes in the Continuum or cutting down on the number of SDN controllers to reduce costs will lead to worse QoS and vice versa.

Addressing this problem is complex, and there are some initial efforts in literature to tackle it. Sarkar et al. [10] proposed a clustering-based heuristic for microservice placement in fog environments. This heuristic tags each workflow with a deadline, and aims to fulfill the deadline for every workflow by placing the microservices appropriately. However, this heuristic does not consider the effects of the SDN network, nor its controllers, and does not optimize the cost of the system. Faticanti et al. took a different approach in [11], mixing the Ford-Fulkerson dynamic programming algorithm with a custom heuristic to optimize the microservice placement in terms of QoS. This approach, however, presents the same limitations: no cost consideration, and no insights on the effect of SDN and SDN controllers. On the other hand, Singh et al. provided an interesting heuristic for SDN controller placement in [12]. This heuristic, known as Varna-based, adapts different concepts from bioinspired algorithms, while adding new considerations, such as different classes or varna of solutions to separate exploration from exploitation, making it easier to navigate the trade-off. The Varna-based heuristic was then used to optimally place SDN controllers in a network. In contrast to other works, this one lacks any consideration of microservice placement, i.e., the purpose for which the network will be used. Finally, the Next-gen IoT Optimization (NIoTO) framework [13] is the closest solution to the current proposal. NIoTO tries to optimize both QoS and cost in the Computing Continuum, considering both MSAs and SDN. NIoTO optimally places SDN controllers and microservices, successfully considering the influences across them. However, NIoTo is based on exact mathematical optimization methods. These methods scale poorly in terms of both resource usage and time, limiting the size of the problems they can solve with reasonable resources and taking very long times for the largest problems [13].

Therefore, there is a need for a method that considers both QoS and cost, as well as the interplay between SDN and MSAs, all while scaling in large scenarios. To address these issues, this work proposes the Microservice and SDN Controller Workflow Heuristic (MCWH). MCWH is a heuristic (i.e., non-exact) algorithm to replicate and place microservices and SDN controllers in the Cloud Continuum, routing their traffic and minimizing the response time and deployment cost of the applications quickly and resource-efficiently. The main contributions of this work are as follows: i) a description of the problems encountered in intensive IoT when MSAs, the Cloud Continuum, and SDN are leveraged, and how QoS and cost are affected by them; ii) the proposal of MCWH, an algorithm that merges various techniques such as machine learning techniques, greedy algorithms, or dynamic programming, to address the microservice and SDN controller placement problem, including the routing of information. MCWH is designed to hold the key constraints in the Cloud Continuum and SDN networks, such as computing device and link capacity, providing similar solutions to the optimal approaches in significantly shorter times and supporting larger scenarios; iii) an assessment of MCWH in terms of large scenario support, optimization times, costs, and response times achieved through an evaluation in realistic IoT scenarios; and iv) a comparison of MCWH with alternative solutions in terms of optimality gaps for both costs and response times, as well as scalability in terms of time and large scenario support. MCWH is novel compared to other heuristic solutions in that it integrates SDN and MSAs, as well as cost and response time considerations. Moreover, MCWH addresses the research gap left by the NIoTO framework, which has these considerations, by supporting large scenarios due to its scalability as a heuristic approach.

The remainder of this paper is structured as follows. Section 2 presents the system architecture in which MCWH is based with an illustrating use case, while the design of MCWH is detailed in Section 3. The evaluation performed and its results are described in Section 4. Finally, Section 5 concludes the paper and presents future research lines.

2. System architecture and use case

To make it easier to understand the problem tackled by MCWH, this section presents an Internet of Medical Things (IoMT) use case that illustrates the system model that is the basis for MCWH. The use case is derived from the work of Gia et al. [14]. Thus, it focuses on an MSA-centric IoMT application designed for monitoring, evaluating, and storing patients' vital sign data, specifically blood pressure and Electrocardiograms (ECG). The case study, including the infrastructure, MSA application, and the execution of an example workflow, is depicted in Figure 1.

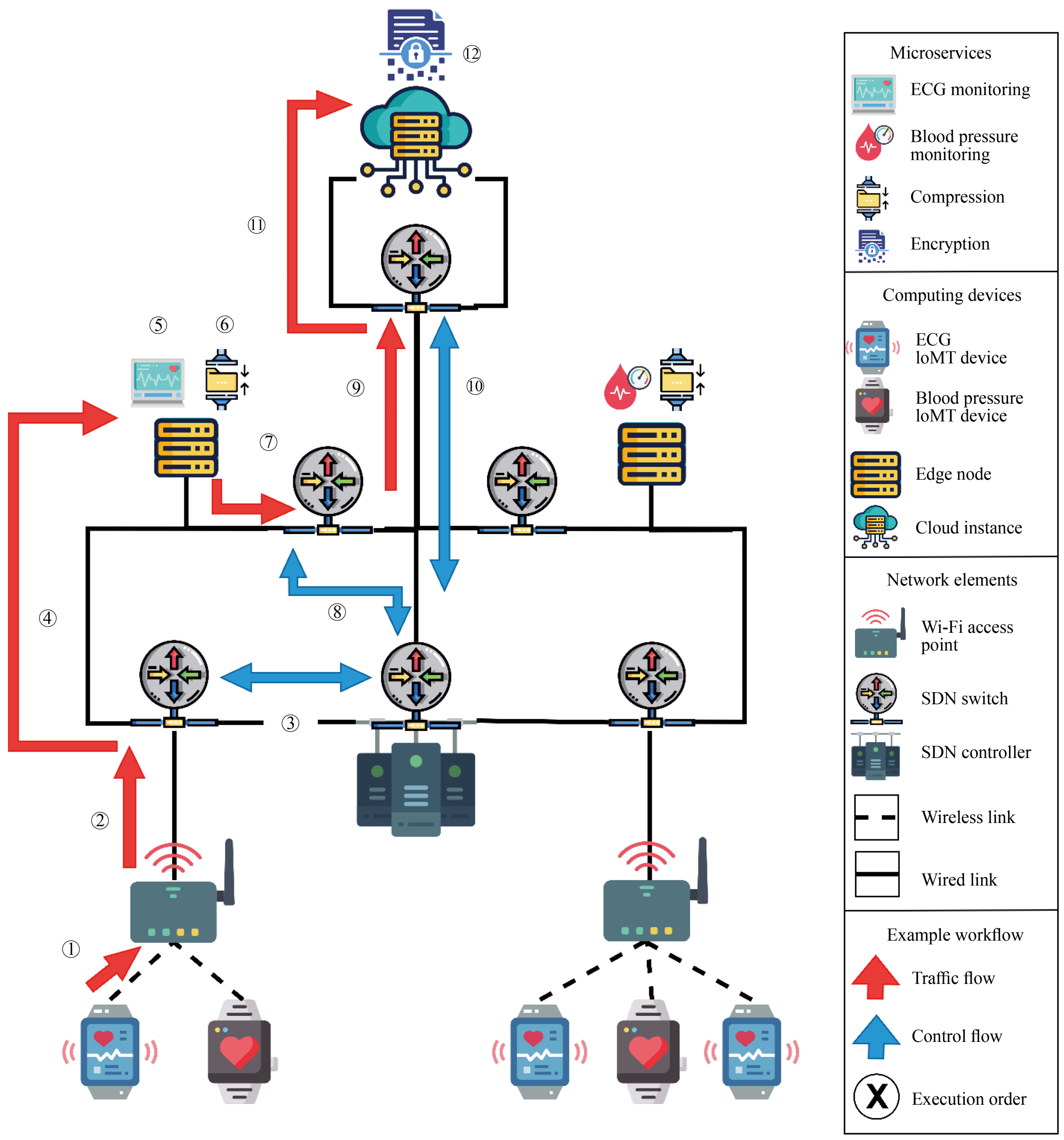

Figure 1. System model with an example workflow

The MSA supporting the IoMT application is composed of four distinct microservices (see legend of Figure 1). The first microservice is the ECG monitor, responsible for gathering raw data from the patient's ECG and transforming it into a comprehensive ECG report or "commented ECG" [14]. A commented ECG comprises the ECG raw data as a whole, along with interpretations of different relevant waveforms, points, and complexes [14]. On the other hand, a second microservice, the blood pressure monitoring microservice, serves the purpose of collecting patient blood pressure readings over a set time interval and calculating key statistics [14]. The third microservice compress the ECG reports. Given the relatively small size of the blood pressure data, it can be encrypted and stored directly, but commented ECGs are too voluminous [14, 15]. Therefore, the MSA integrates a fourth microservice to performs encryption, given ECG and blood pressure information is particularly confidential. This architectural design defines two separate data workflows: one for ECG data, which includes analysis, compression, and encryption before being stored; and another for blood pressure data, which undergoes analysis and encryption only.

The infrastructure would be deployed in a hospital, where the application would be leveraged. This infrastructure also follows the specifications of this IoMT use case [14]. Starting with the computing devices and in a bottom-up order from Figure 1, we first find the IoMT devices. There are two kinds of IoMT devices, those that track the patient's ECG, and those that can measure the blood pressure instead. These would be in the hands of patients and continuously measure their vital signs. Each of these are considered as computing devices and can run microservices that their capacity allows. Moreover, they can also request for the corresponding workflows (e.g., ECG IoT devices can request for the workflow that monitors, compresses, and encrypts the ECG information). Further up in Figure 1, we find edge nodes. These are close to the IoMT devices, but are significantly more capable than them. In Figure 1, the leftmost edge node has a replica of the ECG monitoring and compression microservices deployed to it, while the rightmost edge node hosts replicas of the blood pressure monitoring and compression microservices. Finally, on the top of Figure 1, we find the cloud, which hosts the compression microservice. The cloud is the most powerful device in the infrastructure, but is also the one with the highest latency from the IoMT devices.

In terms of network elements, the IoMT devices connect to the infrastructure through wireless links, while the rest of the network relies on wired connections. These wired links connect the SDN switches to the Wi-Fi access points and to the edge nodes. The uplink to the Internet-and consequently to the cloud-is also established via wired links. At the central bottom part of Fig.Figure 1, the SDN controller is co-located with an SDN switch, providing the control plane for the entire network. Following the SDN controller placement model, each controller is physically located with an SDN switch, and every switch is managed by a single controller [9]. Furthermore, when an SDN switch receives a data packet, it initiates a control communication with the controller using the standard PACKET-IN and PACKET-OUT FLOW-INSTALL mechanism [9]. We assume a worst-case scenario where no forwarding rules are pre-configured on the switches, meaning that every new flow triggers interaction with the controller.

Finally, Figure 1 also represents an example workflow, and more specifically, an ECG workflow. Following the execution ordered as numbered, the ECG IoT device requests for the monitoring, compression, and encryption of its ECG data. To do so, the raw ECG data is sent to the Wi-Fi access point ① and, from there, to the SDN network fabric ②. As the traffic flow reaches this SDN switch, it sends a PACKET-IN request to the SDN controller with a control flow. This request is responded with a PACKET-OUT from the controller ③. Following the action specified by the controller, traffic is sent to the closest edge node ④. In this node, the ECG is processed by the ECG monitoring microservice ⑤. The resulting commented ECG is compressed ⑥. The compressed ECG is then sent to the cloud through the network ⑦, ⑨, ⑪. The two SDN switches in the path request a control flow as previously seen ⑧, ⑩. Finally, the compressed ECG is encrypted and stored ⑫. The workflows requested by other ECG IoT devices, as well as those from blood pressure IoT devices, are performed analogously.

In terms of optimization, two metrics must be considered in this scenario: the response time of workflows and the economic cost of the deployment. Response time is calculated as the sum of the latency and execution time across the whole workflow. Latency is the sum of the latencies of all the links through which the traffic flow goes through. Latency also includes the latencies of the links through which all the related control flows go through. Execution time is calculated as the number of cycles that the microservice takes to execute, divided by the CPU clock speed of the computing device it executes in. The result is then summed over all the microservices in the workflow. In the same example, execution time would be the sum of the cycles of the ECG monitoring microservice divided by the CPU clock speed of the edge node ⑤, the cycles of the compressing microservice also divided by the CPU clock speed of the edge node ⑥, and the cycles of the encryption service divided by the CPU clock speed of the cloud instance ⑫.

On the other hand, the system also involves deployment costs. These are divided into two main categories: Capital Expenditure (CAPEX) and Operational Expenditure (OPEX). CAPEX refers to the one-time cost of acquiring hardware components, such as SDN switches, wired links, SDN controllers, and computing devices. OPEX, in contrast, represents the recurring cost of keeping the infrastructure operational. It applies not only to the elements with CAPEX but also to wireless links and cloud instances. According to the cost model presented in [13], the OPEX for network elements like SDN switches, links, and controllers is calculated per unit of time and is typically based on their energy consumption. For computing devices, OPEX is calculated per execution cycle [13], and it depends on the type of hosting. Devices that are owned and operated on-premises-such as IoMT devices-have their OPEX determined by their energy usage. Devices provided as-a-service, like cloud instances, incur OPEX based on the pricing schemes offered by the service providers. Edge nodes may fall into either category (on-premises or as-a-service), so their OPEX depends on how they are deployed.

The role of MCWH is to decide the replication and placement of both SDN controllers and microservices, as well as to route both traffic and control flows through the network. This optimization has the objective of minimizing the average response time across all workflows, as well as the total economic cost of the deployment. MCWH improves response time by balancing computing power and closeness to IoMT devices to minimize the response time. At the same time, MCWH considers the maximum capacity of computing nodes in its placement. For deployment cost, MCWH tries to use the minimum amount of devices to save on CAPEX, as well as optimizing their load and selection to minimize OPEX. Moreover, these objectives are tackled together, trading them off to obtain a multi-objective solution with minimal cost and response time.

3. Heuristic description

The problem of microservice and SDN controller replication and placement is NP-hard [2, 5, 13]. To allow developers and operators to address it efficiently, this section details the design of MCWH as a heuristic algorithm, i.e., an algorithm to find inexact solutions efficiently. The heuristic proposed is based on the formal mathematical model introduced in [13]. To ensure this work is self-contained, a summarized version of the model can be found in Appendix A. Nonetheless, we point the interested readers to [13] for a more in-depth description of the mathematical model. MCWH is designed to reduce the time required to obtain a solution, as well as the resources needed, thus supporting larger scenarios. This acceleration is achieved by obtaining solutions that are not optimal, but still provide low response times and economic costs. MCWH is designed to explore the optimality and time-and-resource-efficiency trade-off, obtaining solutions close to optimal while maintaining a polynomic time complexity.

3.1 *Algorithmic description*

First, it is necessary to detail the notation. The infrastructure, which as per the system model includes all computing devices, SDN switches, and the links between them, both wired and wireless, is represented as a graph $G = \{V, L\}$. The graph contains V vertices and L edges connecting them. Each edge $l \in L$ represents a link, either wired or wireless, in the infrastructure. V, on the other hand, is divided into two main subsets: the SDN switches S and the computing devices or hosts H. These subsets do not overlap, i.e., $S \cup H = V$ and $S \cap H = \emptyset$. For MCWH, Wi-Fi access points are modelled as wireless links from another SDN switch or host to the SDN switch the access point is attached to. Moreover, SDN switches with computational capabilities are modelled as an SDN switch connected to a host with such computational capabilities by a link with no latency and infinite capacity. The list of microservice types (e.g., the list of microservices in the legend of Figure 1) is modelled as the set M. Each of these microservices can be replicated if required, and some microservices may not be deployed if they are not requested. Workflow requests are contained in the set W instead, where each workflow $w \in W$ has an ordered set of the microservices it requests as $w.Microservices$ (e.g., for an ECG workflow request w_α, $w_\alpha.Microservices$ contains, in this order, the ECG monitoring microservice, the compression microservice, and the encryption microservice).

Algorithm 1 Pseudocode for MCWH

```
1. function MCWH ($G, W, M$)
```

```
2.     $assignment := \emptyset$; $paths := \emptyset$
3.     $numControllers := KMedoidsSilhouetteMethod(S,\ metric = G.ShortestPathLatency)$
4.     $costs := H.CAPEX + H.OPEX_{cycle} \cdot \min(M.Cycles) \cup H.CAPEX + H.OPEX_{cycle} \cdot \max(M.Cycles) \cup S.CAPEX + S.OPEX \cup L.CAPEX + L.OPEX$
5.     $times := L.Latency \cup \frac{\max(M.Cycles)}{\min(H.CPU)} \cup \frac{\min(M.Cycles)}{\max(H.CPU)}$
6.     $costRange := (\min(costs),\ \max(costs))$
7.     $timeRange := (\min(times),\ \max(times))$
8.     for all $l \in L$ do
9.         $normCost := Normalize(l.CAPEX + l.OPEX,\ costRange,\ (0,\ 50))$
10.        $normTime := Normalize(l.Latency,\ timeRange,\ (0,\ 50))$
11.        $l.CombinedWeight := normCost + normTime$
12.    end for
13.    $firstController := Sort(S, descending, metric = G.BetweennessCentrality)[0]$
14.    $firstController.Controller := firstController$
15.    $firstController.ControlPath = \emptyset$
16.    for all $s \in S$ do
17.        $s.Controller := firstController$
18.        $s.ControlPath := G.ShortestPath(s,\ firstController, metric = CombinedWeight)$
19.    end for
20.    $C := \{firstController\}$
21.    $Relevant := \{firstController\}$
22.    for all $w \in W$ do
23.        $initialNode := w.Requestor$
24.        for all $m \in w.Microservices$ do
25.            $host, path :=$ PlaceMicroservice $G, H, initialNode, m, costRange, timeRange$
26.            $assignment[w][m] := host$
27.            $paths[w][m] := path$
28.            $initialNode := host$
29.            $host.Usage += m.Memory$
30.            $path.l.Usage += m.Data$
31.            $Relevant := Relevant \cup \{s \in path\}$
32.            if $|C| < numControllers$ then
33.                $controller :=$ PlaceController $G.Subgraph(Relevant), Relevant, C$
34.                $C := C \cup controller$
35.                for all $s \in Relevant$ do
36.                    $Relevant := Relevant \cup s.ControlPath$
37.                end for
38.            else
39.                $C :=$ MapControllers$(G.Subgraph(relevant), C)$
40.            end if
41.        end for
42.    end for
43.    $C :=$ MapControllers$(G.Subgraph(relevant), C)$
44.    return $C$, $assignment$, $paths$
45. end function
```

Algorithm 1 shows the pseudocode of MCWH as a whole. It receives as input the infrastructure graph G as a whole (which includes the set of SDN switches S, hosts H, and links L), as well as the set of workflows W and microservice types M (line 1). MCHW starts by preparing sets to save the microservice assignments and traffic paths (line 2), which is necessary as part of the final solution. It then calculates the number of controllers to be placed (line 3), another key part of the solution. To do so, it is first important to note that MCWH is partially based on the k-Medoids unsupervised machine learning algorithm [16]. This algorithm takes three parameters: a natural number k for the number of centroids, the initial centroids of the data, and a dissimilarity metric. k-Medoids then uses this metric to agglutinate the data points into k clusters. k-Medoids tries to ensure each data point in a cluster is as similar as possible to the rest of the points in the cluster, and as dissimilar as possible to points in the rest of the clusters, according to the dissimilarity metric specified. Moreover, the centroid of each cluster approximates the center of such cluster, i.e., the point of data with minimum distance to the rest, which k-Medoids guarantees to be a data point. k-Medoids is used to divide the network into clusters of nodes with minimal latencies to one another, and uses the centroid, which has minimal latencies to the rest of the nodes, to place the controller. Intuitively, if latency is visualized as distance, this method can be interpreted as a clustering of close nodes together, and placing the controller in the center. In heuristic terms, k-Medoids approximates the optimal k nodes that are each most central to their own cluster of nodes [16], which is an approximation to the optimal SDN controller placement. MCWH uses k-Medoids in line 3 to create clusters on the set S, i.e., the SDN switches, using as a dissimilarity metric the shortest-path latency between the switches. However, to use k-Medoids requires determining k, i.e., the number of clusters and centroids to calculate and, by extension, the number of SDN controllers to place, as calculated in line 3. MCWH automatically chooses a value for k using the Silhouette method [17], which evaluates the quality of clustering for any given k. Thus, MCWH performs a sweep on values of $k \in [1, |S|]$, determining the optimal number of clusters to be made of the network. This sweep, summarized in the KMedoidsSilhouetteMethod function, returns the optimal k in this analysis. MCWH interprets this value as the number of controllers to set, following the previous logic. It is noteworthy that MCWH does not consider the clusters nor centroids calculated during this sweep, as they are re-calculated and modified during the algorithm.

This initial part considers latency, which is a key part of QoS, but cost must be traded off with it. This trade-off is complex, as not only the units, but also the scales are different across objectives. The next step is thus to combine both response time and cost into a single objective metric, which requires them to be normalized. MCWH leverages a normalization between 0 and 100 for the joint objective, and hence, each response time and cost are normalized to a range between 0 and 50. This normalization is designed to balance response time and cost, following the original formulation of the problem in [13]. The normalization is performed by linear interpolation. This method was chosen as other normalization techniques, such as mean and standard deviation, would require extensive sampling to ensure the objective would not surpass the stated range, and the sampling would slow MCWH down significantly, which would be especially problematic for a heuristic. Linear interpolation, however, requires the minimum and maximum values for response time and cost, i.e., the original ranges. MCWH calculates the minimum and maximum cost from a set built ad-hoc that includes CAPEX and OPEX for all network equipment, while for hosts, their CAPEX is added to their OPEX per cycle multiplied by the minimum and maximum number of cycles of the microservices in the application (lines 4, 6). For response time, this ad-hoc set includes the latencies and the minimum and maximum execution times (lines 5, 7), calculated as the execution times of the shortest microservice in the fastest host and the longest microservice in the slowest host, respectively. These ranges are used to calculate a new metric foreach link (lines 9-11), named the Combined Weight, which is the sum of the normalized latency of the link and its cost.

MCWH then sets an initial controller in the SDN switch with the highest betweenness centrality in the graph (line 12). Betweenness centrality is used for initialization because of its consistency, compared to a random selection, and its representation of centrality [5], which is used to place the final controller This controller is simply an initialization, and hence, it controls not only itself (lines 13, 14), but all the rest of the switches in the infrastructure (lines 15-17). The path from each switch to the controller is also calculated using the shortest path algorithm, but the weight metric is the combined weight (line 17). This new controller is added to C, the set of all controllers in the network (line 18). Then, to ensure that the interplay between SDN and MSAs is being considered, MCWH introduces the novel concept of relevant nodes. One of the key tenets of MCWH is the combination of microservice and SDN controller placement into a single system, given the relationship between both problems. Hence, MCWH does not simply perform SDN controller placement and microservice placement separately. Instead, it considers the interplay of the decisions made in an aspect to decide on the other and vice versa. To apply this design principle, MCWH considers that not all switches in the infrastructure will be used: only those that are either used to route traffic or control flows (i.e., used by microservice placement), or host an SDN controller, are truly relevant. Relevant node marking allows MCWH to, on the one hand, ensure controllers are placed based on the QoS of traffic and control flows, and on the other hand, reduce the deployment cost, as unused switches and links do not need to be acquired. Hence, the final instruction before starting microservice and controller placement is to include the first controller as a relevant node (line 19).

Next, MCWH enters its main loop for microservice and controller placement (lines 20-36). This process considers each workflow, taking as the initial node of the traffic path its requestor (line 21). Next, for each microservice requested (lines 22-36), MCWH takes four essential steps. First, it uses the supporting function PlaceMicroservice to determine the placement of the microservice and the path of its traffic flow (line 23). These results are obtained as the assignment of a host and a traffic path for each microservice in each workflow (lines 24, 25). In this structure, the initial node for the next traffic flow is the assigned host (line 26). To maintain the node and link capacity considerations, the memory taken by the microservice is considered in use, and thus reserved, at the selected host (line 27). The data sent by the microservice flow is also marked as in use in all its path to avoid link overload (line 28). All the switches in the path of the traffic flow are then marked as relevant to ensure they are considered during controller placement (line 29). MCWH then considers whether it is necessary to place another controller (line 30) according to the number of controllers initially calculated. If so required, the new controller is placed using PlaceController (lines 31, 32) and all the switches in the control path, including the new controller, are marked as relevant if they were not in the set already (lines 33, 34). It is noteworthy that PlaceController does not receive the whole graph G, but only the subgraph that contains the relevant nodes and the links between them. On the other hand, if no more controllers have to be placed (line 35), MCWH moves the controllers and performs a new switch-controller mapping through the MapControllers supporting function. Finally, when all microservices have been placed, MapControllers is called (line 37) to ensure that controllers are placed and mapped according to the final microservice placement. MCWH then returns the placement of the SDN controllers, the workflow-microservice-host assignments, and their traffic paths. The replication of controllers and microservices is implicitly provided by the first two results, simply calculating the instances of each. The switch-controller mapping and control routes are instead considered to be embedded into the S set in the Controller and ControlPath attributes of each switch. Hence, MWCH fulfils its role at deciding SDN and microservice replication and placement, as well as traffic and control routing.

Algorithm 2 Pseudocode for controller placement

```
1. **function** PlaceController ($G$, $C$)
2.     $controller := Sort(S-C,\ descending,\ metric = G.BetweennnessCentrality)[0]$
3.     $controller.Controller := controller$
4.     $controller.ControlPath := \emptyset$
5.     **for all** $s \in (S-C)$ **do**
6.         $s.Controller := controller$
7.         $s.ControlPath := G.ShortestPath(s,\ controller,\ metric = CombinedWeight)$
8.     **end for**
9.     **return** $controllller$
10. **end function**
```

To perform these tasks, MCWH leverages multiple supporting functions. The first of these functions is PlaceController, described in Algorithm 2. This function takes as input (line 1) the infrastructure graph. In practice, it is instead fed a subgraph with only the switches marked as relevant, along with the sets of switches S and controllers C. The method places a controller in the node with the highest betweenness centrality that is not already co-located with a controller (line 2). Then, the controller is set to control the SDN switch it is co-located with (lines 3, 4). It also is set to control the rest of the SDN switches that are not co-located with a controller (lines 5-7). The new controller is returned (line 8) to the calling function. As this function is called using the relevant subgraph instead of the whole infrastructure, controllers are only placed where they are relevant to the traffic flows generated by microservices and the control flows generated by other controllers. However, this function only handles controller placement and switch-controller mapping as an initial method, and does not determine the final switch-controller mapping or placement.

Algorithm 3 Pseudocode for controller mapping and re-placement

```
1. function MapControllers $G, C$
2.     $clusters := KMedoids(G,\ initialCentroids = C,\ metric = CombinedWeight)$
3.     $C := clusters.centroids$
4.     for all $cluster \in clusters$ do
5.         for all $s \in cluster.nodes$ do
6.             if $s = cluster.centroid$ then
7.                 $s.Controller = s$
8.                 $s.ControlPath = \emptyset$
9.             else
10.                $s.Controller := cluster.centroid$
11.                $s.ControlPath := G.ShortestPath(s,\ cluster.centroid,\ metric = CombinedWeight)$
12.            end if
13.        end for
14.    end for
15.    return $C$
16. end function
```

Instead, the final controller placement switch-controller mapping is decided by the MapControllers function, shown in Algorithm 3. This function only needs the infrastructure graph, G, although it is also called with the relevant subgraph, and C (line 1). The function uses k-Medoids to generate clusters over the graph, taking as initial centroids the set C, and using the combined weight as the dissimilarity metric (line 2). This serves to move the controllers around to potentially better locations (i.e., the centroids). It also automatically chooses the switch-controller mappings by assigning each switch to a cluster, i.e., to be controlled by a controller. Moreover, selecting the controllers as initial centroids optimizes this selection, as, if the controllers are already in their optimal positions, or near their optimal positions, k-Medoids converges immediately. The centroids of the clusters then become the new controllers (line 3), and they are set as the controllers of all the switches in their cluster (lines 4-11). The control paths are calculated as the shortest path with the combined weight as a metric. This function returns the set of potentially new controller placements C, as their assignments and control paths are changed in place in the set S.

Algorithm 4 Pseudocode for microservice placement

```
1. function PlaceMicroservice $G$, $initialNode$, $m$, $costRange$, $timeRange$
2.     $candidates := \{H | h.memory - h.usage > m.memory\}$
3.     $bestFitness := -\infty$
4.     for all $l \in L$ do
5.         if $l.Usage + m.Data > l.Capacity$ then
6.             $G.NotEligibleForRouting(l)$
7.         end if
8.     end for
9.     for all $candidate \in candidates$ do
10.        $trafficPath := G.ShortestPath(initialNode,\ candidate,\ metric = CombinedWeight)$
11.        $controlWeight := 0$
12.        for all $s \in trafficPath$ do
13.            $controlWeight += Sum(s.ControlPath,\ metric = CombinedWeight)$
14.        end for
15.        $normTime = Normalize\left(\frac{m.Cycles}{candidate.Power},\ timeRange,\ (0,\ 50)\right)$
16.        $normCost = Normalize(candidate.CAPEX + candidate.OPEX_{cycle} \cdot m.cycles,\ costRange,\ (0,\ 50))$
17.        $fitness = -1(normTime + normCost + controlWeight + Sum(trafficPath,\ metric = CombinedWeight))$
18.        if $fitness > bestFitness$ then
19.            $bestFitness = fitness$
```

```
20.                 $bestCandidate = candidate$
21.                 $bestPath = trafficPath$
22.             end if
23.         end for
24.         return $bestCandidate, bestPath$
25. end function
```

Finally, Algorithm 4 handles the microservice placement in MCWH. Replication is handled implicitly: if the same microservice type is required by multiple workflows and is placed in the same host, a single replica that services all workflows will be placed. Hence, MCWH only needs to choose where to execute each microservice type requested by each workflow. To do so, the PlaceMicroservice function first makes a list of the hosts that are candidates to place the service on (line 2). This list considers any host with enough free memory (i.e., memory that is not being used by other placed microservices) as a potential candidate to place the microservice in. Moreover, any links that would become overloaded if the microservice's data had to traverse them are considered not eligible for routing (lines 4-6). It is noteworthy that this mark of non-eligibility is only valid during the execution of a call of this function, and future calls will not be affected by it. Then, the fitness of each candidate to host the microservice is assessed (lines 7-18). The fitness of a candidate considers the combined weight of the route from the initial node provided to the function, which is the IoT node for the first microservice in each workflow, and the host where the last microservice was deployed to in the rest, to the candidate (line 8). Moreover, for each switch on the path, the combined weight of its control path is also considered (lines 10, 11). MCWH also considers the normalized execution time of the microservice in the host (line 12), and the normalized cost of executing the microservice (line 13). The combined weights and costs are multiplied by -1 to obtain the final fitness (line 14). During this loop, the objective of the function is to find the candidate with the highest fitness, i.e., with minimal response time and cost (lines 15-18). The chosen candidate and the path for traffic calculated in line 8 are then returned to the full MCWH heuristic (line 19).

3.2 *Theoretical performance analysis*

Finally, we analyze the algorithms in terms of complexity in reverse order in which the functions are presented, as the complete MCWH heuristic calls the rest of the algorithms. First, the microservice placement function (Algorithm 4) has a worst-case temporal complexity of $O(|H||L|\log(|S|))$. The most complex part of this algorithm is line 8, which leverages Dijkstra's short path algorithm, which has a $O((|S|+|L|)\log(|S|))$ complexity. In these infrastructures, the number of links is expected to be significantly higher than the number of switches, and hence, it can be approximated worst-case complexity of $O(|L|\log(|S|)$. As line 8 is executed for every candidate, in the worst case, every host would be considered a candidate, thus, the overall complexity results in $O(|H||L|\log(|S|))$. Next, Algorithm 3 has two highly complex points: line 2, as k-Medoids has a worst-time complexity of $O(|S|^2)$, given that it clusters switches, and line 11, with the same shortest path algorithm as before. Moreover, line 11 is executed $|S|$ times, As no switch belongs to two clusters, and hence the doubly-nested loop totals to $|S|$ iterations, this line results in the most significant complexity for the MapControllers function: $O(|S||L|\log(|S|))$ worst-case complexity. The same logic can be applied to Algorithm 2: the most significant complexity is the shortest path algorithm at line 7, executed, in the worst case, $|S|$ times, for a worst-case complexity of $O(|S||L|\log(|S|))$. The worst-case complexity of MCWH (Algorithm 1) depends on the specific characteristics of the scenario, as there are three potential lines with the highest complexity: line 29, line 38, and line 43. Lines 38 and 43 simply call the previous PlaceController and MapController functions, both having the same worst-case complexity of $O(|S||L|\log(|S|))$. As they are called for every microservice in a workflow, this results in a worst-case complexity of $O(|W||M||S||L|\log(|S|))$, and as both have the same complexity, whether the condition of line 37 holds or not is irrelevant. On the other hand, line 29 results in a total worst-case complexity of $O(|W||M||H||L|\log(|S|))$. Depending on whether $|H|$ or $|L|$ is higher, one or the other will be more relevant. While this complexity may seem large, the problem it solves is NP-hard [5], and it can thus be considered acceptable when compared to the factorial (i.e., $O(n!)$) complexity of optimal methods.

4. Evaluation

This section presents an assessment of MCWH's performance in terms of the response times and costs achieved with it, as well as the time required to obtain a solution and its support for large scenarios. Section 4.1 presents the setup and scenarios used in the evaluation, and the results are subsequently analyzed in Section 4.2.

4.1 *Evaluation setup*

To evaluate the performance of MCWH, the IoMT use case described by Gia et al. [14], which was detailed in Section 2, was taken as basis. This work, however, details only one scenario, labeled scenario 1, while scalability analyses, crucial for MCWH, require scenarios of increasing sizes. Hence, scenario 1 was scaled into 3 larger scenarios, labeled scenario 2 through scenario 4. The details for these scenarios are available in Table 1. In all the analyzed scenarios, the objective is to deploy the IoMT application described in Section 2, with the ECG and blood pressure workflows, and the 4 described microservices. Each scenario was scaled realistically to vary the number of available hosts, the number of hosts per type, the requests for each workflow, and the size and topology of the SDN network. Other data specified in Section 2 and extracted from [14], such as the wireless and wired technologies leveraged, is also aligned with the evaluation. The network topologies used in the scenarios were generated with the Erdös-Rényi model, which ensures realistic network generation [18]. The technical information of IoMT devices has been extracted from Arduino UNO hardware to ensure realism, while the edge devices were based on those provided by EDGE.NETWORK, and the cloud, on a Google Cloud Platform E2-Standard-4 instance [13]. Further technical details on performance and costs of these devices were retrieved from the data provided in [13], while the technical details (e.g., memory, cycles) of the microservices were extracted from [15]. We point the interested readers to [13, 15] for further numerical details.

Table 1. Scenario details

Scenario	ECG IoMT devices	BP IoMT devices	Edge hosts	SDN switches	Links
1	3	2	1	7	13
2	8	7	3	20	39
3	20	20	10	50	104
4	25	25	20	150	235

The experiments leverage an implementation of MCWH in Python, using the NetworkX and NumPy libraries. MCWH is compared against the NIoTO framework [13], also proposed by the authors of the present work. To the best of the authors' knowledge, only NIoTO implements both microservice and SDN controller management to ensure the fairness of the comparison. NIoTO implements the same system model as an optimal solution based on Mixed-Integer Linear Programming. The implementation of NIoTO is also made in Python, using the Gurobi solver. Both implementations have been tested for all scenarios, although NIoTO is unable to process the larger ones. Moreover, both implementations have been executed in the same circumstances, in a virtual machine with 16 GB of RAM and 8 virtual cores of an AMD EPYC 7413 CPU. Response time and cost have been calculated using NIoTO's reporting tools [13], and hence, are comparable across both. Moreover, the data for the costs, including CAPEX and OPEX of each device and their usage, was obtained from [5].

This evaluation has two objectives: to validate MCWH as a multi-objective solution by comparing its response times and costs with those achieved by an optimal method [13], and to assess its scalability in large scenarios. The results of the performed analyses are described in the following subsection.

4.2 *Evaluation results*

The first analysis, depicted in Figure 2, assesses the average response time of workflows when the microservices and SDN controllers are replicated and placed using MCWH and NIoTO. It is noteworthy that the number of SDN controllers placed in each scenario was determined by both MCWH and NIoTO as part of their solution. The analysis is performed in all four scenarios tested. It is noteworthy that, due to the nature of the optimal method leveraged by NIoTO, the hardware in which the solvers were tested was unable to find a solution with NIoTO for the two larger scenarios, i.e., scenarios 3 and 4. In contrast, MCWH is able to find a solution in all scenarios, as it requires less resources to find a solution. Analyzing the response time results themselves, in scenario 1, the workflows have an average response time of 3.662 ms when the microservices and controllers are deployed using NIoTO, of which 3.659 ms are due to the execution times of microservices, and latency accounts only for 0.003 ms due to the extensive use of the Computing Continuum, keeping latencies low. If MCWH is used instead, the average response time raises by 1.02 ms, for a total of 4.682 ms. This increase is due to response time, as latency is lowered to 0.002 ms, as MCWH leverages nodes closer to the requesting IoMT devices that are less powerful, and thus, slower in executing the microservices. Overall, the increase in scenario 1 represents an optimality gap of 21.78% for MCWH. Nonetheless, this trend is reversed in the next scenario, in which NIoTO achieves an average response time of 3.806 ms and MCWH yields an average response time 0.341 ms slower, at 4.147 ms. The average latency is of 0.001 ms for NIoTO, and 0.002 ms for MCWH, albeit once more, execution time is more relevant. On the one hand, this increase represents a more acceptable optimality gap of 8.2%. On the other hand, this represents that the optimality gap of MCWH decreases in larger scenarios, as leveraging Computing Continuum nodes closer to IoMT devices becomes increasingly optimal. Finally, MCWH achieves average response times of 4.512 ms and 4.596 ms in scenarios 3 and 4, respectively. In summary, while both NIoTO and MCWH may be used in smaller scenarios, MCWH is superior for larger ones, as it not only provides support for larger scenarios, but its optimality gap also decreases with scenario size.

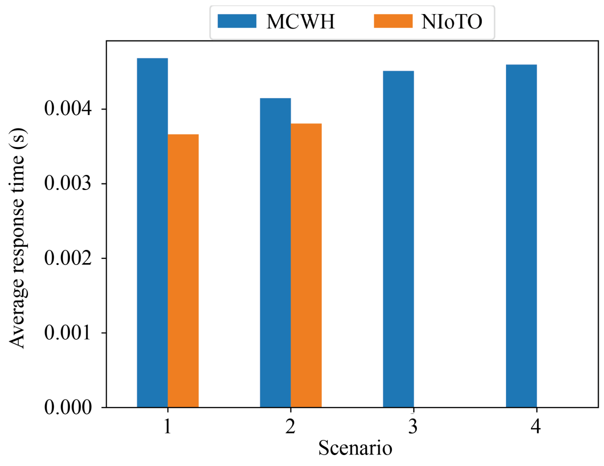

Figure 2. Response time results for MCWH and NIoTO

The next analysis considers cost instead, as shown in Figure 3. Moreover, to provide a more comprehensive representation, Figure 4 provides the results broken down by type of cost, i.e., CAPEX and OPEX. Regarding NIoTO in scenario 1, the deployment could have been made with a total cost of € 4,809.39, including the complete computing and networking infrastructure as well as the application. Out of this cost, approximately 1% (€ 48.58) is due to the OPEX, while CAPEX represents the remainder (€ 4,760.81). If MCWH is used instead, the deployment would cost € 5,414.29, € 604.90 (11.17%) more, out of which OPEX represents approximately 1.6% (€ 86.68). Hence, MCWH's cost increase is mainly due to CAPEX, which represents 85.67 & of the total increase (€ 518.22). Unlike with response time, the optimality gap of MCWH increases in larger scenarios, although at a smaller pace than the response time gap decreases.

This is due to the trade-off nature of the objective: although both are balanced to have equal weights, they cannot be adjusted at an arbitrary granularity, and are instead dependent on the decisions taken at each level. Thus, the optimality gap of MCWH does not necessarily grow at the same rate in both objectives, as shown in these results. In scenario 2, MCWH increases the cost by 18% (€ 2,144.99), from the € 9,769.93 required by NIoTO to € 11,913.92. Broken down by type of cost, OPEX still represents a minor part of the costs, approximately 1% for NIoTO (€ 96.82) and 1.5% for MCWH (€ 182.54), representing 8.5% of the optimality gap. MCWH's deployment of scenario 3 requires € 37,742.37, out of which € 533.94 are due to OPEX (1.41%) and the remaining € 37,208.43 are CAPEX. Finally, scenario 4 could be deployed with € 69,068.78, € 1,082.85 of OPEX (1.6%) and € 67,985.93 of CAPEX. On the one hand, given that NIoTO achieves optimal results, that MCWH presents small optimality gaps w.r.t. response time, the large scenario support that NIoTO cannot provide, and the speed-ups achieved, the results can also be considered acceptable from the point of view of costs. On the other hand, the importance of CAPEX when compared to OPEX is coherent with the use of the Continuum rather than purely the cloud. As more of the infrastructure needs to be acquired, the overall CAPEX rises and becomes more relevant, while cloud deployments would be dominated by OPEX, as Google Cloud instances have no CAPEX, but a significantly high OPEX.

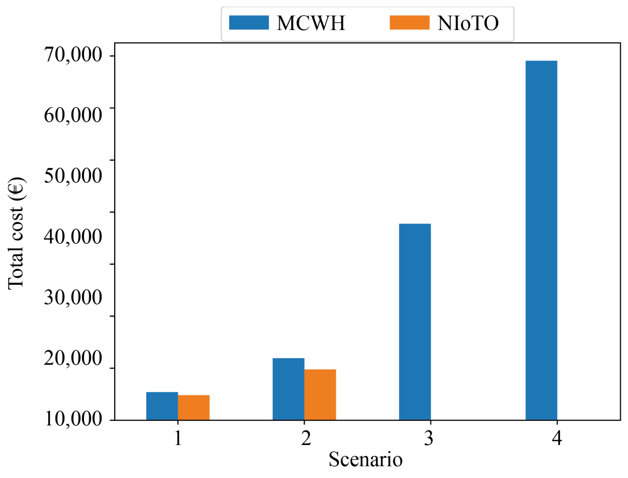

Figure 3. Cost results for MCWH and NIoTO

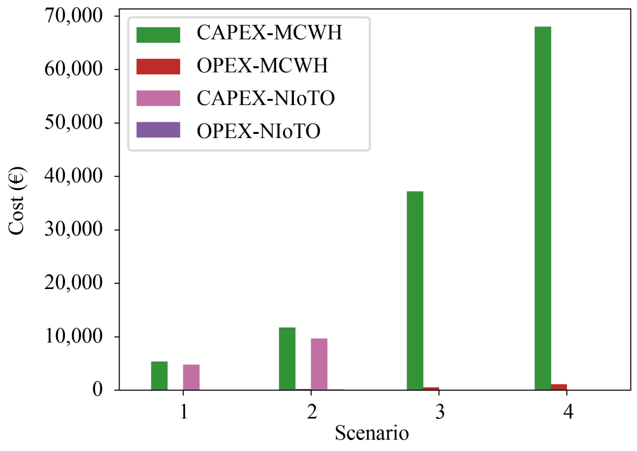

Figure 4. Costs for MCWH and NIoTO, broken down by type

Finally, Figure 5 represents the optimization times (i.e., the execution time of the solvers) for both NIoTO and MCWH. NIoTO requires 0.954 s to solve scenario 1. In contrast, MCWH takes only 0.029 s, representing a speed-up of 31.93× w.r.t. NIoTO. This speed-up further increases in larger scenarios: NIoTO needs 129.676 s to find a solution in scenario 2, while MCWH only takes 1.84 s (70.48× speed-up). For the last two scenarios, however, it is not possible to compare with NIoTO. Focusing thus on MCWH, scenario 3 takes slightly over a minute to solve at 65.08 s, while even scenario 4, with hundreds of devices to consider, only takes 2,428.82 s to solve. These results show that MCWH is significantly lighter than optimal methods, being able to solve larger scenarios where the optimal methods cannot. Moreover, in terms of time, MCWH achieves large speed-ups even in the smaller scenarios. This increase in speed-up as the scenarios grow is due to the significantly lower, polynomial complexity of MCWH explored in Sec. 3.2, in contrast to the factorial complexity of exact methods like NIoTO. It is also noteworthy that, overall, MCWH is not only lighter in terms of time, but also resources, as shown by the inability to obtain results with NIoTO in scenarios 3 and 4. Intuitively, MCWH allows larger scenarios to be solved under the same hardware, and said scenarios will be solved significantly faster, achieving large scenario support.

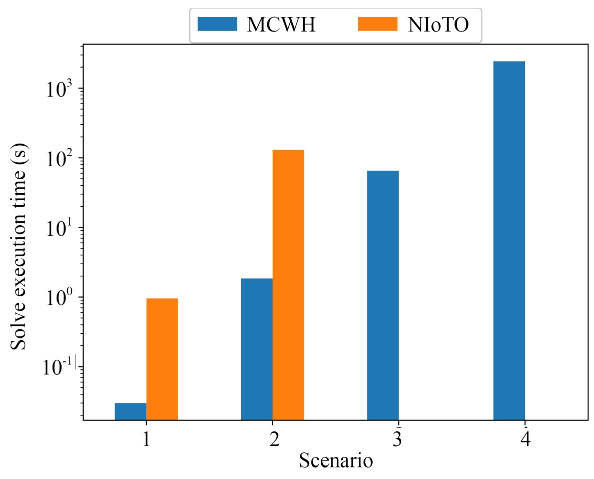

Figure 5. Optimization time results for MCWH and NIoTO

5. Conclusion and future works

As the IoT paradigm becomes increasingly relevant in intensive domains, the need to use MSAs, the Cloud Continuum and SDN will become equally important to meet the requirements of critical IoT applications. In this context, handling of SDN controller and microservice replication and placement, as well as the routing of both traffic and control flows, is crucial to fulfil said objectives. This handling must not only consider QoS, but also cost, trading them off, as well as the crucial nature of the interplay between microservices and the SDN network. Moreover, other proposals are unable to address large problems due to their poor scalability. To address these needs, this work presented MCWH, an algorithmic heuristic to replicate and place SDN controllers and microservices, as well as to route traffic and control flows. MCWH achieves up to 70.48 × speed-up w.r.t. NIoTO [13] with acceptable optimality gaps, and supporting larger scenarios. Hence, MCWH can, on the one hand, enhance the QoS of IoT applications and, on the other hand, reduce their deployment costs. These contributions will allow IoT applications for intensive domains that were deemed technically or financially infeasible to become viable. Moreover, other IoT applications that were already viable can also benefit from lower costs and response times. The main limitation of MCWH, however, is integration. While MCWH's design as a separate architectural module allows it to be integrated into orchestration systems, there are still significant efforts required to integrate it as a native module.

In the future, we expect to integrate MCWH in said orchestrating systems for both SDN networks and MSAs in the Cloud Continuum, automating not only the decision-making but also the execution of said decisions. We also expect to develop meta-heuristic methods to compare them against both NIoTO and MCWH. Finally, we expect to perform further evaluation in real or emulated network testbeds.

Acknowledgement

This work has been funded by the European Union under the MSCA project RENOS (contract number 101205037). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work was also partially funded by the grant DIN2020-011586 and the projects PDC2022-133465-I00, PID2021-124054OB-C31, and TED2021-130913B-I00 funded by MICIU/AEI/10.13039/50100011033 and the “European Union NextGenerationEU/PRTR”, by the Department of Education, Science and Professional Training of the Government of Extremadura (GR24099), and by the European Regional Development Fund.

Conflict of interest

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] Ketu S, Mishra PK. A contemporary survey on IoT based smart cities: architecture, applications, and open issues. *Wireless Personal Communications*. 2022; 125: 2319-2367. Available from: https://doi.org/10.1007/s11277-022-09658-2.

[2] Herrera JL, Bellavista P, Foschini L, Galán-Jiménez J, Murillo JM, Berrocal J. Meeting stringent QoS requirements in IIoT-based scenarios. In: *IEEE Global Communications Conference*. New York, United States: IEEE; 2020. p.1-6.

[3] Brogi A, Forti S, Guerrero C, Lera I. How to place your apps in the fog: state of the art and open challenges. *Software: Practice and Experience*. 2020; 50: 719-740. Available from: https://doi.org/10.1002/spe.2766.

[4] Indrasiri K. *Microservices in Practice-Key Architectural Concepts of an MSA*. Available from: https://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa/ [Accessed 5th July 2025].

[5] Herrera JL, Galán-Jiménez J, Bellavista P, Foschini L, Garcia-Alonso J, Murillo JM, et al. Optimal deployment of fog nodes, microservices and SDN controllers in time-sensitive IoT scenarios. In: *IEEE Global Communications Conference*. New York, United States: IEEE; 2021. p.1-6.

[6] Baktir AC, Ozgovde A, Ersoy C. How can edge computing benefit from software-defined networking: a survey, use cases, and future directions. *IEEE Communications Surveys and Tutorials*. 2017; 19: 2359-2391. Available from: https://doi.org/10.1109/COMST.2017.2717482.

[7] Heller B, Sherwood R, McKeown N. The controller placement problem. In: *Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks*. 2012. p.7-12.

[8] Choudhury B, Choudhury S, Dutta A. A proactive context-aware service replication scheme for adhoc IoT scenarios. *IEEE Transactions on Network and Service Management*. 2019; 16: 1797-1811. Available from: https://doi.org/10.1109/TNSM.2019.2928698.

[9] Das T, Sridharan V, Gurusamy M. A survey on controller placement in SDN. *IEEE Communications Surveys & Tutorials*. 2019; 22(1): 472-503. Available from: https://doi.org/10.1109/comst.2019.2935453.

[10] Sarkar I, Adhikari M, Kumar N, Kumar S. Dynamic task placement for deadline-aware IoT applications in federated fog networks. *IEEE Internet of Things Journal*. 2022; 9: 1469-1478. Available from: https://doi.org/10.1109/JIOT.2021.3088227.

[11] Faticanti F, Pellegrini FD, Siracusa D, Santoro D, Cretti S. Throughput-aware partitioning and placement of applications in fog computing. *IEEE Transactions on Network and Service Management*. 2020; 17: 2436-2450. Available from: https://doi.org/10.1109/TNSM.2020.3023011.

[12] Singh AK, Maurya S, Srivastava S. Varna-based optimization: a novel method for capacitated controller placement problem in SDN. *Frontiers of Computer Science*. 2020; 14: 1-26. Available from: https://doi.org/10.1007/s11704-018-7277-8.

[13] Herrera JL, Galán-Jiménez J, García-Alonso J, Berrocal J, Murillo JM. Joint optimization of response time and deployment cost in next-gen IoT applications. *IEEE Internet of Things Journal*. 2022; 10(5): 3968-3981. Available from: https://doi.org/10.1109/JIOT.2022.3165646.

[14] Gia TN, Jiang M, Rahmani AM, Westerlund T, Liljeberg P, Tenhunen H. Fog computing in healthcare internet of things: a case study on ECG feature extraction. In: *IEEE International Conference on Computer and Information Technology*. New York, United States: IEEE; 2015. p.356-363.

[15] Limaye A, Adegbija T. A workload characterization for the Internet of Medical Things (IoMT). In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. New York, United States: IEEE; 2017. p.302-307.

[16] Park HS, Jun CH. A simple and fast algorithm for *k*-Medoids clustering. *Expert Systems with Applications*. 2009; 36: 3336-3341. Available from: https://doi.org/10.1016/j.eswa.2008.01.039.

[17] Rousseeuw PJ. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*. 1987; 20: 53-65. Available from: https://doi.org/10.1016/0377-0427(87)90125-7.

[18] Erdös P, Rényi A. On random graphs I. *Mathematical Publications of Debrecen*. 1959; 6: 18.

Appendix A
A.1 *Mathematical formulation*

The mathematical formulation in this appendix uses the same notation specified in Sec. 3.1, except for, with slight abuse of notation, w_i, which denotes the i-th microservice in workflow w, i.e., $w_i = w.Microservices[i]$. The $\nVdash(x)$ function is also used, which has a value of 1 if the condition x holds, and 0 otherwise. Moreover, the following decision variables are added: the binary variables α_{whi} will take the value of 1 if microservice w_i, which services workflow w, is deployed to host h, β_{odwhi}, also binary, are set to 1 if the traffic generated as a consequence of routing the traffic for microservice w_i to service w is routed through the link l_{od} (i.e., with origin o and destionation d). γ_s is a binary variable determining whether an SDN controller is placed on switch s, while $\delta_{ss'}$ determines whether the SDN controller placed in switch s' controls switch s. Finally, if the control traffic sent by s is routed through l_{od}, ε_{ods} will take the value of 1.

The formulation, like MCWH, has two objectives: minimizing response time (Equation (1)) and minimizing cost (Equation (2)):

$$\min_{\alpha,\ \beta,\ \gamma,\ \delta,\ \varepsilon} \sum_{w\in W}\left[\sum_{i=1}^{|w.Microservices|}\left(\sum_{h\in H}\left(\alpha_{whi}\frac{w_i.Cycles}{h.CPU}\right)\right)\right. \tag{1}$$

$$\left.+\sum_{h\in H}\left(\sum_{l_{od}\in l}\left(\beta_{odwhi}l_{od}.Latency+\nVdash(d\in S)\sum_{l_{p'd'}\in l}(\varepsilon_{o'd'd}l_{o'd'}.Latency)\right)\right)\right]$$

$$\min_{\alpha,\ \beta,\ \gamma,\ \delta,\ \varepsilon} \sum_{h\in H}\left[h.CAPEX\max\left(\max_{w\in W,\ i\in 1,\ |w.Microservices|}(\alpha_{whi}),\ \max_{w\in W}(\nVdash(w.Requestor=h)\right)\right. \tag{2}$$

$$\left.+\sum_{w\in W}\sum_{i\in 1}^{|w.Microservices|} h.OPEX_{cycle}w_i.Cycles\alpha_{whi}\right]$$

$$+\sum_{s\in S}[(s.CAPEX+s.OPEX)\max(\max(l_{od}\in L, h\in H, i\in_{1,\ |w.Microservices|}\beta_{odwhi}),$$

$$\max(l_{od\in L,\ s'\in S}(\varepsilon_{ods}),\ \max_{s'\in S}(\gamma_s))]$$

The constraints are as follows: Equation (1) ensures each microservice in a workflow is serviced once. Equation (2) guarantees the maximum RAM limit for host. Equation (3) ensures each switch is controller by one and only one SDN controller, which must be placed, as according to Equation (4). Equation (5) defines how traffic flows through the network to communicate microservices in a workflow, wih Equation (6) being a special case for the first microservice of each workflow. Equation (7) specifies this for control flow. Finally, the maximum link capacity is handled by Equation (8). The problem can thus be formulated as Equations (1), (2) subject to Equations (1)-(8).

$$\sum_{h\in H}\alpha_{wih} = 1\forall w\in W,\ i\in 1,\ |w.Microservices| \tag{3}$$

$$\sum_{w \in W} \sum_{i=1}^{|w.Microservices|} \alpha_{wih} w_i.RAM \leq h.RAM \forall h \in H \tag{4}$$

$$\sum_{s' \in S} \delta_{ss'} = 1 \forall s \in S \tag{5}$$

$$\delta_{ss'} \leq \gamma_s \forall s,\ s' \in S \tag{6}$$

$$\sum_{d \in V} \beta_{odwhi} - \beta_{dowhi} = \begin{cases} 0 & \text{if } o \in S \\ \alpha_{whi-1}(1 - \alpha_{whi}) & \text{if } o = h \\ -\alpha_{whi-1}\alpha_{wh1} & \text{otherwise.} \end{cases} \tag{7}$$

$\forall o \in V,\ h \in H,\ w \in W,\ i \in [2,\ |w.Microservices|]$

$$\sum_{d \in V} \beta_{odwh1} - \beta_{dowh1} = \begin{cases} 0 & \text{if } o \in S \\ \nVdash(w.Requestor = h)(1 - \alpha_{wh1}) & \text{if } o = h \\ -\nVdash(w.Requestor = h)\alpha_{wh1} & \text{otherwise.} \end{cases} \tag{8}$$

$\forall o \in V,\ h \in H,\ w \in W$

$$\sum_{d \in V} \varepsilon_{ods} - \varepsilon_{dos} = \begin{cases} 0 & \text{if } o \in H \\ 1 - \delta_{so} & \text{if } o = s \\ -\delta_{so} & \text{otherwise.} \end{cases} \tag{9}$$

$\forall o \in V,\ s \in S$

$$\sum_{h \in H} \sum_{w \in W} \left[\left(\sum_{i=1}^{|w.Microservices|} \beta_{odwhi} w_i.Data \right) \right] <= l_{od}.Capacity \forall l_{od} \in L \tag{10}$$