Received: 27 June 2023 | Revised: 1 August 2023 | Accepted: 7 August 2023

DOI: 10.1002/nem.2246

SPECIAL ISSUE PAPER

WILEY

Fog computing out of the box: Dynamic deployment of fog service containers with TOSCA

Suvam Basak | **Satish Narayana Srirama**

Cloud & Smart Lab, School of Computer and Information Sciences, University of Hyderabad, Hyderabad, Telangana, India

Correspondence
Satish Narayana Srirama, Cloud & Smart Lab, School of Computer and Information Sciences, University of Hyderabad, Hyderabad, Telangana, India.
Email: satish.srirama@uohyd.ac.in

Funding information
MHRD, India, Grant/Award Number: F11/9/2019-U3(A); Science and Engineering Research Board, India, Grant/Award Number: CRG/2021/003888

Abstract

The conventional cloud-centric Internet of Things (IoT) application fails to meet the latency requirement of time-critical applications. The idea of edge and fog computing arrived to distribute workloads across the fog devices located in the local area. However, achieving seamless interoperability, platform independence, and automatic deployment of services becomes the major challenge over heterogeneous fog devices. This paper proposes an integrated and standards-based fog computing federation framework, FogDEFT, that adapts OASIS–Topology and Orchestration Specification for Cloud Applications (TOSCA) for service deployment in fog. The framework standardizes the distributed application design with TOSCA Service Template to deploy Docker Containers in Swarm mode and manages interoperability over heterogeneous fog devices. The framework uses a lightweight TOSCA compliant orchestrator to dynamically deploy various fog applications (user-developed services) on the fly.

KEYWORDS

docker, dynamic deployment, fog computing, Internet of Things, TOSCA

1 | INTRODUCTION

The Internet of Things (IoT) applications initially focused on cloud-centric architectures, where all the sensor data are moved to the cloud platform. After analyzing and processing the sensor data, control signals are sent back to the actuators for actuation on that site. However, due to connectivity issues, high propagation delay, and energy usage, this architecture fails to meet the requirements of real-time applications criteria of low latency and high reliability.[1] The solution to this issue is to move this processing component from the cloud platform to the local hybrid distributed processing architecture known as fog computing.[2,3] As an extension, this solution reduces the overall network load as well. This fog federation allows seamless coordination and cooperation between heterogeneous devices (gateway devices such as Raspberry Pi, drones, network switches, and routers) and make use of as much of resources available locally in fog devices.[4] The biggest challenge in this architecture is having reliability, portability, interoperability, and platform independence on top of a set of heterogeneous devices.[5–7]

The cloud community also faced a similar problem with a lack of standardization and vendor lock-in or portability of composite cloud applications across the different cloud service providers.[8] OASIS–Topology and Orchestration Specification for Cloud Applications (TOSCA) addressed the problem.[9] TOSCA automates the entire life cycle of a cloud application with operations like create, configure, start, stop, and delete.[10] In a nutshell, TOSCA is a modeling language

used to create the blueprint of composite cloud applications. Developers can model their applications in a YAML* file, known as TOSCA Service Template, and deploy them on a cloud platform through a TOSCA-compliant orchestrator. A TOSCA-compliant orchestrator uses implementation scripts (shell script, python script, or any IT Automation tool like Ansible, Chef, Puppet) to implement the TOSCA standard.

1.1 | Motivation

Configuring an array of fog nodes for deploying applications on the fly is painful. In an ideal scenario, a generic service deployment framework would handle all the configuration and deployment complexities under the hood. That provides the convenience and abstraction of single command deployment of any application anywhere on the fly. Hence, the cloud community's adoption of TOSCA standards motivates us to extend that to fog services instead of creating a new framework from scratch.

Therefore, this paper proposes an extension of the TOSCA standard for fog computing framework: **FogDEFT** (**Fog** computing out of the box: **D**ynamic d**E**ployment of **F**og service containers with **T**OSCA), to provide user-friendly development and deployment paradigm for fog applications. With FogDEFT, an application developer will describe the fog service's conceptual structure (or blueprint) with the standard TOSCA Service Template. Then a lightweight orchestrator tool will take TOSCA Service Templates and dynamically deploys the services on a set of fog devices (with a single command). This deployment means starting one or more Docker Containers (in swarm mode or could be in standalone mode) in such a way that containers can talk to each other like a typical microservice architecture and exchange sensor data and actuation signals seamlessly, irrespective of their hardware architecture and placement inside a fog federation.

1.2 | Contributions

The main contributions of this paper are as follows: (i) developing a novel versatile, dynamic deployment framework for fog computing, which is adaptable to any application use cases; (ii) creating node and relationship types for describing fog services in TOSCA for any domain-specific fog application; (iii) relevant actuation scripts for created node and relationship types; (iv) creating TOSCA Service Template for any domain-specific custom/user-developed services; (v) dynamic deployment/undeployment of services on top of fog nodes on the fly with a single command; (vi) demonstrate deployment of fog application across multiple networks.

The rest of the paper is organized as follows. Section 2 briefly discussed related work in this domain. The main challenges and problem description is discussed in Section 3. Section 4 discusses fog device organization and the techniques to address the heterogeneity of these devices. Section 5 contains the background of TOSCA and the process of standardizing fog services with a case study. The performance and resource utilization are discussed in Section 6. Finally, Section 7 discusses the conclusion and future work.

2 | RELATED WORK

The deployment of fog and edge applications has been studied extensively during the past decade. The literature in the domain can be categorized as follows.

2.1 | Dynamic deployment of applications

Ferry et al[11] carried out a case study of GeneSIS in smart buildings. They showed that GeneSIS could support security by design from the development (via deployment) to the operation of IoT systems and keep up security and adapt to evolving conditions and threats while maintaining their trustworthiness. Hassan and Qasha[12] propose a new approach to generate a deployable model for the distributed IoT systems based on a simplified, user-friendly declarative description of the smart devices' communication, configuration, installation, and computation with the IoT system parts with

*https://yaml.org.

Ansible-based YAML description. The work minimizes the efforts for the deployment of the distributed IoT applications on various infrastructures, including the cloud. Tomarchio et al[13] proposed a TOSCA-based framework, "TORCH," for deploying and orchestrating classical and containerized cloud applications on multiple cloud providers. The main benefit of the framework is the possibility to add support to any cloud platform at a very low implementation cost, and it allows deployment management through a simple web tool. Dautov et al[14] propose a hierarchical architecture for provisioning software updates from the cloud to terminal devices via edge gateways in a targeted manner through a last-mile deployment agent, placed on edge gateways via the centralized cloud in the form of containerized microservices, that receive firmware updates from the cloud and install them on connected IoT devices at the edge. Song et al[15] describe joint research on an industrial use case with a Smart Healthcare application provider on a model-based approach for automatically assigning multiple software deployments to hundreds of Edge gateways (fleet) and uses a set of hard and soft constraints to achieve correct, even distribution of software variants.

2.2 | Deployment of fog applications

Donassolo et al[16] proposed another orchestration framework called "FITOR," an automated deployment and microservice migration solution for IoT applications. The framework uses Optimized Fog Service Provisioning (O-FSP) based on a greedy approach that outperforms other relevant strategies in terms of (i) acceptance rate, (ii) provisioning cost, and (iii) CPU usage. Ferry et al[17] developed a framework for continuous deployment for decentralized processing across heterogeneous IoT, edge, and cloud infrastructures called Generation and Deployment of Smart IoT Systems (GeneSIS). GeneSIS provides (i) a domain-specific modeling language to model the orchestration and deployment of Smart IoT Systems and (ii) an execution engine to support automatic deployment across IoT, edge, and cloud infrastructure resources. Venticinque and Amato[18] proposed a new fog service placement methodology. The methodology's effectiveness is demonstrated in the energy domain with smart grid. Davoli et al[19] developed a modular orchestration system called "FORCH." The orchestrator is aware of different service models (SaaS/PaaS/IaaS) and dynamically deploys services and manages resources on the fog nodes. Sami and Mourad[20] proposed a new framework for deploying fog service on-demand on the fly based on Kubeadm and Docker with the presence of volunteering devices. Moreover, the framework optimizes the container placement problem with an Evolutionary Memetic Algorithm (MA) that uses heuristics to make decisions. Sami et al[21] proposed an efficient resource and context-aware approach for deploying containerized microservices on-demand called Vehicular-OBUs-As-On-Demand-Fogs. The scheme embeds adaptable networking architecture combining cellular technologies and the vehicular ad-hoc wireless network (802.11p) and a Kubeadm-based approach for clustering with docker container-based microservices deployment. The solution provides an on-demand fog and service placement solution on vehicles based on an evolutionary memetic algorithm.

2.3 | Dynamic deployment of fog applications

Hoque et al[22] have carried out a technical evaluation of docker container and container orchestration tools, their capability, limitations, and how containerization can impact application performance. The result shows that significant adjustments are required to meet the fog environment needs, and they have proposed a framework based on the docker swarm to address issues with the help of "OpenloTFog" toolkits. In 2013, Li et al. put the first effort to use TOSCA, the new cloud standard, for IoT applications and demonstrated the feasibility of modeling IoT components gateways and drivers for building Air Handling Unit (AHU) with the first edition of TOSCA.[23] da Silva et al[24] automatically deployed an IoT application with OpenTOSCA based on Mosquitto Message Broker running on the cloud, and the publishers and subscribers were running in two different Raspberry Pis. Later, in,[25] they automatically deployed an IoT application out of the box where a python script on Raspberry Pi pushes the data to the message broker on a cloud, and another virtual machine is hosting a web-based dashboard to present the sensor data. They validated this deployment with three case studies of emerging middleware (i) Eclipse Mosquitto, (ii) FIWARE Orion Context Broker, and (iii) OpenMTC. In,[26] A Tsagkaropoulos et al. presented TOSCA extensions for modeling applications relying on any combination of technologies and discussed semantic enhancements, optimization aspects, and methodology that should be followed for edge and fog deployment support. Furthermore, added a comparison with other cloud application deployment approaches. In,[27] HE Solayman and RP Qasha, used TOSCA for deploying IoT applications for Intensive Care Unit (ICU) based on Docker Containers. The demonstration shows automation of IoT application provisioning in

heterogeneous environments consisting of hardware components and cloud instance message broker for network communication between components containerized with docker containers. ZN Samani. et al. proposed idea of resource aware positioning in multilayered approach. That minimize the resources wastage to support maximum deployment in dynamic fog.[28] M Chahoud et al. in[29] discussed the real time on demand deployment of services for Federated Learning (FL) application with docker container and Kubernetes in client devices.

Therefore, almost all the related work discussed and summarized in Table 1 falls into these categories: (i) Fog nodes are preconfigured with some agent to assist the service deployment. (ii) New architecture not following well-known or already adopted standards. (iii) Not versatile enough to fit any fog application and deploy on the fly. (iv) Based on the cloud-centric IoT application, as it was the de-facto architecture over past decades. Unlike fog, the cloud provides virtually infinite resources. Hence, in most cases, the adoption of fog might not work without re-engineering.

Therefore, here we propose the idea of "fog out of the box" where the applications services are dynamically deployed on the fly without any cloud involvement (except the cloning of the code or container images from registry/repositories). The fog federation is dynamically created with a standardize TOSCA Service Template and container orchestration tool, Docker Swarm. The orchestration process will either run on the fog node itself or any PC/laptop/workstation that remotely orchestrates services on the fog nodes.

3 | PROBLEM DESCRIPTION

The development of a framework for deploying standardized user-developed services out of the box throws a few challenges. These challenges can be summarized into two categories:

TABLE 1 A summary of related work and their focus.

Work	Architecture framework	Performance	Virtualization	Agent	DPCA	Service placement	TOSCA
Ferry et al.[11]	✓		✓	✓			
Hassan and Qasha[12]	✓	✓					
Tomarchio et al.[13]	✓	✓	✓				✓
Dautov et al.[14]			✓	✓			
Song et al.[15]	✓	✓	✓	✓			
Donassolo et al.[16]	✓	✓			✓	✓	
Ferry et al.[17]	✓		✓	✓			
Venticinque and Amato[18]						✓	
Davoli et al.[19]	✓						
Sami and Mourad.[20]			✓		✓	✓	
Sami et al.[21]	✓		✓		✓	✓	
Hoque et al.[22]	✓	✓	✓				
Li et al.[23]							✓
da Silva et al.[24]							✓
da Silva et al.[25]							✓
Tsagkaropoulos et al.[26]							✓
Solayman and Qasha[27]			✓				✓
Samani[28]	✓				✓		
Chahoud[29]	✓		✓	✓	✓		
FogDEFT	✓	✓	✓				✓

Abbreviations: DPCA, deployment/placement/cost.

TABLE 2 Collection of devices used for experiments.

Device name	Processor architecture	Memory size	Operating system
Raspberry Pi 3 Model B	ARMv7l 32-bit	1 GB	Raspbian OS
Raspberry Pi 4	ARMv7l 32-bit	4 GB	Raspberry Pi OS 11 & 10
Raspberry Pi 4	ARMv8 64-bit	4 GB/8 GB	Raspberry Pi OS 11
PC and workstations	Intel x86-64 AMD64	4 GB/8 GB	Ubuntu 20.04 Debian 11
Virtual machines	Intel x86-64 AMD64	2–4 GB	Ubuntu 20.04 Debian 11

3.1 | Interoperability and platform independence over heterogeneous hardware

The idea of a fog federation is to include all devices available locally on-premises for computational purposes. Then the local processing task is handled with the accumulated computational capability of these devices. The locally available devices primarily include network and gateway devices and some on-premises core computational devices (e.g., from private clouds). These devices are built by different manufacturers, based on specific hardware architecture, optimized for particular tasks, and could run on individual software. For example, Table 2 shows the collection of heterogeneous devices we used for our experiments.

These are all diverse types of hardware commonly we come across in typical scenarios. Each of them has a different type of features, capabilities, and performance. The Raspberry Pi has 40 GPIO pins to interact with sensors and actuators. However, personal computers do not have that capability. Personal computers have much more processing power than Raspberry Pi. Even comparing Raspberry Pi 3 and Raspberry Pi 4, there are also significant differences in the processing power. Therefore, a significant amount of heterogeneity over hardware and software can be envisioned, as relevant to the fog infrastructure.

So, while orchestrating the services on such fog devices, we must be able to place these services across the nodes (heterogeneous devices) based on their capabilities and requirements of services. Therefore these services must serve their purpose independent of their platform. Furthermore, these services should be able to talk to each other like a conventional microservice architecture irrespective of the placement of the containers over the fog devices, which will ensure interoperability across the fog federation.

3.2 | Modeling of user-developed services with TOSCA

TOSCA's development acknowledges the issue of standardization and portability of cloud applications. TOSCA and relevant basics are discussed in Section 5. EU H2020 RADON Project[†] extended TOSCA and provides reusable TOSCA types of application runtimes, computing resources, and function as a Service (FaaS) platforms in the form of abstract and deployable modeling entities.[‡30] Another extension of TOSCA called TOSCAData focuses on modeling data pipeline-based cloud applications.[31] Because TOSCA is platform agnostic, the language provides an extension mechanism to extend the definitions with additional vendor-specific or domain-specific information. We can extend/adapt the TOSCA to model user-developed fog applications.

Therefore, supporting fog service with TOSCA requires the creation of appropriate nodes and relationship types. These nodes and relationship types will represent the components of the fog services. Then a developer can write a TOSCA Service Template with node and relationship types for user-developed fog applications. A lightweight TOSCA-compliant orchestrator can take the TOSCA Service Template and deploy the services on the fog nodes out of the box.

4 | FOG DEVICE ORGANIZATION AND ARCHITECTURE

All the fog devices are in the same network (could be wired LAN or wireless WiFi) as shown in Figure 1. All the VirtualBox (if present) network configurations should be bridged adapters in their network settings, so they behave like

[†]https://radon-h2020.eu/.
[‡]https://github.com/radon-h2020.

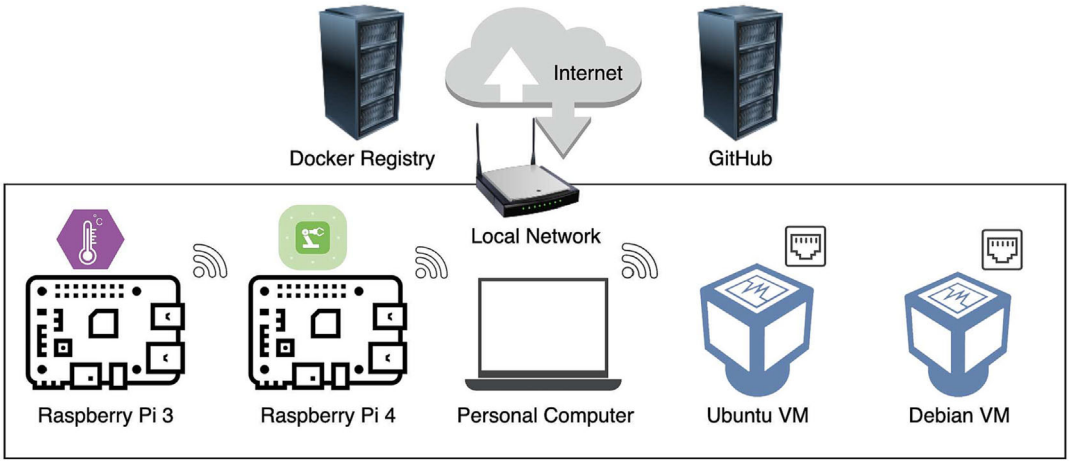


FIGURE 1 Fog devices organization in local area network.

physically connected to the interface using a network cable. On-demand, all the devices should have internet access to pull the docker images from the docker registry or clone source code from the online repository.

4.1 | Addressing heterogeneity over fog federation

4.1.1 | Addressing heterogeneity with Docker

Virtualization is the key technology for achieving platform independence over heterogeneous devices. Plenty of mature solutions are available for hardware virtualization. However, in this context, fog devices are resource-constrained devices. The conventional hardware virtualization takes enormous resources and adds significant overhead to the systems that do not suit fog computing requirements. Therefore, OS-level virtualization offered by containerization technology is the key. This containerization technology uses kernel features, control groups or cgroups (limits and accounts for the system resources), and namespace (logical isolation from the host system with scope) to create logical isolation inside a host system. These containers are decade-old technology such as LXC, LXD, and LXCFS. However, setting up such containers on the fly is difficult as they are deficient. Here, Docker[§] (Docker uses LXC[¶]) comes in and provides a user-friendly high-level tool with many functionalities to make setting up containers easier for the end-users like us. These containers take the kernel support from the host machine and run with minimal overhead and possible to start containers within seconds in these resource-constrained fog devices.

4.1.2 | Portability over heterogeneous CPU architectures with Buildx

The conventional way to shift a docker container is as a docker image. A docker image is created at the developer end and pushed to the Docker Registry. Then we can start the container by pulling the image from the Docker Registry at the deployment end. Usually, the development and deployment sides used to have the same or different chips based on the same architecture (e.g., Intel Core at developer and Intel Xeon at production). However, fog devices are heterogeneous and consist of different processor architectures. An image built on one processor architecture is not going to work on another processor architecture. A straightforward example is Intel Core at the developer end and an ARM-based processor on a Raspberry Pi.

The solution to this problem is through Docker Multi-architecture Manifests. Creating images of different architectures of the same application component to support multiple architectures and docker automatically pulls the compatible images from the Docker Registry while deployment. There are two ways to build such types of images. (i) The old

[§]https://docs.docker.com/.
[¶]https://linuxcontainers.org/.

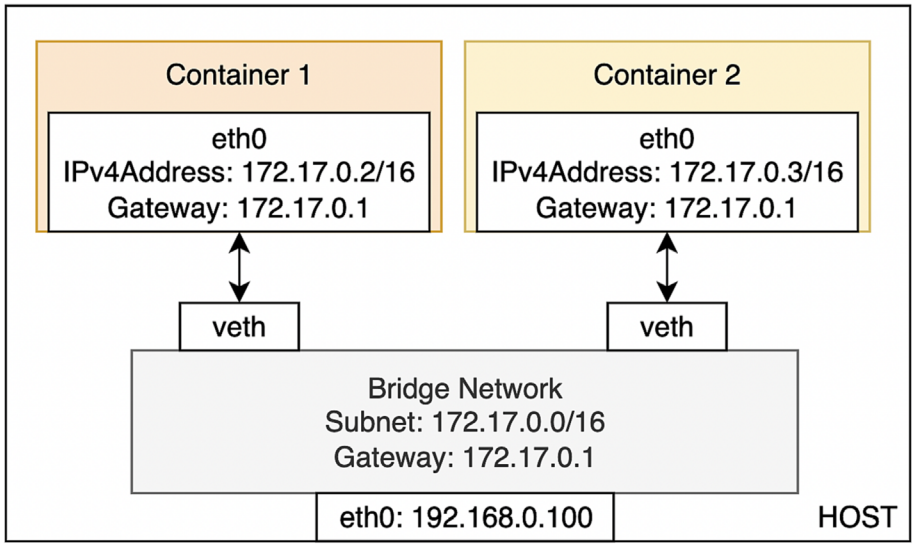


FIGURE 2 Networking inside a Docker host.

way is to build the image on each architecture natively, and then create a combined manifest file and push it to the Docker Registry. (ii) With the new approach, Docker introduces a CLI tool Buildx[#] to build multiarchitecture images, combine them in a manifest file, and push them to the Docker Registry with a single command. Buildx uses QEMU emulation support from the Linux kernel to emulate multiple processor architectures and build multiarchitecture images parallelly.

However, there could be some issues, like not every base image supports multiarchitecture. For some uncommon applications, Buildx may fail for many reasons, such as the unavailability of dependency. In that case, the older way is the only way to go.

4.1.3 | Interoperability over heterogeneous devices with swarm

Let us assume all our fog devices are installed with Docker Engine so we can treat them as Docker Host.

A Bridge Network[‖] (`172.17.0.0/16`) will be created on each Docker Host by default. After starting, all containers get attached to this Bridge Network. In Figure 2, Container 1 and Container 2 get the IP address `172.17.0.2`, `172.17.0.3`, respectively. On a Docker Host, these containers can talk to each other through their IP addresses of each other. If the Bridge Network is user-defined, that is, by default DNS enabled, the container names also can be used to communicate with each other.

However, in fog computing, we are supposed to use many fog devices. Then how will the communication occur between the containers running on different Docker Hosts? That is where Overlay Network[**] comes into the picture. It is a shared network over all the Docker Hosts; we used Container Orchestration Tool called Docker Swarm.

Docker Swarm[††] (a pool of Docker Hosts virtually acts like a single machine from an external view) is a native clustering engine for or by Docker. Any service in a standalone Docker Container can equally run well in swarm mode (there is some limitation to be discussed in the upcoming Section 4.2.1). Docker Swarm, by default, creates an Ingress Network. This Ingress Network is one type of Overlay Network spread across all Docker Hosts joined in the Swarm shown in Figure 3. It has an inbuilt loadbalancer and routing mesh.[‡‡] The loadbalancer distributes the traffic across the multiple replicas of a service running in different Docker Hosts. The routing mesh automatically redirects the traffic to

[#]https://docs.docker.com/buildx/working-with-buildx/.
[‖]https://docs.docker.com/buildx/working-with-buildx/.
[**]https://docs.docker.com/network/overlay/.
[††]https://docs.docker.com/engine/swarm/.
[‡‡]https://docs.docker.com/engine/swarm/ingress/.

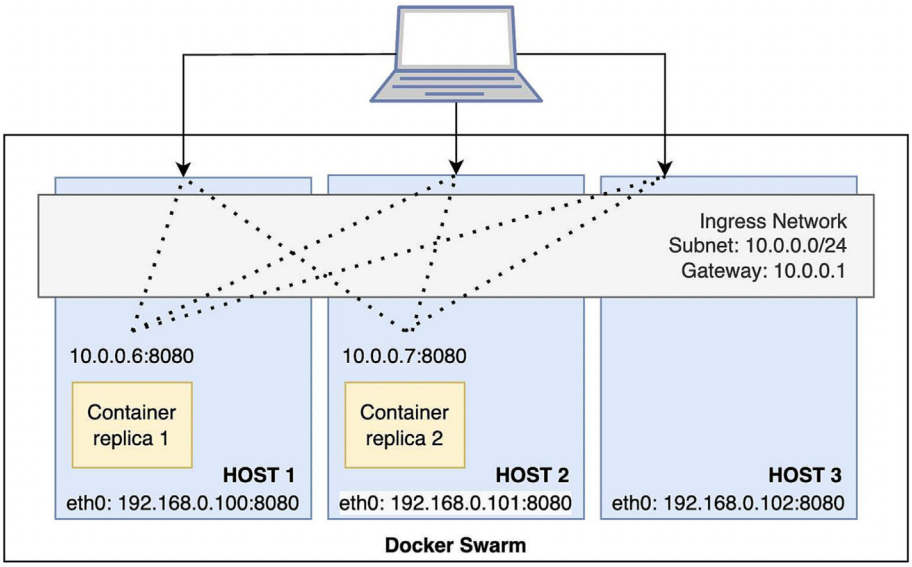


FIGURE 3 Networking across multiple Docker hosts in (Swarm mode)

that specific container where the service is running. Therefore, the service becomes available through all the IP addresses of fog devices. So all the fog devices can use any service running in the Swarm of fog nodes without knowing the actual device where that container is running. This communication is private traffic inside Swarm. With port mapping service made available to the external devices through all the IP addresses of fog nodes. The port mapping is similar to the bind-mount in the file system. The container's internal port is mapped with one host port. Any traffic coming to that docker host port is redirected into that mapped container's internal port. In Figure 3, HOST 1, 2, and 3's port `8080` of Docker Host are mapped with the docker service port `8080`. Any traffic coming to ports `8080` of HOST 1, 2, and 3 is redirected to containers running inside HOST 1 and HOST 2. The HOST 3 is not running the container, but still, service is available through `192.168.0.102:8080` because of the routing mesh.

4.2 | Addressing different capabilities over heterogeneous hardware

Here, capability is not about the processing, but about the capability of sensing and actuation. All fog devices included in the fog federation are not having the hardware support to interface a sensor and actuators. Even this issue is not about the placement of the container. Docker Swarm offers functionality (labels and constraints) to specify the suitable Docker Host to deploy a container. The problem is with containerizing the sensing and actuation components. The container is OS-level virtualization technology to create logical isolation from the platform. However, sensing and actuation required direct platform and hardware interaction. Therefore, it is a conflict of interest. For the processing and communication (web server or message broker) components of an application, platform independence is achievable. However, we are still heavily dependent on the hardware platform for sensing and actuation.

Here are three issues we came across while experimenting:

1. Secure computing mode is a kernel feature that restricts the actions inside a container. By default, out of 300+ system calls, 44 system calls are disabled.[§§]
2. We have to interact with the hardware directly for sensing and actuation, but the container is an isolated environment.
3. Installation of packages such as "`RPi.GPIO`" inside a container fails because the base image is regular Linux distributions (e.g., ubuntu:20.04 and debian:buster-slim). So, the platform is unsupported.

We figured out these two ways to dynamically deploy a sensing and actuation service on a Raspberry Pi.

[§§] https://docs.docker.com/engine/security/seccomp/.

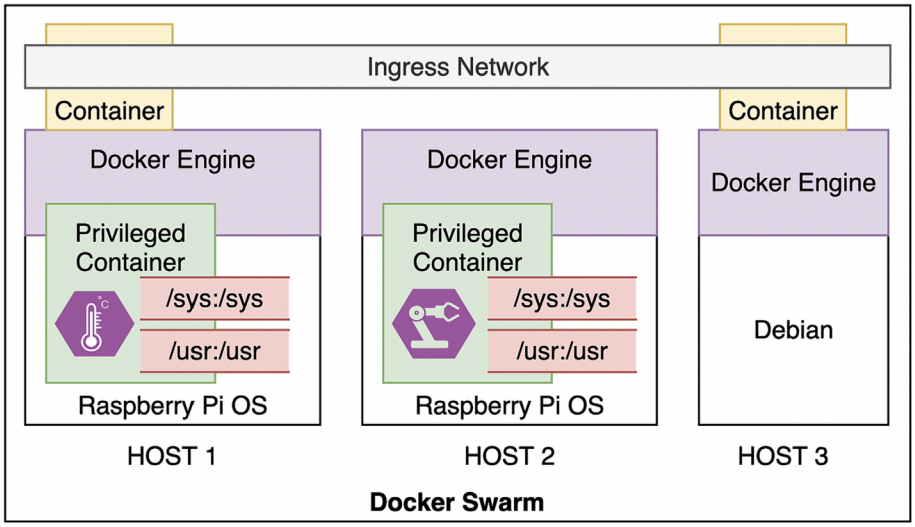


FIGURE 4 Privileged containers with bind-mount on a Docker host.

4.2.1 | Privileged container with bind mounts

These three issues of containerizing sensing and actuation are possible to address as follows.

1. All docker containers are unprivileged by default, limiting the host machine's kernel features and system calls. Docker enables access to entire kernel features on a host for a privileged container. A privileged container can load a kernel module and function as same as a process natively running in a host machine.
2. Linux treats everything as a file. The path to the Raspberry Pi's GPIO pin is "`/sys/class/gpio`." So we bind-mounted[¶¶] the "`/sys`" directory of the container to the "`/sys`" directory of the Raspberry Pi. Bind mount means the container will mount one directory from the host machine, so here containers "`/sys`" directory is mapped with Raspberry Pi's "`/sys`" directory.
3. Similarly, All the libraries and packages are installed on the Raspberry Pi natively. That directory is bind-mounted inside the container ("`/usr`" directory is bind-mounted with the container's "`/usr`" directory).

After these three steps, executing a script inside a container for sensing and actuation becomes possible shown in Figure 4.

However, there are some limitations in Docker Swarm. We get all the features in a standalone container; a few are not available in Swarm mode. The privileged container is one such feature that is not available in Docker Swarm mode yet. So, the sensor and actuation service deployment must be as a standalone container. This standalone container can communicate with other services through the host IP address or any IP address of the fog node inside Docker Swarm. The routing mesh will redirect the traffic to the correct container. This communication is acting as an external device communicating from outside with the services running in the swarm. Otherwise, we can create another Overlay Network and attach the docker services running and standalone container.

4.2.2 | Native background service

In the containerization approach, the container is just providing the source code. Almost everything else is entirely dependent on the platform. This is unnecessarily adding up overhead on the fog devices. Therefore, it is better to execute natively on the host machine as a background service. Therefore instead of starting a container, a background service of sensing or actuation will be created and started on the host machine operating system during deployment shown in Figure 5. This communication with the services running in the swarm is possible through any IP address of

[¶¶] https://docs.docker.com/storage/bind-mounts/.

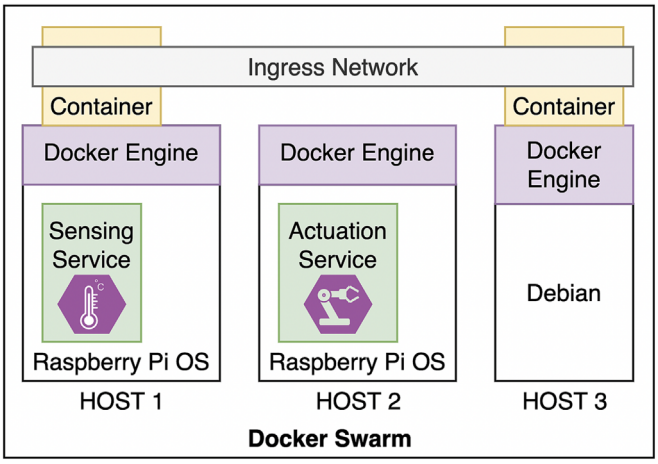


FIGURE 5 Background service for sensing and actuation.

the hosts in the swarm cluster exactly like the previous method. Our experiment tells us that this mechanism is much more robust and resource-efficient than the containerization process discussed in Section 4.2.1.

It is important to note that all these operations, creating Docker Swarm, joining all the fog devices to the swarm cluster, deploying the services, starting privileged containers, or background services, will be completely automatic on the fly without any human intervention. The following section discusses the orchestration and modeling of the fog application with TOSCA and deployment on fog devices out of the box.

5 | TOSCA BACKGROUND AND MODELING THE FOG APPLICATION

TOSCA is a standardized specification to describe software applications to run in the cloud. TOSCA describes not only the application, but also describes the dependency and supporting infrastructure of an application in the cloud. These end-to-end descriptions (from base infrastructure, networks, and software to the running composite application) make TOSCA vendor-independent, and interoperability and portability enabled service definitions for the cloud applications.

TOSCA has two basic building blocks: Nodes and Relationships. Nodes are the infrastructure (servers, networks, virtual machines) or software components (services and runtime environments). Relationships describe the relationship between nodes (a virtual machine hosts a web server). In a TOSCA Service Template (typically a YAML file), we create a Topology Template. The Topology Template consists of Node Templates and Relationship Templates describing the composite application's blueprint.

Each Node and Relationship Templates has a Node Type and Relationship Type. Suppose we relate TOSCA with object-oriented programming concepts. In that case, Node Types or Relationship Types are the classes, and Node and Relationships are the objects. TOSCA Simple Profile v1.3[##] comes with some normative Node Types (Compute, SoftwareComponent, WebServer, WebApplication, DBMS, Database, ObjectStorage), Relationship Types (HostedOn, ConnectsTo, DependsOn, AttachesTo, and RoutesTo), Capabilities Types, Data Types (string, integer, list, dictionary). For modeling, user-developed applications custom Node Types, Relationship Types, and Capability Types are created by extending available Normative Types. Each Node and Relationship Type can have a set of attributes, properties, requirements, capabilities, and implementation. In the implementation, Nodes interface with the system and execute implementation scripts (Shell scripts, Python scripts, or Ansible Playbook scripts), based on the orchestration platform.[||||] A Node's lifecycle is maintained with associated implementation scripts with interface operations: create, configure, start, stop, and delete.

It is important to note that TOSCA is just a standard. The only thing TOSCA gives is a string for the implementation, and it is up to the orchestrator to interpret that string and make sense of it.

[##] https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html.
[||||] https://wiki.oasis-open.org/tosca/TOSCA-implementations.

TABLE 3 Nodes and relationship types.

Nodes types	Relationship types
1. Swarm Leader	1. Token transfer
2. Swarm Worker	
3. Docker Service	
4. Docker Container	
5. System Service	

5.1 | Application modeling with TOSCA

The Radon project[***] and xOpera examples[†††] repository consist of a list of TOSCA node types that are publicly available. However, These two projects and the development of TOSCA are for the cloud platforms. Therefore, these node types are for the deployment of pure cloud applications. Hence, significant modifications are required to create relevant node types for deployment of IoT applications on fog nodes. In general, there will be five generic nodes[‡‡‡] and one relationship[§§§] type required for user-defined applications. These nodes and relationship types are listed in Table 3.

1. Swarm Leader: The node initiates the Docker Swarm and works as the swarm manager. The attributes (Listing 1, lines 4–13) specify the joining token and advertise address of the swarm. Swarm leader has the requirement of type Compute, so it must be hosted on a Compute type node (line 15). Furthermore, the node can host any positive number (line 21) of docker services (theoretically). In interfaces, two operations (lines 34–35), `create` and `delete`, are implemented with Ansible Playbooks. The orchestrator will invoke these Ansible Playbook scrips during deployment and undeployment, respectively. Inside the inputs section (lines 26–32), the host node's IP address is concatenated with port "`2377`" will be available as an input inside Ansible Playbooks. Now, it is up to tasks written into Ansible Playbooks to make the deployment and undeployment happen on the host node.
2. Swarm Worker: The node joins the docker swarm as a worker node. Must be hosted on the Compute type node and depend on the Swarm Leader type node specified by the Token Transfer relationship type (Listing 2).
   Swarm Leader and Swarm Worker nodes have a similar capability to host Docker Services. In interfaces, `create` and `delete` operations are implemented with three inputs: `worker_join_token` (to join the Docker Swarm), `ip_address` (Host IP address), and `join_addr_port` (Docker Swarm joining address from Swarm Leader) (Listing 3).
3. Docker Service, Docker Container, and System Service: These three node types are more or less the same. Docker Service and Docker Container pull a docker-compose file from the online repository. The system service pulls the scripts and service configuration files for background service creation. Then again, it is up to the implementation in the Ansible Playbook to make the container or service creation happen on the host system. The only difference is the requirement. Docker Service node type deploys the service in swarm mode, so has the `dependency` on the Swarm Leader nodes and all Swarm Worker nodes inside the swarm (Listing 4).
   Whereas background services and standalone containers run on a single node, the Compute node is the only requirement. Therefore, in Listing 4, the requirement of a `host` is sufficient. The dependency section is not required for Docker Container and System Service.

1. Token transfer: The relationship type specifies the relationship between Swarm Leader and Swarm Worker (Listing 5). Which is derived from normative dependOn relationships. The relationship sets the attributes joining tokens and advertised address of Swarm Leader node with `pre_configure_source` interface operation.

[***] https://github.com/radon-h2020/radon-particles.git.
[†††] https://github.com/xlab-si/xopera-examples.git.
[‡‡‡] https://github.com/cloud-and-smart-labs/fog-service-orchestration/tree/swarm/orchestrator/tosca/nodetypes.
[§§§] https://github.com/cloud-and-smart-labs/fog-service-orchestration/tree/swarm/orchestrator/tosca/relationshiptypes.

Listing 1 Swarm Leader node type

```
node_types:
 fog.docker.SwarmLeader:
  derived_from: tosca.nodes.SoftwareComponent
  attributes:
   manager_token:
    type: string
    default: undefined
   worker_token:
    type: string
    default: undefined
   advertise_addr:
    type: string
    default: undefined
  requirements:
   - host:
      capability: tosca.capabilities.Compute
      relationship: tosca.relationships.HostedOn
  capabilities:
   host:
    type: tosca.capabilities.Container
    occurrences: [0, UNBOUNDED]
    valid_source_types: [fog.docker.Services]
  interfaces:
   Standard:
    type: tosca.interfaces.node.lifecycle.Standard
    inputs:
     advertise_addr:
      value:
       concat:
        - get_attribute:[SELF,host,private_address]
        - ":2377"
      type: string
    operations:
     create: playbooks/create.yaml
     delete: playbooks/delete.yaml
```

Listing 2 Requirements of Swarm Worker node types

```
requirements:
 - leader:
    capability: tosca.capabilities.Node
    relationship: fog...relationships.TokenTransfer
 - host:
    capability: tosca.capabilities.Compute
    relationship: tosca.relationships.HostedOn
```

Listing 3 Interfaces of Swarm Worker node types

```
interfaces:
 Standard:
  type: tosca.interfaces.node.lifecycle.Standard
  inputs:
   worker_join_token:
    value:
     get_attribute: [SELF,leader,worker_token]
    type: string
   ip_addr:
    value:
     get_attribute: [SELF,host,private_address]
    type: string
   join_addr_port:
    value:
     get_attribute: [SELF,leader,advertise_addr]
    type: string
```

Listing 4 Requirements of Docker Service node type

```
requirements:
 - host:
    capability: tosca.capabilities.Container
    relationship: tosca.relationships.HostedOn
    occurrences: [1, 1]
 - dependency:
    capability: tosca.capabilities.Container
    relationship: tosca.relationships.DependsOn
    occurrences: [0, UNBOUNDED]
```

Listing 5 Interfaces of Token transfer relationship type

```
operations:
 pre_configure_source:
  inputs:
   manager_token:
    value: { get_attribute: [TARGET, manager_token] }
    type: string
   worker_token:
    value: { get_attribute: [TARGET, worker_token] }
    type: string
   join_addr_port:
    value: {get_attribute: [TARGET, advertise_addr]}
    type: string
  implementation: playbooks/pre_configure_source.yaml
```

These are the generic and common types of TOSCA node types. Now, based on the different IoT Applications requirements other types of components may arrive. In that case, these node and relationship types have to be created like the way these five nodes and one relationship types are created.

5.2 | Case study and demonstration

The capability and versatility of the proposed framework have been demonstrated in our previous work[32] with a climate control system for a convention center. In that case study, publisher and subscriber service through a message broker is dynamically deployed on the fly on a complex heterogeneous hardware architecture, that is, Arduino Uno (Microcontroller ATmega328P), Arduino Nano 33 BLE Sense (ARM®Cortex®-M4) connected via serial port of two Raspberry Pi 4 (ARMv7l 32-Bit, Raspberry Pi OS 10) and Raspberry Pi 4 (ARMv8 64-Bit, Raspberry Pi OS 11). Raspberry Pis and Workstation (Intel Xeon W-2145 3.70 GHz × 16) are connected via LAN. Sensors and actuators are connected to Arduino only. Therefore, Raspberry Pis interact with sensors and actuators via Arduino through the serial port and seamlessly exchanges sensing and actuation signals via the message broker. The workstation hosts a control interactive dashboard that showcases sensor data.

In this paper, we produce a much simpler case study to demonstrate the proof-of-concept. Thus, the scope of this paper is limited to (i) the internal mechanics of this versatile framework and (ii) a proof of concept with the dynamic deployment of a simple IoT application, that is, operating LEDs connected through GPIO pins of a few Raspberry Pis with a button from a web page. Five different Node Templates are creating the blueprint of this application as follows:

- Compute Node: all the fog devices.
- Swarm Leader: Docker Swarm manager.
- Swarm Worker: Docker Swarm workers.

- Docker Services: Hosts an Nginx web server, and serves the webpage and Python WebSocket server from where the actuation signal is forwarded to the actuators.
- Docker Containers: LED actuator: turn LED on/off.

5.2.1 | Topology model

The visual blueprint of the structure of this application's TOSCA Service Template would be like Figure 6. The Fog nodes are the fog devices. On top of that, Swarm Leader and Swarm Workers are hosted. Swarm Workers depend on the Swarm Leader. Docker Service is hosted on the Swarm Leader with dependency on all Swarm Workers. Two Docker Services, Python Websocket Server and Nginx web server run as Docker Stack on the Docker Swarm (Swarm Leader and Swarm Workers). However, the Nginx web server depends on the Python WebSocket server mentioned in the docker-compose file. So Nginx webserver service will be deployed after deploying the python WebSocket server. Therefore, LED actuators are hosted on each fog node but depend on Docker Services. So LED Actuators will be deployed after the Docker Services is ready.

5.2.2 | TOSCA service template

The TOSCA Service Template of the application contains the Topology Template given in Listing 6, that describes the application's structure given in the previous section Topology model (Figure 6). The Node Template section defines (lines 2–48) all application components and their properties and dependencies.

The Privileged Containers Node Templates need to be replaced by the System Service Node Templates for background services like Listing 7.

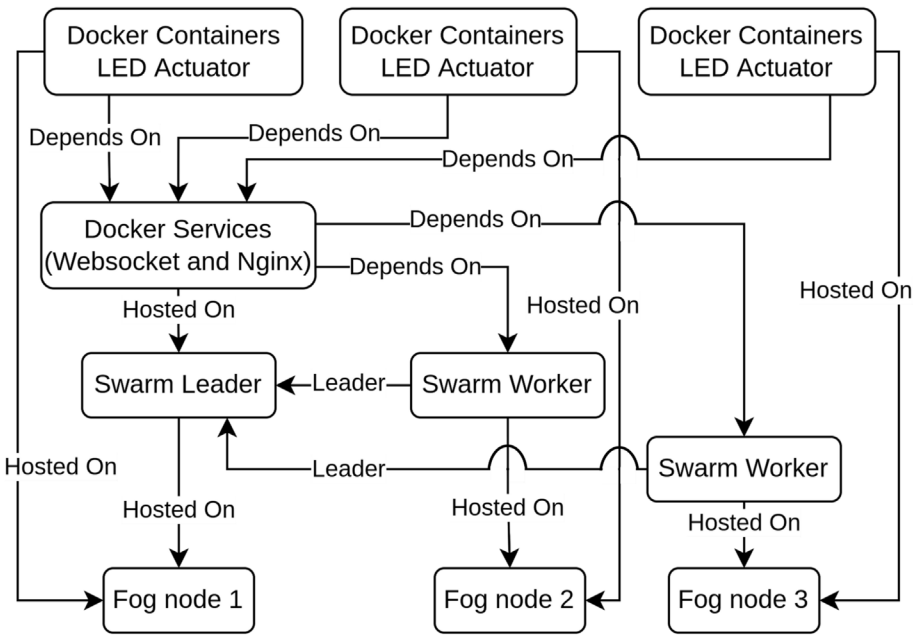


FIGURE 6 Topology or blueprint of the application.

Listing 6 TOSCA Service Template of the application

```
topology_template:
 node_templates:
  fog-node-1:
   type: tosca.nodes.Compute
   attributes:
    private_address: 192.168.0.166
    public_address: 192.168.0.166
  fog-node-2:
    ...
  fog-node-3:
    ...

  docker-swarm-leader:
   type: fog.docker.SwarmLeader
   requirements:
    - host: fog-node-1

  docker-swarm-worker-1:
   type: fog.docker.SwarmWorker
   requirements:
    - host: fog-node-2
    - leader: docker-swarm-leader
  docker-swarm-worker-2:
    ...

  docker-service-1:
   type: fog.docker.Services
   properties:
    name: { get_input: docker_service_name }
    url: { get_input: docker_service_url }
   requirements:
    - host: docker-swarm-leader
    - dependency: docker-swarm-worker-1
    - dependency: docker-swarm-worker-2

  privileged_container-1:
   type: fog.docker.Containers
   properties:
    name: { get_input: docker_compose_name }
    url: { get_input: docker_compose_url }
    packages: { get_input: packages }
   requirements:
    - host: fog-node-1
    - dependency: docker-service-1
  privileged_container-2:
    ...
  privileged_container-3:
    ...
```

Listing 7 System Service node template

```
system-service-1:
 type: fog.system.Service
 properties:
  name: { get_input: system_service_name }
  script_url: {get_input: system_service_script_url}
  service_url: {get_input: system_service_service_url}
  packages: { get_input: packages }
 requirements:
  - host: fog-node-1
  - dependency: docker-service-1
```

For the inputs, we have a separate file ("`inputs.yaml`") to add the required inputs repository or nodes IP addresses. The file structure of the entire package (node, relationship types, service template, inputs, and Ansible Playbooks) of an application is shown in Figure 7.

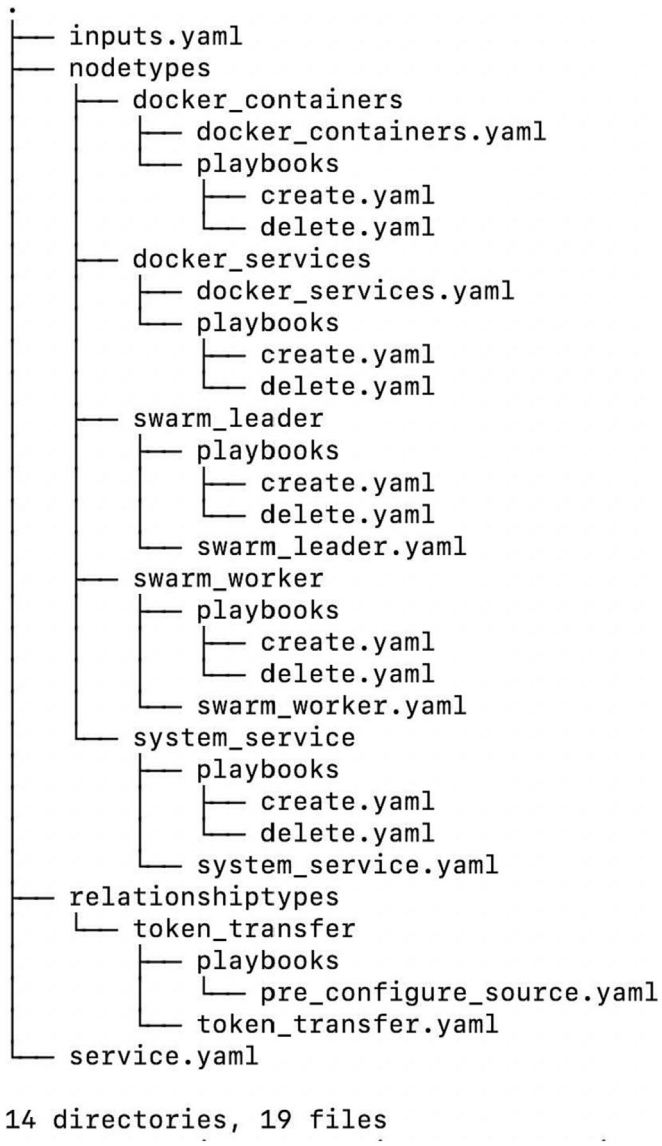


FIGURE 7 Files organization of the framework.

5.3 | TOSCA compliant orchestrator and service orchestration

TOSCA Compliant Orchestrator consists of a TOSCA processor, an engine or tool capable of parsing, validating, and interpreting the Topology Template. Therefore, TOSCA Compliant Orchestrator interprets a TOSCA Service Template to instantiate, deploy, and manage the application. TOSCA Compliant Orchestrator is also called an orchestration engine.

The xOpera[¶¶¶] is a lightweight OASIS TOSCA compliant orchestrator (currently with TOSCA Simple Profile in YAML v1.3) fits for resource-constrained fog devices. It uses the Ansible Automation Tool[###] to implement the TOSCA standard. Hence, all the interface operations are performed with Ansible Playbooks. Therefore, xOpera understands TOSCA Service Template and executes Ansible Playbooks in a particular order based on the node's dependency to make deployment and undeployment happen (Figure 8). This Ansible Playbooks execute on the orchestrator machine and connects fog nodes to push small programs called ansible modules. This connection is over Secure Shell (SSH) by

[¶¶¶] https://xlab-si.github.io/xopera-docs/02-cli.html.
[###] https://docs.ansible.com/.

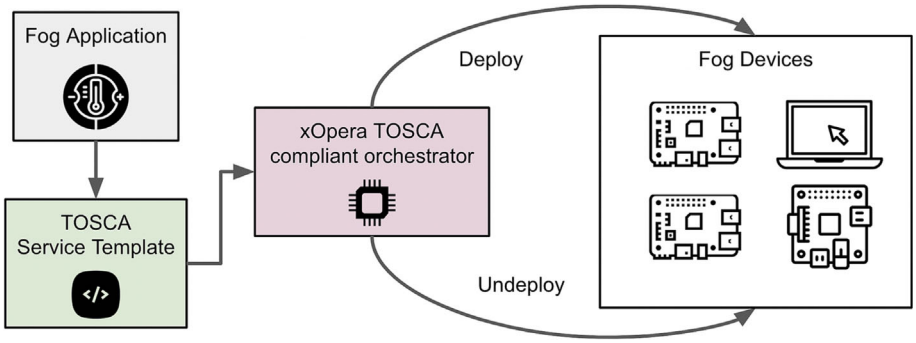


FIGURE 8 Fog service orchestration process.

default. Ansible then executes these modules and then removes them after completion. Hence, the service deployment functions out of the box on the fly.

The xOpera orchestrator and Ansible both heavily rely on SSH infrastructure. The machine where the orchestrator is running must be able to access all the fog devices through the SSH.

5.4 | Orchestration behind NAT

So far, we have discussed service deployment on-the-fly up to the previous subsection. The core mechanism is that all the fog nodes are running SSH servers. The Orchestrator connects to each node through the secure shell. However, the "Out of the Box" core idea is that fog devices are not using exceptional configurations/services. Therefore, standard broadband service connects these devices from the organization/home network to the internet. There are high chances that the Internet Service Provider (ISP) has kept behind the Network Address Translation (NAT) due to the effective use of public IP addresses. It could be multiple layers of NAT in the case of a big organization or urban area. Hence the dynamic deployment is limited to the local network, shown in the organization of the fog federation Figure 1. With this, the scope of work gets stuck into a local area, probably at most one site.

An external coordinator and a modified orchestration tool can bridge this gap. To support this type of deployment, we developed

1. Orchestration Manager (external coordinator): This module is an interactive prompt like a command-line interface running on a static/public IP address. All the orchestrators connect to this manager and act as slaves/workers. The Orchestration Manager broadcasts the commands and configuration. The configuration consists of the name of TOSCA node types and docker-compose files URI.
2. Orchestrator (modified orchestration tool): This Orchestrator is a package of
   - xOpera orchestrator
   - TOSCA node types
   - An agent to connect the Orchestration Manager and TOSCA Service Template generator

   The Orchestrator can also work in standalone mode like a typical xOpera orchestrator. Furthermore, it acts as a slave/worker when connected to the Orchestration Manager. In connected mode when it receives a configuration from Orchestration Manager, it dynamically generates a TOSCA Service Template for that local network behind the NAT.

In our experiment, one of the fog devices runs the Orchestrator as a docker container behind each NAT. The Orchestration manager runs on a Microsoft Azure[IIIII] cloud instance as docker-container. When the respective configuration/command is received, all the Orchestrators dynamically generate the TOSCA Service Template based on the number of fog devices available there and deploy/undeploy on all the fog devices behind that NAT shown in Figure 9.

[IIIII] https://azure.microsoft.com/en-in/services/virtual-machines/.

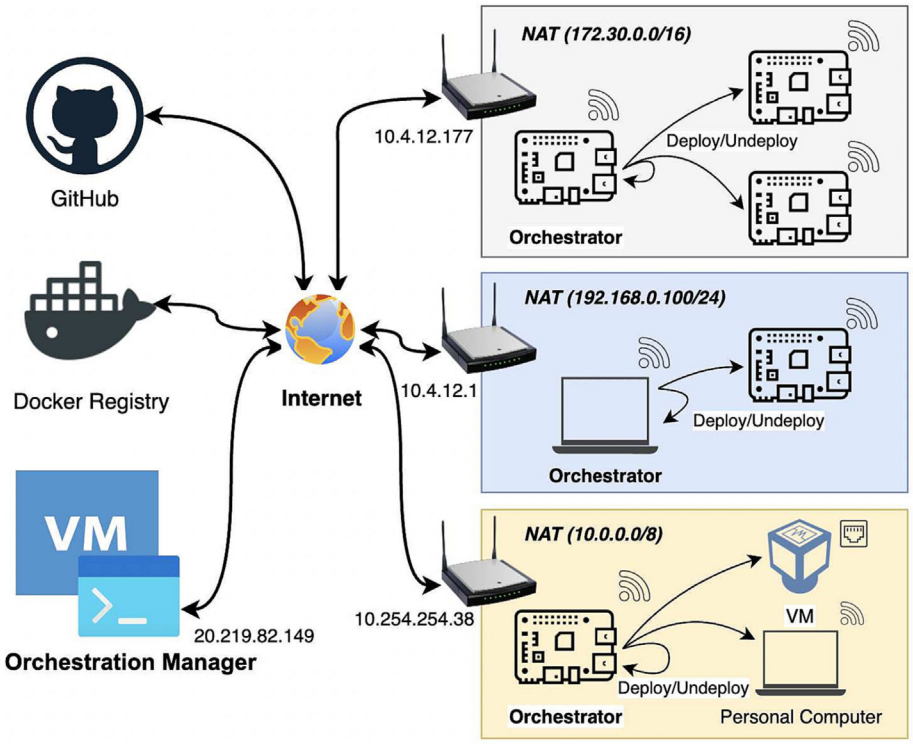


FIGURE 9 Deployment of services behind the NAT.

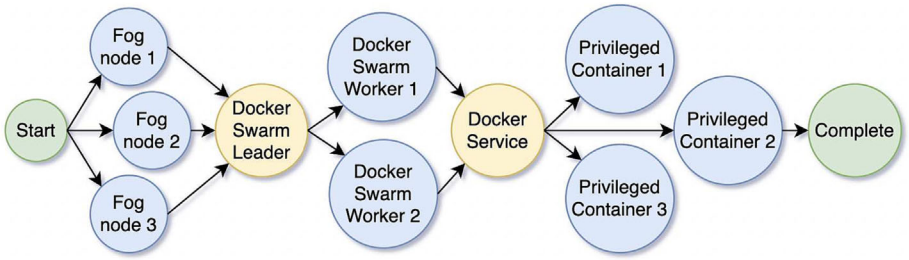


FIGURE 10 Deployment sequence with multiple workers (parallel threads)

6 | RESULTS AND DISCUSSION

The paper's fundamental interest is the evaluation and proof of the concept of dynamic deployment on the fog infrastructure, through the FogDEFT framework. All the extended technologies at their core are meant to serve cloud instances. Therefore, the evaluation results illustrate performance and resource utilization in the fog environment while successively deploying and undeploying the services.

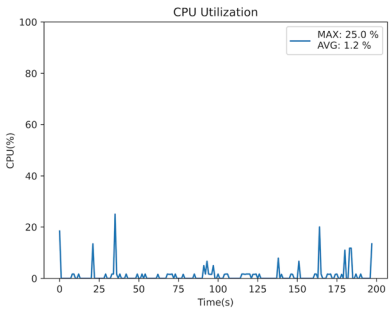
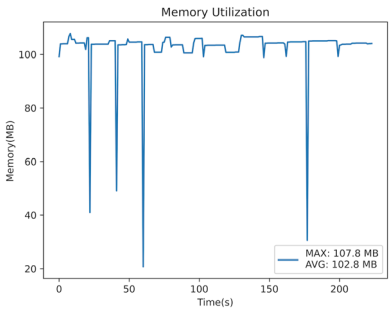
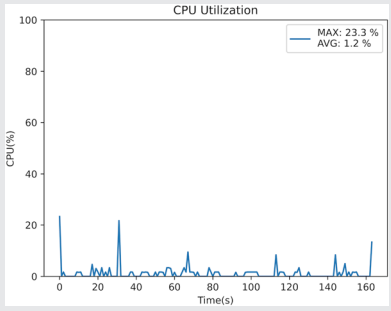
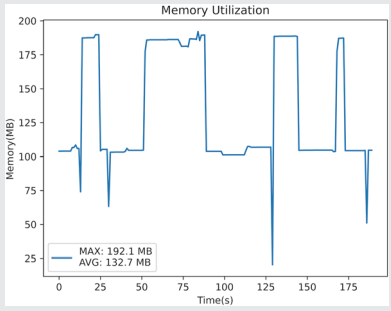
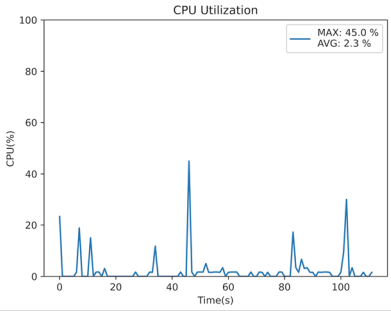
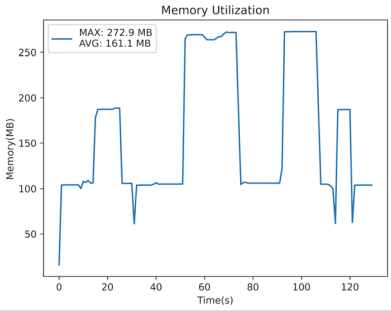
6.1 | Experimental details and assumptions

The experiment was performed in a Local Area Network where the orchestrator node (i.e., personal computer) and fog nodes (i.e., Raspberry Pis) are connected to a WiFi access point, as shown in Figure 1. The orchestrator node has the application service template framework shown in Figure 7 and can interact with fog nodes via SSH. These things for the time being assume all the relevant fog and edge nodes, along with the orchestration tool are present in the same network. This we would like to extend to fog devices across multiple NATs, in our future work, as discussed in Section 7.2.

6.2 | Performance analysis and computational complexity

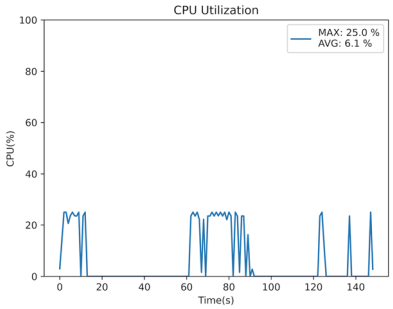
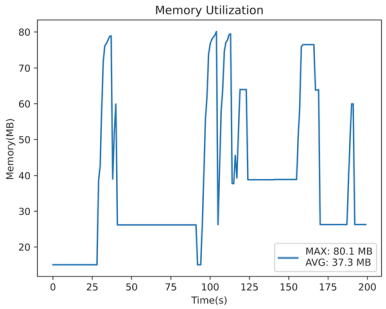
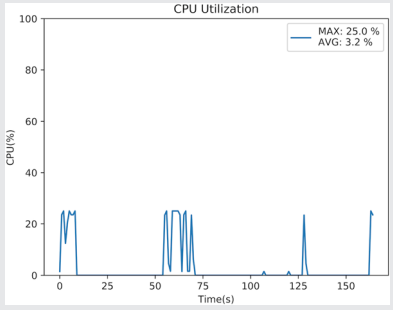
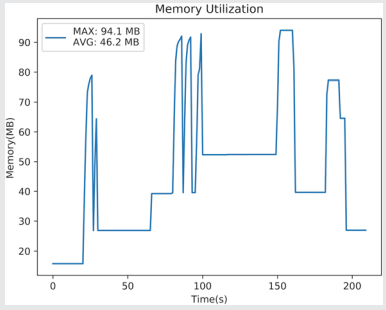
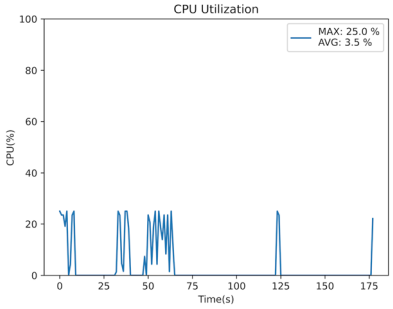
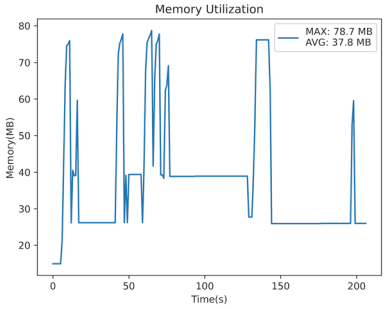
(i) Orchestrator node: Since we used three nodes in the experiment, we employed three deployment threads (workers in the xOpera context) during the orchestration. Based on the dependencies, the orchestrator parallelly deploys the nodes on the fog devices as shown in Figure 10; that is, the fog nodes are first created parallelly in the deployment sequence, followed by the Docker Swarm Leader. All other stuff waited until the completion of the Docker Swarm Leader node because all other nodes directly or indirectly had some dependency on the Docker Swarm Leader node. Then two Swarm Workers are created parallelly, followed by Docker Services that had the dependency on the entire Docker Swarm. After deploying the Docker Service, the orchestrator simultaneously deploys all three Privileged Containers. Similarly, the undeployment sequence is precisely the reverse order of the node's dependency in the Topology Template. The resource utilization of the orchestrator is shown in Table 4. Each row in the table shows the deployment followed by deployment time, CPU, and memory utilization corresponding to the number of deployment threads employed. Therefore, in the case of a single thread, CPU, and memory usage is significantly low but takes the longest time because the deployment order of nodes is sequential. That sequence is always one of the topological sorts of the graph in Figure 6. However, when more deployment threads are employed, deployment and undeployment times decrease significantly, and resource consumption increases drastically (peaks in a CPU and memory usages graph show that), mostly while deploying independent nodes parallelly, as shown in the Figure 10. Therefore, the resource consumption of the orchestrator shows typical time vs. resource trade-offs. However, this is not a matter of concern given (i) the system running the orchestrator is not resource constrained and (ii) a real-world application will not have a high degree of independent services.

TABLE 4 Orchestrator resource utilization and performance.

# W	Time (s)	CPU (%)	Memory (MB)
1	D 139 s U 67 s		
2	D 106 s U 57 s		
3	D 91 s U37 s		

Note: # W: number of workers, D: deploy, U: undeploy.

TABLE 5 Fog nodes resource utilization and performance.

Device	CPU (%)	Memory (MB)
Raspberry Pi 3 Model B (As Worker node)		
Raspberry Pi 4 (As Worker node)		
Raspberry Pi 4 (As Leader node)		

(ii) Fog nodes: The resource utilization details of the fog nodes (where the application services are getting deployed) are shown in Table 5. Interestingly, these results show that irrespective of the devices and their role (Leader/worker) in fog federation, CPU and memory usage is more or less the same and favorable to entitle as a lightweight fog deployment framework. The xOpera uses Ansible Playbook, and Ansible operates on simple, agent-less SSH infrastructure. Hence the service orchestration does not impose any overhead on fog nodes. Also, note that (i) deployment of the service for the first time takes more time for image pulling from the Docker Registry and cloning the code from the repository. The second deployment of the same service will take significantly less time because of cached images/containers on fog devices. (ii) All data represented in Tables 4 and 5 are pure orchestration resource utilization which does not include any other processes or applications we deploy. We ensure this through the custom Python script that explicitly picks the resource consumption of the processes responsible for the orchestration.

7 | CONCLUSION AND FUTURE WORK

We have created TOSCA node types and relationship types for modeling a fog application using TOSCA. Then we deployed an application on heterogeneous fog devices with a TOSCA Service Template with xOpera Orchestrator. Hence, any custom/user-developed application deployment is possible out of the box on the fly with the given TOSCA Service Template. Moreover, Docker containers have become de-facto standards for deploying services with support across various platform architectures. The fog devices have become more potent with the advancement of technologies, so deploying Docker containers with negligible overhead has become a reality. However, there will be some devices that

are left out from Docker support. For these, options for native deployment are always available with the System Service node type. Therefore, it opens plenty of future research areas:

7.1 | Security issue

Deployment of Docker container in privileged mode opens access to the entire kernel of the device. Then bind-mount with the file system and system services opens access to the file system of the host devices. A most straightforward way to handle this security issue is by creating a different user with limited privileges for deploying containers, but it blocks the option to install the required packages on the fly while deploying the service. That could be a future research direction on computing environment.[33]

7.2 | Connectivities

When a service is deployed behind the NAT, it limits the interoperability inside each NAT. Communication between two and more NATs can be established through a cloud with message passing. However, this will also add to the delay, which is a conflict of interest in fog computing.

7.3 | Stateless and stateful applications

In the case studies we carried out most of them are stateless applications. However, stateful applications will store some data on the fog nodes or hold any state of an application. Therefore in these cases, it is crucial to add some features to the framework to migrate the application's state with standardization while moving the application from one fog setup to another one.[34]

ACKNOWLEDGEMENTS

This research is supported by SERB, India, through grant CRG/2021/003888. We also thank financial support to UoH-IoE by MHRD, India (F11/9/2019-U3(A)).

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Satish Narayana Srirama https://orcid.org/0000-0002-7600-7124

REFERENCES

1. Bittencourt, L, Immich, R, Sakellariou, R, et al. The Internet of Things, fog and cloud continuum: integration and challenges. *Int Things*. 2018;3:134-155. https://www.sciencedirect.com/science/article/pii/S2542660518300635
2. Bonomi, F, Milito, R, Zhu, J, Addepalli, S. Fog computing and its role in the Internet of Things. *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12: Association for Computing Machinery; 2012:13-16. https://doi.org/10.1145/2342509.2342513
3. Buyya, R, Srirama, SN. *Fog and Edge Computing: Principles and Paradigms*: John Wiley & Sons; 2019.
4. Vaquero, LM, Rodero-Merino, L. Finding your way in the fog: towards a comprehensive definition of fog computing. *SIGCOMM Comput Commun Rev*. 2014;44(5):27-32. https://doi.org/10.1145/2677046.2677052
5. Biswas, AR, Giaffreda, R. IoT and cloud convergence: opportunities and challenges. In: 2014 IEEE World Forum on Internet of Things (WF-IoT); 2014:375-376.
6. Chang, C, Srirama, SN, Buyya, R. Internet of Things (IoT) and new computing paradigms. In: Buyya, R, Srirama, SN, eds. *Fog and Edge Computing: Principles and Paradigms*: Wiley; 2019:1-23.
7. Srirama, SN. A decade of research in fog computing: relevance, challenges, and future directions. *Softw: Pract Exper*. 2023.
8. Opara-Martins, J, Sahandi, R, Tian, F. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *J Cloud Comput*. 2016;5(1):1-18.
9. Binz, T, Breiter, G, Leyman, F, Spatzier, T. Portable cloud services using TOSCA. *IEEE Int Comput*. 2012;16(3):80-85.

10. Binz, T, Breitenbücher, U, Kopp, O, Leymann, F. TOSCA: portable automated deployment and management of cloud applications. *Advanced web services*: Springer; 2014:527-549.
11. Ferry, N, Nguyen, PH, Song, H, et al. Continuous deployment of trustworthy smart IoT systems. *J Object Technol*. 2020;19(2),16:1-23. doi:10.5381/jot.2020.19.2.a16
12. Hassan, HA, Qasha, RP. Toward the generation of deployable distributed IoT system on the cloud. *IOP Conf Ser: Mater Sci Eng*. 2021; 1088(1):12078. https://doi.org/10.1088/1757-899x/1088/1/012078
13. Tomarchio, O, Calcaterra, D, Di Modica, G, Mazzaglia, P. TORCH: a TOSCA-based orchestrator of multi-cloud containerised applications. *J Grid Comput*. 2021;19(1):1-25.
14. Dautov, R, Song, H, Ferry, N. Towards a sustainable IoT with last-mile software deployment. In: 2021 IEEE Symposium on Computers and Communications (ISCC); 2021:1-6.
15. Song, H, Dautov, R, Ferry, N, Solberg, A, Fleurey, F. Model-based fleet deployment in the IoT–edge–cloud continuum. *Softw Syst Model*. 2022;21(5):1931-1956.
16. Donassolo, B, Fajjari, I, Legrand, A, Mertikopoulos, P. Fog based framework for IoT service provisioning. In: 2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC); 2019:1-6.
17. Ferry, N, Nguyen, P, Song, H, et al. GeneSIS: continuous orchestration and deployment of smart IoT systems. In: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Vol. 1; 2019:870-875.
18. Venticinque, S, Amato, A. A methodology for deployment of IoT application in fog. *J Ambient Intell Humanized Comput*. 2019;10(5):1955-1976.
19. Davoli, G, Borsatti, D, Tarchi, D, Cerroni, W. FORCH: an orchestrator for fog computing service deployment. In: 2020 IFIP Networking Conference (Networking); 2020:677-678.
20. Sami, H, Mourad, A. Dynamic on-demand fog formation offering on-the-fly IoT service deployment. *IEEE Trans Netw Serv Manag*. 2020;17(2):1026-1039.
21. Sami, H, Mourad, A, El-Hajj, W. Vehicular-OBUs-As-On-Demand-Fogs: resource and context aware deployment of containerized micro-services. *IEEE/ACM Trans Netw*. 2020;28(2):778-790.
22. Hoque, S, De Brito, MS, Willner, A, Keil, O, Magedanz, T. Towards container orchestration in fog computing infrastructures. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Vol. 2; 2017:294-299.
23. Li, F, Vögler, M, Claeßens, M, Dustdar, S. Towards automated IoT application deployment by a cloud-based approach. In: 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications; 2013:61-68.
24. da Silva, ACF, Breitenbücher, U, Képes, K, Kopp, O, Leymann, F. OpenTOSCA for IoT: automating the deployment of IoT applications based on the Mosquitto message broker. In: Proceedings of the 6th International Conference on the Internet of Things, IoT'16. Association for Computing Machinery; 2016:181-182. https://doi.org/10.1145/2991561.2998464
25. da Silva, ACF, Breitenbücher, U, Hirmer, P, et al. Internet of Things out of the box: Using TOSCA for automating the deployment of IoT environments. In: Closer; 2017:330-339.
26. Tsagkaropoulos, A, Verginadis, Y, Compastié, M, Apostolou, D, Mentzas, G. Extending TOSCA for edge and fog deployment support. *Electronics*. 2021;10(6):737. https://www.mdpi.com/2079-9292/10/6/737
27. Solayman, HE, Qasha, RP. Portable modeling for ICU IoT-based application using TOSCA on the edge and cloud. In: 2022 International Conference on Computer Science and Software Engineering (CSASE); 2022:301-305.
28. Samani, ZN, Mehran, N, Kimovski, D, Benedict, S, Saurabh, N, Prodan, R. Incremental multilayer resource partitioning for application placement in dynamic fog. *IEEE Trans Parallel Distrib Syst*. 2023;34(6):1877-1896.
29. Chahoud, M, Otoum, S, Mourad, A. On the feasibility of federated learning towards on-demand client deployment at the edge. *Inform Process Manag*. 2023;60(1):103150. https://www.sciencedirect.com/science/article/pii/S0306457322002515
30. Casale, G, Artač, M, Van Den Heuvel, W-J, et al. RADON: rational decomposition and orchestration for serverless computing. *SICS Softw-Intensive Cyber-Phys Syst*. 2020;35(1):77-87.
31. Dehury, CK, Jakovits, P, Srirama, SN, Giotis, G, Garg, G. TOSCAdata: modeling data pipeline applications in TOSCA. *J Syst Softw*. 2022; 186:111164. https://www.sciencedirect.com/science/article/pii/S0164121221002508
32. Srirama, SN, Basak, S. Fog computing out of the box with FogDEFT framework: A case study. In: 2022 IEEE 15th International Conference on Cloud Computing (CLOUD); 2022:342-350.
33. Chen, C-M, Li, Z, Kumari, S, Srivastava, G, Lakshmanna, K, Gadekallu, TR. A provably secure key transfer protocol for the fog-enabled social internet of vehicles based on a confidential computing environment. *Vehic Commun*. 2023;39:100567. https://www.sciencedirect.com/science/article/pii/S2214209622001140
34. Rathod, S, Joshi, R, Gonge, S, Pandya, S, Gadekallu, TR, Javed, AR. Blockchain based simulated virtual machine placement hybrid approach for decentralized cloud and edge computing environments. In: Srivastava, G, Ghosh, U, Lin, JC-W, eds. *Security and Risk Analysis for Intelligent Edge Computing*: Springer International Publishing; 2023:223-236. https://doi.org/10.1007/978-3-031-28150-1_12

How to cite this article: Basak S, Srirama SN. Fog computing out of the box: Dynamic deployment of fog service containers with TOSCA. *Int J Network Mgmt*. 2023;e2246. doi:10.1002/nem.2246