

# Model-driven Approach for Deployment of Container-based Applications in Fog Computing

Nenad Petrović, *Faculty of Electronic Engineering, University of Niš*

**Abstract**— Fog Computing extends the Cloud Computing to the Edge of the network, closer to the things that produce and act on data including different types of devices, ranging from personal computers, laptops, workstations to IoT devices, wearable gadgets and sensors. The heterogeneity and variety of devices doesn't make just the activities of application design harder, but serious effort is also required when it comes to configuration, deployment, re-deployment and testing of the applications executed in Fog environments and it is often time-consuming, trial and error process. As the data could be produced and consumed at both Edge and Cloud, it means that application also needs to be compliant with given data privacy and security policies due to legal constraints, laws and regulations which restrict the freedom of data movement. Therefore, it should also be possible to move computation tasks between devices regardless of their location (Cloud or Edge), even in the case of different computing architectures, which means that the applications should be as flexible as possible. In this paper we focus on automating the deployment activities and increasing the flexibility of applications executed in Fog environment by providing a visual tool utilizing the concepts of metamodeling, automated code generation and container-based virtualization.

**Index Terms** — Fog Computing, Container-based virtualization, Metamodeling, Internet of Things

## I. INTRODUCTION

Fog Computing, also referred to as Edge Computing, is an emerging paradigm aiming to extend Cloud computing capabilities to fully exploit the potential of the Edge of the network where traditional devices, as well as new generations of devices – such as smart, wearable gadgets and mobile devices (the so-called “IoT devices”) are considered. It involves the research and application of enabling technologies that allow computation at the network Edge so that computing happens near data sources [1].

However, despite the rapid evolution of the data-processing speed, the bandwidth of the network that carries data to and from the Cloud has not increased appreciably. Thus, with Edge devices generating more data, the network is becoming Cloud computing's bottleneck. In these cases, the processing time of time-critical applications is often limited by the network delay [1][2]. Also, when we have a large number of devices sensing the information, uploading the data generated by them would induce additional network congestion, increasing this way the network delay, even further. In order to tackle network delay issues, we move the data processing closer to the data generators and consumers.

The concept of enabling a computer to sense information

without any human intervention has been applied to many other fields, such as healthcare, home technology, environmental engineering and transportation [1][2]. Due to such variety, the second major issue appears. Keeping the privacy and dealing with security of the information in more sensitive domains is crucial, in most cases regulated by law, and dramatically restricts freedom of data movement. Due to laws and regulations, in some cases the data is not allowed to leave the boundaries of the institution storing the data. Thus, in these cases, the data has to be processed within the Edge, instead of being uploaded over vulnerable network to the Cloud.

There are two approaches to tackle this problem – move the data or move the computation tasks [2]. In this case, we focus on the latter – the computation task movement.

The goal of the research is to explore state-of-the-art technologies and concepts in order to construct an execution environment which would give us ability to easily deploy and move computation tasks between devices with different computing architectures, located either in Cloud or Edge satisfying the given limitations and policies. As the result of the research, a visual tool for automated test and deployment of data processing applications in Fog environments is developed, utilizing the concepts of container-based virtualization as computation task abstraction and metamodeling approach for representation and abstraction of the application architecture in order to impose rules and constraints when it comes to computation task deployment and movement. User draws a deployment diagram of the application using the tool, which is later validated, translated to infrastructure management code and, finally, deployed to corresponding resources.

In comparison with approach already presented in [9][11] where the focus is on Cloud applications and hypervisor-based virtualization, this work extends the idea by introducing the support for Fog Computing and mixed architecture environment relying on container-based virtualization which is more suitable for the scenario when the tasks also have to be executed on low-power IoT devices.

## II. BACKGROUND

In what follows, the underlying concepts and theoretical foundations of the developed tool are presented.

- *Container-based virtualization*: Containers could be described as a lightweight virtualization approach that creates virtual environment at a software level inside the host machine, also known as operating system-level virtualization. It removes the overhead of using hypervisors by creating virtual machines in the form of containers (act as guest systems) thereby sharing the resources of the underlying host operating system. It provides different

levels of abstraction in which a kernel is shared between containers and more than one process can run inside each container. This way, the whole system can become more resource-efficient as there is no additional layer of hypervisor, and thus no full operating system which can occupy a lot of storage space for each virtual machine. Therefore, container-based virtualization is much more IoT-devices friendly. Considering the fact that most IoT devices use low-power ARM CPUs, it was important to explore if there are compatible ports of container-based platforms. Fortunately, there is Docker ARM port compatible with Raspberry Pi, which is widely used container-based virtualization technology. Thus, we decided to use Docker containers as computation task abstraction in our execution environment, as they are lightweight, easily portable and also support popular ARM-based Linux devices, as Raspberry Pi which gives capability to move tasks between Cloud and Edge by using ARM computation task counterparts for each task in the system [3].

- *Docker Swarm infrastructure*: The underlying container management system is Docker Swarm, a native Docker technology which is fully supported by ARM Docker port, out of the box. There are two types of devices inside the cluster: master node and worker nodes. The Swarm is created at master node which generates a token which worker nodes use in order to join the Swarm. Master node is responsible for container management and service creation, allocating the right container to the right device which had joined the Swarm previously. As container repository, we use public Docker Hub repositories which should contain two versions of each task – ARM and x86, with same name but different suffix. This way, we ensure that, in case of computation task movement, the device can find the right container (ARM or x86). Fig.1 illustrates the Docker Swarm infrastructure [3].

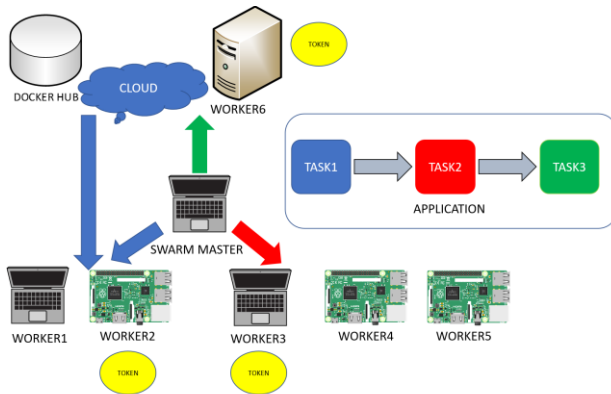


Fig. 1. Docker Swarm infrastructure

- *Model-driven engineering*: A software development methodology focused on creation and exploitation of domain models aiming at abstract representations of the knowledge, rather than computing concepts. The tool produced as an outcome of the research allows designers to use the modeling approach for deployment, re-deployment, analysis, optimization and architectural configuration of the applications that are executed in Fog environments. The user draws a diagram (model), which, translated into code, manages the deployment of the given infrastructure [4].

- *Automated code generation*: A mechanism which

generates a computer program based on higher level abstractions defined by developer [4]. In our case, the deployment configuration given by a diagram is translated into code (Docker and shell commands).

- *Infrastructure-as-a-code*: Approach where the code (possibly automatically generated from the model) is able to provide precise instructions to management tools that support the automated deployment and configuration of applications on the specified resources [5]. When it comes to our approach, Docker Swarm commands are used to manage the task allocation and deployment, while the tasks become Docker containers running on corresponding devices.

- *Metamodeling*: Analysis, construction and development of rules, constraints, models and theories applicable and useful for modeling a predefined class of problems. In our tool, user-drawn models are defined and constrained by a metamodel, which defines the deployment diagram elements, their attributes, configuration parameters, relations and rules they have to comply with [4]. The deployment of the application contains devices which are able to execute tasks corresponding to Docker containers. The metamodel (Fig. 2.) consists of the following elements and relationships between elements, as given in Table I.

TABLE I  
METAMODEL ELEMENTS OVERVIEW

| Name       | Description   |
|------------|---|
| Deployment | Deployment represents a highest level abstraction in the metamodel and contains devices executing allocated tasks.  |
| Device     | Represents a physical or virtual device executing a task given by a container. Considering that our execution environment use case is Fog Computing, we should consider heterogeneous devices (both IoT ARM-based devices and conventional x86 PCs) whose location could either Cloud or Edge. The location of the device is necessary in order to allocate the right tasks due to data privacy and other regulations. Also, it is important to keep track of memory available and allocated to tasks for each of the devices. Each device is actually a part of Docker Swarm infrastructure cluster as a worker which is executing a certain task. Therefore, in order to join a Docker Swarm cluster, we need to provide a valid Swarm master token. In that case, we also need an IP address in order to SSH (assuming that we have access to devices) and join them to the cluster. Otherwise, if devices already belong to some Swarm cluster, we can leave these two fields empty. In that case, we are used name attribute as alias to identify devices within the running cluster. In order to SSH, devices could require credentials in order to log in (username and password). |

|                       |   |
|-----------------------|---|
| Task                  | Computation task abstractions, represented as Docker containers. In our case, they are PHP servers packed as Docker containers which execute data processing with arbitrary number of inputs and one output. All the containers should be present in Docker Hub public repository, and each of them should have also its ARM counterpart, so we could perform the computation task movement between devices. Each device, depending on its architecture, executes the corresponding container version. Also, each of the containers occupies a certain amount of memory which should not be larger than device memory available.  |
| Task Dependency       | A directed association, describing the fact that tasks can be connected to each other, so the output of one task can be used as the input of another one, which makes a computation flow.   |
| Execution Flow        | A sequence of inter-connected tasks, using Task Dependency relations. This abstraction is used in order to support the testing of the deployed services.  |
| Execution Environment | For tasks, it represents the environment where each of them could be executed due to data movement constraints, legal regulations, security and privacy policies. According to this, some of the tasks are allowed to be executed only within the Edge of the network, without leaving the physical boundaries of the organization. On the other side, there are tasks whose execution location is not constrained, so they can be executed also in Cloud. Each device physically resides in Cloud or Edge. Therefore, the metamodel constraint rules take care of matching execution environment matching for each of the tasks. |

As addition to the given metamodel, constraints and rules for given elements are also defined using the metamodeling platform. For example, each task needs to be inside one and exactly one device. One device can execute many tasks. However, each task needs to have all its inputs defined before testing, whether they come from the parameter list (set by user) or dependencies with other task. Also, the memory that task occupies should less or equal to memory available. Each device has its location as an attribute, which could be: Cloud, Edge or both. According to this, we should match the execution environment mapping between devices and tasks. This mechanism encapsulates the fact that some of the tasks process data whose movement is constrained due to legal regulations, security or privacy policies, therefore, it should be processed within the boundaries of the organization, without being sent to Cloud providers. The part of allocating the right version to right kind of device (ARM/x86) done within the code generation phase, and is defined inside the transformation code, when the corresponding suffix is appended.

### III. IMPLEMENTATION

The tool is utilizing model-to-code transformations and Docker Swarm cluster management commands in order to automatically deploy the container-based application represented by user-created diagram. Fig. 3 presents the overview of components involved and operations performed to deploy the application given by a model, using the tool we have developed.

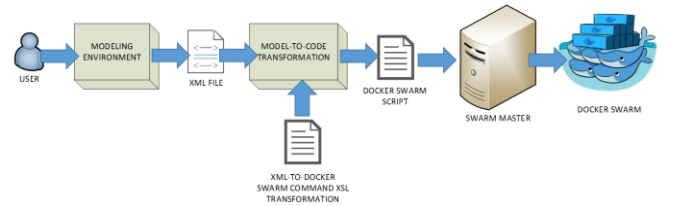


Fig. 3. Working principle overview

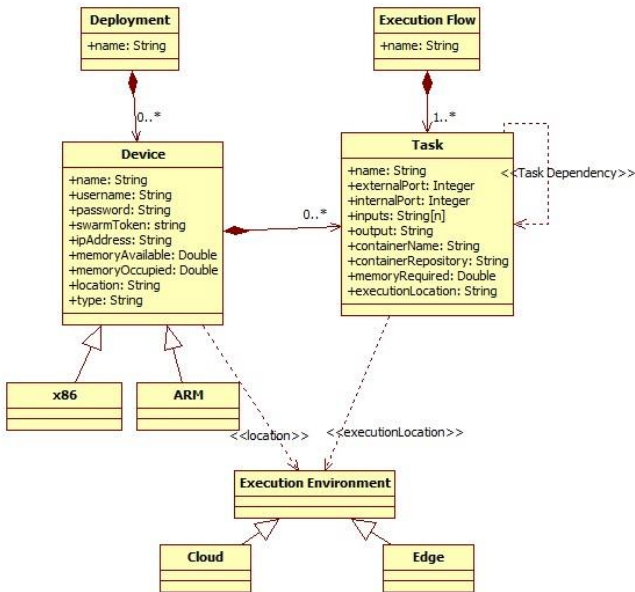


Fig. 2. Metamodel for deployment, testing and computation task movement in Fog environments supporting IoT devices

First, user models a container-based application using a set of given concepts and relationships between them inside the modelling environment. The modeling environment is developed using ADOxx meta-modeling platform [6] which provides the necessary capabilities to develop a GUI-enabled modelling tool based on UML-alike metamodel, described using ADOxx Definition Language (a meta-modelling language used by ADOxx platform). The main advantage of ADOxx meta-modeling platform is the support for creating a complete standalone product based on metamodel definition, including all the visual elements. Shapes are defined as value of GraphRep attribute for each element.

Once the modelling is done, it is possible to check if the model is valid. A set of rules based on the execution environment constraints are implemented within the metamodel in order to achieve this. For example, if we have a task that uses sensible data for computation, enforced by these rules, we can deploy it only to device which resides within the boundaries of the organization. After that, once the considered model is valid, the necessary transformations can be performed and the corresponding application is deployed. The previously mentioned rules and constraints

are also described using ADOxx Definition Language. Fig. 4. displays the screenshot of the modelling environment.

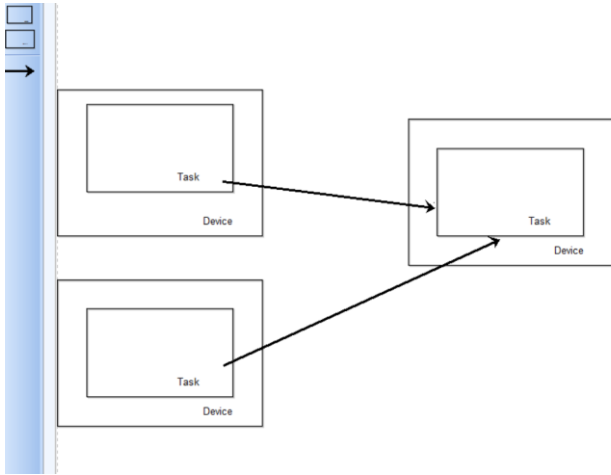


Fig. 4. Screenshot of the modeling tool

Before the deployment, the model is internally exported to XML format. Then, it is converted to Docker Swarm commands, executed by Docker Swarm master, which are actually responsible for container creation and deployment on each of the devices considered. This way, all the devices present in deployment model become actually Docker Swarm workers, running corresponding services on given ports. The transformation is defined by XSL file and executed by Saxon [7] engine which is also a stand-alone product, but also available within ADOxx environment. In Listing 1, the XSL transformation [8] is given as a pseudo-code:

```
foreach(Device device in deployment)
{
  if(device.swarmToken!=null &&
  device.ipAddress!=null)
  {
    ssh(device.ipAddress,
    device.username,device.password);

    joinSwarm(device.swarmToken);
  }

  ssh(master_ip_address,
  master_username, master_password);

  addAlias(device.ipAddress,device.name);

  foreach (Task task in device.Tasks)
  {
    write("docker service create -p
    task.externalPort:task.internalPort --
    replicas    number_of_replicas --network
    overlay_network_name --name task.name
    --constraint 'node.label.alias==
    device.name'
    task.containerRepository/
    task.containerName_device.type");
  }
}
```

Listing 1. Pseudo-code for model to Docker Swarm infrastructure code construction

For each device present in the diagram, it is checked if it has both IP Address and Swarm token set. If it is true, it means that the device needs to be added to the Swarm cluster. In that case, we need to SSH to the device and execute the Swarm join command with a token as a parameter. Otherwise, if the parameters are missing, it means that the device has already joined the Docker Swarm as a worker and we can directly proceed to the step of container scheduling. For this step, it is required to SSH into Swarm master (assuming that the Swarm master has been already run). Then, we can add the alias of the device and allocate all the tasks that should be executed inside the device, each of them running on separate internal/external port. If the device has already downloaded the Docker image for the container, the service can be started immediately, otherwise it downloads the corresponding image, according to the device type (x86/ARM).

Furthermore, a simple test functionality is available for the services deployed. It is performed by transformation of task dependency relationships within the execution flow, into a sequence of HTTP requests, as presented in Listing 2.

```
curl
MASTER_IP:task.externalPort/task.name+".php"+
?task.inputs[0].parameter=task.inputs[0].value
&task.inputs[1].parameter=task.inputs[1].value
&...&task.inputs[n].parameter=task.inputs[n].value
```

Listing 2. Construction of HTTP requests for testing the deployed services

From the infrastructure perspective, all the HTTP requests are sent to the Docker Swarm master. After that, the master resolves the Swarm worker device where the service is actually deployed according to the port used to expose the service. Fig 5. illustrates the behavior of the Docker Swarm infrastructure when we test the deployed application using the previously described mechanism [3].

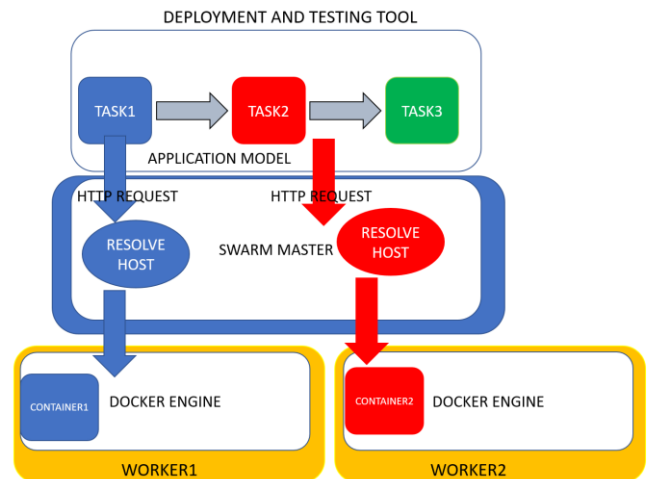


Fig. 5. Testing the deployed services using the Docker Swarm infrastructure

## IV. EVALUATION

To check if the tool works correctly, we have experimented with several deployment configurations, covering various cases: execution in Cloud, execution in Edge and execution in mixed environment (Fog), for



varying number of devices and task dependency depth. Furthermore, we compared the times obtained for setting up the same application with and without tool to figure out how much is the speed up of the whole procedure.

For evaluation purposes, we created infraction test application [3], which consists of several tasks, packed as Docker containers, each of them running PHP or database server and performing some data-related operations: data storage/retrieval or computation. Fig. 6 presents the structure of the test application.

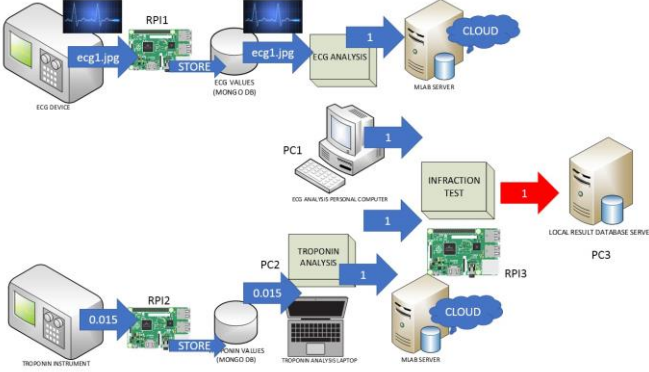


Fig. 6. Test application overview

Table II shows an overview and description of tasks used to construct our case study application. Each of the tasks actually corresponds to a separate Docker image. The first column presents a name of the task. The second column contains the name of the technology used by the task. The third column (environment) tells us if the task is executed in Cloud or within the Edge of our network during the experiment. The last column indicates the computing architecture of the device executing the considered task.

TABLE II  
TASK DESCRIPTION

| Task Name            | Tech.   | Env.  | Architecture |
|----------------------|---------|-------|--------------|
| ECG read data        | MongoDB | Edge  | ARM          |
| ECG processing       | PHP     | Edge  | ARM          |
| ECG write data       | MongoDB | Cloud | X86          |
| Troponin read data   | MongoDB | Edge  | ARM          |
| Troponin processing  | PHP     | Edge  | X86          |
| Troponin write data  | MongoDB | Cloud | X86          |
| ECG read result      | MongoDB | Edge  | ARM          |
| Troponin read result | MongoDB | Edge  | ARM          |
| Infraction check     | PHP     | Cloud | X86          |
| Result storage       | MySQL   | Cloud | X86          |

Table III shows a summary of results obtained during the experiments. In all cases, we developed the deployment model, relying on the consistency and validation check controls embedded in tool, and then we generated the code which has been executed by Docker Swarm master, responsible for allocating tasks to corresponding devices. The infrastructure used for testing consists of following devices: three Raspberry Pi model 3, two laptops and two virtual machines in cloud. One of the laptops had role of Docker Swarm master, while the other devices were used as Swarm workers executing the allocated tasks.

The first column in Table III includes the names we have assigned to the considered application. Each of the applications is constructed from tasks described in Table II. The *Max depth* column represents the length of the longest path in model's dependency graph. For example, level 1

represents an application with services that are all stand-alone and where none of the services depends on any other service. Level 2 represents a two-tier application, where a service (for example, a database) needs to be deployed before another one (the main application) can be deployed and configured. Thus, in general, *Max depth* in conjunction with the used technologies gives an idea of the complexity of the system under deployment. To quantify the time needed to generate and deploy applications, we repeated the experiments 10 times and computed the average duration for generating the shell script commands (third column), and deploying it (forth and fifth columns). *Gen. time* presents the time needed to automatically generate Docker Swarm commands from user-drawn deployment diagram. *Cold Deploy Time* stands for the time which takes to deploy the application taking into account also the time needed to download corresponding images from Docker Hub (in our laboratory the download speed was 100Mbps), while *Warm Deploy Time* is deployment time in case when all the devices had already obtained the necessary images and just need to run them. *Design Time* is time needed to draw a deployment diagram using the tool.

In what follows, the results obtained during experiments are presented (Table III).

TABLE III  
EVALUATION RESULTS

| Blueprint       | Max Depth | Gen. Time [s] | Cold Deploy Time [s] | Warm Deploy Time [s] | Design Time [s] |
|-----------------|-----------|---------------|----------------------|----------------------|-----------------|
| ECG test        | 3         | 2.58          | 75                   | 8.56                 | 32              |
| Troponin test   | 3         | 2.72          | 84                   | 8.90                 | 35              |
| Infraction test | 5         | 4.11          | 198                  | 11.42                | 57              |

From the Table III, we can see that the shell script code generation time strictly depends on the number of devices and tasks (the size of a model), while the warm deploy is much quicker than cold deploy, as expected, due to exclusion of time spent to download the necessary data.

After that, we compared the time spent to completely set up the application using the tool and without it, for the case of infraction test application. It took more than 10 minutes to set it up manually, without any tool intervention in case of cold deploy and around 7 minutes in case of warm deploy. Note that this estimate assumes that user is familiar with Docker Swarm infrastructure.

## V. DISCUSSION

The author has already presented in [9] a tool for automatic deployment of data-intensive applications using a similar metamodeling approach, based on [11]. However, the previous work is focused on Cloud applications, based on traditional virtualization methods and using TOSCA YAML [12] deployment configuration. This paper presents a solution which extends support to Fog execution environments, with IoT devices support, based on Docker containers, inspired by approach presented in [1][2][3], utilizing Docker Swarm as the underlying infrastructure.

Considering the results obtained, we can conclude that this tool does not only extend the support to IoT use cases in Fog environment, but also provides support for computation

task movement and speeds up the application configuration, setup and deployment process. In case of cold deploy, we accelerate the procedure 2.5 times, while it is around 7 times for warm deploy. This is due to fact that, in case of warm deploy, when the download time is eliminated, we actually compare the code generation and design time against the time needed for manual setup operations. However, the relative speed-up obtained in previous work [9][11] was greater. This can be explained by the fact that the manual virtual machine setup and configuration generally takes more effort compared to container-based approach, while the code transformation component of the tool more or less has the same performance in both cases. When it comes to absolute deployment duration, it is much quicker in container-based approach, as there is no need to spawn a full operating system (using hypervisor) each time we deploy a task, which is an advantage in favor of container-based systems, especially when we take into account that low-power Raspberry Pi devices are involved.

## VI. CONCLUSION AND FUTURE WORK

The result of our work is a tool which enables flexibility of applications executed in Fog environments by providing the capability of computation task movement, abstracted as Docker containers between devices with different computing architecture (ARM/x86) and location (Cloud/Edge). Metamodeling is used to define rules and constraints which not only ensure the correct deployment configuration according to the given rules, but also speeds up the configuration and deployment procedure, from 2.5 up to 7 times.

However, the capabilities for increasing scalability and fault-tolerant features of applications executed in Fog environments, offered by Docker Swarm infrastructure, which could future improve the flexibility are not exploited in this paper, and, therefore, considered as topic for future work. Furthermore, increasing the flexibility by enabling the data movement capabilities and support for other container cluster infrastructures, such as Kubernetes [10].

Furthermore, inspired by [13], the idea is to extend and customize the approach presented in this paper in order to apply it for model-driven, semantic-enabled EDL[14] testbed code generator for IoT experimentation environments, in order to support RAWFIE [15] Horizon 2020 project.

## ACKNOWLEDGMENT

This work has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Grant Agreement No 645220, project RAWFIE (Road-, Air- and Water- based Future Internet Experimentation).

## REFERENCES

- [1] Pierluigi Plebani, David Garcia-Perez, David Bernbach, Frank Pallas, Stefan Tai, "Moving Data in the Fog: Information Logistics with the DITAS Cloud Platform", 2017, pp. 1-7.
- [2] Pierluigi Plebani, David Garcia-Perez, Maya Anderson, David Bernbach, Cinzia Cappiello, Ronen I. Kat, Frank Pallas, Barbara Pernici, Stefan Tai, Monica Vitali, "Information Logistics and Fog Computing: The DITAS Approach", 2017, pp. 130-136.
- [3] N. Petrovic, "Enabling Flexibility of Data-Intensive Applications on Container-Based Systems with Node-RED in Fog Environments", Milan, Country: Italy. of Politecnico di Milano, 2017, ch. 3, pp. 18-62.
- [4] M. Brambilla, J. Cabot, Manuel Wimmer, "Model-Driven Software Engineering in Practice", 2012, pp. 13-16.
- [5] E. Di Nitto, E. Di Nitto, D. Tamburi, M. Guerriero, M. Artac, T. Borovsak, "DevOps: introducing infrastructure-as-code", in Proc. 39th International Conference on Software Engineering Companion ICSE-C'17, Argentina, 2017, pp. 497-498.
- [6] ADOxx Metamodelling Platform, [On Line]. Available on: <https://www.adoxx.org/live/meta-modelling-platforms-hierarchy>
- [7] Saxon XSLT, [On Line]. Available on: <http://saxon.sourceforge.net/>
- [8] XSLT intro, [On Line]. Available on: [https://www.w3schools.com/xml/xsl\\_intro.asp](https://www.w3schools.com/xml/xsl_intro.asp)
- [9] N. Petrovic, "Tool for Modelling and Automatic Deployment of Data-Intensive Applications", IEESTEC 10<sup>th</sup> Student projects conference, Niš, Serbia, 2017, pp. 25-30.
- [10] Kubernetes, [On Line]. Available on: <https://kubernetes.io>
- [11] E. Di Nitto, D. Tamburi, M. Guerriero, M. Artac, T. Borovsak, „Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements – 2.4 Deployment Abstractions“, [On Line]. Available on: [http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2017/07/D2.4\\_Deployment-abstractions-Final-version.pdf](http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2017/07/D2.4_Deployment-abstractions-Final-version.pdf)
- [12] Apache ARIA TOSCA Orchestration Engine, [On Line]. Available on: <http://ariatosca.incubator.apache.org/>
- [13] Nejkovic, V., Tosic, M., Jelenkovic, F., Ontologies framework for semantic driven code generation over testbed premises, 7th International Conference on Information Society and Technology, Kopaonik, Serbia, March, 12th-15th 2017.
- [14] K. Kolomvatsos, M. Tsiroukis and S. Hadjiefthymiades, "An Experiment Description Language for Supporting Mobile IoT Applications", pp. 461-486, [On Line] Available on: [https://www.riverpublishers.com/pdf/ebook/chapter/RP\\_9788793519114C15.pdf](https://www.riverpublishers.com/pdf/ebook/chapter/RP_9788793519114C15.pdf)
- [15] RAWFIE: Road-, Air- and Water- based Future Internet Experimentation, [On Line]. Available on <http://www.rawfie.eu/>