Journal of Grid Computing (2021) 19: 47
https://doi.org/10.1007/s10723-021-09589-5

MiCADO-Edge: Towards an Application-level Orchestrator for the Cloud-to-Edge Computing Continuum

Amjad Ullah · Huseyin Dagdeviren · Resmi C. Ariyattu · James DesLauriers · Tamas Kiss · James Bowden

Received: 3 November 2020 / Accepted: 17 September 2021 / Published online: 2 November 2021

Abstract Automated deployment and run-time management of microservices-based applications in cloud computing environments is relatively well studied with several mature solutions. However, managing such applications and tasks in the cloud-to-edge continuum is far from trivial, with no robust, production-level solutions currently available. This paper presents our first attempt to extend an application-level cloud orchestration framework called MiCADO to utilise edge and fog worker nodes. The paper illustrates how MiCADO-Edge can automatically deploy complex sets of interconnected microservices in such multi-layered cloud-to-edge environments. Additionally, it shows how monitoring information can be collected from such services and how complex, user-defined run-time management policies can be enforced on application components running at any layer of the architecture. The implemented solution is demonstrated and evaluated using two realistic case studies from the areas of video processing and secure healthcare data analysis.

Keywords Application-level orchestration · Cloud-Fog-Edge ecosystems · Cloud-to-Edge continuum · IoT applications orchestration · Orchestration of microservices · Deployment and run-time management

A. Ullah (✉)
School of Computing, Edinburgh Napier University, Edinburgh, UK
e-mail: a.ullah@napier.ac.uk

H. Dagdeviren · R. C. Ariyattu · J. DesLauriers · T. Kiss
School of Computer Science and Engineering, University of Westminster, London, UK

H. Dagdeviren
e-mail: H.Dagdeviren@westminster.ac.uk

R. C. Ariyattu
e-mail: R.Ariyattu@westminster.ac.uk

J. DesLauriers
e-mail: J.Deslauriers@westminster.ac.uk

T. Kiss
e-mail: T.Kiss@westminster.ac.uk

J. Bowden
Hochschule für Technik und Wirtschaft, Berlin, Germany
e-mail: bowden@htw-berlin.de

1 Introduction

1.1 Background and Motivation

Cloud computing has immensely changed the provision of computing, both for personal and business users. Since its inception, adoption of cloud services has continued to grow and it is expected that worldwide public cloud service revenue will grow by 33 percent, from 266.4 billion dollars in 2020 to 354.6 billion dollars in 2022 [1]. This is not surprising considering the inherent characteristics of cloud computing that offer economic benefits as well as operational efficiencies to enterprises [2]. Given its benefits and

wide-spread adoption, cloud computing is regarded today as mainstream by many developers. Therefore, we find that new research and developments in IT are often 'cloud-enhanced' in the sense that they build on or extend the capabilities of the cloud platform (or vice versa) to deliver new solutions [1]. Recently, the focus of one such enhancement has been to bridge the gap between the cloud and devices located at the edge of the network [3–6]. The need for developing this capability has emerged from the proliferation of connected devices via the Internet, known as Internet of Things (IoT), which in turn has caused exponential growth in data that needs processing, storing and analysing. The number of connected IoT devices worldwide are expected to grow up to 36 billion by 2025 [7, 8] and they are expected to generate 79.4 ZB of data [9].

The introduction of IoT has fuelled a new breed of applications in various domains such as healthcare, manufacturing and transport, which are often referred to as IoT applications. These applications require IoT devices to capture and possibly process data from the environment. With IoT devices becoming more prevalent, the data they send to the cloud will continue to grow at a rapid rate. A traditional cloud computing architecture is impractical, if not inadequate, to run IoT applications due to its centralised approach, which results in the following two issues: i) the network and the response time slows, and ii) the cost for businesses increases due to network charges. These issues call for more data processing and computation to be done at or near the edge of the network with data only being sent to the cloud when required. This gives rise to two new architectural solutions called fog computing and edge computing, which aim to address the aforementioned issues.

The terms fog computing and edge computing are often used interchangeably to loosely refer to moving processing or computation away from the central cloud to nodes that are closer to endpoints at the network edge. Though they both aim to reduce the amount of data sent to the cloud in data-dense applications, there are subtle differences between the two. As shown in Fig. 1, fog computing is an intermediate layer between cloud and edge that represents the nodes between the cloud to the IoT sensors and actuators, possibly spanning across multiple layers of the network topology. In contrast, in edge computing, the nodes where the computation takes place are normally very close to the IoT devices in terms of network proximity, often only one or a few hops away from the IoT devices, or even embedded within the connected device [10].

One of the key research challenges in the cloud-to-edge continuum is resource orchestration. The term refers to the automation of the deployment and execution of an application in an efficient way while considering the available resources and various Quality of Service (QoS) requirements. It involves the coordination of computing tasks, usually in a unified workflow, and the automation of allocating appropriate resources to the tasks considering the defined QoS requirements to fulfil the required service-levels.

In recent years, a number of commercial and open-source cloud orchestration solutions have emerged that are considered to be mature and reliable. At the level of application containers, Kubernetes [11] and Docker Swarm [12] are two well-known and widely-used solutions for cloud orchestration. There are also a number of higher-level frameworks such as Apache Brooklyn [13], Cloudify [14], Cloudiator [15], Alien4Cloud [16], MODAClouds [17] and MiCADO [18] that support automated application deployment and run-time orchestration. Furthermore, major cloud providers also offer specific tools for orchestration, such as Amazon's AWS CloudFormation [19], OpenStack HEAT [20], Microsoft Azure's Resource Manager (ARM) templates [21], and Google's Deployment Manager [22]. These solutions improve resource utilisation and introduce a great deal of agility by making application development, deployment, execution and maintenance easier in cloud environments.

While cloud orchestration is a mature research area, orchestration in the cloud-to-edge computing continuum is less well studied [23]. Current cloud orchestration solutions have not been designed with edge and fog computing in mind, hence, there is a lack of solutions that can support the orchestration of applications in an extended cloud-to-edge environment. In such environments, applications are deployed in a more complex and heterogeneous infrastructure, where the computing nodes are not just located in a central cloud but are distributed across discrete networks that span multiple layers of a topology in the cloud-to-edge continuum. Therefore, there is a need to re-engineer the existing orchestration solutions in order to extend their capabilities towards the edge of the network. To the best of our knowledge, there is currently no production quality, robust, comprehen-

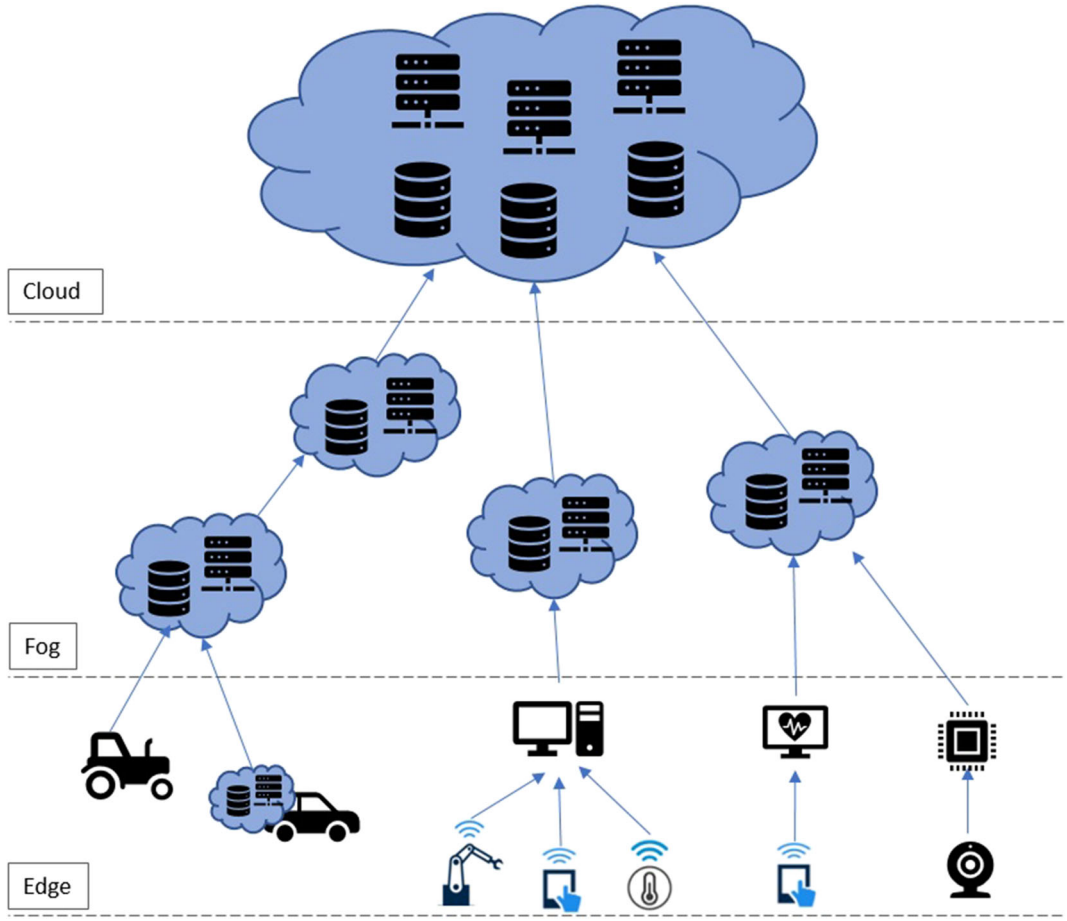


Fig. 1 A generic architecture for cloud-to-edge environments

sive and efficient solution available that supports the automated deployment and run-time management of applications spanning all layers from cloud, to fog and the edge. Our research aims to provide such a solution by extending an existing mature application-level cloud orchestration framework, called MiCADO, towards fog and edge devices.

1.2 Requirements and Contributions

Detailed requirements towards the proposed solution have been derived from real-life case studies implemented within the framework of two currently active research projects funded by the European Commission: DIGITbrain and ASCLEPIOS.

The DIGITbrain project [24] aims to extend the traditional digital twin concept towards the Digital Product Brain that steers the behaviour and performance of an industrial product (mechatronic system or manufacturing machine) by coalescing its physical and digital dimensions and by memorising the occurred (physical and digital) events throughout its entire life-cycle. Twenty one application experiments are implemented within the project, involving over sixty manufacturing and technology companies (independent software vendors and consultancy companies) from Europe. A common feature of these applications is that large amounts of data are collected using various sensors on factory floor, processed locally by edge and fog nodes, and if necessary parts of this data are transferred to the cloud for further computation (e.g. to run computation intensive machine learning algorithms).

The vision of the ASCLEPIOS (Advanced Secure Cloud Encrypted Platform for Internationally Orchestrated Solutions in Healthcare) project [25] is to

maximise and fortify the trust of users in cloud-based healthcare services by exploiting modern cryptographic approaches to build a cloud-based eHealth framework that protects users' privacy and prevents both internal and external attacks. ASCLEPIOS demonstrates the applicability of the developed solution on healthcare applications provided by three European hospitals, with the intention of deploying these applications alongside the ASCLEPIOS framework in a multi-cloud testbed. The ASCLEPIOS cloud testbed features a mix of private (at the University of Westminster and at the Norwegian Centre for E-Health Research) and public (Amazon Web Services and Microsoft Azure) clouds. Facilitating portability and multi-cloud support plays an important role in the development of both the ASCLEPIOS framework and the healthcare applications it is designed to support. Additionally, deploying and managing applications that collect and process data using edge devices, such as smart watches and contactless sensors for monitoring sleep patterns in outpatient settings, will be considered in the future.

During the requirements collection phase of these projects, the following requirements, highlighting a number of features, have emerged for an orchestrator intended for the cloud-to-edge computing continuum:

R1. The orchestrator should be able to **automatically deploy applications** that are composed of **multiple interconnected microservices** on resources that are **spanning cloud, edge and fog nodes**. The deployment should be automated in the sense that a **single deployment descriptor** should be sufficient to execute a complex application. This requirement is common in both projects with several application scenarios in manufacturing and healthcare demanding such a feature. DIGITbrain especially aims to support its end users with the capability of dynamically composing digital twins from data, model and algorithm tuples and deploying such applications on demand.

R2. The orchestrator should be **agnostic to technological solutions** implementing the cloud, edge and fog layers. Therefore, it should not tie in with one particular cloud, edge or fog technology and should support easy portability of applications from one technology provider to another. This requirement is also common in both projects. Application developers and providers do not want to be locked in to one particular cloud technology or edge device, and require easy **portability** of their solution in the future. This portability may be required, for example, in case of new features emerging in one cloud/edge technology that is not supported by others, or because of specific security requirements that can only be fulfilled by private cloud solutions.

R3. The orchestrator should support **multi-cloud applications** where microservices of the same application can be deployed on resources operated by various cloud service providers. This requirement emerged in ASCLEPIOS where certain features, for example supporting specific implementations of security solutions, were not available in the cloud where the core part of the application was running.

R4. The orchestrator should support the **runtime management and optimisation** of the deployed applications. Application developers/owners should be able to define a **set of policies that describe and govern the desired behaviour** of the deployed application, and the orchestrator should be able to execute changes in the application's setup/deployment based on such dynamically changing requirements. For example, the orchestrator should be able to scale up computing nodes in all (cloud, edge, fog) layers of the architecture, or migrate computation between nodes and layers in order to facilitate more efficient application execution. This requirement comes from DIGITbrain where complex simulation and machine learning applications are executed in a multi-tiered infrastructure, and where the initial resource requirements of such applications may be unknown or undecided.

R5. **Types of policies** supported by the orchestrator should be as **flexible and extendable** as possible. In other words, the policies should not be hard-coded and application developers/owners should have the means of **implementing custom policies on-demand**. Therefore, the solution should support automated scaling and reconfiguration of applications using **dynamically defined policies**. This requirement is also primarily emerged in DIGITbrain where, due to

the complex nature of the targeted applications, a wide variety of policies may be required. An example of such complex policies include simulation experimentation where a large number of simulation runs need to be completed by a certain deadline, and where resources need to be dynamically allocated to meet this deadline while minimising costs.

R6. Both deployment and scaling should **support containerised applications**. However, **support for virtual machines (VM)** is also a strong requirement as with many application scenarios, the containers are deployed in VMs. Additionally, some applications (e.g. complex Windows-based applications) may not be suitable for containerisation at all and require deployment and run-time management in **VM only environments**. Finally, some edge or fog devices may not support virtual machines, but are able to **execute containers directly** on the hardware. Both projects are dealing with a large variety of applications, and as a consequence, supporting both containers and virtual machines is a strong requirement, coming from both projects.

R7. The resulting solution should be **robust and production quality**. Especially, **security and reliability** of the orchestrator are crucial features, demanded by the real-life application scenarios in both healthcare (ASCLEPIOS) and manufacturing (DIGITbrain).

Based on the requirements above, in this paper, we present a framework, called MiCADO-Edge, that facilitates the application deployment and run-time orchestration in the cloud-to-edge continuum. More specifically, our contributions include the following:

1. Evaluate a wide range of existing cloud-to-edge orchestration solutions against a well-defined set of requirements collected from realistic IoT applications.
2. Provide a novel orchestration framework that supports the deployment and run-time management of IoT applications in heterogeneous computational environments which can be comprised of different cloud, fog and edge resources.
3. Provide a high level abstraction layer, using a standardised description language called TOSCA (Topology and Orchestration Specification for Cloud Applications [26]), to support application developers with describing their application topology and the required computational resources independently from cloud vendors.
4. Provide an automated method for the seamless integration of non-cloud resources into the centralised cluster and further facilitate the description of those resources via the TOSCA-based abstraction layer, giving application developers a unified interface for describing the overall IoT application.
5. Provide a policy-based method to automate the scaling and reconfiguration of applications in the cloud-to-edge continuum by enabling application developers to define high-level dynamic scaling policies through the TOSCA interface.
6. To demonstrate the applicability of our proposed solution, the implementation of the following two realistic application scenarios using MiCADO-Edge are presented: (i) a real-time face detection application is deployed and scaled on cloud, fog and edge nodes, and (ii) a healthcare application requiring specific resources from multiple different clouds is deployed.

The rest of this paper is organised as follows. Section 2 discusses the MiCADO framework upon which our solution and contribution is based. Section 3 details the technical work undertaken to extend MiCADO. Section 4 presents the results achieved via the application scenarios. Section 5 looks at related work in the orchestration sphere related to the cloud-to-edge ecosystem. Finally, Section 6 describes conclusions and future work.

2 MiCADO Framework

MiCADO [18] is an application-level cloud agnostic orchestration and auto-scaling framework that was developed in the European COLA (Cloud Orchestration at the Level of Application) Project [28]. A number of cloud service providers and middlewares are supported by MiCADO, including both commercial clouds such as Amazon AWS, Microsoft Azure, Google Cloud Platform, or Oracle Cloud Services, as well as private cloud systems based on OpenStack or OpenNebula. MiCADO is fully open source and implements a microservices architecture in a modular way. The modular design [29] supports varied imple-

mentations where any of the components can easily be replaced with a different realisation of the same functionality. The concept of MiCADO is described in detail in [30]. In this section only a high-level overview of the framework is provided to explain its architecture, building blocks and modular implementation.

The cloud agnostic design of MiCADO is based on two major principles. First is the need for a generic orchestration framework providing support for launching and managing applications in various clouds. Therefore, the framework is tied to no specific cloud service provider and supports a mix of public, private and community clouds. It also provides flexibility at the application level, regardless of the underlying cloud. This includes automated deployment and optimised run-time orchestration with features such as automated scaling and enhanced security. Second, a single generic interface to this framework is required. The interface acts as an abstraction layer over the various underlying components of the framework and describes the application, its cloud resources and any policies which govern performance, cost, security or other non-functional application requirements.

The high-level architecture of MiCADO is presented in Fig. 2. The input to MiCADO is a TOSCA-based Application Description Template (ADT) [27] defining the application topology (containers, virtual machines and their interconnection) and the various policies [31] (e.g. scaling and security policies) that govern the full life-cycle of the application. MiCADO consists of two main logical components: Master Node and Worker Node(s). The Submitter component on the MiCADO Master receives and interprets the ADT as input. Based on this input, the Cloud Orchestrator creates the necessary virtual machines in the cloud as MiCADO Worker Nodes and the Container Orchestrator deploys the application's microservices in Docker containers on these nodes. After deployment, the MiCADO Monitoring System monitors the execution of the application and the Policy Keeper performs scaling decisions based on the monitoring data and the user-defined scaling policies. Optimiser is a background microservice performing long-running calculations on demand for finding the optimal setup of both cloud resources and container infrastructures.

Currently there are various implementations of MiCADO based on its modular architecture, which enables changing and replacing its components with different tools and services. As Cloud Orchestrator, the latest implementation of MiCADO can utilise either Occopus [32] or Terraform [33]. These both are capable of launching virtual machines on various private or public cloud infrastructures. However, as the clouds supported by these two orchestrators differ, MiCADO can support a wider variety of targeted resources. For Container Orchestration, MiCADO uses Kubernetes [11]. The monitoring component is based on Prometheus [35], a lightweight, low resource consuming, but powerful monitoring tool. The MiCADO Submitter, Policy Keeper [36] and Optimiser components were custom implemented for MiCADO during the COLA project.

In its original design and implementation, MiCADO was intended to orchestrate applications on cloud resources only. However, due to its modular structure, MiCADO is a good candidate to be extended towards the cloud-to-edge continuum. Such an extension, which completely changes the scope of orchestration and enables automated deployment and run-time application management utilising fog and edge nodes, is explored in this paper.

3 Multi-Level Orchestration for Cloud-To-Edge Ecosystem: MiCADO-Edge

This section introduces the key contributions of our research work including the necessary enhancements to the MiCADO framework that extend its capabilities to perform orchestration and the run-time management of IoT applications in the cloud-fog-edge ecosystem. The following paragraphs discuss the architecture of our proposed extension and provide details of the enhancements.

3.1 Architecture

Figure 3 depicts the proposed high level architecture of MiCADO-Edge, the extended version of MiCADO that supports multi-level orchestration and run-time management for the cloud-to-edge ecosystem. MiCADO-Edge does not differentiate between fog and edge layers, but rather generalises these layers as non-cloud layer, which represents both edge and/or fog workers. From MiCADO's perspective, both edge and fog workers are implemented and connected to the MiCADO Master node in a uniform way and hence-

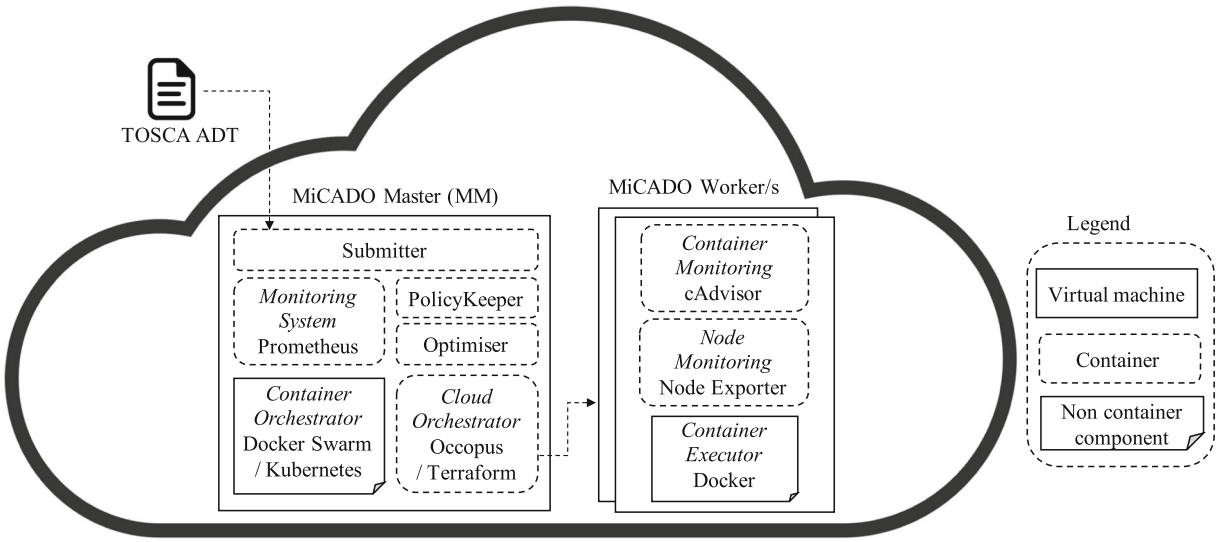


Fig. 2 High-level architecture of MiCADO

forth these workers will be referred to as non-cloud workers.

The extended architecture accepts the application description in the form of an ADT, just as in the original implementation of MiCADO. The MiCADO Master node, having received the ADT, is responsible for deploying the application on cloud and non-cloud workers based on the application topology described in the ADT. MiCADO uses Kubernetes for the automatic deployment, scaling, and management of

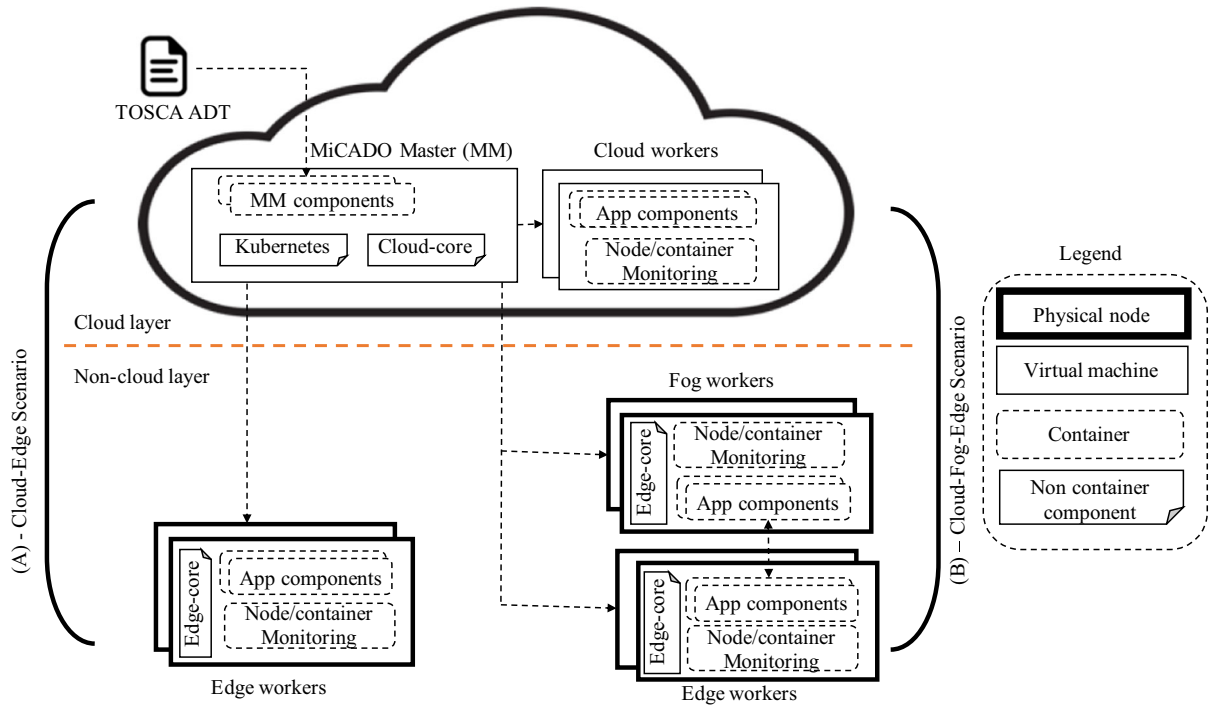


Fig. 3 High level architecture of the proposed solution

application containers across a cluster of cloud nodes. Performing such activities on non-cloud workers (in combination with the cloud) requires a mechanism that extends the current Kubernetes cluster towards non-cloud workers. There are several open-source initiatives that aim to provide such an extension. One of those solutions is KubeEdge, which was selected for our implementation based on its current relative maturity. However, due to the modular architecture applied, KubeEdge can be easily replaced with other solutions providing the same functionality, if required in the future. KubeEdge is built upon Kubernetes and facilitates the extension of the orchestration capabilities of Kubernetes to host containerised applications on non-cloud workers. The main motivations for selecting KubeEdge for our implementation include the following:

- It extends a Kubernetes cluster towards non-cloud workers.
- It enables the seamless and automatic configuration of non-cloud workers when joining the central Kubernetes cluster.
- It is an open source solution with a significant community behind it , which is in-line with the generic modular open source nature of the MiCADO framework.
- It provides support for heterogeneous edge devices such as the Raspberry Pi, and also for fog nodes such as a laptop or a PC outside a central cloud.

KubeEdge consists of two main components, Cloud-core and Edge-core. These components, as illustrated in Fig. 3, are integrated into our architecture. The Cloud-core runs on the Kubernetes master node (in the cloud) that manages the entire Kubernetes cluster. Edge-core, on the other hand, runs separately on each individual non-cloud worker. These KubeEdge components facilitate the underlying infrastructure support for network, application deployment and synchronisation of metadata between cloud and non-cloud workers.

In addition to deployment, MiCADO is also responsible for managing the application's life cycle, including any run-time reconfiguration, if required. Such reconfigurations are based on user defined policies provided in the ADT and/or on any manual update from users (application owners) during the application's life cycle. For such run-time management of applications, the MiCADO Master constantly requires updated monitoring information from the worker nodes. MiCADO supports Prometheus based metrics collection. For this purpose, MiCADO deploys monitoring services including *cAdvisor* and *Node Exporter* on each worker. Furthermore, if required, any other application specific Prometheus-based monitoring service (often referred to as a Prometheus exporter) can also be utilised. The *Node/container monitoring* label in Fig. 3 represents such components. The deployed monitoring services on each worker node become targets for the Prometheus server. Prometheus regularly pulls the updated information from all workers, which is then processed by the Policy Keeper to make reconfiguration decisions. Utilising KubeEdge, these monitoring and run-time management capabilities of MiCADO can also be extended to the non-cloud layers of the architecture.

The implementation of the proposed extended architecture involved the following three key challenges:

1. The automation of the seamless integration of non-cloud resources to the centralised cluster.
2. The definition of non-cloud workers and the description of application components in the ADT.
3. The automated deployment and run-time management of application microservices on non-cloud workers.

The following sections discuss each of these challenges and the way they have been addressed.

3.2 Integration of Non-Cloud Resources to the Centralised Cluster

IoT applications typically need computational resources from both cloud and fog/edge environments. Therefore, for the deployment of such applications, the orchestrator needs access to the hybrid - cloud and non-cloud - set of resources. The cloud resources are usually created by the orchestrator at run-time. However, the non-cloud resources are physical nodes that are already available. Before deploying the IoT application, these resources must join the set of resources created by the orchestrator. In MiCADO-Edge, the integration of these resources to the centralised cluster are handled through KubeEdge. This integration involves the completion of the following tasks:

- The automatic setup and execution of the Cloud-core part on the MiCADO Master node, involving configuration of its Kubernetes cluster.
- The automatic setup and joining of non-cloud workers to the Kubernetes cluster running on the MiCADO Master node.

The aforementioned tasks are achieved during the setup of the MiCADO Master node. MiCADO uses Ansible Playbooks [37] to prepare and configure the Master node. Ansible Playbooks facilitate configuration management on multiple machines. During the MiCADO Master installation process, all non-cloud workers are specified using the Ansible Playbook inventory file. The code snippet in Fig. 4 shows an example of such a file. The *micado-target*, in this example, represents the machine of the MiCADO Master node, whereas *edge-device-A* and *edge-device-B* are the two edge (non-cloud) nodes. The installation process will first set up the MiCADO Master, followed by the installation and configuration of the Cloud-core part. Finally, the non-cloud workers are configured and automatically join the Kubernetes cluster. The non-cloud workers must be provided with unique names, which will be referenced during the authoring of the ADT in the application deployment phase to indicate hosts for application components. The complete scripts containing the Ansible tasks for setting up the Cloud-core and the Edge-core components are available in the MiCADO GitHub repository [38].

```
all:
  hosts:
    micado-target:
      ansible_host: ip_address_of_micado_master_vm
      ansible_connection: ssh
      ansible_user: ubuntu
      ansible_become: True
      ansible_become_method: sudo
      ansible_python_interpreter: /usr/bin/python3
  children:
    edgenodes:
      hosts:
        edge-device-A:
          ansible_host: ip_address_of_edge_device_1
          ansible_connection: ssh
          ansible_user: ubuntu
          ansible_become: True
          ansible_become_method: sudo
          ansible_python_interpreter: /usr/bin/python3
        edge-device-B:
          ansible_host: ip_address_of_edge_device_1
          ansible_connection: ssh
          ansible_user: ubuntu
          ansible_become: True
          ansible_become_method: sudo
          ansible_python_interpreter: /usr/bin/python3
```

Fig. 4 Example of MiCADO installation hosts/inventory file

3.3 Description of Non-Cloud Nodes and Application Components - TOSCA ADT Enhancements

TOSCA is an OASIS [39] standard for describing complex application topologies in the cloud [26]. A standard TOSCA template in YAML defines the various components of a cloud application (software, storage, networks, virtual machines) as *nodes*, which may have *requirements* for, or share *relationships* with, other nodes in the template. TOSCA also supports *policies* for defining rules for scalability, monitoring, placement or security that will govern application behaviour at run-time. MiCADO has adopted TOSCA-based ADTs as its interface for defining the application containers, cloud compute resources, and scaling and security policies that describe a complete application.

At the time of writing, the current version of the TOSCA standard (v1.3) has yet to formally include representations of fog, edge or IoT devices within the specification. To extend MiCADO to support orchestration at the edge, a new type – *tosca.nodes.MiCADO.Edge* – has been introduced. Defined nodes of this type support relationships with the other nodes in the ADT – for example, an application container may have a *HostedOn* relationship with an edge (non-cloud) node, which MiCADO can enforce at run-time to ensure that a given application component runs only on a specific edge device. Nodes of the *Edge* type have a single optionally-defined property: *public_ip*. If the public IP of an edge device (non-cloud node) is defined, MiCADO can automatically configure ingress to certain application containers running on that edge device. This specific feature of the ADT is currently used to automate monitoring of Prometheus metrics on non-cloud nodes, which would otherwise require manual configuration from the user. The code snippet in Fig. 5 provides an example of ADT. The first few lines define the necessary metadata for the template. Then, user-defined inputs are described, followed by the nodes that make up the application – in this case an edge device, and an application container with a requirement for the edge device as its host.

In the context of the cloud-to-edge ecosystem, in addition to the definition of a new edge type, it may

```
tosca_definitions_version: tosca_simple_yaml_1_3
imports: [ micado_types.yaml ]
description: Sample ADT for edge orchestration
topology_template:
  inputs:
    remote_ports:
      type: list
      default: [4095, 4096, 4097, 4098]
  node_templates:
    edge-device:
      type: tosca.nodes.MiCADO.Edge
      properties:
        public_ip: 18.130.25.25
    application-container:
      type: tosca.nodes.MiCADO.Container.Application.Docker
      occurrences: [ 1 , 5 ]
      instance_count: 4
      properties:
        ports:
        - target: { get_input : [ remote_ports, INDEX ] }
      requirements:
      - host: edge-device
```

Fig. 5 Sample MiCADO ADT for orchestration of a container on an edge device

also be required to define multiple instances of the same TOSCA node, where each instance may require a different set of values. An example of this could be a streaming client application that needs to be deployed on multiple edge devices, with each edge device sending the streamed data to a different receiver. Although defining such an application topology in an ADT could be achieved by manually copying a node and modifying the single required property value, it resulted in repetition and made it difficult to introduce policies on a particular node type (since each copy became a different node type). Therefore, to simplify the description of such applications, the support to dynamically assign values to various parameters of the same TOSCA node from a configurable range of values was required.

The latest standard of the TOSCA specification, i.e. v1.3, introduced such a feature. According to this, multiple instances of a TOSCA node can be defined, whose properties could be dynamically assigned values from a user-defined array. However, currently the OpenStack TOSCAParser [40], i.e. the parser used by MiCADO, only supports TOSCA versions 1.0 - 1.2. Hence, to support the aforementioned feature, an additional wrapper was implemented in MiCADO. This wrapper validates the ADT and deals with this unsupported feature of TOSCA. An example of applying this feature can be seen in Fig. 5. The feature is supported by the following TOSCA concepts within the definition of a node:

- **inputs**: User-defined inputs that can receive assigned values at deployment time,
- **occurrences**: The lower and upper bounds of the number of instances of the defined node,
- **instance-count**: The desired number of instances of the defined node,
- **get-input** by INDEX: Reference to the user-defined array to pick and assign values from.

3.4 Automated deployment and run-time management of microservices on non-cloud nodes

Sections 3.2 and 3.3 described how to integrate non-cloud workers to the centralised cluster and how to describe the extended topology in an ADT. These are all necessary steps to set up the infrastructure. This section focuses on the process of automated deployment, monitoring and run-time management of microservices running on edge nodes.

3.4.1 Deployment of Microservices on Edge Nodes

Figure 6 presents an integrated view of the necessary parts of MiCADO and their interactions with the KubeEdge architecture [41]. The aim of Fig. 6 is to explain how MiCADO-Edge, with the help of KubeEdge, achieves the deployment and run-time management of the application's microservices on non-cloud nodes. The MiCADO Submitter translates the provided ADT. If the application requires cloud workers (virtual machines), the Submitter instructs the Cloud Orchestrator to dynamically create those worker nodes on the specified cloud. Once created, the worker nodes automatically join the MiCADO Kubernetes cluster. On the other hand, the non-cloud workers are already part of the cluster, set up beforehand by the Ansible Playbook according to the process explained in Section 3.2. Once the infrastructure is ready, the Submitter instructs Kubernetes to deploy the application's microservices. The core Kubernetes handles scheduling of microservices across cloud workers, while the *EdgeController* inside Cloud-core takes care of microservices scheduling on non-cloud workers.

The *EdgeController* is an extended Kubernetes controller that is responsible for managing the edge nodes connected to the cluster, as well as their pods' metadata. Any change on the Kubernetes side is recognised by *EdgeController*, which directs it to the

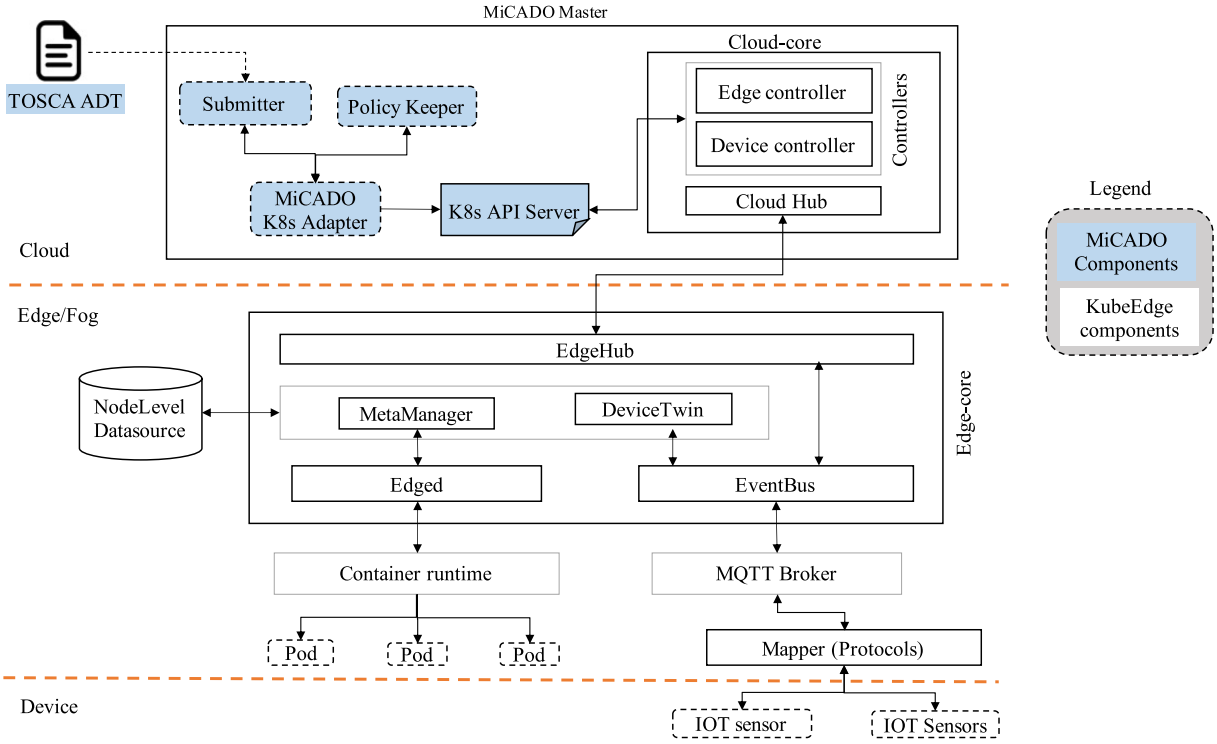


Fig. 6 Detailed architecture of KubeEdge in interaction with the relevant components of MiCADO

specific edge nodes. These changes are performed by the *CloudHub* module, which is a web socket server responsible for facilitating necessary caching, and sending/receiving messages to/from the *EdgeHub* module of Edge-core running on the non-cloud nodes.

EdgeHub is the corresponding web socket client module of the Edge-core that is responsible for communicating with the *CloudHub* module for syncing updates across cloud and edge. Any pod-related updates from the cloud side are passed to the *MetaManager*, while device-related updates are passed to the *EventBus* module. At the current stage of the integration, MiCADO-Edge only extends the deployment and run-time management of pods to the edge. Therefore, any details of device-related activities are outside the scope of this paper. The *MetaManager* module of Edge-core is responsible for two key activities: (i) it stores/retrieves pod metadata in/from a SQLite database running on the non-cloud node, and (ii) it mediates messages between *EdgeHub* and the *Edged* module, which is responsible for the management of pods using a container run-time (e.g. Docker [34]). Changes initiated on non-cloud nodes automatically trigger reactions on the cloud side, using the aforementioned process in reverse order.

3.4.2 Collecting Monitoring Information from Non-Cloud Nodes

MiCADO uses the Prometheus monitoring system to collect metrics from cloud workers. Monitoring in MiCADO can be enabled or disabled from the ADT based on the requirements of an application. If monitoring is enabled, the *cAdvisor* and *Node exporter* services are deployed on each worker. These services enable the Prometheus server to automatically gather monitoring information from the target worker nodes at regular (but configurable) intervals. Similar to the cloud worker nodes, this functionality of collecting monitoring information has been extended to the

non-cloud nodes. However, the procedure for collecting monitoring information from cloud and non-cloud workers is different.

In the case of cloud workers, targets are registered in Prometheus using a Pod IP that is automatically assigned to each container by the container networking interface (CNI) in Kubernetes. These Pod IPs are allocated from a single subnet for all cloud workers, so monitoring services deployed to cloud workers are directly accessible via their respective Pod IPs. KubeEdge, on the other hand, does not support CNI, so containers deployed to non-cloud workers are not assigned a Pod IP in the same subnet as containers on cloud workers. Therefore, containers across cloud and non-cloud nodes cannot communicate over these IPs, so monitoring services cannot be registered with Prometheus in the same way as above. To resolve this problem the following two options are considered:

1. Non-cloud nodes push the generated metrics to the MiCADO Master node at regular intervals.
2. The metrics are gathered by the Prometheus server running on the MiCADO Master node, as is done in the case of the cloud workers (but using the public IPs of the non-cloud nodes).

MiCADO-Edge adopted the second method due to its similarity with the existing MiCADO solution. Furthermore, its implementation required only minimal changes, which involved the registration of the public IPs of non-cloud nodes and the target ports used by the monitoring services, with the Prometheus server. The only drawback to this approach is that read-only monitoring information is now accessible to any actor with the public IPs and the specific ports of the monitoring services. However, this problem can be solved by establishing mutual client authentication between the Prometheus server and the monitoring services deployed in non-cloud nodes, which will restrict access to monitoring information to the Prometheus server only. Such a feature will be added to MiCADO-Edge as future work, unless KubeEdge addresses the extension of CNI to edge nodes, which will automatically resolve this issue. The code snippet in Fig. 7 presents an example of using a public IP (x.y.138.187) that is statically provided as the target source to the Prometheus server in order to collect monitoring information.

3.4.3 Run-time Management

In addition to automated deployment, MiCADO also supports the run-time management of applications. This includes complying with the policies defined in the ADT, and modifying/re-configuring the application's microservices accordingly. Policies in MiCADO support automated scaling of applications (e.g. allocating more or fewer containers and/or virtual machines to run the application's microservices), or dynamically re-configuring security settings. Additionally, MiCADO also handles re-deployment in the case of any failure, e.g. when a particular service (or worker node) crashes. All these features have been extended to the cloud-to-edge ecosystem and implemented for non-cloud workers. The major challenge in this case was to enforce configuration changes in non-cloud nodes based on user-defined policies.

In the MiCADO framework, the Policy Keeper component is responsible for the execution and enforcement of application policies. Although policy enforcement has been extended to non-cloud worker nodes, the implementation of the Policy Keeper itself did not require any modification. This is due to the generic and modular nature of the MiCADO framework. Once a worker node (either cloud or non-cloud worker) joins the cluster and containers are deployed on it, the Policy Keeper does not differentiate between the different types of workers and the containers running on it. At run-time, the Policy Keeper periodically evaluates the policies defined in the ADT and triggers the required changes in the Kubernetes API Server using the MiCADO Kubernetes Adapter, as it is shown in Fig. 6. If changes are required in edge nodes, these can be propagated by the main *CloudHub* module to the *EdgedHub* module running on the specific non-cloud nodes, similar to how it was described in Section 3.4.1. A further demonstration of MiCADO's run-time management capabilities in the case of non-cloud worker nodes will be provided in Section 4.1.

A specific challenge to address in the case of non-cloud workers is that of handling their volatile nature, meaning that such workers can be unreliable and may lose connectivity with the central cloud cluster. In such scenarios, the remote node has the ability to automatically rejoin the cluster, once connectivity is reestablished. This automatic rejoining to the cluster is an inherent characteristic of the KubeEdge

```
policies:
    - scalability:
        type: tosca.policies.Scaling.MiCADO.Container.Network.target-node
        targets: [ target-node ]
        properties:
          sources:
          - x.y.138.187:8080
          - x.y.138.187:9100
          constants:
            SERVICE_NAME: 'fd-edge-processor'
            COOLDOWNPERIOD: 180
          min_instances: 1
          max_instances: 5
```

Fig. 7 An example for the use of Edge node public IP as a target source for collecting monitoring information

architecture, as its components (from cloud-to-edge) constantly synchronise information with each other. As a result, at any point in time, if a non-cloud node becomes unavailable, the central cloud cluster automatically removes it from the set of available worker nodes. On the edge side, the Edge-core, responsible for the management of application containers on the non-cloud node, does not interrupt the running components. However, once the connection is restored, the non-cloud node automatically joins the central cluster and synchronises information with the cloud side.

4 Case Studies and Results

This section demonstrates the proposed MiCADO-Edge framework using two realistic case studies. The following subsections briefly explain these case studies and provide details of how MiCADO-Edge can be utilised to deploy and manage these applications. These case-studies represent two widely different scenarios.

The first case study illustrates how a realistic application consisting of a number of microservices can be automatically deployed in the targeted cloud-to-edge ecosystem and how this application can be managed and scaled by MiCADO at run-time. This case study demonstrates evidence that MiCADO-Edge fulfills requirements R1 (microservices are deployed automatically in the cloud-to-edge continuum), R4 (scaling policies are used to increase the number of processing nodes in case the load increases), R5 (scaling policies are defined by the application developer at the time of deployment), and R6 (containers and virtual machines are both used for the deployment and run-time management of the application).

The second case study, a demonstrator application for the H2020 ASCLEPIOS [25] project, shows how real multi-cloud scenarios, where different microservices of the same application run simultaneously in different clouds and administrative domains, can also be achieved. While this scenario does not include actual edge devices, it demonstrates the multi-cloud capabilities of MiCADO-Edge for microservices of the same application, a feature that was not available in MiCADO. This case study provides evidence for fulfilling requirements R1 (the application's microservices are deployed automatically), R2 (various cloud service providers and middleware are utilised by the application), R3 (microservices of the application are running in different clouds), and R6 (the deployment utilises both containers and VMs).

These two case studies demonstrate that the requirements R1 to R6 have been fulfilled. However, neither of the case studies dealt with R7. The requirement R7 mostly focuses on the security aspects. In general, the MiCADO framework facilitates various security enablers that guarantee the security of the system against different attack scenarios within the cloud environment. However, the extension of security enablers towards edge nodes are not yet explored, but will be considered in future work.

4.1 Real-time Face Detection

The Real-time Face Detection (FD) application captures a video stream via cameras, e.g. CCTV footage, and processes it to detect faces. In the context of IoT applications, this case study has been previously used by Wang, et al. in [42] and McChesney, et al. in [43] as a benchmark application.

The original implementation, which is open source and available on GitHub [44], consists of the following three components:

- Client: The *Client*, running on an edge device, captures a video stream and sends it to the *Pre-processor* component.
- Pre-processor: The *Pre-processor*, running on a fog node closer to the edge device, receives the incoming stream, converts it to a greyscale image and sends it to a *Cloud server*. The conversion to greyscale reduces the size of the image by one-third, thereby reducing the overall communication overhead.
- Cloud server: The *Cloud server*, running in a virtual machine in the cloud, receives the greyscale images and detects faces in them.

For the purposes of this paper, the aforementioned implementation has been adopted with the following changes. These changes are performed to demonstrate the capabilities of the proposed multi-level orchestration framework using realistic settings. The updated implementation is available in GitHub [45].

- All components are implemented as Docker containers.
- The face detection functionality has been moved from the *Cloud server* to the *Pre-processor*. This can further reduce the traffic to the central cloud by ignoring those images that do not contain any faces. The new *Cloud server* implementation is only used for storing the greyscale images that contain faces.
- The functionalities of the *Pre-processor* component are distributed into three components, namely *Receiver*, *Processor*, and *Sender*. Such distribution enables independently applying different policies to each service. For example, it enables running multiple instances of the *Receiver* service to receive video streams from different edge devices, or introducing scaling policies only for the *Processor* component.

The following paragraphs discuss the details of the experimental setup, important aspects from the deployment perspective, and the obtained computational results.

4.1.1 Experimental Setup

The aim of the experiment is to demonstrate the automatic deployment and run-time management of the following application scenario:

There are *N* number of edge devices, a fog node, and a cloud-based storage. The edge devices capture video streams and send them to the fog node. The fog node runs *N* instances of the *Receiver*, one instance of the *Sender*, and at least one instance of the *Processor* service. The *Processor* is bound by a scaling policy that determines the number of instances based on the network bytes received by all instances of the *Receiver* service. Lastly, the cloud storage - running on a dynamically created virtual machine in the cloud - is responsible for storing the received greyscale images that contain faces. Figure 8 depicts this scenario in the context of the MiCADO framework.

For this experiment, the following settings were applied for the various nodes of the case study, as introduced in Fig. 8. In this particular scenario, the monitoring services, i.e. node and container exporters, are only deployed on the cloud and fog nodes (as can be seen in the figure). Monitoring on these nodes is required for enforcing the applied scaling policy (see Section 4.1.3). However, such monitoring is not required on the edge devices because they are not expected to be bound by any scaling policy.

- Edge layer: One Raspberry Pi with a plugged-in camera with [ARMv7 Processor rev 3, Model 4, 4 GB RAM] as the edge device.
- Fog layer: A laptop with [4 CPUs, 12GB RAM, and Ubuntu 18.04] as the fog node.
- Cloud layer: Two virtual machines, i.e. MiCADO Master and a cloud worker, on Amazon. The specification of these machines are [2GHz CPU, 3GB RAM, 15GB DISK, and Ubuntu 18.04] and [1GHz CPU, 2GB RAM, 8GB DISK, and Ubuntu 18.04] respectively.

4.1.2 ADT and Deployment (Requirements R1 and R6)

Figure 9 presents some of the important segments from the ADT of the FD application deployment. The complete ADT can be found in the MiCADO TOSCA GitHub repository [46]. Figure 9a shows the definition of the fog node and an edge device. The names of these nodes, e.g. hpfog, and piedge, must match the names of the hosts provided at the time of the MiCADO installation (see Section 3.2). Figure 9b shows some of the inputs in the ADT. The inputs allow

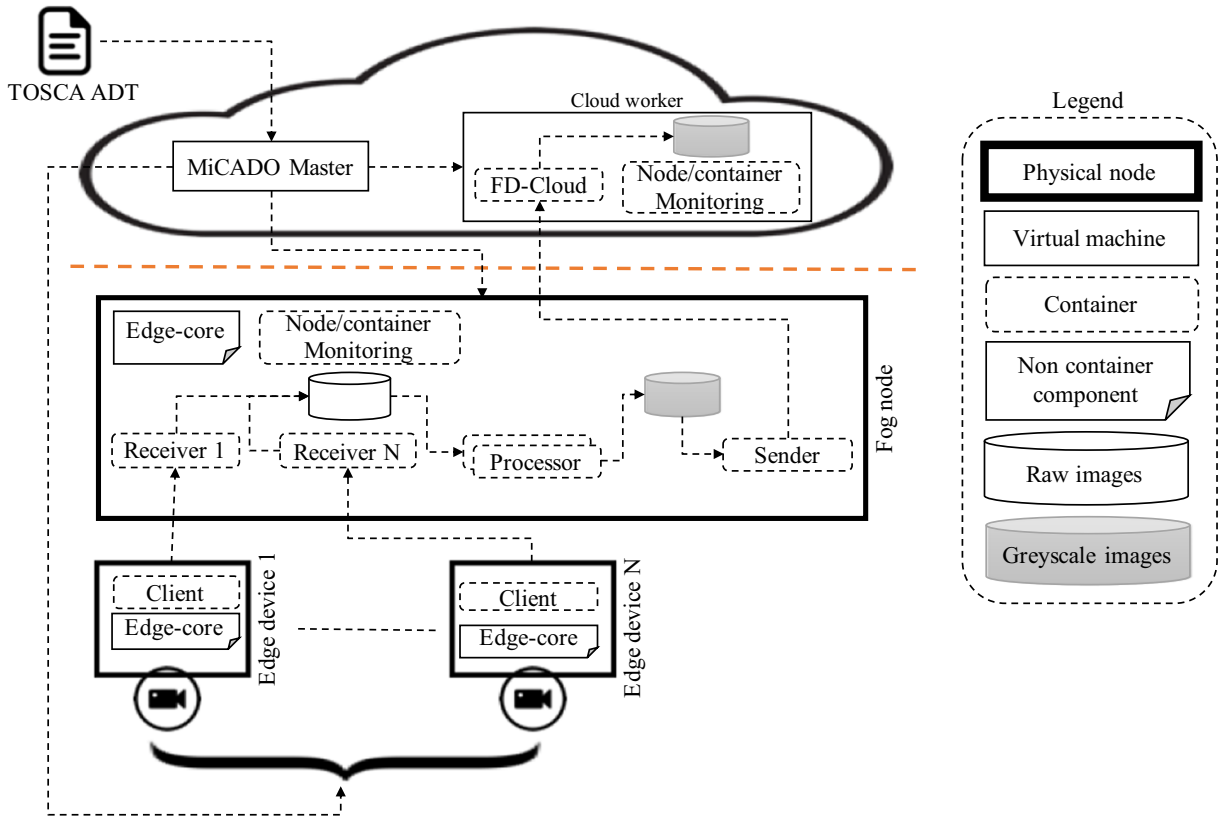


Fig. 8 Architecture of Real-time Face Detection case study in the context of the MiCADO-Edge framework

the definition of lists that can be used for components that require the creation of multiple instances, where each instance requires a different set of values (discussed earlier in Section 3.3), e.g. the definitions of *fd-pi-client* and *fd-receiver* components provided in Fig. 9c and d respectively. The *fd-receiver* component, in this case, requires exactly six instances. Each instance is responsible for listening on a specific port to receive data from a different streaming client. The port assignment for each instance is automatically handled by MiCADO using the *ports* list. Similarly, MiCADO creates multiple instances (based on the value of the *clientcount* variable) of the *fd-pi-client*. MiCADO deploys these instances to the respective edge nodes, i.e. from the *clienthosts* list, by passing a different port value (obtained from the *ports* list) as an environment variable to each instance. In fact, using this topology, MiCADO creates a one to one link from each instance of the *Client* to a corresponding instance of the *Receiver* component.

4.1.3 User-Defined Auto-Scaling Policy (Requirements R4 and R5)

MiCADO empowers application developers to define custom scaling policies for their applications using the Python programming language. These policies are defined in the ADT. This section, in the context of the FD case study, presents an example for the container level scaling of the *Processor* service. The code snippet in Fig. 10 shows the relevant segment of the policy. Based on this policy, Section 4.1.4 will further discuss and present the results obtained at run-time.

The *targets* tag in Fig. 10 indicates that the policy applies to the *fd-processor* service. The *sources* tag informs the Prometheus server, running on the MiCADO Master node, of the target IPs and port numbers for the collection of monitoring information. The *sources*, in this case, includes the fog node that hosts the *fd-processor*, *fd-receiver* and *fd-sender* services. The *constants* tag allows application developers to

```
hpfog:
  type: tosca.nodes.MiCADO.Edge
  properties:
    public_ip: public_ip_fog_node
piedge:
  type: tosca.nodes.MiCADO.Edge
  properties:
    public_ip: public_ip_pi_node
```

(a) Definition of fog node and edge device

```
topology_template:
  inputs:
    ports:
      type: list
      default: [4092, ---, 4097]
    reccount:
      type: integer
      default: 6
```

(b) Definition of inputs for dynamic values

```
fd-pi-client:
  type: --.App.Docker.Deployment
  occurrences: [1, UNBOUNDED]
  instance_count: {get_input: clientcount}
  properties:
    image: uowcpc/fd-pi-client
    securityContext:
      privileged: True
    env:
    - name: REMOTE_HOST
      value: {get_input: ip_hp_node}
    - name: REMOTE_PORT
      value: {get_input:[ports,INDEX]}
  requirements:
  - host: {get_input:
            [clienthosts, INDEX]}
  - volume: docker-pi-vc-host-vol
```

(c) Client service running on edge device

```
fd-receiver:
  type: --.App.Docker.Deployment
  occurrences: [1, UNBOUNDED]
  instance_count: {get_input:
                    reccount}
  properties:
    image: uowcpc/fd-edge-receiver
    env:
    - List environment variables
    ports:
    - containerPort: {get_input:
                        [ports, INDEX]}
      hostPort: {get_input:
                        [ports, INDEX]}
      protocol: 'UDP'
  requirements:
  - host: hpedge
```

(d) Receiver service running on fog node

Fig. 9 ADT Segments of FD case study

define any constants, (e.g. *COOLDOWN* in this case) that can be used inside the Python based scaling code. The *queries* section can include any Prometheus based queries/expressions to obtain (and/or compile) information from the gathered monitoring data of worker (both cloud and edge) nodes. In this particular case, the query labelled as *RCVDBYTES* calculates the total network bytes received by all instances of *fd-receiver* service in the last 2 minutes. The *min_instances* and *max_instances* indicate the minimum and maximum allowed instances of the *fd-processor* service that the policy will enforce at run-time.

The scaling policies from the ADT are iteratively executed by the Policy Keeper component of MiCADO. In each iteration, the Policy Keeper executes the queries to get up-to-date monitoring information and runs the user provided scaling code. The scaling logic in this particular example is expressed in

Fig. 10 Auto-scaling policy for the FD case study

```
policies:
- scalability:
    type: tosca.policies.Scaling.MiCADO
    targets: [ fd-processor ]
    properties:
      sources:
      - public_ip_fog_node:8080
      - public_ip_fog_node:9100
      constants:
        COOLDOWN: 180 # in seconds
      queries:
        RCVDBYTES: "sum(rate(container_network_receive_bytes_total
        {container_label_io_kubernetes_pod_name=~'fd-receiver.*'}[2m])*100)"
      min_instances: 1
      max_instances: 5
      scaling_rule: |
      # User provided python based scaling code
      # COOLDOWN and RCVDBYTES can be used as variables
```

(1), i.e. the required number of *fd-processor* instances at any point in time is proportional to the size of received network bytes by all instances of *fd-receiver* service.

$$m\_container\_count = ceil\left(\frac{rcv\_bytes\_inMB}{2}\right) \quad (1)$$

The rcv_bytes_inMB in the above equation represents the value of received network bytes in Mega Bytes, *ceil* returns the smallest integer value that is bigger than or equal to the value result from the division, and $m\_container\_count$ is the required number of *fd-processor* containers at that point in time.

4.1.4 Computational Results

The demonstrator application was deployed in the previously described testbed using the presented ADT (demonstrating the fulfillment of requirements R1 and R6), and its auto-scaling behaviour was observed based on various input loads (providing evidence for fulfilling requirements R4 and R5). The results are presented in Fig. 11. Figure 11a shows the network traffic received by all instances of the *Receiver* service, Fig. 11b tracks the number of instances of the *Processor* service, and finally Fig. 11c shows the network traffic received by the *cloud storage* service.

The experiment starts with one instance of the *Client* service running on a Raspberry Pi. The *Client* continuously captures the video stream and sends still images from the stream to the *Receiver* service. The incoming traffic from one client averages around 750 Kilo Bytes per second (KB/s). To demonstrate the effect of the scaling policy, the *Client* instances are manually increased from 1 to 6 at the following times to generate additional network traffic: 12:54, 13:02, 13:12, 13:20, and 13:26. This results in increased network traffic on the *Receiver* service from ≈ 750 KB/s to ≈ 4.8 Mega Bytes per second(MB/s). During this increase, based on the scaling policy, MiCADO automatically increased the number of *Processor* instances from one to two at 13:02, when the received network traffic exceeded 2 MB/s, as can be seen in Fig. 11b. Similarly, another scale-up can be observed at approximately 13:24, when the received network traffic became higher than 4 MB/s. The increase in network traffic (i.e. the number of streamed images) has a proportional impact on the number of greyscale images being sent to the *cloud storage* service. This is evident from Fig. 11c at 12:54 and 13:02, i.e. when the number of *Client* instances are increased from one to two and then from two to three. The network traffic received by the *cloud storage* service after 13:17 remains steady because there were many greyscale images still to be sent by the *Sender* service.

Figure 11b shows a scale down action at ≈ 13:28, when the network traffic is reduced to less than 4 MB/s. A few moments later, the traffic suddenly increased and again reached higher than 4 MB/s. This situation, as per the applied policy, required another scale-up. However, it only happened with some delay (and not immediately), at 13:31. The delay is due to the applied cool-down period of 3 minutes. The cool-down period, in general, is referred to as the minimum duration that is applied between two consecutive scaling actions to avoid the oscillating behaviour, where constant scale-up/down operations are performed one after the other [47]. MiCADO empowers application developers to use any custom cool-down period within their scaling policy in the ADT. An example of this is the use of the *COOLDOWN* constant in Fig. 10, which is then further used in the scaling logic part of the policy and enforces that no consecutive scaling actions can be performed within 180 seconds.

Finally, to experience the reaction of MiCADO in case of a decrease in network traffic, two *Client* instances are manually stopped at 13:35 and 13:40. These changes are picked up by MiCADO automatically and are followed by the execution of necessary scale-down actions, as can be seen in Fig. 11b, at 13:38 and 13:44 respectively.

4.2 Sleep Healthcare: a Case Study for Multi-Cloud Usage

The case study described in Section 4.1 demonstrated how MiCADO-Edge can be utilised to deploy and manage applications in multi-level cloud-to-edge environments. However, the developed solution also enables MiCADO to deploy and manage applications where the various interacting microservices are deployed in different clouds. Multi-cloud deployment may be necessary when some specific and required services are only provided by one particular cloud service provider. Such a scenario, the Sleep Healthcare application, is described below, followed by the detailed description of its implementation based on MiCADO-Edge. The motivation for multi-cloud

(a) Network traffic received by the *Receiver* service

(b) Instances of the *Processor* service

(c) Network traffic received by the *Cloud storage* service

Fig. 11 Computational results from the FD case study

deployment in this particular case was the fact that certain components of the application needed to be deployed in virtual machines supporting Intel Software Guard Extensions (SGX) [48]. As such specific VMs are not available in the private cloud environment where the application is currently deployed, adding VMs from Microsoft Azure (with SGX support) was necessary.

4.2.1 Sleep Healthcare

Sleep Healthcare is one of the demonstrator applications within the H2020 ASCLEPIOS project [25]. The Sleep Healthcare application aims at bringing data sharing and analysis capabilities on inpatient and outpatient sleep medicine to the cloud. Using the cloud for storing and processing the different sleep measurements can be highly beneficial.

However existing solutions such as [50, 51] suffer from security and privacy failings [49] that make them ill-suited for cloud deployment. Searchable Encryption (SE) is a promising new technology that allows queries on encrypted data in a way that the cloud provider can neither reveal the metadata search term, nor the query result. In the context of the Sleep Healthcare application, SE technologies would allow medical professionals to manage biosignal recordings from different inpatient and outpatient settings

to enhance both the process and the precision of sleep diagnosis and therapy control, while preserving patient data privacy. Sleep Healthcare utilises one such SE technology called Symmetric Searchable Encryption (SSE).

The SSE scheme, [52] also developed within the scope of the ASCLEPIOS project, enables searches on outsourced encrypted data while preserving the privacy of both the data and any search queries. The SSE scheme mainly consists of two core components, i.e a Trusted Authority (TA) and an SSE Server. The SSE Server facilitates storage of the encrypted data, whereas the TA is responsible for storing the metadata that is required for the facilitation of searching the encrypted data.

For the sake of security, the TA and SSE Server components must be executed in a Trusted Execution Environment (TEE) [53], for example with Intel SGX [48] capabilities. Based on this restriction, the Sleep Healthcare application has the realistic requirement to use Microsoft Azure - being the only cloud with support for SGX - to deploy the two SSE components. However, such a restriction is not mandatory for the other components, (i.e. xnat, keycloak) of the application. Therefore, the Sleep Healthcare demonstrator needs to deploy only the SSE components in Microsoft Azure, while the other components can be deployed to an in-house private cloud to save on operating costs.

4.2.2 Multi-cloud Sleep Healthcare Test-bed Setup

Figure 12 depicts the overall architecture of using MiCADO-Edge for the deployment and run-time management of the Sleep Healthcare use case across two different cloud environments. Such multi-cloud deployment was not supported by the original implementation of MiCADO. However, using the edge extension described earlier in this paper, it is possible to add worker nodes to a MiCADO deployment, even if they are running on a different cloud than the MiCADO Master, and to deploy and manage containers within these VMs. In such deployments, the workers running on the external cloud will act as non-cloud workers and will connect to the MiCADO master node via KubeEdge.

In the case of the Sleep Healthcare application, as it is illustrated in Fig. 12, the Application Server component runs on the primary cloud (in this case the University of Westminster private OpenStack cloud), where the MiCADO Master node is also deployed. This VM and its containers are deployed and managed by MiCADO-Edge as in the original MiCADO implementation. On the other hand, the SSE Server and TA Server components are deployed to a cloud that is remote from where the MiCADO Master is running (in this case on the Microsoft Azure public cloud). While MiCADO was capable of deploying these VMs on this remote cloud previously (by specifying the requirements of VMs running across multiple clouds in the ADT), connecting these VMs and their containers to the Kubernetes cluster only became possible through the use of KubeEdge. The worker nodes running on the remote cloud are all configured as non-cloud nodes, running the Edge-core component of KubeEdge.

- Primary cloud: University of Westminster private OpenStack (Nova Compute). One VM [2 CPUs, 4GB RAM, 40GB DISK and Ubuntu 18.04] as the MiCADO Master node, and one VM [1 CPU, 2GB RAM, 40GB DISK, and Ubuntu 18.04] as the Application Server node.
- Remote cloud: Microsoft Azure public cloud. Two VMs [2 CPUs, 8GB RAM, 16GB DISK, and Ubuntu 18.04] as the SSE Server and TA Server nodes, respectively.

4.2.3 Multi-cloud Sleep Healthcare ADT

The code snippets in Fig. 13 highlight the distinction between a normal MiCADO worker, which is to be launched on the primary cloud versus the worker node that must be launched on the other cloud (i.e. Microsoft Azure in this case). The node type *tosca.nodes.MiCADO.Nova.Compute* in (Fig. 13a), informs MiCADO to dynamically create a worker with the given specification on OpenStack. On the other hand, the node type *tosca.nodes.MiCADO.Edge.Azure* in Fig. 13b informs MiCADO that a worker node is required on the other cloud (i.e. on Microsoft Azure). In this case, MiCADO will dynamically create a virtual machine with the required specification. In addition, the *Edge-core* component will also be installed and automatically configured on this external node. This configuration will lead the newly created virtual machine to join the MiCADO cluster and run the specified con-

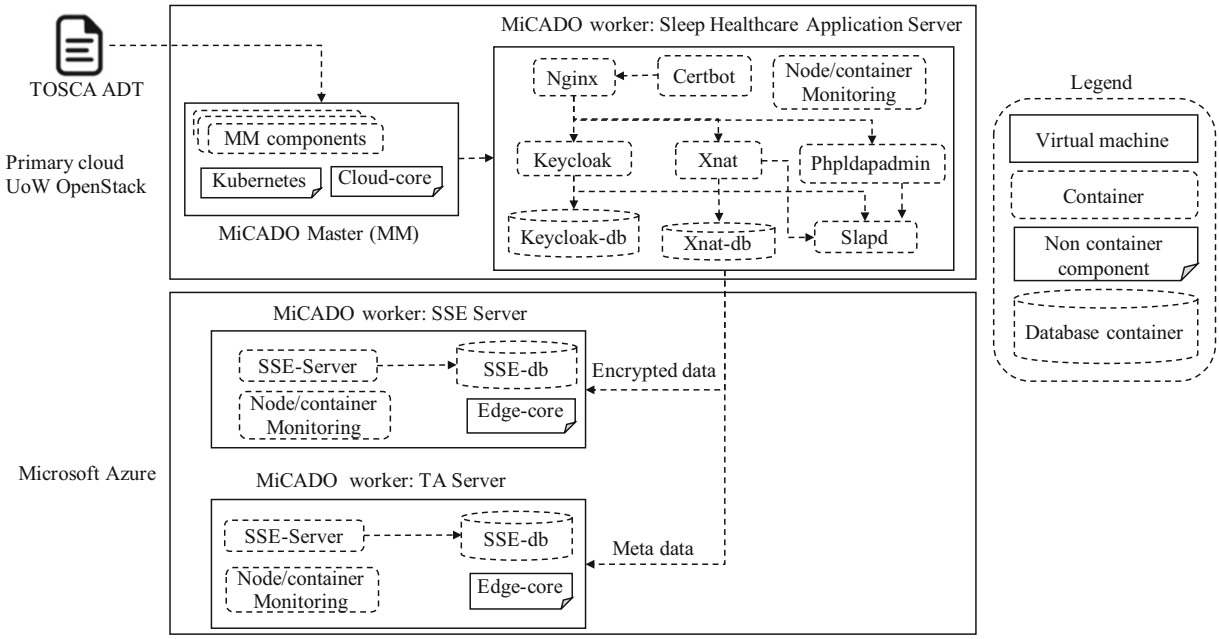


Fig. 12 Architecture of Sleep Healthcare in integration with the SSE scheme in the context of MiCADO-Edge framework

tainers. The full ADT, which includes the description of the required resources and overall application topology of the Sleep Healthcare application is available in the MiCADO TOSCA GitHub repository [54].

4.2.4 Multi-cloud Sleep Healthcare Deployment & Results

Once the ADT is finalised and submitted to MiCADO, it deploys the Sleep Healthcare application along with the necessary SSE components across both cloud environments. After the deployment is complete, MiCADO starts managing the application. The monitoring information from the SSE components deployed on Azure can be gathered in a similar way to how it was done for edge nodes, as described in Section 3.4. Figure 14 presents some examples of the obtained monitoring information from the experiment. Figure 14a shows the CPU utilisation of the three deployed VMs on the Azure and OpenStack clouds (see Fig. 12). In Fig. 14a, the VMs indicated by labels *51.134.4.182* and *51.140.51.82* are the ones that run on Azure, while the third VM runs on OpenStack. Figure 14b, on the other hand, presents the CPU usage of each application container running on the aforementioned three VMs. Containers with labels starting with *sse* or *ta* are running on the Azure VMs, while the rest runs on OpenStack. Using the obtained monitoring information, MiCADO can perform any reconfiguration operation on either cloud, as required. This

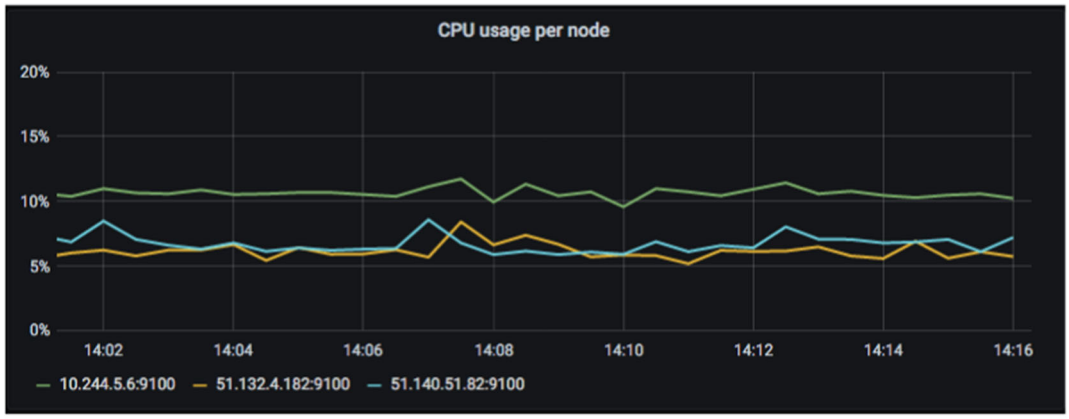
```
nova-worker:
  type: tosca.nodes.MiCADO.
                Nova.Compute
  properties:
    # VM Specifications
```

(a) Normal worker for OpenStack

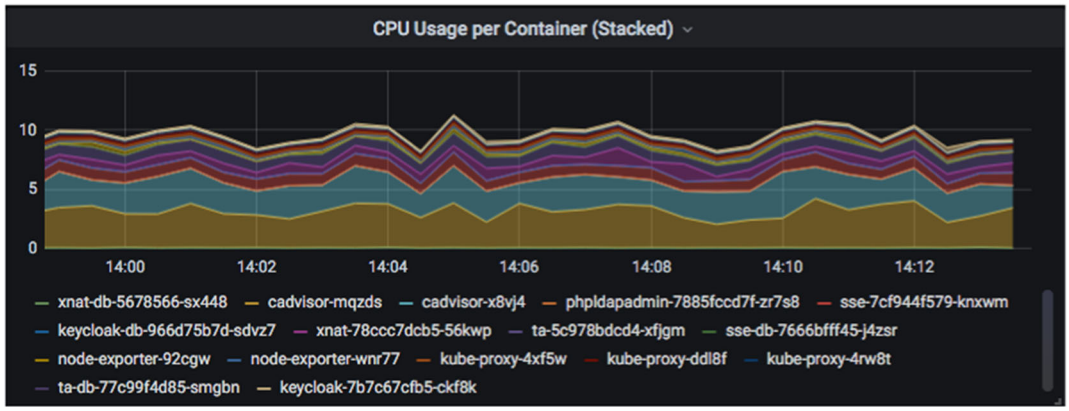
```
azure-sse-server-worker:
  type: tosca.nodes.MiCADO.Edge.Azure
  properties:
    # VM specifications
    config_drive: true
    context:
      path: "system/cloud_init_worker
                    _tf_edge.yaml"
```

(b) Edge worker for the Azure cloud

Fig. 13 Distinction between primary cloud worker and Azure cloud worker



(a) CPU utilisation per node



(b) CPU usage per container

Fig. 14 Computational results from the Sleep Healthcare case study

setup is a strong evidence that MiCADO-Edge fulfills requirement R3 (support for multi-cloud applications), as well as R1 (automated deployment of microservices using a single ADT), R2 (cloud-agnostic nature as two different cloud technologies were applied) and R6 (applications deployed in containers hosted by virtual machines).

5 Related Work

Existing cloud-to-edge orchestration solutions can be divided into two major categories. There are infrastructure-level solutions that often act as middlewares by allowing access to microservices running in the edge and fog layers and connecting these with components executed in the central cloud. While such solutions are essential for the presented MiCADO-Edge implementation, these are not direct competitors of our work. However, such solutions are necessary for extending MiCADO to the edge and can be reused and integrated with the proposed architecture. On the other hand, there are some significant and notable efforts to create comprehensive, application-level cloud-to-edge orchestrators. These orchestrators can be directly compared to MiCADO-Edge. The rest of this section provides a short overview of the most prominent research efforts in both categories and puts these into the context of our proposed solution.

5.1 Infrastructure-Level Solutions to Access the Fog and Edge Layers

There are a number of proprietary and open source initiatives offering capabilities to run applications in the cloud-to-edge continuum. These infrastructure-level

solutions provide an important bridge for connecting nodes in the edge and fog layers with nodes in the cloud layer. Some notable initiatives from this category include the following:

Cloud vendor specific propriety solutions All major cloud vendors offer proprietary solutions, often in the form of a middleware, for application deployment in the cloud-to-edge continuum. Such examples include Amazon GreenGrass [55], Microsoft's Azure IoT core [56] and Google's Cloud IoT Core [57]. These solutions make it possible to run local compute, messaging, management, monitoring, synchronisation and machine learning interface capabilities on connected devices in a secure way. The application modules are executed locally as Docker containers, making the deployment and orchestration easier. However, all three solutions have been developed with their respective cloud platforms in mind. Hence, they are not cross-platform solutions and will cause some degree of vendor lock-in.

Open source cloud agnostic solutions Project EVE [4], a Linux Foundation Edge (LF Edge) project, provides a flexible foundation for IoT edge deployments with the choice of any hardware, application and cloud. EVE supports both virtual machines (VMs) and containers. The orchestration of the underlying hardware and installed software is achieved through the open EVE API. EVE is complementary to other LF Edge application frameworks such as EdgeX Foundry [60] and Fledge [5]. The aim of EdgeX Foundry is the simplification and standardisation of the foundation for edge computing architectures in the Industrial IoT market, while Fledge is a Kubernetes compatible container orchestrator for edge devices. Fledge works closely with Project EVE to provide system and orchestration services and a container run-time for Fledge applications and services.

Other notable open source initiatives include KubeEdge [3], Kubefed [58] and Submariner [59]. All these solutions built upon Kubernetes and aim to extend native containerised application orchestration capabilities to hosts at the edge of the network by providing fundamental infrastructure support for network, application deployment and metadata synchronisation between cloud and edge. These solutions provide almost similar features, however, there exists fundamental difference in their architectures. The KubeEdge architecture is based on a centralised approach, where every non-cloud node joins the centralised cluster. In contrast, Kubefed and Submariner follow a federated approach, where the non-cloud workers formulate a sub-cluster of their own.

In the context of this paper, the above-mentioned Infrastructure-level solutions are important and related because MiCADO-Edge requires such solutions as a component to access and manage fog and edge resources. However, in comparison with MiCADO-Edge, these are lower-level solutions in the sense that they do not offer policy-based scaling and automated deployment of complex microservices architectures. Moreover, these solutions also lack the high level abstraction layer for describing application and computational resources using a standardised high-level description language. Such an abstraction layer improves re-usability, portability and helps in avoiding the issue of vendor lock-in.

5.2 Application-Level Cloud-to-Edge Orchestration

This sub-section provides an overview of the most related application-level cloud-to-edge orchestrator solutions and compares them to the proposed MiCADO-Edge framework.

Reference architectures At a conceptual level, several reference architectures for fog and edge computing have been proposed (e.g. [61] and [62]) that highlight the necessary functional layers required for orchestration. While these studies provide important input and a starting point for our work, they only present high-level descriptions of the applied concepts and principles, when compared to a working reference implementation of a framework, such as MiCADO-Edge.

Orchestration solutions with reference implementations At the level of more concrete reference implementations, there are several research efforts that concentrate on the orchestration of microservices for the fog and edge layers. Notable examples include [63–69]. These efforts typically apply Docker containers to deliver services and utilise either Docker Swarm [63, 64] or Kubernetes for container orchestration [66]. The authors in [65] present an ad-hoc OpenStack-based Heat Orchestration Template (HOT) service manifest for deploying microservices to selected fog

computing nodes. However, unlike MiCADO-Edge, this solution does not support run-time management of resources and services. The ENORM framework [67], on the other hand, provides comprehensive autoscaling features. However, automated deployment of complex architectures is not supported. Moreover, ENORM autoscaling is based on predefined policies instead of the dynamic user defined policies supported by MiCADO-Edge. Lastly, the solutions described in [68] and [69] are both similar to MiCADO-Edge as they also offer deployment, orchestration and auto-scaling of edge resources. However, they support orchestration only at the container level, as opposed to the VM/container orchestration supported by MiCADO-Edge. Moreover, these solutions are all specific to a particular cloud environment in their presented implementations (e.g. both [68] and [69] are implemented specifically in OpenStack and support only specific types of applications and experimental testbeds). In comparison, MiCADO-Edge is cloud agnostic and supports flexible application deployment and user-defined scaling policies that are not specific to middleware or to application type.

Offloading specific orchestration solutions While the previously mentioned solutions concentrate on microservice orchestration for the fog and edge layers, there are also several orchestration solutions [70–74] that focus primarily on offloading workloads from edge/fog nodes to the cloud and Quality of Service (QoS) optimisation. Foggy [71] and FORCH [73] facilitate dynamic resource provisioning and automated application deployment in fog computing architectures. They minimise the developer effort required to deploy, update, and maintain large-scale geo-distributed IoT applications. However, Foggy only seems to focus on application placements and does not appear to support auto-scaling of applications during run-time, while FORCH does not deal with offloading or scaling of applications. Taherizadeh et al. [70] propose a new distributed computing architecture that performs offloading from edge to cloud nodes and supports smart applications where IoT devices dynamically move from one geographic location to another. The main focus of this work is on application offloading and in their testbed application they appear to set up application deployment in a custom way. Orchestration frameworks ISYMPHONY [72] and PiCasso [74] consider orchestration in the edge layer only, and not in the entire cloud-to-edge continuum. Their orchestration engine component deploys services based on the required QoS specifications and the status of the resources of the hosting devices. These solutions are all based on lightweight Dockerised services and do not support virtual machine level orchestration. Moreover, their major focus is computation offloading from fog/edge nodes towards the cloud, and they do not provide automated, dynamic policy-based scaling or full life-cycle management of interconnected microservices in the way that MiCADO-Edge does.

Table 1 presents a comparison of the aforementioned related works with MiCADO-Edge, based on a number of attributes that are derived from requirements R1 to R7 presented in Section 1.2. These attributes are more granular and make comparison easier by focusing on a single easily identifiable characteristic. The attributes are defined below. At this point, we should emphasise that our goal has been to design an orchestration framework that fulfils all seven requirements with the support for the derived attributes we present. We do not necessarily claim that our solution is superior to all related solutions, we merely present that MiCADO-Edge is the only solution that satisfies the requirements that emerged from the DIGITbrain and ASCLEPIOS projects. The various attributes used for comparison in Table 1 and their meaning are as follows:

1. High-level abstraction layer - The support for defining application and computational resources using a standardised high-level description language such as TOSCA [26] or CAMP [75].
2. Cloud agnostic - The ability of an orchestration solution to operate seamlessly between different cloud platforms.
3. Multi-cloud - The support for deploying and managing application workloads across multiple cloud environments simultaneously.
4. Extendable cloud middle-ware support - The support for facilitating the addition of new cloud service providers.
5. Modular design - The flexibility in the context of replacing parts of its implementation with a different realisation of the same functionality.
6. Virtual Machine support - The support for dynamically provisioning and management of

Table 1 Comparative summary of the orchestration solutions in the cloud-to-edge ecosystem

Attributes	Relevant requirements	Orchestration solutions										
		Enorm [67]	Cloud4IoT [69]	Zanni et al. [68]	Forch [73]	Brito et al. [63]	Alam et al. [64]	Villari et al. [65]	Foggy [71]	ISYM PHONY [72] PiCasso [74]	Taheri zadeh et al. [70]	MiCADO-Edge
1. High-level abstraction layer	R1, R2					x					x	x
2. Cloud agnostic	R2	x			x	x	x		x	x	x	x
3. Multi-cloud	R3											x
4. Extendable cloud middleware support	R2, R3											x
5. Modular design	R2		x		x				x			x
6. Virtual Machine support	R6											x
7. Container support	R6	x	x	x	x	x	x	x	x	x	x	x
8. Seamless integration of non-cloud (fog/edge) resources	R1, R2											x
9. Automated deployment	R1	x	x	x	x	x	x	x	x	x	x	x
10. Run-time management	R4	x	x	x	x	x	x	x	x	x	x	x
11. Reconfiguration	R4		x	x		x			x	x	x	x
12. Auto-scaling	R4	x	x	x						x		x
13. Dynamic and user-defined auto-scaling policies	R5											x
14. Security handling	R7					x						x
15. Dynamic and user-defined networking and security policies	R7											x
16. Heterogeneity	R7		x									x
17. Fault diagnosis	R7				x	x	x	x				x
18. Automatic re-connectivity of non-cloud volatile nodes	R7									x	x	x
19. Support for standardised monitoring mechanism	R3, R4	x		x	x				x			x

virtual machines, as well as the support for the deployment and management of application components on virtual machines (with or without containerisation).

7. Container support - The support for the deployment and management of application components through containers.
8. Seamless integration of non-cloud (fog/edge) resources - The support for an automated method through which an orchestrator facilitates application owners the integration of non-cloud (fog/edge) resources with the centralised cluster.
9. Automated deployment - The support for the automated deployment and configuration of applications across (multi-)cloud and non-cloud resources.
10. Run-time management - The support for the run-time management of applications across (multi)-cloud and non-cloud resources.
11. Reconfiguration - The support for a method that enables making changes to an already running application. The method can be manual, e.g. where amendments to application topological structure can be manually resubmitted by the application owners, or automatic, e.g. where migration of certain components can be made possible based on the fulfilment of some run-time rules.
12. Auto-scaling - The support for the automated scaling of computational resources based on some pre-defined elasticity policies, e.g. system state specific rule-based scaling policies.
13. Dynamic and user-defined auto-scaling policies - The support for dynamic and complex user-defined policies based on a large range of system- or application-level metrics (as opposed to the pre-defined policies in attribute 12). Such a feature aim to enable application owners the freedom to write their scaling policies without any restriction in comparison to the pre-defined scaling policies where the users are usually restricted to provide threshold values based on some already established criteria.
14. Security handling - The support for security enforcement enablers that provide security guarantees of the overall system against different attack scenarios in relation to the orchestration frameworks [76].
15. Dynamic and user-defined network and security policies - The support for user-defined network and security policies related to their applications at the time of deployment. Examples of such policies could be application level port filtering and firewall rules, TLS control, proxy settings, secret provisioning, etc.
16. Heterogeneity - The support for the simultaneous use of cloud and non-cloud computational resources that are different in terms of underlying hardware, architecture, and/or operating systems.
17. Fault diagnosis - The support for detection of system and/or application level faults at run-time, e.g. unexpected termination of a virtual machine due to fault in cloud provider system, an application level un-handled run-time exception forced to stop a container, a volatile edge node lost connection to the centralised cluster, etc.
18. Automatic re-connectivity of non-cloud volatile nodes - Continuous Internet connectivity can not be guaranteed in the cloud-to-edge environment and therefore, non-cloud workers may lose connection to the centralised cluster. In this regard, this attribute represents the support for the automatic re-connectivity of volatile non-cloud workers to the centralised cluster in case Internet connectivity is lost and then later restored.
19. Support for standardised monitoring mechanism - The support for gathering system-level (e.g. cpu/memory utilisation) and application-level metrics (e.g. number of active http requests) from both cloud and non-cloud workers, as well as the ability to define custom metrics for collection (e.g. number of running jobs in a batch processing application).

6 Conclusions and Future Work

This paper explored how an existing application level cloud orchestrator can be extended to support the cloud-to-edge computing continuum. Although specific technologies were used for the implementation (i.e MiCADO and KubeEdge), the presented solution is generic in the sense that the applied principles can be utilised for other similar software solutions. More-

over, due to the highly modular nature of the implementation, any component can be easily replaced in the future with a different implementation if necessary. The capabilities of the resulting cloud-fog-edge orchestrator were illustrated with two realistic case studies. One of these case studies also demonstrated how the implemented solution can be used to support multi-cloud scenarios.

Further development of MiCADO-Edge is currently ongoing within the DIGITbrain and ASCLEPIOS projects. Following requirements collection, DIGITbrain has designed the concept of a generic platform that supports the execution of digital twin instances that are dynamically generated from data, model and algorithm tuples. The execution mechanism of the platform is based on MiCADO-Edge and supports the automated deployment and run-time management of digital twins throughout the cloud-to-edge continuum. Based on the proof of concept prototype described in this paper, MiCADO-Edge is being developed into a production quality solution and will also be extended on demand with specific features required by the application experiments (e.g. support for Windows applications and data streaming). Additionally, it is also envisaged that MiCADO-Edge will be integrated with FIWARE [77] and more specifically with its context broker in order to further facilitate and automate the connection of various edge and fog nodes to the orchestrator. In parallel, ASCLEPIOS is concentrating on specific security requirements raised by healthcare application scenarios. In this context, MiCADO-Edge should facilitate the deployment and management of application microservices that are executed in Trusted Execution Environments, such as Intel SGX. Therefore, integration of MiCADO with such deployment mechanisms is currently ongoing.

Acknowledgements This work was funded by the following projects: ASCLEPIOS – Advanced Secure Cloud Encrypted Platform for Internationally Orchestrated Solutions in Healthcare – project, No. 826093, European Commission (EU H2020); DIGITbrain - Digital twins bringing agility and innovation to manufacturing SMEs, by empowering a network of DIHs with an integrated digital platform that enables Manufacturing as a Service – project, No. 952071, European Commission (EU H2020).

References

1. Gartner forecasts worldwide public cloud revenue to grow 17% in 2020 (2019). https://www.gartner.com/en/newsroom/press-releases/2019-11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020, Accessed 5 Oct 2020
2. Marston, S., Li, Z., Bandyopadhyay, S., Ghalsasi, A.: Cloud computing - the business perspective. In: 2011 44th Hawaii International Conference on System Sciences, pp. 1–11 (2011)
3. Kubeedge (2020). https://kubeedge.io/en/, Accessed 4 Oct 2020
4. Project eve (2020). https://www.lfedge.org/projects/eve/, Accessed 4 Oct 2020
5. Goethals, T., De Turck, F., Volckaert, B.: Fledge: Kubernetes compatible container orchestration on low-resource edge devices. In: Internet of vehicles : technologies and services toward smart cities, 6th International Conference, IOV 2019, Proceedings, pp. 174–189. Springer (2020)
6. Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., Kong, J., Jue, J.P.: All one needs to know about fog computing and related edge computing paradigms: A complete survey. J. Syst Architect. **98**, 289–330 (2019). https://doi.org/10.1016/j.sysarc.2019.02.009
7. Mercer, D.: Global connected and iot device forecast update. https://www.strategyanalytics.com/access-services/devices/connected-home/consumer-electronics/reports/report-detail/global-connected-and-iot-device-forecast-update (2019)
8. Columbus, L.: Roundup of internet of things forecasts and market estimates, 2016. Forbes Magazine. https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016/#6a558beb292d (2016)
9. The growth in connected iot devices is expected to generate 79.4zb of data in 2025, according to a new idc forecast (2019). https://www.idc.com/getdoc.jsp?containerId=prUS45213219
10. IEEE standard for adoption of openfog reference architecture for fog computing. IEEE Std 1934-2018, pp. 1–176 (2018)
11. Kubernetes : Production-grade container orchestration (2020). https://kubernetes.io/, Accessed 4 Oct 2020
12. Docker swarm (2020). https://docs.docker.com/engine/swarm/, Accessed 4 Oct 2020
13. Apache brooklyn (2020). http://brooklyn.apache.org/, Accessed 4 Oct 2020
14. Cloudify orchestration platform - multi cloud, cloud native & edge (2020). https://cloudify.co/, Accessed 4 Oct 2020

15. Cloudiator (2020). http://cloudiator.org/, Accessed 4 Oct 2020
16. Alien 4 cloud (2020). https://alien4cloud.github.io/, Accessed 4 Oct 2020
17. Modaclouds multi-cloud devops alliance: Modaclouds releases multi-cloud devops toolbox (2020). http://multiclouddevops.com/, Accessed 4 Oct 2020
18. Micadoscale (2020). https://micado-scale.eu/, Accessed 4 Oct 2020
19. Amazon: Aws cloudformation: Speed up cloud provisioning with infrastructure as code. https://aws.amazon.com/cloudformation/, Accessed 18 Oct 2020 (2020)
20. OpenStack: Openstack orchestration. https://wiki.openstack.org/wiki/Heat, Accessed 18 Oct 2020 (2020)
21. Azure resource manager (arm) templates (2020). https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/overview, Accessed 19 Oct 2020
22. Google cloud depyment manager (2020). https://cloud.google.com/deployment-manager, Accessed 19 Oct 2020
23. Ghobaei-Arani, M., Souri, A., Rahmanian, A.A.: Resource management approaches in fog computing: a comprehensive review. J. Grid Comput, 1–42 (2019)
24. Digitbrain h2020 project (2020). https://digitbrain.eu/, Accessed 4 Oct 2020
25. Asclepios eu h2020 project (2020). https://www.asclepios-project.eu/, Accessed 4 Oct 2020
26. Oasis topology and orchestration specification for cloud applications (2020). www.oasis-open.org/committees/tosca, Accessed 4 Oct 2020
27. Pierantoni, G., Kiss, T., Terstyanszky, G., DesLauriers, J., Gesmier, G., Dang, H.-V.: Describing and processing topology and quality of service parameters of applications in the cloud. J. Grid Comput., 1–18 (2020)
28. Cola - cloud orchestration at the level of application, h2020 eu project (2020). https://project-cola.eu/, Accessed 4 Oct 2020
29. DesLauriers, J., Kiss, T., Ariyattu, R.C., Dang, H.-V., Ullah, A., Bowden, J., Krefting, D., Pierantoni, G., Terstyánszky, G.: Cloud apps to-go: Cloud portability with tosca and micado. Concurrency and Computation: Practice and Experience, Accepted (2020)
30. Kiss, T., Kacsuk, P., Kovács, J., Rakoczi, B., Hajnal, Á., Farkas, A., Gesmier, G., Terstyánszky, G.: Micado - microservice-based cloud application-level dynamic orchestrator. Fut. Gener. Comput. Syst. **94**, 937–946 (2019)
31. Kiss, T., DesLauriers, J., Gesmier, G., Terstyánszky, G., Pierantoni, G., Oun, O.A., Taylor, S.J.E., Anagnostou, A., Kovács, J.: A cloud-agnostic queuing system to support the implementation of deadline-based application execution policies. Future Gener. Comput. Syst. **101**, 99–111 (2019). https://doi.org/10.1016/j.future.2019.05.062
32. Kovács, J., Kacsuk, P.: Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures. J. Grid Comput. **16**(1), 19–37 (2018)
33. Terraform (2020). www.terraform.io, Accessed 4 Oct 2020
34. Docker (2020). www.docker.com, Accessed 4 Oct 2020
35. Prometheus (2020). https://prometheus.io/, Accessed 4 Oct 2020
36. Kovács, J.: Supporting programmable autoscaling rules for containers and virtual machines on clouds. J. Grid Comput. **17**(4), 813–829 (2019)
37. Ansible documentation (2020). https://docs.ansible.com/ansible/latest/index.html, Accessed 19 Oct 2020
38. Micado - autoscaling framework for docker services on cloud (2020). https://github.com/micado-scale/ansible-micado/tree/edge, Accessed 19 Oct 2020
39. Oasis (2020). https://www.oasis-open.org/, Accessed 30 Oct 2020
40. Openstack parser (2020). https://github.com/openstack/tosca-parser, Accessed 4 Oct 2020
41. Kubeedge (2020). https://github.com/kubeedge/kubeedge, Accessed 19 Oct 2020
42. Wang, N., Matthaiou, M., Nikolopoulos, D.S., Varghese, B.: Dyverse: Dynamic vertical scaling in multi-tenant edge environments. Future Generation Computer Systems (2020)
43. McChesney, J., Wang, N., Tanwer, A., de Lara, E., Varghese, B.: Defog: fog computing benchmarks. In: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, pp. 47–58 (2019)
44. Dyverse - dynamic vertical scaling in multi-tenant edge environments (2020). https://github.com/qub-blesson/DYVERSE, Accessed 19 Oct 2020
45. Real time face detection (fd) demo application (2020). https://github.com/UoW-CPC/DYVERSE, Accessed 19 Oct 2020
46. Micado tosca adt repository (2020). https://github.com/micado-scale/tosca/tree/develop/ADT/edge-fog, Accessed 19 Oct 2020
47. Ullah, A.: Towards a novel biologically-inspired cloud elasticity framework. Ph.D. Thesis, University of Stirling, UK (2017)
48. Costan, V., Devadas, S.: Intel sgx explained. IACR Cryptol. ePrint Arch. **2016**(86), 1–118 (2016)
49. Sweeney, L.: k-anonymity: A model for protecting privacy. Int. J. Uncertain. Fuzz. Knowl.-Based Syst. **10**(05), 557–570 (2002)
50. Beier, M., Jansen, C., Mayer, G., Penzel, T., Rodenbeck, A., Siewert, R., Witt, M., Wu, J., Krefting, D.: Multicenter data sharing for collaboration in sleep medicine. Fut. Gener. Comput. Syst. **67**, 466–480 (2017)
51. Beier, M., Penzel, T., Krefting, D.: A performant web-based visualization, assessment and collaboration tool for multidimensional biosignals. Front. Neuroinform. **13**, 65 (2019)
52. Bakas, A., Michalas, A.: Power range: Forward private multi-client symmetric searchable encryption with range queries support (2020)
53. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: what it is, and what it is not. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 1, pp. 57–64, IEEE (2015)
54. Asclepios adt repository (2020). https://github.com/micado-scale/tosca/tree/asclepios/ADT/sleep, Accessed 19 Oct 2020
55. Amazon greengrass (2020). https://aws.amazon.com/greengrass/, Accessed 4 Oct 2020
56. Microsoft azure iot (2020). https://azure.microsoft.com/en-gb/overview/iot/, Accessed 4 Oct 2020
57. Google cloud iot (2020). https://cloud.google.com/iot-core, Accessed 4 Oct 2020
58. Kubernetes federation project (2020). https://github.com/kubernetes-sigs/kubefed, Accessed 4 Oct 2020

59. Submariner, connected kubernetes overlay networks (2020). https://github.com/submariner-io/submariner, Accessed 4 Oct 2020
60. Edgex foundry (2020). https://www.edgexfoundry.org, Accessed 4 Oct 2020
61. Ostberg, P., Byrne, J., Casari, P., Eardley, P., Anta, A.F., Forsman, J., Kennedy, J., Le Duc, T., Marino, M.N., Loomba, R., Lopez Pena, M.A., Veiga, J.L., Lynn, T., Mancuso, V., Svorobej, S., Torneus, A., Wesner, S., Willis, P., Domaschka, J.: Reliable capacity provisioning for distributed cloud/edge/fog computing applications. In: 2017 European Conference on Networks and Communications (EuCNC), pp. 1–6 (2017)
62. Velasquez, K., Abreu, D.P., Gonçalves, D., Bittencourt, L., Curado, M., Monteiro, E., Madeira, E.: Service orchestration in fog environments. In: 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud), pp. 329–336 (2017)
63. de Brito, M.S., Hoque, S., Magedanz, T., Steinke, R., Willner, A., Nehls, D., Keils, O., Schreiner, F.: A service orchestration architecture for fog-enabled infrastructures. In: 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC), pp. 127–132 (2017)
64. Alam, M., Rufino, J., Ferreira, J., Ahmed, S.H., Shah, N., Chen, Y.: Orchestration of microservices for iot using docker and edge computing. IEEE Commun. Mag. **56**(9), 118–123 (2018)
65. Villari, M., Celesti, A., Tricomi, G., Galletta, A., Fazio, M.: Deployment orchestration of microservices with geographical constraints for edge computing. In: 2017 IEEE Symposium on Computers and Communications (ISCC), pp. 633–638 (2017)
66. Pahl, C., Helmer, S., Miori, L., Sanin, J., Lee, B.: A container-based edge cloud paas architecture based on raspberry pi clusters. In: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), pp. 117–124 (2016aug)
67. Wang, N., Varghese, B., Matthaiou, M., Nikolopoulos, D.: Enorm: A framework for edge node resource management. IEEE Trans. Serv. Comput. **PP** (2017)
68. Zanni, A., Forsstrom, S., Jennehag, U., Bellavista, P.: Elastic provisioning of internet of things services using fog computing: An experience report. In: 2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), pp. 17–22 (2018)
69. Pizzolli, D., Cossu, G., Santoro, D., Capra, L., Dupont, C., Charalampos, D., De Pellegrini, F., Antonelli, F., Cretti, S.: Cloud4iot: A heterogeneous, distributed and autonomic cloud platform for the iot, pp. 476–479 (2016)
70. Taherizadeh, S., Stankovski, V., Grobelnik, M.: A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers. Sensors **18** (2018)
71. Yigitoglu, E., Mohamed, M., Liu, L., Ludwig, H.: Foggy: A framework for continuous automated iot application deployment in fog computing. In: 2017 IEEE International Conference on AI Mobile Services (AIMS), pp. 38–45 (2017)
72. Yigitoglu, E., Liu, L., Looper, M., Pu, C.: Distributed orchestration in large-scale iot systems. In: 2017 IEEE International Congress on Internet of Things (ICIOT), pp. 58–65 (2017)
73. Davoli, G., Borsatti, D., Tarchi, D., Cerroni, W.: Forch: An orchestrator for fog computing service deployment. In: 2020 IFIP Networking Conference (Networking), pp. 677–678 (2020)
74. Lertsinsrubtavee, A., Ali, A., Molina-Jimenez, C., Sathiaseelan, A., Crowcroft, J.: Picasso: A lightweight edge computing platform. In: 2017 IEEE 6th International Conference on Cloud Networking (CloudNet), pp. 1–7 (2017)
75. Cloud application management for platforms version 1.1 (2020). http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html, Accessed 1 Feb 2020
76. Paladi, N., Michalas, A., Dang, H.-V.: Towards secure cloud orchestration for multi-cloud deployments. In: Proceedings of the 5th Workshop on CrossCloud Infrastructures & Platforms, pp. 1–6 (2018)
77. Fiware: The open source platform for our smart digital future (2020). https://www.fiware.org/, Accessed 19 Oct 2020