EDITOR: Schahram Dustdar, dustdar@dsg.tuwien.ac.at

DEPARTMENT: INTERNET OF THINGS, PEOPLE, AND PROCESSES

A Fault-Tolerant Workflow Composition and Deployment Automation IoT Framework in a Multicloud Edge Environment

Osama Almurshed, Omer Rana, and Yinhao Li, *Cardiff University, Cardiff, CF10 3AT, U.K.*
Rajiv Ranjan, *Newcastle University, Newcastle upon Tyne, NE1 7RU, U.K.*
Devki Nandan Jha, *University of Oxford, Oxford, OX1 2JD, U.K.*
Pankesh Patel, *University of South Carolina, Columbia, SC, 29208, USA*
Prem Prakash Jayaraman, *Swinburne University of Technology, Hawthorn, VIC, 3122, Australia*
Schahram Dustdar, *TU Wien, 1040, Vienna, Austria*

With the increasing popularity of IoT application, e.g., smart home, smart manufacturing, the importance of underlying system availability, safety, reliability, and maintainability become crucial in the development processes. IoT applications are expected to continuously provide reliable services and features, which put the fault-tolerance mechanism as a priority. Current IoT fault-tolerant systems are designed to overcome any faults caused by human activities and physical errors to preserve the correct IoT workflow execution. However, addressing fault-tolerant interaction in multicloud edge environment and failed service deployment automation remains challenging. This article proposes a novel fault-tolerant model that offers the self-detection and automatic recovery of faults to increase IoT applications' reliability and to address the infrastructure level failure in the heterogeneous IoT environments. Based on the proposed model, we implement a fault-tolerant workflow composition and deployment automation system, which is empowered by a layered architecture and a time-dependent failure model. The efficiency and effectiveness of the proposed system are validated and evaluated with a real-world IoT application.

Internet of Things (IoT) is mainly driven by data that is transferred between resources including cloud, edge, and IoT devices. It raises the importance of IoT systems in terms of availability, safety, reliability, and maintainability. Reliability is a primary aim to implement for IoT applications concerned with Quality of Service (QoS). Reliability is threatened by the occurrence of failures, where an IoT system can hardly offer potential services. There are three main methods to mitigate failures including fault correction, fault avoidance, and fault tolerance, where fault tolerance refers to detect and recover faults in runtime. In IoT environment, applications are expected to continuously provide reliable services and features, which put fault-tolerance mechanism as a priority.

Consider that the Newcastle City Council bids for a new IoT project for flood forecasting. Their engineers plan to develop and deploy a flood forecasting application, which provides a real-time rainfall map and main road risk level to avoid flood damage in the city. In this application, the raw streaming rainfall data captured by CCTV and sensors around the city is supposed to be

1089-7801 © 2021 IEEE
Digital Object Identifier 10.1109/MIC.2021.3078863
Date of publication 14 May 2021; date of current version 19 July 2022.

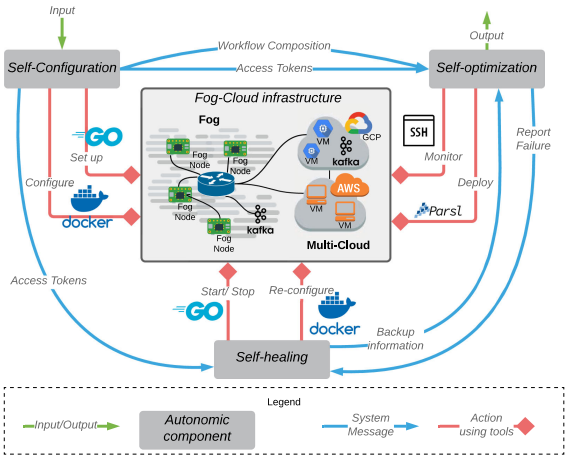

FIGURE 1. System Overview of Proposed Approach.

delivered and analyzed by a given rainfall model and flood forecasting model. The heterogeneous infrastructure deployed in this flood application may include CCTV, rainfall sensors as IoT devices, Raspberry Pis as edge devices, and various virtual machines provided by cloud providers as cloud resources. It raises a challenge to select appropriate infrastructure combinations to meet QoS requirements (end-to-end latency and reliability) and the city council budget limits. Meanwhile, battery-driven IoT devices and Raspberry Pis have a chance to fail with artificial or natural impacts. To avoid infrastructure failure, a robust multilevel reconfiguration mechanism is necessary to apply, which increase the system availability and reliability. Moreover, when renewing a failed resource, a centralized deployment agent is required which can manage and deploy every task across different IoT application resources to maintain the IoT applications' availability.

This article presents a framework that enables a user to compose any IoT application with defined QoS parameters, then automatically set up the infrastructure according to the recommended configuration. As shown in Figure 1, the system is divided into three components: Self-configuration, self-optimization and self-healing. Self-configuration takes user QoS requirements and a workflow model of an IoT application, and then it returns ready to use infrastructure. Moreover, the self-configuration automatically sets up the infrastructure and sends the access tokens to self-optimization and self-healing. Then, self-optimization loads configuration to operate and monitor infrastructure according to QoS. During failure events, self-optimization reports self-healing with system failures. Self-healing component restores operation to normal by backing up the infrastructure with resources and recover the faulty ones.

Our system uses IoTWC[1] for configuring, and greedy nominator heuristic (GNH)[2] for deploying IoT application. IoTWC returns suitable infrastructure for IoT application with defined QoS parameters while GNH uses redundant deployment to avoid the additional application delay caused by the infrastructure's unreliability, e.g., lack of response or temporal resource freezing (i.e., application-level failure). The Flood-PREPARED[3] application is utilized to test the functionality of the proposed framework that solves challenging issues associated with developing and operating IoT applications, such as avoiding dependence issues using container technology. Flood-PREPARED[3] is a real-time surface water flooding data monitoring and management application. It is a system-based approach to predict flooding from intense rainfall events. The experiment contains a platform that operates on raw streaming data and analyses using a given ML model near the edge. Moreover, multicloud infrastructure supports computing power, e.g., additional storage and processor type (FPGA or GPU). The application has chained service functions in the form of a directed acyclic graph (DAG). The functions apply logical data operation, machine learning (ML) model training or ML prediction. Each service function has its execution requirements, which will determine the place of execution, i.e., cloud or edge.

In summary, we propose a novel fault-tolerant model that offers self-detection and automatic recovery of faults to increase IoT applications' reliability to address the infrastructure level failure in the heterogeneous multicloud edge environment. Meanwhile, based on the fault-tolerance model, we propose a workflow composition and deployment automation system, which utilizes a layered architecture and a time-dependent failure model to offer deployment automation and infrastructure recovery. Additionally, we evaluate the efficiency and effectiveness of the proposed system with a real-world IoT application.

IOT FAULT-TOLERANCE MODEL

This section presents a novel IoT fault-tolerance model and discusses details of the fault detection and recovery mechanism.

A self-adaptive software adjusts itself when it does not accomplish the purpose, e.g., not meeting the IoT application requirements. The adjustment can vary from parameters and methods to components and system resources. System issues are detected during consistent monitoring after which an action to restore the system to its regular functionality is decided. The adaption is in the form of properties, i.e., self-configuration, self-healing, self-optimization that can be in a single system component or distributed within the

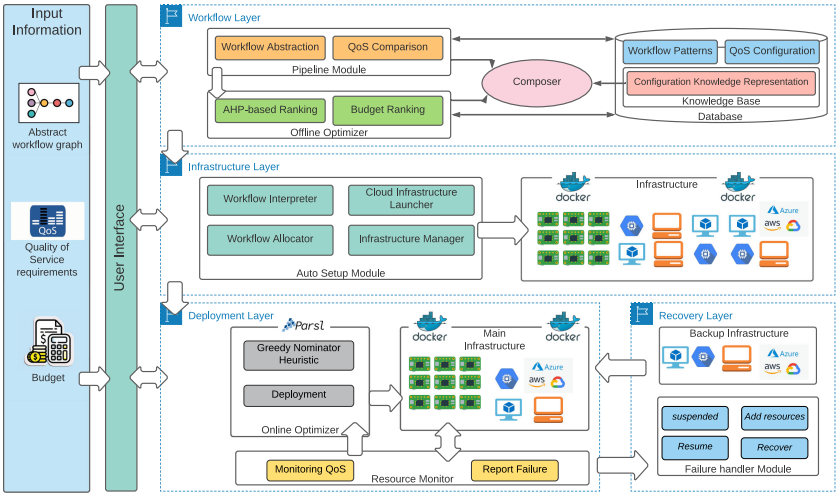

FIGURE 2. IoT fault-tolerant workflow composition and deployment automation system architecture.

architecture layers. For example, self-healing in Figure 1 is in the deployment layer and the recovery layer (layers in Figure 2). The deployment layer uses redundant deployment as a proactive fault-tolerance mechanism, whereas the recovery layer restores failed resources as a reactive fault-tolerance mechanism.

In our proposed IoT reactive fault-tolerance model, an IoT application consists of two main components: controller and resource pool. The controller is a fog/edge node that controls the resource pool. It deploys the application, monitors resources, and switches to/from recovery mode. The controller manages all the available resources and can be categorized into two resources: primary and backup. Primary resources refer to the resources utilized in a particular application as a priority (usually considered edge resource and part of cloud resources). Backup resources are for backup purpose (usually cloud resources). Since the available edge devices may have different architecture and configuration, it may not always be possible to have a backup device with the same configuration. Consequently, we choose on-demand cloud resources as the backup to meet the end-to-end latency constraints.

The system has two operational modes: regular mode and recovery mode. Regular mode is when the controller deploys IoT applications in the primary resources. The recovery mode controller deploys in the temporal infrastructure (i.e., primary resources with chosen cloud backup resources) and recovers failed resources. In the case of intolerable failure, the controller switches to recovery mode. After recovering failed nodes, it switches to the regular mode and resumes the deployment in the primary resources.

The centralized controller contains a self-optimization component and a self-healing component. Self-optimization manages continuous IoT applications deployments and detects an intolerable error in the primary resources group. Moreover, it operates in both primary resource (i.e., regular mode) and temporal infrastructure (i.e., recovery mode). Self-healing role can be summarized in three main actions: switch OFF failed resources, switch ON backup resources and recover failed resources. It prepares the temporal infrastructure for the self-optimization component by adding cloud resources to unfailing resources from primary resources. During recovery mode, the self-healing component starts a set of recovery procedures including failed resources reboot, environment setup, and availability check. After recovering failed nodes, the self-optimization component resumes the deployment in the regular mode. Self-healing reduces the cost of on-demand cloud VMs by switching OFF backup cloud resources during the regular mode.

SYSTEM DESIGN

This section presents the IoT workflow composition and fault-tolerant system architecture and discusses the system execution workflow.

System Architecture

This system is implemented as a web application that provides a user interface for users to explore and compose their IoT components and execute the fault-tolerant IoT workflow applications. Execution is initiated by merely inputting an abstract workflow graph, QoS requirements, and desired budget information. An

abstract workflow graph represents a DAG including abstract level workflow activities specified by Li *et al.*[1]. All the complex composition, deployment, and recovery procedures are hidden to ease the interaction. The layered schematic architecture of our proposed system is shown in Figure 2. There are four main layers of our proposed system, as explained in the following sections.

Workflow Layer

The workflow layer is employed in managing IoT workflows. It consists of four main components, as discussed below:

I. *Pipeline Module:* This component is involved in acquiring user input from the User Interface and creating a DAG-based IoT workflow. It has two modules workflow abstraction and QoS parameter comparison.
II. *Offline Optimizer:* Since there are many possible solutions for the deployment of each data analysis task,[4] it is necessary to find an optimal solution that satisfies all the nonfunctional QoS requirements in the defined budget. However, finding an optimal solution for only one data analysis task can be proven to be NP-hard.[5] To solve this problem, we propose a heuristic model divided into two segments AHP-based ranking and budget-based ranking.
III. *Database:* This is the most important component as it stores not only the basic workflow patterns and QoS configurations, but it also contains a knowledge base, which acts as a knowledge source for offline optimizer and composer. Knowledge base contains the predefined ranking and configuration knowledge representation (CKR) for different infrastructure resources. CKR is also computed for all the available resources using a knowledge management system IoT-CANE.[6]
IV. *Composer:* For each data analysis task, the composer takes the optimized infrastructure component from offline optimizer and queries the database for the desired CKR. Finally, it combines all the configurations and returns to the user a unified format file, which can be easily used for the deployment purpose. The details of this workflow layer are discussed in IoTWC.[1]

Infrastructure Layer

The infrastructure layer is designed to manage the setup of infrastructures used in IoT applications. It consists of an auto setup module, which acquires composed workflow DAG with configuration information from the Workflow Layer. It manages infrastructure setup procedures using four components.

I. *Workflow Interpreter:* When a unified format IoT workflow file is composed and generated from composer, the interpreter can read and understand the infrastructures required to deploy the particular IoT application. After interpreting, a list data structure containing each edge/cloud CKR is constructed for further deployment.
II. *Workflow Allocator:* It is mainly functioning to allocate and orchestrate the workflow activities and ensure the workflow sequence's success.
III. *Cloud Infrastructure Launcher:* As cloud providers, such as AWS, Azure, and Google Cloud Provider offer different SDKs and APIs for developers to program and operate. A unified, centralized cloud infrastructure launcher becomes necessary for the cloud environment setup. This component provides solutions for virtual machine launching, Docker installation, communication establishment, etc. Previous CKR information is adopted to specify the cloud provider, VM type, deployment location, network properties, and other relevant requirements.
IV. *Infrastructure Manager:* After cloud infrastructures and fog/edge infrastructures are prepared, the Infrastructure Manager component offers CRUD (Create, Read, Update, Delete) operations for such infrastructures and attached containers. Because Docker is installed with REST API enabled on all launched infrastructures, the container management becomes comfortable with simple constructed RESTful requests. Meanwhile, releasing the failed and unoccupied infrastructures is possible with simple commands from the Infrastructure Manager.

Deployment Layer

The deployment layer is based on master–worker architecture. The master orchestrates the workflow and is in the fog, whereas workers are geodistributed over edge cloud infrastructure. Greed nominator heuristic (GNH)[2] decides where to deploy the functions and utilizes Parsl to control the dataflow over the infrastructure. The deployment layer includes the following parts:

I. *Parsl:* It controls the computation of geodistributed resources including managing connection and provisioning virtual resources to deploy DAG applications. Data flow kernel (DFK) of

Parsl handles error and steers dataflow between computing nodes.

II. *Master–Worker Control Pattern:* The master node handles the task allocation, monitors resources (worker nodes), tracks failure, and report intolerable failure. The worker nodes provide their resources to compute and pass the result to the master node or message broker depending on the user specification. The Master node also has a DFK and an online optimizer to orchestrate the deployment.
III. *Online Optimizer:* A Greedy Nominator Heuristic (GNH) optimizer aims to i) lower the end-to-end latency, ii) leverage redundancy to operate during failure events, iii) increase the number of replicas to the critical function, and iv) balance redundancy and deployment cost. The greedy algorithm calculates the maximum possible replicas for each function using MaxReplicas [see (1)] function, where i is the virtual function sequence in the application. The application size is represented by n, and m is a constant set by the user to adjust the redundancy number

$$MaxReplicas = 1 + \left(1 - \frac{i}{n+1}\right)m. \quad (1)$$

The GNH is scalable with the increasing number of nodes in the infrastructure. It searches in two phases i) Nomination phase and ii) Announcement phase. At the Nomination phase workers are divided between nominators, then each nominator provides partial decision. The announcement phase decides the final redundant deployment out of the nominators' output. Redundant deployment is funnel shaped, where earlier functions will have higher replicas.

Recovery Layer

The recovery layer is the layer that manages the backup infrastructure and faulty nodes through the Failure handle module. It receives a failure report from the Deployment layer then supports the central infrastructure with a preconfigured cloud instance to form a temporal infrastructure.

I. *Backup infrastructure:* It represents virtual machines in the cloud with all the package dependency and the required configuration to run the IoT application workflow. Each node in the primary infrastructure has a backup from Backup infrastructure with proper configurations to replace specific requirements.
II. *Failure handle module:* This is the component that controls the backup infrastructure and faulty nodes to pause and resume. Node failure means that the virtual instance is down, or its performance is deteriorating. Recovering a node is restoring the instance to its original configuration. This is achieved by decoupling the software from the hardware. However, in case of a hardware failure, e.g., a malfunction within the electronic circuits, the services are moved from the faulty node to another backup node until the physical node is repaired/replaced and is ready for task execution.

System Execution Workflow

The following section illustrates the main steps to execute IoT workflow composition and fault-tolerant system.

1) *Define the IoT Workflow:* At the beginning, the Workflow Layer is designed to compose a different type of IoT application. The application is defined in terms of data analysis and connection patterns using the User Interface. Users can drag and drop the abstract patterns and rename them based on their adequacy. The pipeline module converts the user input into the workflow sequence, which is then stored in the database.
2) *Define the QoS requirement comparison:* The Workflow Layer provides a two-way comparison scheme for all the QoS parameters. Users can enter a priority value in the box provided by User Interface, which is converted into a comparison matrix by the Pipeline Module. Finally, the comparison matrix is stored in the database for further infrastructure ranking.
3) *Optimized infrastructure generation:* The Offline Optimizer retrieves the application workflow, QoS, and infrastructure information from the database and finds the optimized infrastructure for each workflow component.
4) *Compose the workflow:* The composer retrieves the optimized infrastructure resource information from the offline optimizer and queries the knowledge base to get the CKR information for the respective infrastructure resource. Finally, it will compose the workflow.
5) *Infrastructure Setup: The* composed workflow information in JSON format can be transferred to workflow interpreter to interpret as a set of CKR information along with infrastructure

locations. It is then passed to workflow allocator for infrastructure orchestration. Cloud infrastructure launcher retrieves information from the workflow allocator and invokes API from cloud providers to launch necessary cloud resources with virtual programming environment installed. Next, an infrastructure pool containing primary and backup resources is prepared for further deployment.

6) *Workflow deployment:* When the online optimizer receives a success message from infrastructure layer, a decision of virtual functions deployment allocation is generated by Greedy Nominator Heuristic module. Next, Parsl is enabled as a distributed parallel deployment tool to manage network connections and resource provisioning. The DataFlow Kernel of Parsl handles errors and controls dataflow between computational resources.
7) *Failure Handling:* During workflow application execution, resource monitor monitors and detects infrastructure level failure, then passes such failure information to failure handler module. The centralized controller can pause the failure resources first, then migrate tasks from failed nodes to backup resources to enable availability. Next, a recovery mechanism is employed to recover failed resources, which then put them back to the work. Finally, successful deployment information is displayed on the user interface to indicate workflow deployment success.

EVALUATION

We use real-time surface water flooding data monitoring and management application (Flood-PREPARED[3]) to evaluate our proposed system. The following sections present the experiment setup, failure model along with experiment results analysis.

Experiment Setup

The application is executed over a cross-cloud edge environment. We utilize two cloud service provides: Amazon Web Service (AWS) and Google Cloud Platform (GCP). The infrastructure has three GPU nodes, one in the fog and two in the GCP cloud. In total, the environment has ten computing nodes, half of it is primary, and the other half is backup nodes.

The HIPIMS requires GPU to execute the CUDA program. Therefore, it can run only on GPU instance with NVIDIA Tesla P100. All other functions can execute on every computing nodes. The Controller and Kafka server are in the fog infrastructure. Meanwhile, the fog is in Cardiff, U.K. The fog infrastructure contains a commodity machine and Raspberry Pi 4B. The commodity machine has 6 cores CPU and 32 GB memory, and Raspberry Pi 4B has 4 cores and 4 GB memory. Raspberry Pi 4B is the controller and runs some virtual instance tasks inside it. The CPU instance in GCP is e2-medium with 2 vCPUs, and 4 GB memory, whereas the GPU nodes are n1-standard-64 with 64 vCPUs and 240 GB memory. The GCP's nodes are in Brussels, Belgium (europe-west1-b). The round-trip time (RTT) of 14B size Python's object from the fog to GCP is 99 ms ($\pm$24.8). On the other hand, AWS instances are of t2.micro type, which has 1 vCPUs, and 1 GB memory. The AWS zone is eu-west-2, which is located in London, U.K. The RTT between AWS and fog is 93 ms ($\pm$4.73).

Failure Model

This section describes a time-dependent failure probability model from which we simulate node failure.

There are two types of failure 1) application-level failure and 2) host-level failure. Application-level failure is the failure of IoT application caused by service function delay or not completed. Host-level failure occurs when the node is down or repeatedly freezes, causing service delay or even outage (which takes from 30 s up to 2 min to recover). In the failure model, it simulates host-level failure. Based on the Weibull distribution, we determined the probability of not completing a submitted task, i.e., deployed virtual function. The model parameters are as follows:

A. Start time is the timestamp when the node is started to deploy virtual functions, whether it is at the beginning or after a recovery session.
B. Current time is the count that started after the Start time represented by x, i.e., $x =$ current timestamp—start time.
C. Time-to-failure (λ) is the time where the services in a node go down.
D. Reliability variable (k) is the Weibull shape parameter, which determines how reliable the node is. In case when the failure rate is constant then $k = 1$, whereas $k < 1$ or $k > 1$ means failure rate changes over time

$$f(x; \lambda, k), \quad = \quad 1 - e^{-(x/\lambda)^k}.$$

A random choice based on the probability of failure, i.e., $f(x; \lambda, k)$, decides whether it met the deadline or not.

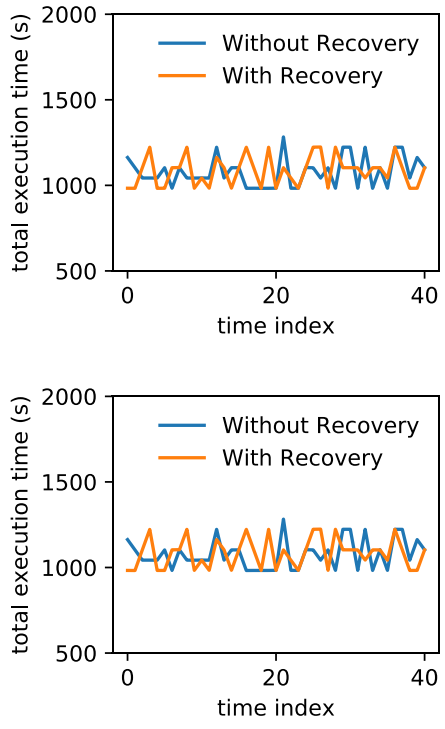

FIGURE 3. (a) Total execution time comparison. (b) Failure rate comparison.

Experiment Result

This section describes the experimental results based on the proposed failure model and the Flood-PREPARED workflow.

For the experiment parameters, we set a fixed λ (86 400 s) and vary k from 0.5 to 0.8 for different resources. The final results of total workflow execution time and failure rate comparison in terms of recovery algorithm utilizing are shown in Figure 3.

We can clearly see from Figure 3(a) that the total workflow execution time has a rare difference if the recovery algorithm is enabled. Our recovery algorithm is designed to minimize the backup resources' cost while maintaining the total workflow execution time. Meanwhile, the failure rate increase over time without the proposed recovery algorithm enabled. However, when the recovery algorithm is employed, the failure rate will drop after a time interval because the particular resource is in recovery processes.

Figure 3(b) shows that replicas of GNH guarantee task completed with no failure. Nevertheless, our recovery strategies reduce the cost of on-demand backup nodes by 95.87% (i.e., $141.18 per month) in total. Cost is reduced due to backups are shutdown unless they are needed.

CONCLUSION

This article has described a novel framework that is utilized to compose IoT workflow from a given DAG and QoS requirements. It automatically launches the cloud infrastructures with recommended configurations, finally allocate and deploy the desired virtual functions across edge and cloud environments. Meanwhile, the proposed system detects and recovers infrastructure level failures to enable continuous services with tolerable end-to-end latency. The proposed approach is designed for general composition and deployment purpose in the IoT environment. The recovery mechanism can be applied in various IoT application without parameter or argument changes. This system has been validated and evaluated with real-world surface water flooding data monitoring and management application. The results of experiments prove the efficiency and effectiveness of our system. In future, an execution feedback-based intelligent infrastructure recommendation approach can be employed to increase the configuration recommendation accuracy and decrease useless budget waste.

REFERENCES

1. Y. Li, D. N. Jha, G. S. Aujla, G. Morgan, A. Y. Zomaya, and R. Ranjan, "IOTWC: Analytic hierarchy process based Internet of Things workflow composition system," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2020, pp. 1–10.
2. O. Almurshed, "Greedy nominator heuristic (GNH): Virtual function placement using mapreduce," in *The Annual Research Student Poster exhibition, Cardiff School of Computer Science and Informatics*. Cardiff, U.K. : Cardiff Univ., 2020.
3. S. Barr *et al.*, "Flood-prepared: A nowcasting system for real-time impact adaption to surface water flooding in cities," *ISPRS Ann. Photogrammetry, Remote Sens. Spatial Inf. Sci.*, vol. 6, pp. 9–15, 2020.
4. D. N. Jha *et al.*, "Challenges in deployment and configuration management in cyber physical system," in *Handbook of Integration of Cloud Computing, Cyber Physical Systems and Internet of Things*. Berlin, Germany: Springer, 2020, pp. 215–235.
5. D. N. Jha, P. Michaľak, Z. Wen, R. Ranjan, and P. Watson, "Multiobjective deployment of data analysis operations in heterogeneous IoT infrastructure," *IEEE Trans. Ind. Informat.*, vol. 16, no. 11, pp. 7014–7024, Nov. 2020.
6. Y. Li *et al.*, "IoT-cane: A unified knowledge management system for data-centric Internet of Things application systems," *J. Parallel Distrib. Comput.*, vol. 131, pp. 161–172, 2019.

OSAMA ALMURSHED is currently working toward the Ph.D. degree with the Cardiff School of Computer Science & Informatics, Cardiff University, Cardiff, U.K. Contact him at almurshedo@cardiff.ac.uk.

OMER RANA is currently a professor of performance engineering with Cardiff University, Cardiff, U.K. Contact him at ranaof@cardiff.ac.uk

YINHAO LI is currently working toward the Ph.D. degree with the School of Computing, Newcastle University, Newcastle upon Tyne, U.K. Contact him at y.li119@ncl.ac.uk.

RAJIV RANJAN is a full professor of Computer Science with Newcastle University, Newcastle upon Tyne, U.K. Contact him at raj.ranjan@ncl.ac.uk.

DEVKI NANDAN JHA is a postdoctoral research associate with the University of Oxford, Oxford, U.K. Contact him at devki.jha@eng.ox.ac.uk.

PANKESH PATEL is a senior researcher with AI Institute, University of South Carolina, Columbia, South Carolina, USA. Contact him at ppankesh@mailbox.sc.edu.

PREM PRAKASH JAYARAMAN is an associate professor with the Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne, Australia. Contact him at pjayaraman@swin.edu.au.

SCHAHRAM DUSTDAR is a full professor of Computer Science heading the Research Division of Distributed Systems at TU Wien, Vienna, Austria. Contact him at dustdar@dsg.tuwien.ac.at.