Multiobjective Deployment of Data Analysis Operations in Heterogeneous IoT Infrastructure

Devki Nandan Jha, *Student Member, IEEE*, Peter Michalák, *Student Member, IEEE*, Zhenyu Wen, Rajiv Ranjan, *Senior Member, IEEE*, and Paul Watson

***Abstract*—The growth of Internet of Things (IoT) technology brings many new opportunities for applications in areas including smart healthcare, smart buildings, and smart agriculture. These applications must normally distribute the computations, required for extracting value from sensor data, over the IoT infrastructure platforms (e.g., sensors, phones, field-gateways, and clouds). This can be very challenging for IoT application developers due to the heterogeneity of the aforementioned platforms, potentially conflicting nonfunctional requirements (e.g., battery power, latency, and cost), and related deployment criteria, which is impossible to resolve manually. To address the above challenges, we have developed the *PATH2iot* framework that decomposes a complex IoT application into self-contained micro-operations. Based on the deployment criteria, *PATH2iot* automatically distributes the set of micro-operations across IoT infrastructure platforms, while respecting their run-time data and control flow dependencies. In our previous work, we have shown how to use the *PATH2iot* to optimize the battery life of a healthcare wearable. In this article, we describe a new research that significantly extends *PATH2iot*, which introduces a heuristic model capable of making optimal deployment decisions based on multiple conflicting nonfunctional requirements and selection criteria (user preferences). It does so by leveraging a well-known multicriteria decision-making method called the analytic hierarchical processes (AHP). The applicability of the deployment model is validated based on a real-world digital healthcare analytics use case. The results show that our model is able to find the optimal deployment solution for different user preferences.**

***Index Terms*—Analytic hierarchical process (AHP), Internet of Things (IoT), smart healthcare, streaming data.**

Manuscript received April 30, 2019; revised August 2, 2019; accepted November 20, 2019. Date of publication December 23, 2019; date of current version July 29, 2020. This work is supported in part by the Engineering and Physical Sciences Research Council, Centre for Doctoral Training in Cloud Computing for Big Data under Grant EP/L015358/1 and in part by the Digital Catapult Teaching Grant Award under Grant KH161118. Paper no. TII-19-1680. *(Corresponding author: Devki Nandan Jha.)*

The authors are with the School of Computing, Newcastle University, Newcastle Upon Tyne NE1 7RU, U.K. (e-mail: d.n.jha2@newcastle.ac.uk; p.michalak1@newcastle.ac.uk; zhenyu.wen@newcastle.ac.uk; raj.ranjan@newcastle.ac.uk; paul.watson@newcastle.ac.uk).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2019.2961676

I. Introduction

ADVANCES in Internet of Things (IoT) technology are transforming the society and the economy through their widespread impact, e.g., smart homes, smart healthcare, and smart agriculture. This will continue to grow: research by Cisco predicts that 50 billion smart devices will be connected to IoT by 2020 [1]. To extract and process the massive streaming data collected from these data sources, several stream processing engines are developed that run in cloud providing a common programming frameworks, e.g., Apache Storm [2] and Amazon Kinesis [3].

However, this cloud-based approach is not suitable for many critical IoT applications for three main reasons. First, some applications require close coupling between the IoT data generators and actions taken based on the analysis of the data [4]. For these applications, the centralized cloud-based analytics approach might introduce unacceptable message transfer delays, and there may be a major problem due to network failure. Second, sending all the raw data from sensors to the cloud for analysis is not possible in cases where it may require higher bandwidth than is available or affordable [5]. Third, sending all data to the cloud may flatten the battery of devices, such as healthcare wearables, too quickly [6].

To addresses these challenges, an alternative approach is to run part of the application on, or close to, the sensors that generate the data. In the literature, an IoT application can be represented using a set of queries (Q_is) which can be modeled as a directed acyclic graph (DAG) with data transformation operations (O_is) as its nodes, and dataflow dependencies (or control flow dependencies for computational synchronization, if/as needed) between data transformation operations O_i as its vertices (see Fig. 1) [7]. This approach has been made possible by the introduction of what has become known as edge devices. Development of smarter IoT and edge devices, with some local storage and processing (e.g., healthcare wearables), opens up a tremendous opportunity for local analytics. Smartphones and field gateways can perform some basic analytic operations on the data, as well as acting as a network bridge between IoT devices and the cloud [8], [9].

Unfortunately, distributing applications across such a wide range of infrastructure (clouds, edge devices, sensors) (see Fig. 1) is an extremely challenging task for a systems

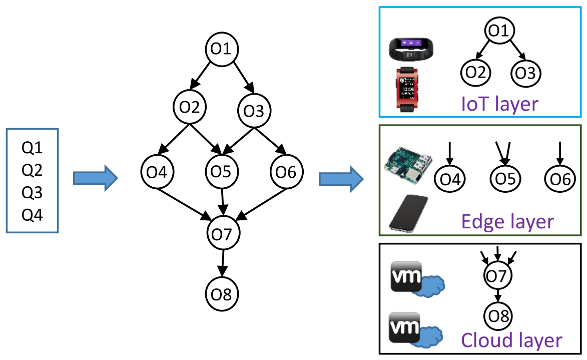

Fig. 1. Distributed deployment of IoT application.

programmer, especially for critical IoT applications such as in healthcare and city management. Key challenges include:

1) *Heterogeneity of IoT infrastructure*—Applications need to be deployed across a wide variety of heterogeneous platforms with differing programming interfaces and capabilities (e.g., sensors may have very limited computational capabilities). Data must be communicated between them over a variety of networks, each with its own idiosyncrasies and limitations.
2) *Meeting multiple, nonfunctional requirements*—Optimizing several requirements is challenging as the requirements may be conflicting to each other, e.g., performance, energy, cost, and dependability.

These challenges lead to the following research questions:

RQ 1. *How to model the problem of mapping complex IoT application across distributed IoT infrastructure with heterogeneous hardware and software configurations?*

RQ 2. *How to compute an optimal set of hardware and software configurations for each micro-operations of an IoT application considering conflicting nonfunctional requirements?*

Early efforts centered on the deployment of IoT applications across cloud and edge datacentres are mostly theoretical. Moreover, these solutions have not considered automatic computation partitioning and deployment. Nor have they considered the optimization of multiple conflicting nonfunctional requirements during the deployment process.

In our previous work [6], we developed the *PATH2iot* system that allows an application administrator to define the overall computation in a high-level declarative event processing language (EPL). The system decomposes the set of queries into a DAG that can be easily optimized. Next, it automatically partitions the applications into deployable operations, distributes them across the IoT infrastructure based on the battery power of the IoT device as the single nonfunctional requirement, and performs the physical deployment. However, this article did not focus on multiple nonfunctional requirements for making deployment decisions.

A. Contributions

To address the limitations of our previous work, this article makes the following new contributions:

1) We provide a formal model for the deployment of a general IoT application across a distributed IoT infrastructure (e.g., sensor/wearable, edge/phone, cloud).
2) We prove that the partitioning and deployment of an IoT application across a distributed IoT infrastructure is a strong NP-hard problem.
3) We introduce a new heuristic model *ABMO (AHP-based multiobjective optimization)* based on analytic hierarchical processes (AHP) [10] for finding the optimal deployment plan taking into account multiple, potentially conflicting nonfunctional requirements. This includes user preferences for making decisions based on a set of nonfunctional requirements.
4) A comprehensive experimental evaluation is carried out using a real-world, digital healthcare scenario for verifying the performance of the proposed decision-making technique.

The rest of this article is organized as follows. Section II discusses the relevant related work, while a formal model for the *PATH2iot* framework is presented in Section III. It also discuss the complexity analysis of the *PATH2iot* problem. Section IV describes the system model of *PATH2iot*, along with our proposed *ABMO* module. Section V evaluates the proposed framework on the real-world healthcare IoT application. Section VI concludes this article.

II. Related Work

The problem of distributed stream processing has been extensively studied in the literature. Frameworks such as Stream [11], Flextream [12], Naiad [13] from academia and Apache Storm [2], Amazon Kinesis [3], and Google MillWheel [14] from industry are common examples of stream processing systems. However, these approaches are usually limited to a cloud environment. Recent studies [8], [15], [16] show that data streams generated by IoT often require local processing to satisfy different application requirements. Distributing the processing across sensors and edge devices along with the cloud introduces new problems mainly due to the heterogeneity of processing and storage capacity, the power status of the battery-constrained devices, the mobility of the IoT devices, and the interaction between them.

There are very few models that consider edge/fog and IoT devices for distributed stream application deployments. Hong *et al.* [17] propose a framework for mobile fog computing that helps in the deployment of streaming IoT applications. Sarkar and Misra [18] propose a theoretical model for the fog computing infrastructure. They analyze the performance latency and energy performance of fog compared to the cloud environment. Saurez *et al.* [19] propose a programming infrastructure – "Foglets" – for distributing the deployment across edge and cloud. Foglets are used in the distributed deployment with latency and sensor mobility parameters taken into account. Benchmarks are used to test the application. Most of these models are theoretical and also consider only one or two parameters to be optimized.

In [20], an infrastructure module, LEONORE, is presented that provisions applications on resource-constrained edge devices. Cao *et al.* [21] present the work that distributes the analytics of health-monitoring application across edge processing. A similar work is done in Kea [22], a system for computational offloading of sensor data processing that also takes into consideration hardware capabilities, communication energy, and latency costs. However, the focus of the system is limited to smart phone sensor data, compared to our system that views computational placement holistically.

A general model to support QoS-aware deployment is presented in [23], where a multicomponent IoT application is deployed across fog infrastructure. The fog infrastructure here considers both edge and cloud environments. A simple Java-based prototype, FogTorch, is presented to illustrate the proposed model. Although the deployment of an application across the IoT infrastructure is considered, it does not address how to optimize the deployment solution. Also, this is an abstract model that does not consider how to generate the distributed IoT application and how to perform the physical deployment. Recent work in [24] addresses the problem of dynamic computation offloading in wearable healthcare devices. An improvement of 21.1% in battery life is achieved by partitioning the computation between the wearable and a mobile device: the approach is derived and validated using simulation software.

To the best of our knowledge, this article is the first to propose, implement, and evaluate an optimized solution for distributed stream processing that considers how to optimize for heterogeneous nonfunctional requirements. Our framework, *ABMO*, extends the capability of *PATH2iot*, is based on AHP [10] that incorporates user preferences along with a high-level computation description, nonfunctional requirements, and a resource catalogue, in order to find an optimized deployment plan. AHP is a well-known multicriteria decision-making method used in a variety of domains [25], [26].

III. Formal Model

In this section, we give some basic concepts and formally define our problem. Different symbols used in this article is given in Table I.

A. Basic Concepts

Definition 1: An IoT application A is a triple $\langle DS, Q, \Gamma\rangle$ where

1) DS represents a continuous stream of data generated by the IoT device.

2) $Q = \{q_1, q_2, \ldots, q_k\}$ is the set of k queries defined by the user as a description of the computation. The set Q is logically decomposed using a stream optimization function $\mathcal{P}()$ into a set of computational micro-operations O, which need to be deployed on the processing elements, as given in (1)

$$O = \{o_1, o_2, \ldots, o_l\} = \mathcal{P}\{q_1, q_2, \ldots, q_k\}. \quad (1)$$

TABLE I
SUMMARY OF SYMBOLS AND ABBREVIATIONS USED IN THIS ARTICLE

Symbol	Explanation
A	IoT application
DS	Data Stream
Q	Set of queries q_k
Γ	Identity property of the application
O	Set of operations o_l
$\mathcal{P}()$	Stream optimization function
R_H	Hardware requirements of micro-operation o_i
R_S	Software requirements of micro-operation o_i
$Cons$	Constraints for micro-operation o_l
id	Identifier of the component A
r_H	Hardware requirements of application A
r_S	Software requirements of application A
$cons$	Constraints for application A
I	IoT infrastructure
D	Set of IoT devices d
E	Set of edge devices e
C	Set of cloud datacentre components c
S_H	Hardware support
S_S	Software support
λ	Connection between different IoT components
R	Set of non-functional requirements R_i
ϕ	Unspecified values for R_i
Δ	Set of mappings Δ_i for Operation O to λ
$\preceq$	Satisfied by
P_L	Set of logical plans
P_{OD}	Set of all possible deployment plans
P_{PD}	Set of all physical deployment plans
M	Comparison Matrix
CR	Consistency Ratio
CI	Consistency Index
RI	Random Index
eig	Maximum eigen Value of comparison matrix M
W_j	Weight for non-functional requirement j
$\mathcal{N}$	Number of plans to shortlist
EI	Energy Impact
T^3	Total Turnaround Time

The dependency among various micro-operations is represented by a topologically ordered DAG. Each micro-operation o_l has specific hardware and software requirements R_H and R_S, respectively. Some constraints $Cons$ are also associated with the requirement specification.

3) Γ represents the identity property of the application and is represented as $\langle id, r_H, r_S, cons\rangle$, where id is the identifier of application, r_H and r_S are the set of hardware and software requirements, and $cons$ is the set of constraints defined for the hardware/software requirement of the application.

Definition 2: An IoT infrastructure I is a quadruple $\langle D, E, C, \lambda\rangle$, where

1) D is the set of IoT devices d, each represented by a set of 4-tuple $\langle id, Type, S_H, S_S\rangle$, where id is the identifier, $Type$ represents the type of device, and S_H and S_S, respectively, represent the hardware and software support provided by the device d.

2) E is the edge datacentre consisting a set of edge devices e, each denoted by $\langle id, S_H, S_S\rangle$, where id is the identifier, S_H and S_S, respectively, represent the hardware and software support provided by the edge device e.

3) C is the set of cloud datacentre components (VMs or containers) c, each denoted by $\langle id, S_H, S_S\rangle$, where id is the identifier, and S_H and S_S, respectively, represent the hardware and software capabilities provided by c.

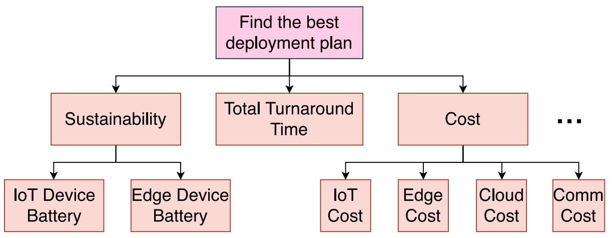

Fig. 2. Nonfunctional requirements.

4) $\lambda \in \{\{D \times E \times C\} \cup \{D \times E\} \cup \{D \times C\} \cup \{E \times C\}\}$ is a set of all the available connections from IoT device to edge to cloud.

Definition 3: The set R of nonfunctional requirements is a sequence of r elements $R = \{R_1, R_2, \ldots, R_r\}$, where each element R_i can have either numeric or Boolean values. Unspecified values of element R_i are denoted by $\emptyset$.

Numerous nonfunctional requirements may be associated with the application. Our approach is general, but those considered in this article are discussed in Appendix *A*. Fig. 2 shows the hierarchical representation for all the nonfunctional requirements considered in this article.

B. Problem Definition

Definition 4: Let $A = \langle DS, Q, \Gamma \rangle$ be an IoT application and $I = \langle D, E, C, \lambda \rangle$ be the IoT infrastructure. A possible deployment plan Δ_i is a mapping from operation $O : \mathcal{P}(Q)$ to $\lambda \in \{\{D \times E \times C\} \cup \{D \times E\} \cup \{D \times C\} \cup \{E \times C\}\}$ $(\Delta_i = O \rightarrow \lambda)$ if and only if:

1) all $o_j \in O$ must be mapped to some IoT infrastructure $I_k \in I$.
2) for each $o_j \in O$, $I_k \in I$, if $(o_j \rightarrow I_k) \in \Delta$, then $R_H(o_j) \preceq S_H(I_k)$, $R_S(o_j) \preceq S_S(I_k)$, and $satisfied(Cons(o_j))$.
3) $\sum_{j=1}^{\max} R_H(o_j) \leq S_H(I_k)$ and $\sum_{j=1}^{\max} R_S(o_j) \leq S_S(I_k)$.

The definition given above considers all the constraints to meet the optional deployment requirements. Condition (1) guarantees that all the operations must be deployed on some IoT infrastructure. Condition (2) allocates the operations o_j only to infrastructure I_k, which satisfies their hardware requirements $R_H(o_j)$ and software requirements $S_H(o_j)$, along with any other deployment constraints $Cons$ defined for the operation o_j. Condition (3) limits the number of operations to be deployed on an infrastructure component so that the hardware and software requirements are enough to satisfy the demands of selected operations $o_j | j \in \{i, \max\}$.

Definition 5: Given the available possible plans Δ, find the best possible plan $\Delta_{\text{best}} \in \Delta$ that optimizes all nonfunctional requirements such that for any other plan $\Delta_i \in \Delta$, $\Delta_i \leq \Delta_{\text{best}} | \forall R_l \in R$, and $\Delta_i < \Delta_{\text{best}} | \exists R_l \in R$.

C. Complexity Analysis

Given an IoT application A and IoT infrastructure I, finding the optimal deployment plan Δ_{best} from the set of possible plans Δ that optimizes all R nonfunctional requirements is strong NP-hard and can be proved by reduction from the bin-packing problem.

Bin-packing is known to be a strong NP-hard problem, which is nonsolvable in any polynomial time [27]. It is defined as follows: given a set of o objects with sizes $s_1, s_2, \ldots, s_o$ and a set of bins $B_1, B_2, B_3, \ldots$ of the same capacities C, find the smallest integer $k \in N$ such that all the o objects got mapped to some bins B_i following the condition that for any $i = \{1, 2, \ldots, k\}$, $\sum_{i \in B_k} s_i \leq C$.

Proposition 1: Finding an optimal mapping for *PATH2iot* problem in NP-hard.

Proof: Considering the formal definition of the bin-packing problem as given above, it is possible to transform the bin-packing problem into the simplest *PATH2iot* problem in a polynomial time. The transformation is as follows. ■

Change all the bins B_i to IoT infrastructure deployment nodes I (IoT device, edge device, or cloud datacentre components) with equal hardware capacities and no software support. Change all the objects o to operations O with s_o hardware requirements, no software requirements, no constraints, and no dependency between operations.

This maps the bin-packing problem into the simplest case of our *PATH2iot* deployment problem. This transformation can be easily achieved in polynomial time. Thus, it is proven that the simplest case of our problem is at least as hard as bin-packing problem, which is already strong NP-hard, making the generic *PATH2iot* problem $\in$ strong NP-hard.

Inherently, as given in Proposition 1, finding a solution of the bin-packing problem in polynomial time leads to finding a solution of the generic *PATH2iot* problem in polynomial time. No such algorithm exists for any NP-hard problem; therefore, we need a heuristic algorithm to find a solution.

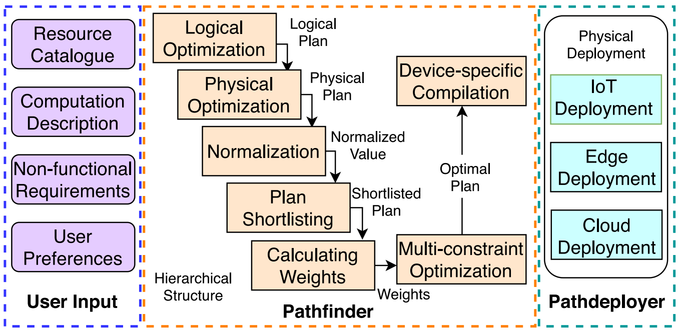

Fig. 3. Holistic representation of the deployment plan.

IV. System Model

Fig. 3 summarizes the internal processing of our proposed *PATH2iot* framework. The framework is divided into three components. The first is the *User Input* that accepts a set of EPL queries which defines – in high-level terms – the computation, the state of the deployment infrastructure, and the nonfunctional requirements to be placed on the system. The second is the *PATHfinder* implementation, where all the deployment decisions are performed automatically, while the third is the

PATHdeployer that performs the physical infrastructure deployment. *PATHfinder* is again divided into three stages, namely, *Initial Optimization*, *ABMO,* and *Device Specific Compilation*. A detailed discussion of each component is given below.

A. User Input

The whole system is driven by the following inputs:

Resource Catalogue: It holds a description of all the relevant features of the IoT infrastructure platforms over which the computations can be distributed. This includes the infrastructure capabilities in terms of hardware support S_H and software support S_S. It represents the state of the infrastructure, i.e., a description of the available cloud resources, IoT, and edge devices with their current state (e.g., active/disabled, battery level, battery capacity, network bandwidth). It also holds the constraints that need to be recognized when making deployment decisions. This includes a definition of user-defined functions (UDFs) that are supported by the system with their placement constraints, along with the energy impact coefficients for the supported operators. The optimizer accesses this information in the form of a JSON file.

Computation Description: To allow automatic partitioning over a set of platforms, it requires the computation to be defined in a high-level declarative language which can be analyzed, distributed, and optimized. In this article, we adopt the approach of [6] and use a complex event processing (CEP)-based relational model in which an EPL is used to define the computation.

Nonfunctional Requirements: This defines all the requirements that are required to be optimized. The set of nonfunctional requirements considered in this article are discussed in detail in Appendix A. The nonfunctional requirements are represented by a top-down hierarchical structure of level L, where lower level elements are grouped under some higher level elements based on a common property, e.g., lower level attributes IoT cost, edge cost, and cloud cost are grouped as they all calculate the cost.

User Preferences: This is one of the key user inputs as it defines the relative importance of the nonfunctional requirements. A user submits pairwise comparison values for all the nonfunctional requirements in the form of a CSV file.

B. PATHfinder

PATHfinder is an internal module of the *PATH2iot* system and is divided into three stages as explained below.

1) Initial Optimization: This stage is divided into two consecutive phases *logical optimization* and *physical optimization*. The details are given below.

1) Logical Optimization: The high-level user description for the computation is decomposed into a DAG, and the operators of the DAG are topologically sorted. Therefore, each operation is executed in a valid sequence. Various stream optimization techniques can be used to used to optimize the DAG [28]. A list of topologically sorted logical plans P_L are created that acts as input for the next step.

2) Physical Optimization: This step generates a set of physical deployment plans based on the logical plans P_L generated by the previous step. For each logical plan, it first creates all possible

TABLE II
SATTY SCALE FOR ASSIGNING THE PRIORITY VALUE

Value	Priority Scale	Value	Priority Scale
1	Equal	7	Demonstrated
3	Moderate	9	Extreme
5	Strong	2, 4, 6, 8	Intermediate

TABLE III
RANDOM INDEX (RI) VALUE

Size	1	2	3	4	5	6	7	8	9	10
RI	0	0	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49

Algorithm 1: Physical Optimization.

```
Input: $P_{OD}$ – list of optional deployment plans, $R_H$ – hardware
       requirements, $S_H$ – software requirements, $Cons$ – defined
       constraints, $I$ – IoT infrastructure
Output: $P_{PD}$ – list of physical deployment plans
1 for all $p \in P_{OD}$ do
2     if $(R_H(p) \preceq R_H(I)$ && $S_H(p) \preceq S_H(I)$ &&
      $Satisfied(Cons))$ then
3         ADD $P_{OD}$ to $P_{PD}$
4     end
5 end
```

deployment plans P_{OD} based on the topological ordering of the operations.

Algorithm 1 is used to shortlist the plans that satisfy the constraints and nonfunctional requirements and generates a list of physical deployment plans P_{PD}.

2) AHP-Based Multiobjective Optimization (ABM): This stage uses the AHP [10] for calculating the rank of each physical deployment plan. The whole process is divided into four steps as given below.

1) Calculating Weights: This step uses AHP to first find the weights for each nonfunctional requirements, based on the user preferences. The preferences are measured by using Satty scale [29] as given in Table II. A reciprocal comparison matrix M is constructed from the user preferences following the rules as given in (2)

$$M_{ij} = \begin{cases} 1, & \text{when } i = j \\ X, & \text{when } i > j \\ 1/M_{ij}, & \text{when } i < j \end{cases}\Bigg\}. \tag{2}$$

Before calculating the weights of the nonfunctional requirements, it is necessary to check whether the provided user-preference values are consistent or not. The consistency of comparison matrix is verified by checking the consistency ratio (CR) value. CR is calculated using the maximum eigen value (*eig*), size of the comparison matrix (*M.size*), and the random index (*RI*) values. The default *RI* values are given in Table III. Equation (3) is used to calculate the CR

$$CR = CI/RI \tag{3}$$

where $CI = (eig - M.size)/(M.size - 1)$. If the comparison matrix M is found to be consistent, AHP is used to calculate the final weights W_j for nonfunctional requirement j. Otherwise, the user is instructed to reenter the preference values. AHP uses

Algorithm 2: Calculating Weights.

```
Input: $Pref_{jk}$ – preference of $j$th non-functional requirement over $k$th
       non-functional requirement, $R_r$ – list of $r$ non-functional
       requirements, $M$ – Reciprocal comparison matrix
Output: $W$ – non-functional requirements weight
construct $M$ using $Pref_{jk}$ following equation (2)
if (!$Consistent(M)$) then
    Notify user to enter new values
    return -1
else
    $W$ = average (principle eigen vector ($M$))
end
Consistent($M$)
$eig = max(eigenvalue(M))$
$CI = (eig - M.size)/(M.size - 1)$
$CR = CI/RI$ |$RI$ is the Random Index as given in Table III
if ($CR < 0.1$) then
    Matrix $M$ is consistent
    return TRUE
else
    Matrix $M$ is inconsistent
    return FALSE
end
```

principal eigenvector to calculate the priority of each nonfunctional requirement. The final weights are computed by averaging the priority values. The pseudo-code for the whole process is explained in Algorithm 2.

2) Normalization: Directly comparing different nonfunctional requirement values is not possible as each requirement can have a different data type and range. Also, the optimal value depends on the type of requirements: in some cases, higher is better, e.g., battery power, while in others, lower is better, e.g., cost. It is necessary to normalize all these values to one type and range.

The normalization is performed according to the data type of the nonfunctional requirements, e.g., Boolean, numerical, etc., and the function whether to maximize or minimize. The steps of normalization process is summarized in Algorithm 3.

3) Plan Shortlisting: The complexity of the *Pathfinder* depends on the number of physical deployment plans. To manage the complexity of the *Pathfinder*, we shortlist $\mathcal{N}$ plans from the full list of physical deployment plans. If the total number of the plans are less than $\mathcal{N}$, we can skip this step.

Algorithm 4 explains the steps involved in the pruning process. It first ranks all the plans and finally selects top $\mathcal{N}$ plans for further evaluation.

4) Multiconstrained Optimization: This step combines the final weights of the nonfunctional requirement with the corresponding values of the plans to find the final rank of the plans. This step aims to find the optimal plan over the selected plans. The rank of each plan P_i is computed using (4), where W_j is the weight for nonfunctional requirement j, and $norm(R_j(P_i))$ is the normalized value for the plan P_i. The plan with highest rank is selected for the physical execution

$$Rank(P_i) = \sum_{j=0}^{r}(W_j) \times norm(R_j(P_i)). \quad (4)$$

3) Device-Specific Compilation: Once the execution plan is derived, the framework converts the selected plan into a deployment configuration which is finally transmitted to *PATHdeployer* for deployment over the available IoT infrastructure.

Algorithm 3: Normalization.

```
Input: $P_{PD}$ – list of physical deployment plans, $R_r$ – list of $r$
       non-functional requirements, $type_{R_j}$ – data type of the
       non-functional requirement $R_j$, $Option_{R_j}$ – option for the
       non-functional requirement $R_j$ to be maximized or minimized,
       $R_j(P_i)$ – value of $j$th non-functional requirement for $i$th plan
Output: $norm(R_j(P_i))$ – normalized value of $j$th non-functional
        requirement for $i$th plan
initialize $norm(R_j(P_i))$ to 0 for all $R_j(P_i)$
for each $R_j \in R_r$ do
    if ($type_{R_j}$ == $Numeric$) then
        if ($Option_{R_j}$ == $maximize$) then
            $Sum(R_j) = \sum_i R_j(P_i)$
            $norm(R_j(P_i)) = R_j(P_i)/Sum(R_j)$
        else if ($Option_{R_j}$ == $minimize$) then
            $Sum1(R_j) = \sum_i R_j(P_i)$
            $norm1(R_j(P_i)) = Sum1(R_j)/R_j(P_i)$
            $Sum(R_j) = \sum_i norm1(R_j(P_i))$
            $norm(R_j(P_i)) = norm1(R_j(P_i))/Sum(R_j)$
        else
            option is not valid
        end
    end
    if ($type_{R_j}$ == $Boolean$) then
        if ($Option_{R_j}$ == $maximize$) then
            $Sum(R_j) = \sum_i R_j(P_i)$
            $norm(R_j(P_i)) = R_j(P_i)/Sum(R_j)$
        else if ($Option_{R_j}$ == $minimize$) then
            swap 0 with 1
            do same as maximize
        else
            option is not valid
        end
    end
    if ($type_{R_j}$ == $Unordered\_set$) then
        find the maximal set $R_{max}$
        $norm1(R_j(P_i)) = size(R_j(P_i) \cap R_{max})/size(R_{max})$
        repeat the same process as in $Numeric$ with $maximize$ using
          the value $norm1(R_j(P_i))$ in place of $(R_j(P_i))$
    end
    if ($type_{R_j}$ == $Numeric\_Range$) then
        find the optimal Range $R_{opt}$
        $norm1(R_j(P_i)) = length(R_j(P_i) \cap R_{opt})/length(R_{opt})$
        repeat the same process as in $Numeric$ with $maximize$ using
          the value $norm1(R_j(P_i))$ in place of $(R_j(P_i))$
    end
end
```

Algorithm 4: Plan Shortlisting.

```
Input: $P_{PD}$ – list of physical deployment plans, $R_r$ – list of $r$
       non-functional requirements, $norm(R_j(P_i))$ – normalized value of
       $j$th non-functional requirement for $i$th plan, $\mathcal{N}$ – maximum number
       of plans to be generated, $grade(P_i)$ – grade of the plan $(P_i)$
Output: $P_{PPD}$ – list of shortlisted physical deployment plans
initialize $grade(P_i)$ to 0 for all $P_i \in P_{PD}$
for each $P_i \in P_{PD}$ do
    for each $R_j \in R_r$ do
        $grade(P_i) = grade(P_i) + norm(R_j(P_i))$
    end
end
sort $grade(P_i)$ in descending order
select top $\mathcal{N}$ plans and store the list in $P_{PPD}$
```

C. PATHdeployer

There are two stages to deploy the optimized plan in *PATH2iot:* cloud deployment and edge and IoT deployment. The two stages are detailed below.

Cloud Deployment: It is the first phase, where the tool verifies through the ZooKeeper that the destination *D2ESPer* instances are available in the infrastructure and then transmits the deployment configuration to them. The configuration information is then parsed and dynamically compiled EPL statements are executed within the Esper CEP engine, which is wrapped inside the *D2ESPer* tool.

Edge and IoT Deployment: It occurs once the cloud deployment has been completed. The configuration information is passed through a REST API endpoint. Preinstalled agents on IoT and edge devices pull regularly configuration from the endpoint, and once it has been received, it starts the processing accordingly.

D. Time Complexity of the Proposed Framework

This section computes the time complexity of our proposed framework. Let o be the number of operations and $\overline{I}$ is the number of IoT infrastructure components. Consider the nonfunctional requirements R is represented in a hierarchical structure with L number of levels. The complexity of each phase is given below.

Path Finder: This phase is divided into three stages, and the complexity of each stage is given below.

a) Initial Optimization: There are two steps for this stage, the complexity of *logical optimization* depends only on the operator reordering operation. For one *logical optimization*, the maximum complexity of operator-reordering is O(o). Varying the window size between a given range, $R1$ and $R2$ with defined step size $Step$ creates $\{(R1 - R2)/Step\}$ plans. Since $R1$, $R2$, and $Step$ are constant, the complexity for creating a set of logical plans, P_L, is O(o). For *physical optimization*, finding all the possible deployment plans for one logical plan takes O($o \times \overline{I}$) searches. Thus, the total complexity for finding all the optional deployment plan P_{OD} for the given set of logical plan P_L is O($P_L \times o \times \overline{I}$). For finding the physical deployment plan P_{PD}, we have to check all the available possible deployment plan, making the complexity as O(P_{OD}) $= P_L \times o \times \overline{I}$.

After reduction, the overall complexity for *initial optimization* step is O($P_L \times o \times \overline{I}$).

b) ABMO: This stage consists of four steps: *calculating weights*, *normalization*, *plan shortlisting,* and *multiconstrained optimization*. *Calculating weights* compares the user preferences for each attribute by using matrix manipulation and computes eigen vector as the priority values. Given n elements, the complexity to calculate the normalized eigen vector is O(n^3). Considering N_{lev} number of attributes at level lev and $r_{sub,lev}$ number of subattributes at level lev of subth attribute at level $(lev - 1)$, the complexity to calculate the normalize eigen vector is given as O($\sum_{lev=1}^{L} \sum_{sub=1}^{N_{lev-1}} (r_{sub,lev})^3$). The complexity value seems large but is comparatively very small when compared with the comparison complexity without using AHP. The total complexity for comparing all the low-level elements without using AHP is $(\sum_{sub=1}^{N_0} (r_{sub,1}))^3$, which is $>> (\sum_{lev=1}^{L} \sum_{sub=1}^{N_{lev-1}} (r_{sub,lev})^3)$ when there are a large number of elements which can be represented in a hierarchical structure. AHP reduces the complexity by breaking the problem in a hierarchical structure resulting in more, smaller comparisons.

For the *normalization* step, the time taken to normalize any nonfunctional requirement value is O(P_{PD}) irrespective of the data type. The complexity to normalize all R nonfunctional requirements is O($R \times P_{PD}$). Therefore, the final complexity for normalization step is O($R \times P_{PD}$). *Plan shortlisting* step finds the grade for each plan which is O($R \times P_{PD}$) complex. The time taken to sort the grade values is O($(P_{PD})^2$) and finally selecting the top $\mathcal{N}$ plan is O(1). Therefore, the final complexity for plan shortlisting is given as O($(P_{PD})^2$). For final step, *multiconstrained optimization* step, the complexity to calculate the rank for plan is O(R). For all $\mathcal{N}$ plan, the complexity is O($R \times \mathcal{N}$) = O(R) as $\mathcal{N}$ is a constant. Finally, the time taken to find the best value from sorted plan is O($\mathcal{N}$) = O(1).

Thus, the total time complexity for the second stage is reduced to O($\sum_{lev=1}^{L} \sum_{sub=1}^{N_{lev-1}} (r_{sub,lev})^3 + R \times P_{PD} + (P_{PD})^2$).

c) Device-Specific Compilation: The complexity of this step is constant as there is only one execution plan for the deployment.

Path Deployer: The complexity of this step is also constant as the deployment is always performed for one selected plan.

Thus, the overall time complexity of our proposed framework is summarized as O($(P_L \times o \times \overline{I}) + \sum_{lev=1}^{L} \sum_{sub=1}^{N_{lev-1}} (r_{sub,lev})^3 + R \times P_{PD} + (P_{PD})^2$).

V. Experimental Evaluation

In this section, we evaluate the performance of our proposed framework on a real testbed.

A. Experimental Setup

For experimentation, we choose a healthcare-based application that captures the physical activity and blood glucose level of a type II diabetes patients [30]. The architecture is as shown in Fig. 1. To analyze the activity, we used Pebble Steel smartwatch that contains an embedded accelerometer. The smartwatch connects to an LG G4 smart phone via a Bluetooth low-energy (BLE) network and the phone is then connected to the cloud via 4 G mobile network. The data analysis uses a *Step Count* algorithm [31] that accepts the raw accelerometer data and generates activity information. Our framework – *PATH2iot* – automatically decides where to place the components of the analysis while optimizing different objectives (maximizing the battery life on the smartwatch and mobile phone, minimizing the turnaround time, etc.). The starting point is a description of the required computation as a set of EPL rules

1) **INSERT INTO** *AccelEvent*
 SELECT getAccelData(25, 60)
 FROM *AccelEventSource*
2) **INSERT INTO** *EdEvent*
 SELECT Math.pow($x \times x + y \times y + z \times z$, 0.5) **AS** ed, ts
 FROM *AccelEvent* **WHERE** vibe $= 0$
3) **INSERT INTO** StepEvent
 SELECT * **FROM** *EdEvent*
 MATCH_RECOGNIZE (**MEASURES** A **AS** ed1, B **AS** ed2 **PATTERN** (A B) **DEFINE** A **AS** (A.ed > THR), B **AS** (B.ed $\leq$ THR))
4) **INSERT INTO** *StepCount*
 SELECT count(*) **AS** steps
 FROM *StepEvent*.win:flexi_time_batch(30, 120, 15, sec)
5) **SELECT** persistResult(steps, "step_sum," "time_series") **FROM** *StepCount*

TABLE IV
POWER CONSUMPTION COEFFICIENTS FOR THE PEBBLE STEEL WATCH

Operation	Energy Impact (mJ)	95% Conf Int
OS_{idle}	1.78	± 0.0370
25 Hz sampling	0.06	± 0.0153
SELECT	0.09	± 0.0416
ED	0.34	± 0.0665
POW	0.03	± 0.1039
WIN	0.06	± 0.0605
net_cost	5.06	± 0.2747
BLE_{active}	12.12	

TABLE V
POWER CONSUMPTION COEFFICIENTS FOR THE LG G4 MOBILE PHONE

Operation	Energy Impact (mJ)	95% Conf Int
OS_{idle}	56.28	± 4.520
n_cost	161.62	± 6.813
RF_active	2497.10	± 226.288

PATH2iot interprets this into a graph of basic operations, and then considers all the possible solutions that map these operations onto the available IoT infrastructure (Pebble watch, mobile phone, and cloud). For each option, a cost is derived, and the one that has the maximal value of the nonfunctional requirements is selected.

In order to calculate the energy consumption of the Pebble Steel watch and the mobile phone, we must know the energy consumption of performing each operation (i.e., per sample received from the accelerometer). A series of experiments were conducted in which the watch and the mobile phone were connected to a Monsoon Power Monitor to measure the energy expenditure with each executed operation. Based on central limit theorem (CLT), we assume that the entire data set follows Gaussian distribution. Therefore, we compute the 95% confidence interval ($Conf$) using (5), where sd and $\bar{X}$ are the standard deviation and mean of the sample and n is the size of the sample. The results are shown in Tables IV and V. Notably, the energy impact shown in the two tables represents the mean of the sample.

$$Conf = \bar{X} \pm 1.96 \times \frac{sd}{\sqrt{n}}. \tag{5}$$

To calculate the battery life resulting from each option, we need to know the overall capacity of the battery. The battery capacity and battery voltage of Pebble watch and LG G4 are 130 mAh, 3.7 V and 3000 mAh, 3.85 V, respectively. The $maxBatteryLife$ of Pebble watch and mobile phone β_D and β_E is calculated as $\beta_D = 130\,\text{mAh} \times 3.7\,V \times 3.6 = 1731.6$ J and $\beta_E = 3000\,\text{mAh} \times 3.85\,V \times 3.6 = 41580$ J.

To calculate the total transmission time from mobile phone to cloud, we considered the Wi-Fi data rate from our phone (30 Mbps). Also, to calculate the total cost, we considered the standard electricity cost (ignoring the average fixed cost) as £0.155/KWh [32] and standard data-rate cost as £0.01/MB [33]. We have neglected the VM cost for our experiment as the VM cost is almost the same for all the cases. Different nonfunctional requirement values for our experiment can be found in Ref. [34].

B. Experimental Results and Analysis

We have compared three scenarios, where different nonfunctional requirements of the IoT infrastructure have been monitored. The detail of these scenarios are as follows:

1) Baseline: The generated raw accelerometer data (under 25 Hz) is streamed from the Pebble Steel watch to the cloud as quickly as possible. Given the software restrictions, this is done in a batch of ten accelerometer samples, and therefore with a frequency of 2.5 messages per second. This scenario gives the best turnaround time but consumes maximum energy.

2) Optimized by Energy: The main focus, in this case, is to optimize the energy consumption of IoT device to increase the battery running hours. This is outlined in our previous article [6], where the optimizer selects the deployment plan where some of the operators are placed on the wearable watch, reducing the amount of data required to be transmitted and pushing windows closer to the data source in this case directly on the wearable device, with fixed window size of 120 s, greatly reducing the energy required to keep the Bluetooth connection opened. The result shows that we achieved a significant improvement in the energy consumption of the wearable device of 453% as compared to the baseline approach. However, the turnaround time in this scenario is higher as compared to the baseline approach.

3) Optimized by Multiple Conflicting Objectives: Depending on the type of application and user requirements, the module automatically selects the best deployment plan. It can be easily converted to the previous scenarios, i.e. baseline or optimized by energy by setting up the user preference highest for turnaround time or sustainability, respectively. To illustrate the effectiveness of our proposed plan, we considered three cases with different user preferences, as given below.

Case 1: In this case, the battery constraints are more important, so making the user preferences inclined toward sustainability. The comparison matrix, C_1, for level 1 is given below. For other levels, we considered equal priority for all the attributes.

$$C_1 = \begin{array}{cccc} & \text{Sus.} & T^3 & \text{Cost} \\ \text{Sus.} & 1 & 7 & 9 \\ T^3 & 1/7 & 1 & 2 \\ \text{Cost} & 1/9 & 1/2 & 1 \end{array}$$

Case 2: For this case, the total turnaround time has higher preference as compared to other nonfunctional requirements. The comparison matrix, C_1, for this case is given below:

$$C_1 = \begin{array}{cccc} & \text{Sus.} & T^3 & \text{Cost} \\ \text{Sus.} & 1 & 1/7 & 2 \\ T^3 & 7 & 1 & 9 \\ \text{Cost} & 1/2 & 1/9 & 1 \end{array}$$

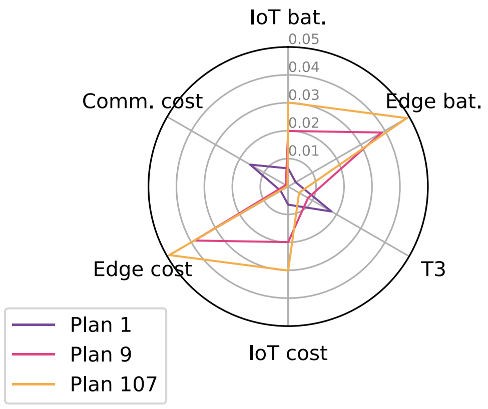

Fig. 4. Normalized nonfunctional requirement values for selected plans.

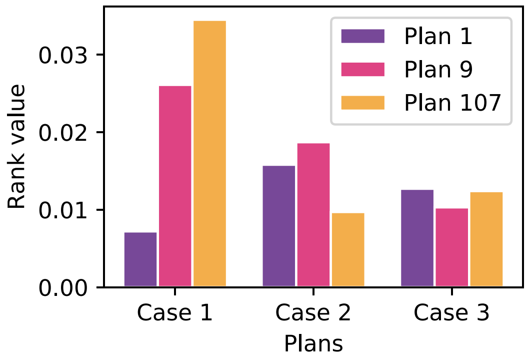

Fig. 5. Final rank in different cases.

Case 3: In this case, cost has given higher preference as compared to other nonfunctional requirements. The comparison matrix, C_1, for this case is shown below:

$$C_1 = \begin{array}{cccc} & \text{Sus.} & T^3 & \text{Cost} \\ \text{Sus.} & 1 & 2 & 1/7 \\ T^3 & 1/2 & 1 & 1/9 \\ \text{Cost} & 7 & 9 & 1 \end{array}$$

After completing the *initial optimization* stage, a total of 108 optimal plans got selected, which act as input for the $ABMO$. After execution of $ABMO$, the final result gives $Plan107$, $Plan9$, and $Plan1$ for the physical deployment in Case 1, Case 2, and Case 3, respectively. Fig. 4 presents a clear comparison of the normalized values for all three selected plans. The figure clearly shows that $Plan1$ has the best normalized value for *communication cost* and $T3$ with the value of (0.0157 and 0.0179) as compared to $Plan9$ and $Plan107$ with values of (0.0010 and 0.0082) and (0.0003 and 0.0045), respectively. However, for remaining nonfunctional requirements, $Plan9$ and $Plan107$ give better results. The detailed description of our implementation and result analysis is available in Ref. [35].

Based on the user preference values, the final rank is calculated as discussed in Section IV-B2. Depending on the users' preferences, our proposed approach always select the optimal solution. Fig. 5 shows the actual rank value for the best three plans. For Case 1, $Plan107$ has the highest rank value of 0.0345 followed by $Plan9$ with the rank value of 0.0261. For Case 2, the highest rank (0.0187) is shown for $Plan9$ followed by $Plan107$ (0.0158). Finally, for Case 3, $Plan1$ gives the best result (rank value of 0.0128) followed by $Plan107$ (rank value of 0.0124).

VI. Conclusion

PATH2iot provides a novel framework to facilitate the partitioning and deployment of IoT application across the distributed IoT infrastructure. This article extended the basic *PATH2iot* framework by adding *ABMO*, a heuristic module leveraging AHP for making optimal decision based on users' preferences and conflicting nonfunctional requirements. The case study showed that *ABMO* always chose the optimal deployment plan based on user preferences. The approach is very general; while we described a healthcare use case in this article, it can be directly applied to any other domain in which nonfunctional requirements are vital. For example, we have other Smart City and transport applications, in which *PATH2iot* is used to minimize the bandwidth needed when transmitting data over a low-bandwidth network. The system also works well when a model derived by machine learning is used to classify, or predict behavior; in this case, the model is simply treated as an operation in the computational graph, and *PATH2iot* is then used to decide where to place it in order to meet the nonfunctional requirements. In addition to this, our model is also compatible with the IoT environment that includes software-defined networks (SDN). To interoperate with the SDN infrastructure, a query operation needs to be composed and deployed based on the network function virtualization (NFV) technique. Finally, while the focus of this article was on stream processing IoT application, the approach can be easily applied to batch query processing workload. To this end, a batch query can be modeled as an operation, within our IoT graph, deployed at either edge or cloud layer. Since batch processing is normally executed over massive historical data, batch query operation is most likely to be mapped to the cloud layer. However, in the future work, we will make the deployment decision more dynamic so that if the environment changes over time, the deployment plans are automatically adjusted to maintain the nonfunctional requirements specified by the user.

Appendix A
Nonfunctional Requirements

The details of different nonfunctional requirements considered in this article are as follows.

Sustainability: This considers the energy impact in terms of the battery life of all the IoT and edge devices. Battery life is very important to consider, as it can be a limiting factor; for example, healthcare wearables that do not last a full day on a single charge of the battery are not going to be used in practice. Battery power is represented in terms of energy impact (EI), which is measured in milliJoule (mJ). As an example, the energy impact for a BLE-connected IoT device EI_D and edge device

EI_E is calculated as given in (6) and (7)

$$EI_D = OS_{idle} + \sum_{i}^{n} c\_cost_i + \frac{msg\_count_D \times n\_cost_D + BLE_{active} \times BLE_{dur}}{cycle\_length} \tag{6}$$

$$EI_E = OS_{idle} + \sum_{j}^{n1} c\_cost_j + \frac{msg\_count_E \times n\_cost_E + RF_{active}}{time\_slice} \tag{7}$$

where OS_{idle} is the power consumption by the IoT device/edge device caused by the operating system, c_cost and n_cost are the overall power consumption of the IoT device (D) and edge device (E) for each computation and transmission, respectively, msg_count specifies the number of messages transmitted from IoT device to edge device or edge device to cloud device, BLE_{active} is the power consumption for the Bluetooth low energy active state that is activated just after the message transfer, BLE_{dur} is the period of BLE_{active} state, $cycle\_length$ is the time duration after which the whole cycle repeats, and $time\_slice$ is the particular period of time considered for edge device.

The energy impact value represents the average power consumed by the IoT device and the edge device. The battery life is inversely proportional to the energy impact. The battery life in hours (bat_D) for IoT device D is computed as given in (8)

$$bat_D \propto (EI_D)^{-1} \Rightarrow bat_D = \beta_D \times (EI_D)^{-1} \tag{8}$$

where β_D is a constant called $maxBat$ that depends on IoT device D and is given by (9), where $bat(C)$ is the battery capacity and $bat(V)$ is the battery voltage

$$maxBat = bat(C) \times bat(V) \times 3.6. \tag{9}$$

Similarly, for an edge device, the battery life in hours bat_E is calculated in terms of energy impact EI_E and a device-specific constant β_E as given in (10)

$$bat_E = \beta_E \times (EI_E)^{-1}. \tag{10}$$

Total Turnaround Time: The raw data generated by the accelerometer is partially processed by the IoT device and is then transferred to edge devices for further processing. The final processing takes place in a cloud datacenter which also saves the result for further processing (e.g., for cross-population analytics that can generate better predictive models). The total turnaround time T^3 is given by summing the time taken by IoT device T_D, edge device T_E, and cloud datacenter T_C from data generation to data computation and is given by equation (11)

$$T^3 = T_D + T_E + T_C. \tag{11}$$

Each layer performs some operation and then sends the data to upper layer. The time taken by IoT device T_D is given in (12). The total time is equal to the cycle length as it includes both computation time and time to send data to the edge device

$$T_D = cycle\_length. \tag{12}$$

The time taken by edge and cloud datacenter are given in (13a) and (13b), respectively

$$T_E = T_E(comp) + T_{E\to C}(trans) \tag{13a}$$

$$T_C = T_C(comp) \tag{13b}$$

where $T_{E\to C}(trans)$ is the transmission time from edge to cloud and $T_x(comp)$ is the computation time taken by either edge device or cloud datacenter. For the sake of simplicity, we are not considering any waiting time or queueing time at any part of the IoT infrastructure.

Cost: Performing the operations on either IoT device, edge or cloud datacenter incurs some cost in terms of electricity charge, setup cost, cloud VM cost, or storage cost. There is an additional cost associated with the data transfer. The total cost ($Cost_{Total}$) is given by the sum of the cost incurred by an IoT device ($Cost_D$), edge device ($Cost_E$), cloud datacenter ($Cost_C$), and communication cost ($Cost_{comm}$). The cost incurred by an IoT device is determined in terms of electricity cost in charging the device plus a fixed setup cost. The electricity cost depends on the power consumed (energy impact) by the device and the per unit electricity rate ρ_{elec}. The value of $Cost_D$ is given in (14a). Similarly, the cost for an edge device depends on the setup cost and electricity cost as given in (14b). The cost for cloud datacenter can be given in terms of launching cost and the processing and storage cost of a VM as given in (14c)

$$Cost_D = (EI_D \times \rho_{elec}) + Cost_D(set_{up}) \tag{14a}$$

$$Cost_E = (EI_E \times \rho_{elec}) + Cost_E(set_{up}) \tag{14b}$$

$$Cost_C = Cost_C(VM_{proc}). \tag{14c}$$

Data is transferred from the IoT device to the edge, and from the edge device to the cloud. For an IoT device, the data transfer costs drain the battery (see Sustainability above) but the data transfer from edge device to cloud datacenter costs not only in terms of the edge device's battery charge cost, but also the network charge, e.g., for transferring data over a 3G or 4G network. The communication cost is calculated by multiplying the quantity of data transferred (msg_count_E) with the data rate charge (ρ_{data}) as given in (15)

$$Cost_{comm} = (msg\_count_e + msg\_header) \times \rho_{data}. \tag{15}$$

References

[1] Cisco, "Fog computing and the Internet of Things: Extend the cloud to where the things are," pp. 1–6, 2015. [Online]. Available: https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf

[2] Apache storm. [Online]. Available: http://storm.apache.org/

[3] Amazon kinesis. [Online]. Available: https://aws.amazon.com/kinesis/

[4] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky, "Key ingredients in an IoT recipe: Fog computing, cloud computing, and more fog computing," in *Proc. 19th Int. Workshop IEEE Comput. Aided Model. Des. Commun. Links Netw.*, 2014, pp. 325–329.

[5] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[6] P. Michalák and P. Watson, "PATH2iot: A holistic, distributed stream processing system," in *Proc. Int. Conf. IEEE Cloud Comput. Technol. Sci.*, 2017, pp. 25–32.

[7] M. Nardelli, S. Nastic, S. Dustdar, M. Villari, and R. Ranjan, "Osmotic flow: Osmotic computing+ IoT workflow," *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 68–75, Mar./Apr. 2017.

[8] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, 2014.

[9] 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.*, ACM, 2012, pp. 13–16.

[10] T. L. Saaty, "Axiomatic foundation of the analytic hierarchy process," *Manage. Sci.*, vol. 32, no. 7, pp. 841–855, 1986.

[11] A. Arasu *et al.*, "Stream: The Stanford data stream management system," in *Proc. Data Stream Manage.*, Springer, 2016, pp. 317–336.

[12] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," in *Proc. 18th Int. Conf. IEEE Parallel Architectures Compilation Techn.*, 2009, pp. 214–223.

[13] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, ACM, 2013, pp. 439–455.

[14] T. Akidau *et al.*, "MillWheel: Fault-tolerant stream processing at Internet scale," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[15] M. Satyanarayanan *et al.*, "Edge analytics in the Internet of Things," *IEEE Pervasive Comput.*, vol. 14, no. 2, pp. 24–31, Apr./Jun. 2015.

[16] R. Fang, S. Pouyanfar, Y. Yang, S.-C. Chen, and S. Iyengar, "Computational health informatics in the Big Data age: A survey," *ACM Comput. Surv.*, vol. 49, no. 1, p. 12, 2016.

[17] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the Internet of Things," in *Proc. 2nd ACM SIGCOMM Workshop Mobile Cloud Comput.*, ACM, 2013, pp. 15–20.

[18] S. Sarkar and S. Misra, "Theoretical modelling of fog computing: A green computing paradigm to support IoT applications," *IET Netw.*, vol. 5, no. 2, pp. 23–29, 2016.

[19] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, "Incremental deployment and migration of geo-distributed situation awareness applications in the fog," in *Proc. 10th ACM Int. Conf. Distrib. Event-Based Syst.*, ACM, 2016, pp. 258–269.

[20] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "A scalable framework for provisioning large-scale IoT deployments," *ACM Trans. Internet Technol.*, vol. 16, no. 2, p. 11, 2016.

[21] Y. Cao, P. Hou, D. Brown, J. Wang, and S. Chen, "Distributed analytics and edge intelligence: Pervasive health monitoring at the era of fog computing," in *Proc. Workshop Mobile Big Data.*, ACM, 2015, pp. 43–48.

[22] R. B. Das, N. V. Bozdog, M. X. Makkes, and H. Bal, "Kea: A computation offloading system for smartphone sensor data," in *Proc. Int. Conf. IEEE Cloud Comput. Technol. Sci.*, 2017, pp. 9–16.

[23] A. Brogi and S. Forti, "QoS-aware deployment of IoT applications through the fog," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1185–1192, Oct. 2017.

[24] H. Kalantarian, C. Sideris, B. Mortazavi, N. Alshurafa, and M. Sarrafzadeh, "Dynamic computation offloading for low-power wearable health monitoring systems," *IEEE Trans. Biomed. Eng.*, vol. 64, no. 3, pp. 621–628, Mar. 2017.

[25] D. N. Jha and D. P. Vidyarthi, "A heuristic for security prioritized resource provisioning in cloud computing," in *Proc. IEEE UP Sect. Conf. Elect. Comput. Electron.*, 2015, pp. 1–6.

[26] A. Mendoza, E. Santiago, and A. R. Ravindran, "A three-phase multicriteria method to the supplier selection problem," *Int. J. Ind. Eng.*, vol. 15, no. 2, pp. 195–210, 2008.

[27] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[28] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, p. 46, 2014.

[29] T. L. Saaty, "How to make a decision: The analytic hierarchy process," *Eur. J. Oper. Res.*, vol. 48, no. 1, pp. 9–26, 1990.

[30] L. Roberts, P. Michalák, S. Heaps, M. Trenell, D. Wilkinson, and P. Watson, "Automating the placement of time series models for IoT healthcare applications," in *Proc. IEEE 14th Int. Conf. e-Sci.*, 2018, pp. 290–291.

[31] N. Zhao, "Full-featured pedometer design realized with 3-axis digital accelerometer," *Analog Dialogue*, vol. 44, no. 6, pp. 1–5, 2010.

[32] "Average variable unit costs and standing charges for standard electricity in UK," [Online]. Available: https://www.gov.uk/government/uploads/system/uploads/attachment_data/fi le/357808/qep_224.xls

[33] "Pay as you go rates on three," [Online]. Available: http://www.three.co.uk/Store/Pay_As_You_Go_Price_Plans

[34] PATH2iot-data. [Online]. Available: https://ln.sync.com/dl/278ca25a0/4hnrq9ng-myq3f55n-qwaj9w69-mx3qe3n4

[35] ABMO. [Online]. Available: https://github.com/DNJha/ABMO/tree/merge2path

Devki Nandan Jha received the M.Tech. degree in computer science and technology from Jawaharlal Nehru University, New Delhi, India, in 2015. He is working toward the Ph.D. degree in computer science with the School of Computing Science, Newcastle University, Newcastle Upon Tyne, U.K.

His research interests include cloud computing, big data analytics, machine learning, and Internet of Things.

Peter Michalák received the B.Eng. degree in computer software engineering from the JAMK University of Applied Sciences, Finland, in 2013, and the B.Sc. degree (with distinction) in computer engineering from the University of Žilina, Slovakia, in 2012. He is currently working toward the Ph.D. degree in computer science with the EPSRC Centre for Doctoral Training in Cloud Computing for Big Data, School of Computing Science, Newcastle University, Newcastle Upon Tyne, U.K.

He previously worked as an R&D Software Developer with Tieto Finland Oy. His research interests include distributed computing, Internet of Things, and real-time event processing.

Zhenyu Wen received the M.S. and Ph.D. degrees in computer science from Newcastle University, Newcastle Upon Tyne, U.K., in 2011 and 2015, respectively.

He is currently a Postdoctoral Researcher with the School of Computing, Newcastle University. His current research interests include multiobjects optimization, crowdsources, AI, and cloud computing.

Rajiv Ranjan received the Ph.D. degree in computer science and software engineering from the University of Melbourne, Parkville, Australia, in 2009.

He is currently a Full Professor in Computing Science with Newcastle University, Newcastle Upon Tyne, U.K. Before moving to Newcastle University, he was Julius Fellow (2013–2015), Senior Research Scientist, and Project Leader with the Digital Productivity and Services Flagship of Commonwealth Scientific and Industrial Research Organization (CSIRO – Australian Government's Premier Research Agency). Prior to that, he was a Senior Research Associate (Lecturer level B) with the School of Computer Science and Engineering, University of New South Wales.

Paul Watson received the Ph.D. degree in parallel computing from the Manchester University, Manchester, U.K., in 1986.

He is currently a Professor of Computer Science, Director of the Informatics Research Institute, and Director of the North East Regional e-Science Centre, Newcastle University, Newcastle Upon Tyne, U.K. He also directs the UKRC Digital Economy Hub on "Inclusion through the Digital Economy." Prior to moving to Newcastle University, Paul worked with the ICL as a System Designer of the Goldrush MegaServer parallel database server. He was also a Lecturer with the Manchester University. His research interests include scalable information management, including data-intensive e-Science, dynamic service deployment, and e-Science applications. He has over 40 refereed publications, and three patents.