RESEARCH ARTICLE

# Continuous QoS-compliant Orchestration in the Cloud-Edge Continuum

Giuseppe Bisicchia*[1] | Stefano Forti[1] | Ernesto Pimentel[2] | Antonio Brogi[1]

[1]Department of Computer Science, University of Pisa, Pisa, Italy

[2]ITIS Software, University of Málaga, Málaga, Spain

**Correspondence**
*Giuseppe Bisicchia, Corresponding author.
Email: giuseppe.bisicchia@phd.unipi.it

**Present Address**
Department of Computer Science, Largo Bruno Pontecorvo, 3, 56127 Pisa (PI)

**Abstract**

The problem of managing multi-service applications on top of Cloud-Edge networks in a QoS-aware manner has been thoroughly studied in recent years from a decision-making perspective. However, only a few studies addressed the problem of actively enforcing such decisions while orchestrating multi-service applications and considering infrastructure and application variations. In this article, we propose a next-gen orchestrator prototype based on Docker to achieve the continuous and QoS-compliant management of multiservice applications on top of geographically distributed Cloud-Edge resources, in continuity with CI/CD pipelines and infrastructure monitoring tools. Finally, we assess our proposal over a geographically distributed testbed across Italy.

**KEYWORDS:**
Cloud-Edge Continuum; Multiservice Applications; Continuous Reasoning; Continuous Management; Application Orchestration

## 1 | INTRODUCTION

To support the growth of the Internet of Things (IoT) devices, new infrastructural architectures have been proposed, relying on computing, storage and, networking resources along the so-called Cloud-Edge continuum[1]. Most of them – e.g., Fog, Edge, Mist computing[2,3] – are based on the idea of employing computational capabilities closer to application end-users or, more generally, to data sources. Such a continuum is characterised by its high dynamicity, device/connection heterogeneity, and availability of resources[4].

In contrast with the Cloud paradigm, the Cloud-Edge continuum can better support the deployment of next-gen IoT applications, usually featuring strict run-time constraints on, for instance, required IoT devices, latencies, and bandwidth availability (e.g., virtual reality, remote surgery, online gaming)[5]. Indeed, such applications are developed in the form of (possibly)

---

[0]**Abbreviations:** IoT, Internet of Things; CI/CD, Continuous Integration/Continuous Deployment; QoS, Quality of Service.
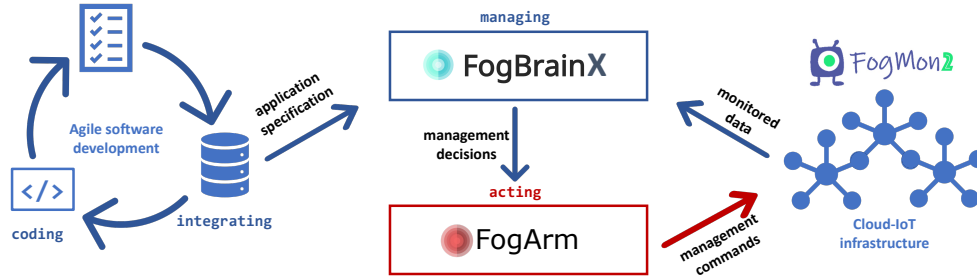
**FIGURE 1** Bird's-eye view of FogArm.

hundreds of interacting microservices, each with its own peculiar requirements. As Cloud-Edge resources, also modern applications rapidly evolve over time, being continuously and collaboratively developed and released through automated tools in the *Continuous Integration/Continuous Deployment* (CI/CD) pipelines[6].

In this context, much literature focused on determining the best QoS- and context-aware placements[1] of multiservice IoT applications to Cloud-Edge infrastructures, by mainly exploiting search-based and mathematical programming solutions, e.g.,[7],[8], and[9]. However, even if the compelling need for QoS-aware methodologies to place and manage application services onto Cloud-Edge infrastructure efficiently is evident[10,11,12,13], most existing proposals only referred to simulated environments due to the lack of orchestration platforms capable of monitoring the needed QoS attributes, and to limited availability of Cloud-Edge testbeds[14]. Particularly, the problem of designing platforms and methodologies for the orchestration and management of multiservice applications in a Cloud-Edge setting is a challenging one, having to deal with the scale and dynamicity of Cloud-Edge networks and of next-gen applications, but has only been marginally addressed.

In light of these needs, new solutions to support the QoS-aware orchestration and management of next-gen multiservice distributed applications suited for Cloud-Edge infrastructures could bring several benefits. To the best of our knowledge, none of the most popular orchestrators for managing digital infrastructures and services (e.g., Docker Swarm, Kubernetes) supports a continuous (i.e., incremental and differential) decision-making process that implements a scalable, QoS- and context-aware orchestration of microservices, ensuring suitable service placement and deployment on top of highly dynamic infrastructures, in continuity with the CI/CD pipeline and always (re-)considering the current infrastructure conditions.

In this article, we design and develop a next-gen prototype orchestrator[2], FogArm, to achieve the continuous and QoS-compliant management of multiservice applications on geographically distributed Cloud-Edge networks, in continuity with CI/CD pipelines and infrastructure monitoring tools. We also assess the performance of our orchestrator over a real-world, geographically distributed testbed.

As shown in Fig. 1 , FogArm, interacts with a monitoring tool (FogMon in our case[15]), and with FogBrainX[16], a declarative *continuous reasoning*[3] the time needed to make informed management decisions. On the other hand, it can reduce the number of management operations needed to adapt the current deployment to the new infrastructure conditions, by avoiding unnecessary service migrations. engine to make informed service placement and migration decisions for next-gen multiservice applications. FogBrainX, through continuous (i.e., incremental, differential) reasoning, provably reduces the time needed to make management decisions when only part of running application deployment is affected by changes in the Cloud-Edge infrastructures (e.g., crash of a node hosting a service, degraded network QoS) or when the application itself changes (e.g., changed application requirements, addition or removal of application services). FogArm integrates these two external tools to enhance its capabilities. FogMon enables FogArm to continuously monitor the state of a distributed, Fog infrastructure and to spot QoS changes (e.g., link degradation, node failure). FogBrainX, instead, is exploited by FogArm to compute *on-demand* suitable management decisions (e.g., migration of a service) to always comply with the QoS application objectives.

To the best of our knowledge, FogArm represents a first complete prototype of a next-gen orchestrator for the continuous QoS-compliant management of multiservice applications on top of geographically distributed Cloud-Edge infrastructures.

---

[1]A *placement* maps each managed microservice to a node of the infrastructure, in such a way all the application's QoS requirements are satisfied.

[2]Freely available at: https://github.com/di-unipi-socc/FogArm

[3]By mainly considering the migration of services suffering due to such changes in the infrastructure or in the application, *continuous reasoning*, permits, on the one hand, scaling to larger sizes of the placement problem by incrementally solving smaller instances of such a problem, thus acting as a booster for existing placement strategies and reducing

The rest of this article is organised as follows. Sect. 2 briefly presents the main peculiarities of the tools exploited by FogArm, viz., FogMon (Sect. 2.1) and FogBrainX (Sect. 2.2). Then, Sect. 3 discusses the architecture and behaviour of FogArm. Sect. 4 illustrates the assessment of FogArm over a real Cloud-Edge testbed. Finally, Sect. 5 briefly discuss related work and Sect. 6 concludes the article by highlighting some possible directions for future work.

## 2 | BACKGROUND

In this section, we briefly present FogMon[15] (Sect. 2.1) and FogBrainX[16] (Sect. 2.2) that are integrated with FogArm to equip it with monitoring and reasoning capabilities, respectively.

### 2.1 | FogMon: Lightweight Infrastructure Monitoring

FogMon is a TRL5 C++, distributed monitoring tool targeting Cloud-Edge computing settings[4]. FogMon measures and statistically aggregates node capabilities (viz., CPU, RAM and HDD) as well as connected and available IoT devices and link QoS (viz., latency and bandwidth). It leverages on a self-organising and self-restructuring peer-to-peer topology, that can run on any TCP/IP network, based on a two-tier *Leader-Follower* architecture and gossiping protocols[17] for communicating among peers.

*Follower* agents have the task of monitoring the capabilities on their associated node. They are divided into groups, each group assigned to a specific *Leader*.

Leaders perform all the tasks of a Follower and periodically aggregate data gathered by the Followers in their group. Furthermore, through gossiping, Leaders share among them the aggregated data collected from their Followers. Moreover, periodically, Leaders publish on a common endpoint a report containing all the gathered information both from their Followers and the other Leaders. The most recent report received is published as the current global report on the monitored infrastructure. The published reports and the communication between peers exploit JSON messages.

Leaders, also compute estimates of bandwidth and latency between Followers belonging to distinct groups. Indeed, Followers inside the same groups directly measure the link performance among them, but to avoid network congestions due to the exponential explosion of possible links between peers, QoS parameters between Followers in different groups are only estimated and not directly measured. In detail, Followers directly measure the network parameters only among Followers in the same group and with their Leader. Furthermore, Leaders directly measure the link performance among them. Thus, the QoS of a link between two Followers in a different group is estimated by composing the measurements between each Follower and its Leader and between the two Leaders.

Peers self-organise into an overlay peer-to-peer network constructed upon a proximity criterion based on latency distances among nodes. Indeed, any new node joins as a Follower and eventually selects its own Leader the one with the minimum measured latency. Periodically, during its activity, or after a failure, a node performs this procedure again to find the best suitable Leader. This approach is designed to face the high dynamicity of the Cloud-Edge continuum and to continuously adapt to a changing environment. For the same reason, also the role of Leader and Follower are dynamically assigned and can vary over time, restructuring the network topology by exploiting the k-medoids algorithm[18]. Finally, the monitored data are also replicated by each Leader that, together with the eventual consistency of such data achieved through gossiping, make FogMon capable of resisting the failure of some Leaders.

FogMon shows a very low footprint in terms both of hardware and bandwidth resources, performing its probing tasks with low overhead. Furthermore, the two-tier peer-to-peer architecture avoids (e.g., due to node or link failures) a single point of failure as well as increases the scalability. FogMon is also released as a Docker image, thus being cross-platform on any Docker-compliant node.

### 2.2 | Declarative Continuous Reasoning in the Cloud-Edge Continuum

FogBrainX[16] is a declarative *continuous reasoning* engine to make informed service placement and migration decisions for next-gen multiservice applications in Cloud-Edge settings. The idea behind the use of *continuous reasoning* is to mainly consider the migration of services in need of attention while preserving as much as possible the placement of the other services.

---

[4]Available at https://github.com/di-unipi-socc/FogMon.

The *continuous reasoning* strategy employed by FogBrainX helps in reducing the time needed to make management decisions while only a part of application deployment is affected by infrastructural changes (e.g., traffic congestion, node failures) or triggers by a CI/CD pipeline (e.g., new service, requirements updates). Through *continuous reasoning*, we can scale our approach to larger applications and infrastructures by incrementally solving smaller instances of the placement problem and at the same time, reducing the number of management operations (e.g., avoiding unnecessary migrations).

FogBrainX reacts to changes in the infrastructure, in the application requirements and service addition and removal. Fog-BrainX is designed as a booster for existing placement strategies, being able to adapt to different approaches, improving their performance and reducing the size of the considered problem.

When FogBrainX is triggered, it verifies if a placement for an application already exists, if not, the default placement strategy (e.g., exhaustive search, heuristics) is applied to find a valid placement. Otherwise, the *continuous reasoning* methodology is performed. In this last case, FogBrainX first determines all services that have been added to the application from the latest commit and the services that have to be migrated (due to infrastructural or requirements changes). Such services are given in input to the default placement strategy to complete the partial placement of the services that have not been migrated, finding a new valid placement.

FogBrainX has been successfully applied through simulations over a lifelike small use case based on real data, and assessed at increasing infrastructure sizes and different variations rates up to thousands of nodes. It showed a speed-up of 50 to 1000× in terms of average inferences across different large-scale infrastructure sizes (i.e., from 320 to 1280 nodes).

## 3 | DESIGN & IMPLEMENTATION OF FOGARM

In this section, we illustrate the architecture and functionalities of FogArm, a next-generation orchestrator prototype designed to perform continuous and QoS-compliant management of multi-service applications on top of highly dynamic and geographically distributed resources such as the Cloud-Edge continuum. Sect. 3.1 discusses the general component-wise architecture of FogArm. Sect. 3.2 illustrates the run-time behaviour of our orchestrator, highlighting the interactions of the components through three main scenarios. Finally, Sect. 3.3 describes the actual implementation of FogArm and its components.

### 3.1 | Architecture of FogArm

FogArm enables:

- the integration with CI/CD pipelines and infrastructure monitoring tools (e.g., FogMon [15]),

- the execution of management decisions made by FogBrainX, and

- user interactions via a Web GUI and a Command Line Interface (CLI).

Fig. 2 sketches the overall architecture of FogArm, with the services that enable the above features. Namely:

**FogArm Core,** which retrieves, in continuity with one or more CI/CD pipelines, the information about the applications to be managed and the current state of the infrastructure, exploited by FogBrainX to determine management decisions. Fog-Arm then transforms such placement decisions, computed by FogBrainX, into executable actions and implements them through Docker Swarm. It also offers a CLI through which users can interact by requesting the execution of actions and/or by monitoring the current state of the managed resources and applications. FogArm Core needs two files for each managed application, listed in Fig. 3 . The standard docker-compose.yml file (Fig. 3 a) contains information to configure the application's services and the requirements.yml file (Fig. 3 b) describes the software, hardware and IoT devices requirements for each service reported in the docker-compose.yml file, as well as latency and bandwidth to other services. These requirements are used to automatically generate a suitable Prolog file exploited by FogBrainX as application specification.

**FogWatcher,** which is a daemon service that monitors whether updates have occurred in the specification of managed applications or the status of the infrastructure. It checks whether the desired and current application placements do not match so as to trigger FogArm Core to enforce appropriate actions. Last, it checks user triggers coming from the Web GUI (e.g., updates on services' requirements). By, automatically triggering FogArm Core, FogWatcher automatically guarantees the
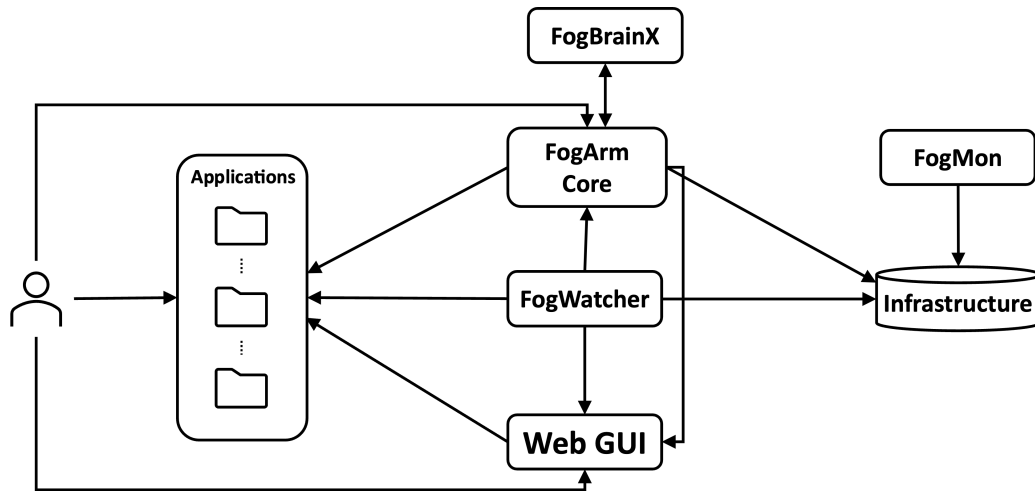
**FIGURE 2** Architecture of FogArm.

```
version: "3.3"

services:
  web:
    image: localhost:5000/stackdemo
    build: .
    ports:
      - "8000:8000"
  redis:
    image: redis:alpine
```

(a) docker-compose.yml

```
services:
  redis:
    hardware: 6
    links:
      web:
        bandwidth: 20
        latency: 150
  web:
    hardware: 3
    links:
      redis:
        bandwidth: 50
        latency: 500
```

(b) requirements.yml

**FIGURE 3** The application specification files.

requirements of each application at run-time, without the need for human intervention. Thus, FogWatcher allows clos-ing the management loop of an application in an automated cycle starting from the CI/CD pipeline, passing through its deployment and any necessary migrations in the presence of infrastructural or application requirements changes.

**Monitoring Tool,** which takes care of retrieving the current state of all nodes and links in the considered infrastructure. This information is converted into a series of Prolog facts ready to be used by FogBrainX. During our experiments, we employed the FogMon monitoring tool (Sect. 2.1). An adapter downloads the latest available FogMon report and translates it into a set of Prolog facts monitored by FogWatcher.

**Web GUI,** which shows all updates on the status of the infrastructure, applications' requirements, and their current and desired placement (Fig. 4 ). It allows users to monitor the global or individual status of nodes and links, and to read and modify

application specifications. It also offers the possibility to observe on which node the various services are currently placed and to manually request (and possibly actuate) the new placement for a given application, or to undeploy an application.
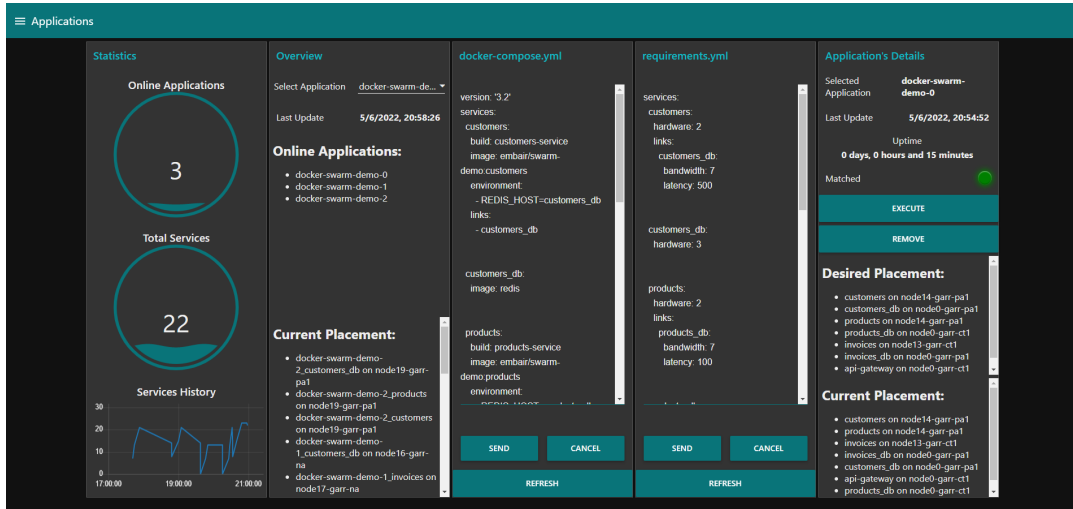


(a) Application page.



(b) Node page.

**FIGURE 4** The WebGUI.

FogArm operates autonomically by constantly monitoring the state of the infrastructure, the managed applications and their placements. It fully exploits the incremental approach of *continuous reasoning* by reducing management operations only to those services in need of attention, as identified by FogBrainX. Furthermore, FogArm is capable of simultaneously orchestrating several different multi-service applications on highly dynamic and geographically distributed infrastructures.

Note that the choice of the CI/CD pipelines and the monitoring tools are completely orthogonal and transparent to FogArm. Indeed, FogWatcher periodically checks if the information about the requirements of the applications and the state of the infrastructure has changed compared to the previous iteration and triggers FogArm Core when needed, independently from the entity (tool or human) that updates those state information.

FogArm leverages Docker Swarm at a low level to be able to focus on the innovative aspects of the orchestration process, delegating the basic mechanisms (e.g., deployment and allocation of resources) to a widely used and validated tool. We chose Docker Swarm for two main reasons. On one hand, it relies on Docker containers to feature flexibility and ease of use, since

Docker containers are the *de facto* standard to deploy microservices. On the other hand, it offers all the low-level features needed for the management of clusters and the deployment of services.

Overall, FogArm translates FogBrainX's decisions into actions on containers by exploiting Docker's constraints [5]. Constraints, enable specifying that a given service must necessarily be deployed to a specific node (by its hostname). Note that when a constraint is specified (or modified), if the service is on the wrong node, Docker automatically takes care of migrating from the node where the service is located to the one requested in the constraint.

Thus, exploiting FogBrainX and Docker, FogArm performs a complete monitor-analyse-plan-execute flow through the continuous monitoring of services' requirements and infrastructure status and, the interaction with Docker in Swarm mode, as depicted in Fig. 1 .

## 3.2 | FogArm's Behaviour

In this section, we discuss and highlight the behaviour of FogArm, in terms of the interactions of its components, illustrating three main scenarios viz., changes from the CI/CD pipeline, infrastructure changes and triggers from the Web GUI. These scenarios are sketched in Fig. 5 .
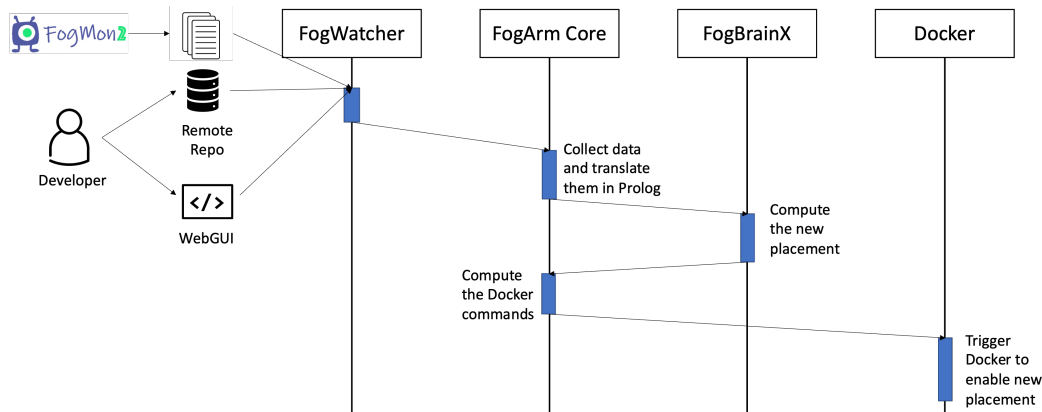


**FIGURE 5** Interaction diagram of FogArm.

**Changes from the CI/CD pipeline**

One of the most common scenarios working with FogArm is when a new commit is received through the CI/CD pipeline. In this case, the remote application repository is updated with the new code and application requirements might change. Periodically, FogWatcher checks whether changes have occurred in each managed repository. When a change is spotted, FogWatcher triggers FogArm Core. FogArm Core collects the application requirements from the local repository (i.e., the docker-compose.yml and the requirements.yml files) and the latest infrastructure report. This information is translated into a set of Prolog facts that are given as input to FogBrainX. Eventually, FogBrainX (possibly) outputs a new placement. Then FogArm Core, computes the differences between the current placement and the new one generated by FogBrainX. If the two differ, FogArm generates a set of Docker commands to reconcile the actual placement with the desired one.

**Infrastructure Changes**

Similarly, if a change occurred on the infrastructure resources, FogMon publish it in the new report. Periodically, FogWatcher verifies whether some changes have occurred in the infrastructure and triggers FogArm Core. The same procedure of the previous

---

[5]https://docs.docker.com/engine/reference/commandline/service_update/

case is then performed, ending with the (possible) new sets of Docker commands to reconcile the current placement into the new one determined by FogBrainX.

**Triggers from the Web GUI**

Finally, the last main source of updates is the Web GUI. Indeed users, besides monitoring the status of the infrastructure and the managed application and services, can also change the `docker-compose.yml` and `requirements.yml` files through the GUI. FogWatcher periodically checks if new updates are published by the Web GUI. If so, first the updates are saved in the local repository and then FogWatcher triggers FogArm Core as in the CI/CD pipeline scenario.

## 3.3 | FogArm's Implementation

In the following paragraphs, we briefly discuss the design and implementation of FogArm components. Sect. 3.3.1 details the technological aspects of the backend components, while Sect. 3.3.2 illustrates the frontend.

### 3.3.1 | BackEnd

**FogArm Core**

It is the central component of the whole FogArm architecture. It is implemented in Python3[6] exploiting the *argparse* library to implement the CLI, the *Docker SDK for Python*[7] to interact with Docker and the *PySwip*[8] library, which together with a Prolog script, makes the interaction with FogBrainX possible. The main task of FogArm Core is to collect all the necessary information on the applications and the state of the infrastructure, consult FogBrainX and actually apply its management decisions.

The main way to interact with FogArm Core, and more generally with FogArm, is through its CLI. Through it, it is possible to add, remove and manage applications, as well as to interact with FogWatcher and consult the status of managed applications and their placements. More in detail, the main CLI commands of FogArm are:

**add** which deploys, for the first time, the application specified in the path argument, if entered, otherwise in the current folder.

**exec** which performs a reasoning step, of one application if specified, otherwise for all the applications, by verifying if the current placement is still valid and possibly carrying out the necessary operations to add, remove or migrate the application's services.

**rm** which removes one or all applications from the infrastructure.

**status** which, for each application displays the desired placement, the current one and checks if the two match or not.

**watcher** which enables starting, stopping, or restarting FogWatcher and displaying if it is running or not.

As for add and exec, the two commands perform rather similar functions, with the only difference that in the case of add, being the first deployment, some additional information is stored by the system for future use and in the case of exec it is possible to specify the application whose reasoning process the user wants to execute also by its name (or it is also possible to request the execution of all applications).

Once the application to orchestrate has been identified, by path or name, and after any additional information has been saved, FogArm Core proceeds by verifying the existence of `docker-compose.yml` and the possible existence of `requirements.yml`[9]. Once the data have been retrieved, this information is used to generate a series of Prolog facts representing the application specifications. These facts together with the Prolog file containing the updated status of the infrastructure are passed to FogBrainX, which checks whether a deployment already exists. If it exists, FogBrainX checks whether this is still valid, if not or if the deployment does not exist, FogBrainX is asked to generate a new placement, possibly by applying *continuous reasoning*.

Three lists are then generated from the computed deployment, comparing the placement obtained with the previous one. The services to be deployed (together with the relative chosen node, i.e., those services that are in the new placement but not in the

---

[6]https://www.python.org/

[7]https://docker-py.readthedocs.io/en/stable/

[8]https://github.com/yuce/pyswip

[9]With `requirements.yml` it possible to "annotate" each service reported in the `docker-compose.yml` file with quality and quantity requirements that the service needs to be able to correctly perform its tasks. The conversion into a set of Prolog facts is then a simple 1-to-1 mapping.

old one), those to be removed (i.e., those services that are in the old placement but not in the new one) and those to be migrated from one node to another (i.e., those services that are in both placements but are assigned to different nodes). If a previous deployment does not yet exist, all services are considered to be added.

Once these three lists have been defined, FogArm Core takes care of actually executing FogBrainX's decisions by interacting with the Docker CLI. If instead, the `status` is requested, the current placement of all managed applications is extracted and, application by application it is compared with the one determined by FogBrainX. Finally, through the `watcher` command it is possible to interact with FogWatcher being able to start, stop, or restart it on request.

### From FogBrain's Output to FogArm's Constraints

As aforementioned, FogArm Core translates FogBrainX's decisions into actions on containers by exploiting Docker's constraints. Through constraints, it is possible to specify that a given service must necessarily be deployed to a specific node (specified in our case by the node's `hostname`).

Currently, FogArm supports five main actions:

- the *initial deployment* of an application,

- the run-time *addition* of a service,

- the *migration* of a service from a node to another node,

- the *removal* of a service, and

- the *removal* of an application

In the following, we illustrate how FogArm enforces the execution of FogBrainX decisions through Docker. We rely on examples to epitomise each case.

***Initial Deployment of an Application***   When a new application has to be deployed for the first time, FogArm executes

```
docker stack deploy AppId
```

where `AppId` is the unique identifier of the application. For each service, then the deployment constraint for the node is embedded with

```
docker service update --constraint-add node.hostname==NodeId AppId_ServiceId
```

where `NodeId` is the unique hostname of that node and `AppId_ServiceId` is composed by the identifiers of the application (i.e., `AppId`) and of the service (i.e., `ServiceId`).

*Example.*   Suppose we have a `WebApp` consisting of two microservices - `Frontend` and `Backend` - and an infrastructure of three nodes: `node1`, `node2`, and `node3`. To ask FogArm to deploy the application[10], our developer just needs to type

```
fogarm add .
```

then FogBrainX computes the initial placement, for instance

```
placement(webapp, [(frontend, node1), (backend, node2)]).
```

Last, FogArm converts such placement in a suitable set of Docker commands and constraints as follows

```
docker stack deploy webapp
docker service update --constraint-add node.hostname==node1 webapp_frontend
docker service update --constraint-add node.hostname==node2 webapp_backend
```

***Run-time Addition of a Service***   When a service is deployed for the first time in an already running application, the constraint is simply added through a suitable instantiation of the command

---

[10]For the sake of simplicity let's suppose the user is currently executing FogArm in the folder of the application's source code.

```
docker service update --constraint-add node.hostname==NodeId AppId_ServiceId
```

where `NodeId` is the unique hostname of that node and `AppId_ServiceId` is composed by the identifiers of the application (i.e., `AppId`) and of the service (i.e., `ServiceId`).

*Example.* Suppose our developer decides to add a `database` to the `webapp`, `FogArm` is then invoked by the developer or by FogWatcher

```
fogarm exec webapp
```

the new placement is computed by FogBrainX as

```
placement(webapp, [(frontend, node1), (backend, node2), (database, node2)]).
```

FogArm eventually takes care of interacting with Docker via the commands

```
docker stack deploy webapp
docker service update --constraint-add node.hostname==node2 webapp_database
```

***Migration of a Service***   If the service was already deployed, but FogBrainX decides to migrate the service to another node, the previous constraint is first removed and a new one is added through the sequence of commands

```
docker service update --constraint-rm node.hostname==OldNodeId AppId_SId
docker service update --constraint-add node.hostname==NewNodeId AppId_SId
```

where `OldNodeId` is the unique hostname of the node where the service is already placed and `NewNodeId` is the unique hostname of the new node where to deploy the service `SId` of the application `AppId`. When a constraint is specified (or updated), if the service is on the wrong node, Docker automatically takes care of migrating such service from the node where it is located to the one specified by the constraint.

*Example.* Assume node1, hosting the `fronted`, crashes, then FogBrainX updates the placement with the new information

```
placement(webapp, [(frontend, node3), (backend, node2), (database, node2)]).
```

and a new set of Docker constraints is generated

```
docker service update --constraint-rm node.hostname==node1 webapp_frontend
docker service update --constraint-add node.hostname==node3 webapp_frontend
```

***Removal of a Service***   Similarly, it is possible to request the complete removal of a service through

```
docker service rm AppId_ServiceId
```

In this way, FogArm Core can manage the deployment and migrations of multi-service applications.

*Example.* After some more work, the developer decides to integrate the database inside the backend, the `database` service is then no longer required

```
placement(webapp, [(frontend, node3), (backend, node2).
```

after which FogArm executes the following commands to remove the `database` service

```
docker service rm webapp_database
```

***Removal of an Application***   Finally, when the removal of an entire application is requested through `rm`, the Docker CLI is once again exploited to remove all the services of a given application, through

```
docker stack rm AppId
```

*Example.* To remove the application, the developer just needs to type

```
fogarm rm webapp
```

and FogArm performs

```
docker stack rm webapp
```

**FogWatcher**

It is developed in Python3 exploiting the *timeloop* library to implement periodic checks. The main task of FogWatcher is to periodically monitor whether updates have occurred in the infrastructure or the specifications of the applications managed, triggering FogArm Core to perform the actions necessary to guarantee the desired QoS for each service, without the need for human intervention.

FogWatcher periodically monitors, with independent and customisable periods[11], four possible sources that may require FogArm Core activation:

**The CI/CD pipeline**

For each managed app, it is periodically checked whether the `docker-compose.yml` or the `requirements.yml` files have changed. For each of the files, a hash (exploiting the HASH256 function) of its content is computed. If the final hash does not correspond to the previous one, then a change has happened and FogArm Core is invoked to carry out a reasoning step through the exec command. In this way, FogArm Core, by invoking FogBrainX, checks for the application whether the required requirements are still satisfied after the specification change and, if not, carries out the necessary operations to add, remove and migrate the services.

**The infrastructure**

Furthermore, also the file containing the updated status of the infrastructure is monitored. FogWatcher takes care of calculating the hash of that file and if the calculated hash does not correspond to the previous one, then for all the managed applications it is required to carry out a reasoning step.

**The current placement**

FogWatcher also periodically checks, for each application, whether the desired placement and the current one correspond. If this is not the case, then the deployment computed by FogBrainX is removed and FogArm Core is invoked. Indeed, in a real system it may happen that after a node or a Docker error, or if there has been a manual intervention on one or more services, the current placement no longer corresponds to that requested by FogBrainX.

**Live changes**

Finally, FogWatcher regularly queries the Web GUI to check if the user has requested any operation, e.g., removal of an application, execution of a reasoning step, update of the compose or requirements file, in which case the request is fulfilled and eventually a reasoning step is carried out.

**Integration with FogMon**

In our implementation, FogMon allows FogArm to be always updated on the state of the infrastructure and to make informed management decisions to comply with application requirements.

After installing a FogMon agent in each node of the infrastructure, such a tool periodically reports the updated status of the infrastructure in the form of a JSON file. Such monitoring data is made available through FogMon's REST APIs.

---

[11]Check periods, as well as other parameters of FogArm's components, can be customised through a global *configuration file*, updatable on the fly.

Then FogArm takes care to periodically consult that endpoint and obtain the latest available report. It checks whether the report obtained is different from the previous one[12] (by comparing the hash of the two reports). If so, the JSON report is mapped into a set of Prolog facts and the input file is updated.

It will be the task of FogWatcher to note that an infrastructural change has happened and to invoke FogArm Core to verify if service management operations are needed.

### 3.3.2 | FrontEnd

**The Web GUI**

It is the main interface with which the user can interact with FogArm. On one hand, the Web GUI offers a higher-level control of applications than the CLI. On the other hand, it enables to simply have an overall view of the entire system at a glance, with the possibility of paying close attention to the characteristics and properties of a single application or specific node and/or link.

The Web GUI is therefore designed to be primarily a monitoring interface with which to observe the actual status and the evolution of the system and be able to pay attention to detailed aspects, but which still offers the main functions for interacting with FogArm and its managed applications.

The Web GUI is implemented through Node-RED[13], a flow-based development tool built on Node.js, and it is divided into two main pages, Application and Nodes. The former focuses on the global status of applications and services, with the ability to analyse a single application. The latter allows users to observe the overall state of the infrastructure and study a single node and/or a single link.

We illustrate in more detail such two views of the Web GUI:

*Applications* The Applications page (Fig. 4 b), allows the users to monitor the overall state of the managed applications as well as analyse in detail the state of a particular application and interact with it, also offering the possibility to view and update its `docker-compose.yml` and `requirements.yml` files.

The page is divided into five panels.

- *Statistics*: It allows having some global statistics on the current state of managed applications. In detail, it displays the total number of applications and services currently deployed as well as how the number of deployed services has varied over time.

- *Overview*: This panel displays which applications are currently deployed and the current placement of each service belonging to those applications (i.e., on which node each service is deployed). It also reports when the last update was received and allows one to select a particular application to focus on.

- *docker-compose.yml & requirements.yml*: They show the last known state of the two related files of the application selected in the previous panel and offer the possibility to modify them live and send the changes to be implemented. Sending a change also automatically activates the execution of a reasoning step. If, on the other hand, a change has been applied but not yet sent, with a refresh button it is possible to return to the unmodified version. Cancel instead deletes all the contents allowing to rewrite the file from scratch.

- *Application's Detail*: Shows various information about the application selected in the Overview panel. In particular, it displays when the last update for that particular application was received, the uptime and, for each service of that application, the desired and actual placement. A LED allows checking at a glance whether the two placements match (green light) or if they differ (red light). It is also possible to request the execution of a reasoning step for that application or remove it from the infrastructure.

*Nodes* The Nodes page (Fig. 4 b) allows monitoring the current state of the whole infrastructure, also offering the users the possibility to focus their analysis on a specific node and/or link.

The Nodes page is divided into five panels.

---

[12]Note that, thanks to the `sensitivity` configurable parameter (i.e., threshold relative difference on average and variance to send differential reports) of the FogMon agents, only variations that exceed the threshold level are reported. Thus, small fluctuations will not result in changes in the report. Hence, a change in the report implies the presence of at least one significant infrastructural change, which therefore needs attention.

[13]https://nodered.org/

- *Overview*: Displays when it received the last update, the current number of available nodes and the evolution of that number over time. Allows also the users to select a particular node (among all the nodes of the infrastructure, not only the currently available ones) and/or link, specifying the two endpoints of the link.

- *Node's Detail*: Shows if the selected node is online and its last knows status, in terms of available free hardware (i.e., RAM) and its evolution over time, and the available IoT devices and software (if any).

- *Link's Detail*: Displays if the selected link is actually available and the last known value of the available bandwidth and latency.

- *Link's History*: Allows the users to view how the bandwidth and latency of the selected link are changed over time.

- *Applications*: Shows when the last updates for the applications are received and the current placement of the whole available services of all the deployed applications. If a node is selected, it also displays which services are deployed on that specific node (if any).

## 4 | EXPERIMENTAL ASSESSMENT

In this section, we first illustrate the experimental assessment, that is performed on a real distributed cloud infrastructure managing real microservices applications[14], of FogArm at varying infrastructure sizes and a number of applications over a real-world testbed (Sect. 4.1). We then show how *continuous reasoning* can boost decision-making and application management times in a testbed made of 60 nodes and running 400 services (Sect. 4.2).

In each experiment, the infrastructure is configured through an automatic script to guarantee the reproducibility of the procedure and results.

## 4.1 | Scalability assessment

We perform a scalability assessment of FogArm to evaluate how scaling the size of the infrastructure and the number of managed applications and services might affect the orchestrators' performance.

**Experimental Setup.**

The assessment is divided into three experiments at increasing size of the infrastructure (viz. 15, 30 and 60 nodes) and number of application[15] replicas (viz. 10, 25 and 50). Each composed of 8 services (viz. a total of 80, 200 and 400 services, respectively). Testbeds span three regions spread across Italy (viz., Catania, Palermo and Turin), with nodes evenly distributed among each region. Each node hosts a FogMon agent to monitor available resources[16]. We use replicas of the "Docker Swarm Demo" application[17], composed of 8 services (Fig. 6  ) and which mimics a real-world application. For each replica, we actuate changes through the CI/CD pipeline, by randomly producing a new commit and/or updated service requirements[18]. Finally, we run and monitor experiments for 5 hours after the initial deployment of all applications. During each experiment FogArm was invoked from a minimum of 100 times to a maximum of 250 times, performing from a minimum of 23 migrations to a maximum of 247 migrations when applying *continuous reasoning*, and a maximum of 340 migrations when performing the exhaustive search.

**Experimental Results.**

maybe here the other graphs?

---

[14]We run all experiments over Virtual Machines (VMs) featuring 1 vCPU and 6GB of RAM, and running Ubuntu 20.04.3 LTS, provided by the GARR Consortium and spread across 3 regions (viz., Catania, Palermo and Turin). Nodes run Python 3.8.10, Docker 20.10.12, docker-compose 1.25.0 and SWI-Prolog 8.4.2 to support the correct execution of FogBrainX and FogArm. A node in the Catania region is chosen as the leader from which the orchestration process is actually executed, interacting with the FogArm CLI and the Web GUI.

[15]Available at: https://github.com/michal-bures/docker-swarm-demo

[16]In detail, whenever a new FogMon report is received, we artificially reduce the available node's RAM of a value picked at random from a Gaussian distribution centred at 750MB with a standard deviation of 375MB. For each link, we artificially increase the latency by adding a random value from a Gaussian distribution centred at 50 ms and with a standard deviation of 25 ms. Similarly, the bandwidth is artificially reduced by a value picked at random from a Gaussian distribution centred at 12.5% of the available bandwidth and a standard deviation of 6.25%. Last, nodes and links have a failure probability of 5%.

[17]https://github.com/michal-bures/docker-swarm-demo

[18]Each service can require from 250MB to 750MB of available RAM. For each service-to-service communication a latency from 200ms to 750ms and an available bandwidth from 10Mbits/s to 30Mbits/s. Furthermore, each service has a different probability, from 75% to 100%, of being added to the last generated commit, so to also experiment with the addition and/or removal of services at run-time.
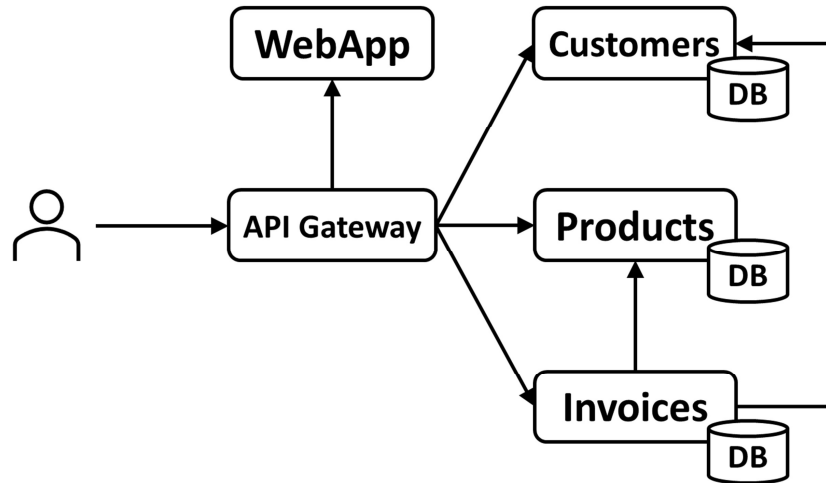
**FIGURE 6** Example application.
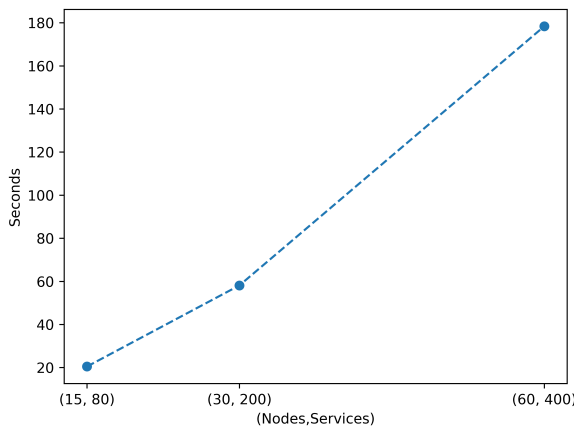


(a) Average times.
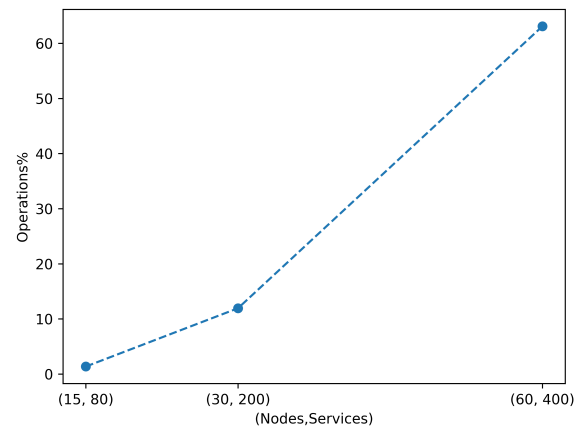


(b) Average migrations.

**FIGURE 7** Results for the scalability assessment.

Fig. 7 a illustrates the experimental results in terms of the average FogArm execution times at increasing sizes of the infrastructure and number of managed services. Execution times sum up both the time needed by FogBrainX to make management decisions and by FogArm to actuate such decisions by interacting with Docker Swarm. Fig. 7 b shows the average percentage of migrations per execution step over the total number of services that could be migrated (i.e., thus excluding from the considered set of services those that are about to be added or removed).

Execution times (Fig. 7 a) increase as the scale of the experiments increases, even if in all considered scenarios, the minimum time was 0.05 seconds.. Indeed, we experience an average execution time of 20 seconds and a median of 0.33 seconds when managing 80 services on 15 nodes (i.e., around 250 ms on average for each service). When managing 400 services, instead we experience an average execution time of 180 seconds (i.e., around 450 ms per each managed service) and a median of 130 seconds. We have, then, an increase of less than 2× on the average execution times, while increasing by 5× the number of services and 4× the number of nodes. This behaviour relates to the variation of the average amount of required migrations over

the experiments. Indeed, in smaller scenarios, we experience near-to-zero migrations, while we reach more than 60% average migrations per execution step in larger scenarios (Fig. 7 b).

Overall, we observe an exponential increase both in execution times and migrations. Indeed, as the size of the infrastructure and the number of managed services increase, more resources are required by FogMon to monitor the infrastructure state and by Docker to enact management decisions. Additionally, the larger number of managed services leads to a more dynamic system that, therefore, requires more migrations to maintain its optimal state. When the number of services and nodes increases, the chances of resource congestion and failures naturally increase and cause more migrations to satisfy application requirements. These results further highlight the need for efficient resource management strategies that can scale with the size of the infrastructure and the number of managed services.

## 4.2 | Continuous Reasoning assessment

**Experimental Setup.**

In this section, we compare the *continuous reasoning* approach of FogArm against a version exploiting exhaustive search, i.e., possibly migrating services that are not affected by infrastructure changes or CI/CD triggers. The exhaustive search is a commonly adopted strategy, as we will discuss in Sect. 5, and was, therefore, a natural candidate as a baseline to compare the performance of management decisions with and without continuous reasoning.

We consider the same settings of the largest scenario of the scalability assessment (i.e. 60 nodes evenly spread in 3 regions across the Italian national territory and 50 applications with most 8 services, for an overall number of 400 services). We also employ the same approach for generating application commits and changing infrastructure capabilities. Also in this case experiments last 5 hours. FogArm was invoked every 5 minutes - around 60 times - both for the *continuous reasoning* and exhaustive search. Continuous reasoning performed around 250 migrations, the exhaustive approach performed around 350 migrations.

**Experimental Results.**

Fig. 8 a compares the average execution times featured by FogArm exploiting the *continuous reasoning* and the exhaustive search. The *continuous reasoning* strategy allows the orchestrator to save 35 seconds on average (i.e., around 15%) while determining and enforcing a given placement in comparison with the exhaustive search. We can relate such improvement with the different number of migrations performed on average by the two strategies. Indeed, the exhaustive search enforces, on average, 33% more migrations, increasing also the execution time required to enforce a given placement.

It is worth noting that these results confirm those of the simulations we carried out in [16]. Such results can be explained considering that FogBrainX, and hence FogArm, tries to preserve the current deployment as much as possible. FogArm, therefore, migrates only those services in need of attention due to infrastructure changes (e.g., due to link QoS degradation or node overloading) or CI/CD triggers.

On the other hand, the exhaustive search does not pay any attention to preserving the current deployment, limiting itself only to finding a placement among all those admissible and possibly leading to migrating more services than necessary. Overall, the results indicate that the *continuous reasoning* strategy is more efficient than the exhaustive search strategy in terms of execution time because it performs fewer migrations on average. Indeed, the *continuous reasoning* approach is designed to take into account and preserve the current deployment as much as possible while minimising the number of migrations required to achieve the desired placement.

Finally, note that the exhaustive search is not QoS-aware, so we have no guarantees that the placement found meets the given requirements. FogArm, instead, is designed to find only QoS-aware placements and to modify such placements when a CI/CD trigger or an infrastructure change occurs.

## 4.3 | Threats to Validity

In this section, we briefly discuss some threats to validity that might affect our experimental assessment. In pursuit of this objective, we employ commonly acknowledged criteria to analyse and assess validity, adhering to the framework outlined by Wohlin *et al.* [19].

**Internal Validity.** It concerns the cause-and-effect relationship, i.e., whether the observed effect is a result of changes introduced by the researcher or stems from some other unidentified cause. In our experiments, this means that the behaviour

(a) Average times.

(b) Average migrations.
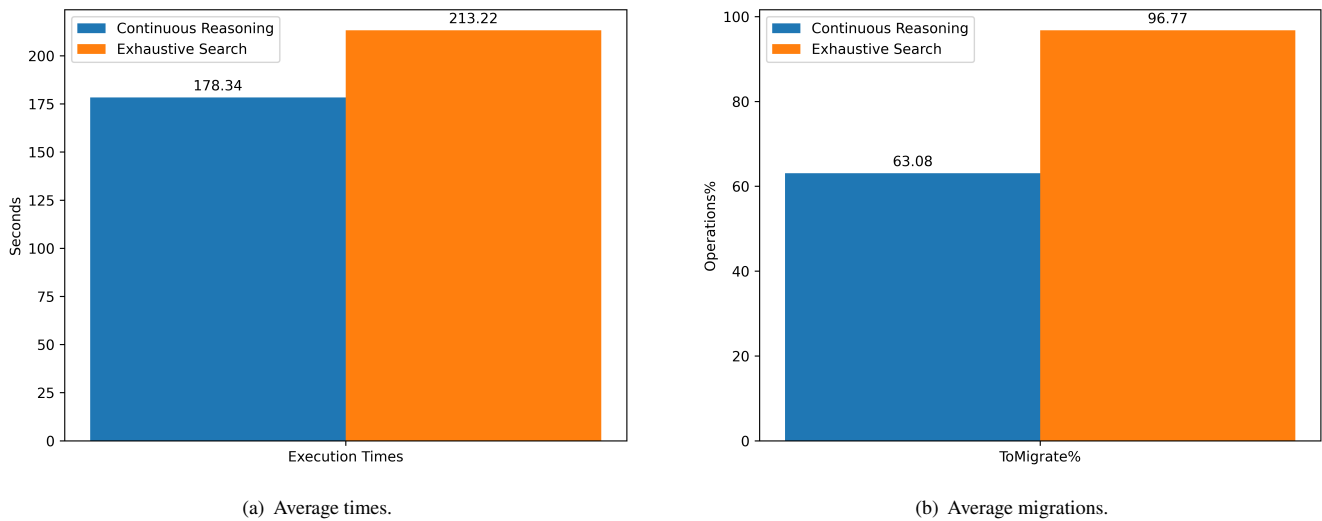
**FIGURE 8** Results for the continuous reasoning assessment.

of our experimental setup incorporates factors beyond our control. To the best of our knowledge, our prototype implementation of FogArm is faithful to the description provided in this article. All measurements related to execution times and migrations were conducted under consistent machine configurations throughout all experiments to prevent any instrumentation biases and they were guaranteed by relying on a set of automatic scripts to deploy and set the infrastructure as well as to run the actual experiments, thus requiring no manual intervention during the whole experimentation.

**External Validity.** It focuses on the possibility to generalise the obtained results. In this context, the experimental assessment is crafted to encompass, at varying scales, all defining characteristics of realistic medium-scale Cloud-Edge infrastructures and medium to large multi-service applications. These characteristics include considering different features of the infrastructure nodes (i.e., the hardware capacity, the offered software and available IoT devices) and links (i.e., bandwidth and latency - both upload and download), as well as service requirements (in terms of hardware, software and IoT devices) and service-to-service interactions (in terms of minimum bandwidth and maximum latency). Our approach is further justified by previous work, which proved the potential of FogBrainX to model and analyse various Cloud-Edge scenarios (e.g. [16]). To further enhance our results we artificially - and in a controlled way - varied the parameters of the infrastructure nodes and links so as to mimic different infrastructure behaviours and to improve the reproducibility of our results. It is important to acknowledge that fully generalising our experimental results to every conceivable Cloud-Edge scenario is not feasible. However, we are confident that the chosen setup represents reasonably general settings. Furthermore, the open-source availability of our code is intended to empower other researchers and practitioners to assess FogArm (or its extensions) on diverse infrastructures, applications, and scenarios.

**Construct Validity.** It underscores our capacity to measure the intended variables. To address potential threats to construct validity, we have employed common and popular open-source software and tools to conduct our experiments (as Docker), already published and assessed tools (such as FogBrainX and FogMon) or common and heavily employed Python libraries (as Fabric and OpenStack).

**Conclusion Validity.** It outlines the ability to draw statistically valid conclusions from experimental measurements. Our experiments were carried out on increasing scale infrastructure up to realistic medium-scale Cloud-Edge infrastructure of 80 nodes deployed on a national scale. Each experimentation was performed during a time span of 5 hours executing hundreds of operations and migrations for hundreds of deployed and interacting microservices. Across all the experiments both the infrastructure data and application specifications significantly change.

As per the external validity, the assessment discussed above would benefit from assessments carried out on other real monitored infrastructure data and to larger infrastructures — which, to the best of our knowledge, are currently not yet available - while orchestrating more applications and services.

# 5 | RELATED WORK

The problem of designing platforms and methodologies for the orchestration and management of multi-service applications in the Cloud-Edge continuum is a very well-known problem [7,10,11,12,13]. The main difficulties are given by the scale, heterogeneity and diversity of the node's infrastructures. Another important factor is the high dynamicity, in terms of resource capabilities variation (e.g., memory and bandwidth), failures of nodes and links and, devices or users distribution, with their possible movements [4]. Additionally, applications are composed of several different and heterogeneous software components with possible dependencies among them [20].

Addressing these issues, [21] proposes a device-aware, greedy approach to incrementally build service provisioning solutions. The authors divided infrastructure into two layers (viz., Fog and End devices), with the Fog layers subdivided into two sublayers (viz., High Fog and Mist). Considering such structure, the process iteratively decomposes an application into sub-components and greedily places each component considering its requirements. The process is repeated until all solution components are provisioned. With respect to FogArm, in [21] a structural division of the infrastructure is proposed, while in our work we consider infrastructure as a graph of nodes. Indeed, we believe that a plain representation fits better the heterogeneity and pervasiveness of the Cloud-Edge continuum.

Still dividing the Fog level, [22] proposes a hybrid choreography/orchestration hierarchical strategy for service management in Fog environments, dividing the infrastructure into three layers (viz., IoT, Fog and Cloud). At the IoT level devices are organised in virtual clusters supporting the possibility of mobile devices. At such a level the management devices cooperate among them, in a choreography fashion, offering low response time and high resilience in the presence of a device movement (i.e., shift into a different virtual cluster). At the Fog level, both strategies are performed depending on whether the fog device is in the south (i.e., closer to the IoT) or north (i.e., closer to the Cloud) region. Choreography is performed in the south region, orchestration in the north region and on the Cloud. The three-level architecture offers higher dynamism to the lower levels while keeping a global view of the higher levels, in which possible optimisation is performed. As discussed for [21] also here a structural representation of the infrastructure is proposed. Furthermore, it is not clear how the services' requirements are managed and checked by their *Resource Manager*. Finally, no prototypes are implemented for this work and there is also a lack of experimental results, instead, we proposed and assessed a fully automatic orchestrator, also gathering experimental results in an actual infrastructure.

In [23], an orchestration middleware for IoT systems is presented. The orchestrator features a three-level architecture. Each layer, from the topmost to the bottom, comprises, namely, a system description language to describe fog infrastructures and services, persistent data storage and a management engine to formulate constraints that encode system properties and requirements in the form of satisfiability modulo theory (SMT). Such an engine enables the use of SMT solvers to determine a valid configuration at run-time. Despite, the description language proposed is powerful and flexible it requires a greater effort by the developers with respect to that required by FogArm in compiling the `requirements.yml` file and in [23] users are required to compile also a description of the possible nodes' templates while in FogArm the management of the nodes is fully transparent and automated. Finally, in the proposed version only the nodes' failures are accounted for, on the contrary, FogArm is capable of managing nodes, networks and service failures.

In [24], the authors propose a container scheduling system in a Cloud-Edge environment. Their system leverages native Docker and features a master/worker-based design pattern. Their approach is designed to minimise edge resource utilisation rate and save cloud resource costs. The strategy is divided into two stages, viz., *node filtering* to filter out nodes that do not meet the minimum resource requirements of the container, then the remaining nodes are comprehensively *scored*, and then *sorted* according to the value of the score. However, their approach does not scale with the size of the managed Cloud-Edge infrastructure due to the exhaustive analysis that must be carried out on all the nodes. FogArm, instead, features a *continuous reasoning* approach that enables high-scalability.

Working on Osmotic computing, [25] discusses an orchestration architecture in which managed IoT applications, deployed in distributed environments, are modelled as a graph of MicroELements (MELs). A MELs graph models microservices, which implement specific functionalities, as well as microdata, representing information flows from/to IoT devices. The proposed orchestrators, through a Deep Learning process, generate MELs deployments based on previous experiences and eventually execute the obtained deployments manifest. However, this work is only a theoretical work so neither a working prototype nor an experimental assessment is proposed. Furthermore, using a Deep Learning process usually decreases the explainability of the orchestrator, making it difficult to understand why a certain management decision was taken. Instead, the engine of FogArm,

FogBrainX, thanks to its declarative nature is explainable and it is possible to trace all the decision steps performed. A model-driven approach is also exploited in [26], which proposes an attribute-driven framework for application development and service orchestration, assisting developers through the entire development lifecycle through a set of formal rules.

Exploiting Software Defined Networking, [27] presents a service orchestration mechanism to meet the latency and reliability requirements of IoT applications. Through a target optimisation function, a differentiated task offloading strategy is applied considering task attributes as well as communication and computation energy consumption and pre-estimated task offloading costs. Similarly, the problem of the placement of Virtual Network Function is studied in [28], through a multi-objective optimisation problem model that is converted to a problem which is solved by a Markov approximation technique.

Moving on to industry tools, the most popular solutions are based on container orchestration. In this field, the orchestrator manages the entire lifecycle of a container ranging from its creation to its destruction or termination and scaling or migrating containers if needed. Among the most widely used we have Docker in its Swarm mode[19] and Kubernetes[20], targeting clusters and datacenters. Both solutions do not offer high awareness of the services' requirements. Docker Swarm offers a system of constraints based on the labelling of nodes and services (i.e. placing a service if its labels match those of the nodes), while Kubernetes allows specifying simple CPU, memory and storage constraints. At the same time, these solutions are carried out manually, thus not coping with the high dynamism of Cloud-Edge infrastructures.

Finally, Topology and Orchestration Specification for Cloud Applications (TOSCA)[29] is one of the first and main proposals for standardising the service orchestration in an extensible and flexible way[30,31]. TOSCA is an open-source language to define an interoperable model of cloud applications. TOSCA describes the components as well as the relationship and dependencies between them and their requirements and capabilities, thus enabling portability and automated management. With TOSCA applications are described as a typed, direct topology graph, representing components as nodes and the dependencies between them as links. For each component is also possible to describe its requirements as well as the needed operations and policies.

In [32], the TOSCA standard and the Docker ecosystem are exploited to propose an orchestration strategy for the management of multi-component applications based on a TOSCA-based representation. The approach allows specifying software components and Docker containers to form an application and automatically deploy and manage such applications. With respect to FogArm in this work, the management of the applications is performed in a single machine and not on a distributed infrastructure. Furthermore, the orchestrator requires already developed management plans and it is not capable of finding the placement automatically. On the contrary, [33], proposes a TOSCA-based orchestration tool for automating the process of federating Kubernetes container clusters even across different cloud providers. However, both the orchestrators illustrated do not support the connection with a CI/CD pipeline, and thus the modification of the applications' topology or requirements at runtime.

In [34], the authors present a technology-agnostic method in their paper, utilizing Business Process Model and Notation (BPMN) and TOSCA standards for modeling function orchestrations and their deployment. They introduce a unified, technology-agnostic approach for function orchestration modeling, employing BPMN, and a TOSCA-based method for deploying function orchestrations. This allows for the declarative modeling of serverless application deployments with serverless function orchestrations using a standards-based toolchain. It's worth noting that their system focuses on serverless function orchestration deployment, while our proposal is more versatile, catering to simple multi-container applications. Despite this, we aim to explore how FogArm can enhance support for serverless functions in a dedicated manner.

Focusing on continuous methodologies, in [35] the authors leverage the continuous reasoning approach of FogBrainX to propose a new DevOps practice to support application management in continuity with CI/CD pipelines based on requests for microservice workflows. In [36], instead, the authors illustrate a framework for the continuous evaluation of the QoS of distributed mobile applications.

Concluding, to the best of our knowledge, none of the existing orchestration solutions (e.g., Docker Swarm, Kubernetes), unlike FogArm, supports *continuous reasoning* or more generally a continuous (i.e., incremental and differential) scheduling process that makes QoS- and context-aware management of microservices, possibly ensuring the optimisation of the allocation of services on highly dynamic infrastructures, in continuity with the CI/CD pipeline. Furthermore, most existing proposals only referred to simulated environments due to the lack of orchestration platforms capable of monitoring the needed QoS attributes, and to the limited availability of Cloud-Edge testbeds[14].

[19]https://docs.docker.com/engine/swarm/
[20]https://kubernetes.io/

FogArm, instead, is capable of autonomously adapting the deployment of the application in response to changes to the application specification coming from the CI/CD pipeline and to variations infrastructural detected through a distributed monitoring tool. When triggered FogArm applies a *continuous reasoning* approach, through the interaction with FogBrainX. Finally, FogArm is assessed in an actual geographically distributed infrastructure over the Italian national territory.

# 6 | CONCLUSIONS

## 6.1 | Summary

In this paper, we proposed FogArm, a next-gen orchestrator prototype that performs fully automated and QoS-compliant continuous management of multiservice applications on top of highly dynamic and geographically distributed infrastructures.

To perform the orchestration process FogArm interacts with different tools viz., FogMon to gather the current status of the infrastructure's resources, FogBrainX to exploit its *continuous reasoning* approach to find a valid placement in a continuous and scalable way and finally, Docker Swarm to implement the low-level operations through Docker's constraints.

FogArm continuously monitors the status of the infrastructure, the application's requirements and the current deployments searching for changes. When a change occurs, FogArm verifies through FogBrainX if a new placement is required. If so, FogArm generates the suitable management operations to accomplish the desired placements and interacts with Docker to perform the operations.

Through FogArm, developers are required only to define the requirements of each application's service. The whole process of deployment and management is fully automated without any required user action. FogArm works also in continuity with the CI/CD pipelines, supporting the current iterative and incremental development process. Additionally, users can interact with FogArm through a CLI or a Web GUI.

To the best of our knowledge, FogArm represents a first complete prototype of a next-gen orchestrator for the continuous QoS-compliant management of multi-service applications on geographically distributed Cloud-Edge infrastructures. FogArm proved to be able to scale up to tens of nodes and hundreds of managed services while also reducing execution times and migrations thanks to *continuous reasoning*. Our experiments have shown how FogArm can scale even on high dynamic, geographically distributed infrastructures with up to 60 nodes spread across Italy while managing up to 400 services from 50 applications, and continuously interacting with the CI/CD pipelines. Furthermore, the *continuous reasoning* methodology proved to save more than 15% of the execution time (i.e., around 35 seconds) while migrating on average 33% services fewer than the version of FogArm featuring only the popular exhaustive search strategy. Interestingly, the obtained results are in line with those we obtained in Forti *et al.* [16].

## 6.2 | Known Limitations

FogArm is still an initial prototype of a next-gen orchestrator based on *continuous reasoning* to achieve the continuous and QoS-compliant management of multi-service applications on geographically distributed Cloud-Edge networks, also capable of working in continuity with the CI/CD pipeline and infrastructure monitoring. Thus, we consider here some possible limitations to our proposal. The current implementation of FogArm works by interacting with Docker, but to improve the execution times it could be interesting to substitute Docker with a more advanced tool (e.g., Kubernetes) or to work directly with a container run-time environment (e.g., containerd), thus excluding the interactions with intermediates.

Such improvements should be accompanied by the development of strategies and techniques to support stateful migrations of services, thus enabling the persistency of the data over time and nodes. While this manuscript primarily focuses on providing key low-level functionalities for our orchestrator, the absence of higher-level features such as strategies and techniques for supporting data persistency across time and nodes do not support smooth data transfer during migrations and maintaining data consistency. However, developing strategies to enable data persistence, particularly during stateful migrations of services is still an active research topic and very few orchestrators currently support them. We plan to investigate how to incorporate them in future releases.

Currently, FogArm reasons only on software, IoT devices and RAM requirements. However, FogBrainX could be extended to support more expressive policies managing a richer infrastructure model including, for example, CPU, HDD and security requirements. Additionally, in the current prototype, the user has to manually insert the service's requirements. However, a

useful extension could include a process of data mining to automatically generate, possibly exploiting a system of code's annotations, the service requirements. It would be very interesting to design such a process also exploiting a *continuous reasoning* methodology to speed up the process of requirements extraction.

As a final limitation, our experimental assessment was carried out by comparing the *continuous reasoning* approach only with the exhaustive search. Indeed, the nature of the solution - a next-gen orchestrator for the managing of multi-service applications on Cloud-Edge settings - limits direct comparisons with other state-of-the-art research and orchestrators. To the best of our knowledge, there are no other orchestrators with compatible characteristics to perform a more extensive and holistic comparison. To overcome the current limitation on suitable orchestrators, we plan to develop more sophisticated placement policies to embed directly on FogArm to be able to carry out a more extensive experimental assessment comparing different strategies.

## 6.3 | Directions for Future Work

To conclude, we discuss some future research lines:

**Developing new placement strategies**  Continuous reasoning is designed to boost the performance of a given placer, reducing the size of the considered problem and re-using previously computed results as much as possible. Following this principle, FogBrainX could support several different placement strategies (e.g., genetic algorithms), possibly providing either a logic programming implementation of the desired approach or a logic interface to the implementation of the placer. Furthermore, the application and infrastructure models can be enriched by considering other QoS requirements/capabilities (e.g., security properties, and energy consumption). Additionally, data persistency, data consistency, and a stateful migration mechanism could be studied to better support the application orchestration.

**Extending run-time decisions**  Our methodologies could be extended by considering other management decisions (e.g., application scaling, service adaption), possibly including explanations on *why* a certain management decision was (not) taken. This would enrich the capabilities of our orchestrator, enabling both more sophisticated application management and improving visibility into the decision-making process.

**Workload stress test**  A further assessment of FogArm could involve studying its behaviour under stressful conditions through increasing workload on the managed services, thus even overloading both the node and links on the infrastructure through experimenting with the actual flow of data and users' interaction even in difficult condition with service crash or data loss.

## ACKNOWLEDGEMENTS

## References

1. Yousefpour Ashkan, Fung Caleb, Nguyen Tam, et al. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *J. Syst. Archit.*. 2019;98:289–330.

2. Qiu T., Chi J., Zhou X., Ning Z., Atiquzzaman M., Wu D. O.. Edge Computing in Industrial Internet of Things: Architecture, Advances and Challenges. *IEEE Commun. Surv. Tutorials.* 2020;22(4):2462-2488.

3. Uehara Minoru. Mist computing: Linking cloudlet to fogs. *Computational Science/Intelligence and Applied Informatics.* 2018;:201–213.

4. Svorobej Sergej, Bendechache Malika, Griesinger Frank, Domaschka Jörg. Orchestration from the Cloud to the Edge:61–77. Cham: Springer International Publishing 2020.

5. Bellavista Paolo, Berrocal Javier, Corradi Antonio, Das Sajal K., Foschini Luca, Zanni Alessandro. A survey on fog computing for the Internet of Things. *Pervasive Mob. Comp.*. 2019;52:71 - 99.

6. Bobrovskis Sergejs, Jurenoks Aleksejs. A Survey of Continuous Integration, Continuous Delivery and Continuos Deployment.. in *BIR workshops*:314–322; 2018.

7. Herrera Juan Luis, Galán-Jimnez Jaime, Garcia-Alonso Jose, Berrocal Javier, Murillo Juan Manuel. Joint Optimization of Response Time and Deployment Cost in Next-Gen IoT Applications. *IEEE Internet of Things Journal.* 2022;10(5):3968–3981.

8. Salaht Farah Ait, Desprez Frédéric, Lèbre Adrien. An Overview of Service Placement Problem in Fog and Edge Computing. *ACM Computing Surveys (CSUR).* 2020;53:1 - 35.

9. Mahmud Redowan, Ramamohanarao Kotagiri, Buyya Rajkumar. Application Management in Fog Computing Environments: A Taxonomy, Review and Future Directions. *ACM Comput. Surv..* 2020;53(4).

10. Velasquez Karima, Abreu David Perez, Assis Marcio RM, et al. Fog orchestration for the Internet of Everything: state-of-the-art and research challenges. *Journal of Internet Services and Applications.* 2018;9(1):1–23.

11. Wen Zhenyu, Yang Renyu, Garraghan Peter, Lin Tao, xu Jiudong, Rovatsos Michael. Fog Orchestration for Internet of Things Services. *IEEE Internet Computing.* 2017;21:16-24.

12. Costa Breno, Bachiega Joao, Carvalho Leonardo Rebouças, Araujo Aleteia P. F.. Orchestration in Fog Computing: A Comprehensive Survey. *ACM Comput. Surv..* 2022;55(2).

13. Wen Z., Yang R., Garraghan P., Lin T., Xu J., Rovatsos M.. Fog Orchestration for Internet of Things Services. *IEEE Internet Computing.* 2017;21(02):16-24.

14. Smolka Sven, Mann Zoltán Ádám. Evaluation of fog application placement algorithms: A survey. *Computing.* 2022;:1–27.

15. Gaglianese Marco, Forti Stefano, Paganelli Federica, Brogi Antonio. Assessing and enhancing a Cloud-IoT monitoring service over federated testbeds. *Future Gener. Comput. Syst..* 2023;147:77–92.

16. Forti Stefano, Bisicchia Giuseppe, Brogi Antonio. Declarative continuous reasoning in the cloud-IoT continuum. *Journal of Logic and Computation.* 2022;32(2):206-232.

17. Jelasity Márk. Gossip:139–162. Berlin, Heidelberg: Springer Berlin Heidelberg 2011.

18. Schubert Erich, Rousseeuw Peter J.. Faster k-Medoids Clustering: Improving the PAM, CLARA, and CLARANS Algorithms. in *Similarity Search and Applications* (Amato Giuseppe, Gennaro Claudio, Oria Vincent, Radovanović Miloš, eds.):171–187Springer International Publishing; 2019; Cham.

19. Wohlin Claes, Runeson Per, Höst Martin, Ohlsson Magnus C, Regnell Björn, Wesslén Anders. *Experimentation in software engineering.* Springer Science & Business Media; 2012.

20. Barika Mutaz, Garg Saurabh, Zomaya Albert Y, Wang Lizhe, Moorsel Aad Van, Ranjan Rajiv. Orchestrating big data analysis workflows in the cloud: research challenges, survey, and future directions. *ACM Computing Surveys (CSUR).* 2019;52(5):1–41.

21. Donassolo Bruno, Fajjari Ilhem, Legrand Arnaud, Mertikopoulos Panayotis. Fog Based Framework for IoT Service Provisioning. in *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*:1-6; 2019.

22. Velasquez Karima, Abreu David Perez, Gonçalves Diogo, et al. Service Orchestration in Fog Environments. in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*:329-336; 2017.

23. Pradhan Subhav, Dubey Abhishek, Khare Shweta, et al. CHARIOT: Goal-Driven Orchestration Middleware for Resilient IoT Systems. *ACM Trans. Cyber-Phys. Syst..* 2018;2(3).

24. Tang Bing, Luo Jincheng, Zhang Jiaming. Container-based data-intensive application scheduling in hybrid cloud-edge collaborative environment. *Software: Practice and Experience.* 2023;.

25. Carnevale Lorenzo, Celesti Antonio, Galletta Antonino, Dustdar Schahram, Villari Massimo. From the Cloud to Edge and IoT: a Smart Orchestration Architecture for Enabling Osmotic Computing. in *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*:419-424; 2018.

26. Rafique Wajid, Zhao Xuan, Yu Shui, Yaqoob Ibrar, Imran Muhammad, Dou Wanchun. An application development framework for Internet-of-Things service orchestration. *IEEE Internet of Things Journal.* 2020;7(5):4543–4556.

27. Huang Mingfeng, Liu Wei, Wang Tian, Liu Anfeng, Zhang Shigeng. A Cloud–MEC Collaborative Task Offloading Scheme With Service Orchestration. *IEEE Internet of Things Journal.* 2020;7(7):5792-5805.

28. He Wenchen, Guo Shaoyong, Liang Yun, Qiu Xuesong. Markov Approximation Method for Optimal Service Orchestration in IoT Network. *IEEE Access.* 2019;7:49538-49548.

29. Lipton P, Lauwers C, Tamburri D. Oasis topology and orchestration specification for cloud applications (tosca) tc. *March2017.* 2017;.

30. Bellendorf Julian, Mann Zoltán Ádám. Cloud topology and orchestration using TOSCA: A systematic literature review. in *European Conference on Service-Oriented and Cloud Computing*:207–215Springer; 2018.

31. Luzar Anže, Stanovnik Sašo, Cankar Matija. Examination and comparison of tosca orchestration tools. in *European Conference on Software Architecture*:247–259Springer; 2020.

32. Brogi Antonio, Rinaldi Luca, Soldani Jacopo. TosKer: a synergy between TOSCA and Docker for orchestrating multicomponent applications. *Software: Practice and Experience.* 2018;48(11):2061–2079.

33. Kim Dongmin, Muhammad Hanif, Kim Eunsam, Helal Sumi, Lee Choonhwa. TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform. *Applied Sciences.* 2019;9(1).

34. Yussupov Vladimir, Soldani Jacopo, Breitenbücher Uwe, Leymann Frank. Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA. *Software: Practice and Experience.* 2022;52(6):1454–1495.

35. Herrera Juan Luis, Berrocal Javier, Forti Stefano, Brogi Antonio, Murillo Juan M. Continuous QoS-aware adaptation of Cloud-IoT application placements. *Computing.* 2023;:1–23.

36. Laso Sergio, Berrocal Javier, Fernández Pablo, Ruiz-Cortés Antonio, Murillo Juan M. Perses: A framework for the continuous evaluation of the QoS of distributed mobile applications. *Pervasive and Mobile Computing.* 2022;84:101627.