Orchestration of Containerized Applications in the Cloud Continuum

Orchestration of containerized applications in the Cloud continuum

Giuseppe Di Modica [†], Antonino Galletta [*], Lorenzo Carnevale [*], Ahmad Alkhansa [‡],
Alessandro Costantini [‡], Daniele Cesini [‡], Paolo Bellavista [†], Massimo Villari [*]

[†] Department of Computer Science Engineering, University of Bologna, Italy
{giuseppe.dimodica, paolo.bellavista}@unibo.it

[*] MIFT Department, University of Messina, Italy
{angalletta, lcarnevale, mvillari}@unime.it

[‡] Center for Research and Development on Information and Communication Technologies (CNAF)
Italian Institute for Nuclear Physics (INFN), Bologna, Italy
{alessandro.costantini, daniele.cesini, ahmad.alkhansa}@cnaf.infn.it

Abstract—Cloud Service orchestration is a hot topic addressed by both the academia and the business world. The increasing diversification of the Cloud offer, coupled with the strong need of flexible and scalable applications demanded by customers, is calling up for efficient and agile mechanisms that streamline the devops practices. The novel opportunity of running software applications or just application portions on resources located on the Edge of the network poses an additional challenge to the already tough problem of flexibly and seamlessly provisioning services. Orchestrators must deal with a continuum of heterogeneous computing resources, contributed by both the Edge and the Cloud, that promises several benefits yet at a higher management cost. In this paper, we introduce a continuum-enabled orchestrator and discuss an application provisioning paradigm that helps the users to easily configure and deploy composite applications. Finally, we showcase the potential of the orchestrator in a simple yet practical smart-city scenario.

Index Terms—Service orchestration, Computing Continuum, Cloud, Edge, IoT, Machine Learning, OCR

I. INTRODUCTION

Cloud computing is a computing paradigm that allows the flexible sharing of scalable resources on an on-demand basis model. Due to the wide spread and large acceptance of the Cloud paradigm in almost all business sectors, we have witnessed a proliferation of cloud service offers in the IT market. From the cloud user perspective, the increased complexity in the management of cloud resources, along with the challenges posed by building and managing scalable applications, has called for new methodologies, practices and software tools that would simplify the provisioning, monitoring and maintenance of applications in the Cloud. With the advent of Edge computing, which aims at integrating the computing power offered by the remote Cloud with private, yet less powerful computing resources available on the customer's premises, the aforementioned problem has further exacerbated. In the Edge-Cloud landscape, which many refer to as the *Cloud continuum*, more agility is demanded to ensure a seamless, flexible, robust and scalable provisioning of distributed applications over heterogeneous computing environments.

This paper builds on the scientific results achieved by IoTwins[1], a EU-funded project which aimed at delivering an open and low-cost platform to code, deploy and maintain distributed Digital Twins (DTs)-based applications in industrial manufacturing settings. With regards to the application deployment goal, that is the primary concern of this work, in IoTwins a TOSCA-compliant [2] orchestration platform was implemented that enforces the provisioning of containerized application in the *continuum* computing, i.e., in a distributed computing environment where Cloud and Edge types of resources coexist. While the focus of the IoTwins project was mainly put on industrial control applications, the IoTwins platform was designed to also support the build, deployment and maintenance of distributed DTs in domains other than the industrial manufacturing one[1]. The platform offers users services to configure generic Docker containers as application building blocks, wire such blocks to form a complex service chain and deploy the chain in an heterogeneous environment comprising both Cloud and Edge resources. Also, the platform is equipped with a repository of general-purpose and ready-to-use dockerized services that users can leverage to easily build data pipelines along the continuum.

The aim of this work is to show that the service provisioning approach developed in the IoTwins project is also viable in a typical smart-city scenario. In the paper, we thoroughly discuss of the Orchestrator component of the IoTwins platform. In particular, we present its architecture and provide some implementation details of the software prototype. The rest of the paper is organized as follows. In Section II we briefly describe some relevant literature works that relate to ours. In Section III, we introduce our orchestrator and discuss some architectural and implementation details. In Section IV, we demonstrate the use of the orchestrator to provision a composite application in the Cloud continuum. Conclusions and future directions are summarised in Section V.

---

[1]In the course of the project, distributed DTs were developed to support the management of complex facilities

II. RELATED WORK

The literature is full of frameworks, software prototypes, models and proposals that address the problem of service orchestration in cloud and multi-cloud environments. Hereafter, we report a subset of the most relevant research contributions. For a more comprehensive and systematic review of the literature, the reader may refer to [3].

Several proposals have been developed by academic and public-funded initiatives. MiCADO (Microservices-based Cloud Application-level Dynamic Orchestrator) [4] is an open-source multi-cloud orchestration and auto-scaling framework for Docker containers, orchestrated by Kubernetes (or alternatively by Docker Swarm). MODAClouds (MOdel-Driven Approach for the design and execution of applications on multiple Clouds) [5] is an open-source design-time and run-time platform for developing and operating multi-cloud applications with guaranteed QoS. mOSAIC [6] is an open-source API and platform for multiple clouds designed and developed within the homonymous EU-funded research project. Application deployment and portability across multiple clouds are facilitated by means of a common API and a high-level abstraction of cloud resources. Cloud4SOA [7] is a multi-cloud broker-based solution developed under the homonymous project, which addresses semantic interoperability and portability challenges at the PaaS layer. TORCH [8] is a TOSCA-based framework for the deployment and orchestration of resources in heterogeneous and multi-cloud environments. The framework proposed by the authors assists the cloud customer in defining application requirements by using standard specification models. Unlike other multi-cloud orchestrators, TORCH adopts a strategy that separates the provisioning workflow from the actual invocation of proprietary cloud services API. The main benefit is the possibility to add support to any cloud platforms at a very low implementation cost. Tolerancer [9] is a resource orchestrator able to face failures in Edge-Cloud environments by monitoring the status of hardware resources and deploying docker containers among the available resources.

In the business sector, there is quite a number of products that are worth being cited. For space reasons, we report here the ones that most relate to the same issues we face in our work. Cloudify [10] is an open-source orchestration framework based on TOSCA. It provides services in order to model applications and automate their entire life-cycle through a set of built-in workflows. Application templates are referred to as blueprints, which are YAML documents written in Cloudify's Domain Specific Language. OpenStack Heat [11] is a service for managing the entire life-cycle of infrastructure and applications within OpenStack clouds. It implements an orchestration engine to launch multiple composite cloud applications based on the native OpenStack Heat Orchestration Template format (HOT). Heat provides support for TOSCA via the independent Heat-Translator project [2] which translates TOSCA templates to HOT. Terraform [12] is an open-source infrastructure-as-code tool for building, changing, and versioning infrastructures in a

platform-agnostic way. It uses its own high-level configuration language known as Hashicorp Configuration Language (HCL), or optionally JSON, in order to detail the infrastructure setup. Despite being non-compliant with any model standards, HCL supports reusability via modules and module composition.

Finally, since in this paper we deal with a license plate recognition use case, we discuss some relevant literature contributions that proposed innovative approaches in the field.

An Automatic License Plate Recognition System for increasing the security of parking, gas station, and highways has been proposed in [13]. The main idea behind this work is to install some cameras in parking, gas station, and tolls, that upload the video to the Cloud where a software tool following the YOLO approach[14] implement a license plate recognition process. Despite the accuracy of the proposed system is very high (higher than 99%), the system can not be installed on mobile nodes. A similar work has been proposed in [15], where authors developed a software tool on MATLAB. The accuracy of the proposed system is lower than other tools proposed in the literature, but is still acceptable. A solution able to reconfigure dynamically video surveillance cameras has been proposed in [16]. The authors considered a Smart City where general purposes cameras can be reconfigured to accomplish different tasks like license plate recognition and vehicle count. In particular, the proposed system is built by means of the Function as a Service (FaaS) paradigm that allows to reconfigure the Edge devices and setting up new services. A license plate recognition system based on Raspberry Pi has been proposed in [17]. The authors leveraged Tesseract, an open-source tool for Optical Character Recognition. The proposed system is able to run the computation on the Edge without sending any frame to the cloud; yet, the accuracy is quite low if compared to Cloud-based solutions.

III. A TOSCA-COMPLIANT AND EDGE-ENABLED SERVICE ORCHESTRATOR

In this section, we discuss architectural and implementation details of the INDIGO PaaS orchestrator, one of the core components of the IoTwins platform [18], [19]. Provided with input instructions, the orchestrator carries out a set of actions that enforce the provisioning of multi-component applications in a distributed and crossing-domain computing environment.

Users provide instructions to INDIGO PaaS in the form of declarative statements expressed in the TOSCA language[2]. TOSCA is a widely-accepted OASIS standard that specifies a language to define the topology of cloud applications and their orchestration. The orchestrator takes in input TOSCA-compliant instructions and translates them into a workflow of application provisioning actions (e.g., deploy, configure, run) that are executed with the support of a workflow engine. The capability of transparently provisioning applications in both Edge and Cloud environments is one of the distinctive feature of the orchestrator, which proved to fulfill the need of flexible provisioning strategies that is typical of heterogeneous computing environments. The high-level architecture of the service orchestrator is depicted in the Figure 1. In the following, we

---

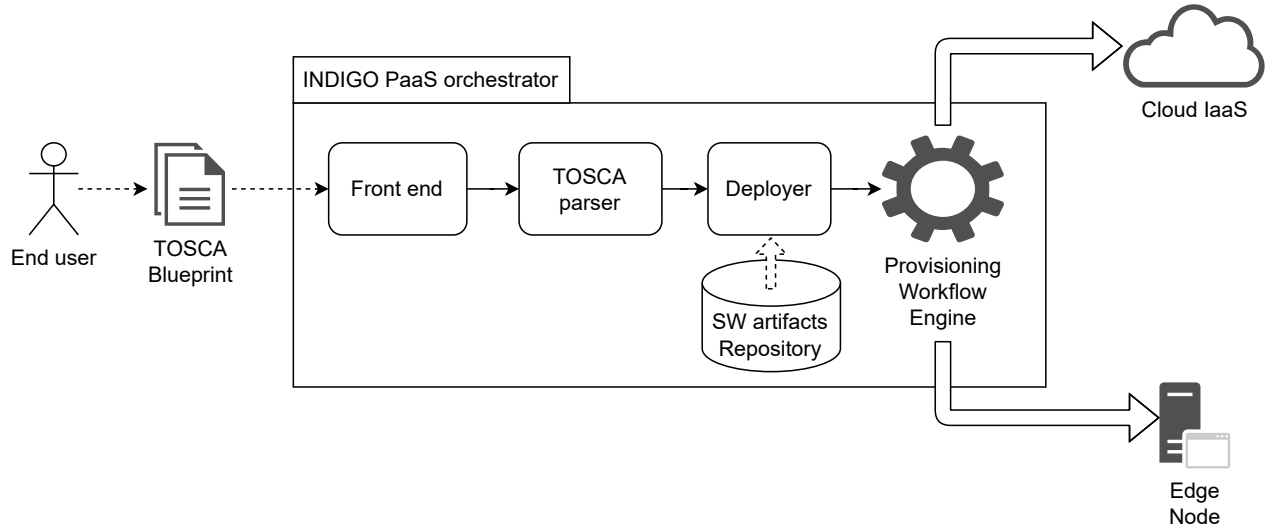[2]https://wiki.openstack.org/wiki/Heat-Translator

Fig. 1. Architecture of the INDIGO PaaS Orchestrator

provide a description of its components and the respective open-source software/libraries that were used to implement their functionalities.

The end-user on the left side of the picture defines the provisioning instructions in the form of a TOSCA blueprint. The *Front end* component provides the user with the interface to submit the TOSCA blueprint. Three types of interfaces are currently offered: a command-line one, implemented by the orchent[3] tool; a web-based interface, that also helps the user in step-by-step building the TOSCA blueprint; RESTful API. Once the blueprint has been ingested, the *TOSCA parser* is called upon to interpret the instructions and transform them into a format that is interpretable to a workflow engine. The TOSCA parsing is carried out with the support of the Alien4TOSCA libraries[4]. The *Deployer* is the component responsible for executing the provisioning task. In particular, it is in charge of i) configuring the application deployment script, by also resolving the links to the software artifacts stated in the application dependencies; ii) feeding the workflow engine with the deployment script. The *Software Artifacts Repository* contains all software artifacts that may be required for the correct application deployment (e.g., app installation tarballs, software libraries, Docker images, etc.). The open source Harbor software[5] was used to implement this component. The *Provisioning Workflow Engine* executes the deployment scripts. Scripts complies with the BPMN notation[20], a well-known standardized language used to model business process workflows; the BPMN-compatible Flowable engine[6] was eventually chosen to act as provisioning engine. In Listing 1, an excerpt of a sample service topology description written in the TOSCA language is shown. The blueprint contains instructions

to schedule a virtual machine from the Cloud infrastructure, featured with given CPU/mem capacity and equipped with a 10GB block storage.

While the orchestrator supports several application provisioning schemes, in this paper we put the focus on the capability of the orchestrator to provision containerised applications, i.e., applications that rely on the runtime environment and the services provided by a container engine. The latest version of the orchestrator prototype fully supports the Docker containerization technology.

A. Support for the Edge

As mentioned earlier in this section, the orchestrator can handle the deployment of multi-component applications, i.e., applications made up of multiple components, each implemented as a Docker container that can transparently be deployed and run in either the Cloud or in an Edge machine. In pursuing the application provisioning goal, the orchestrator will have to: a) reserve a (virtual) computing resource from the available computing infrastructure; b) on top of that, set up and install a Docker runtime environment; c) in such environment, deploy, configure and run the dockerized application components.

In its original implementation, the orchestrator was capable of executing Cloud application provisioning tasks on top of relevant private and public IaaS Clouds, such as Openstack-empowered private clouds [21] and the Amazon AWS. In the course of the project, such a capability was extended to also support the provisioning of Docker applications on top of Edge nodes. In particular, off-the-shelf personal computers (PCs) were selected as target nodes of this extension. PCs are machines with a sufficient computing capacity to run general purpose (yet small-sized) applications. For the extension purpose, PCs were equipped with an Ubuntu Linux distribution on top of which a Mesos Cluster framework [22] is laid. In its turn, this framework is used by the orchestrator to deploy

```
node_templates:
 simple_node:
  type: tosca.nodes.indigo.Compute
  capabilities:
    endpoint:
      properties:
        network_name: PUBLIC
    scalable:
      properties:
        count: 1
    host:
      properties:
        num_cpus: 1
        mem_size: 2 GB
    os:
      properties:
        distribution: Ubuntu
      requirements:
      - local_storage:
        node: block_device
        capability: tosca.capabilities.Attach.
        relationship:
          type: tosca.relationships.AttachesTo
          properties:
            location: /data
            device: vdb
  block_device:
    type: tosca.nodes.BlockStorage
    properties:
      size: 10 GB
```

Listing 1: TOSCA blueprint for the provisioning of a VM equipped with a block storage

and execute application components in the form of Docker containers. At design time, the choice of Mesos Cluster is motivated by the need of leveraging a computing runtime that allowed the implementation of a Cluster of nodes (not just a single one) to be exploited at the Edge, and yet be part of unique pool of resources under the control of the orchestrator. With that extension, in fact, the orchestrator was empowered to exploit a "continuum" of virtual and heterogeneous computing resources (those offered by Cloud and Edge, respectively) on top of which to execute the provisioning of distributed, multi-component applications.

Along with the Apache Mesos, two more frameworks were deployed in the Edge machine: Marathon [23], a production-grade container orchestration platform that can launch applications and provide scaling and self-healing for containerized workloads, and Chronos [24], a fault-tolerant scheduler that runs on top of Apache Mesos and handles short-lived jobs.

B. A simplified application provisioning paradigm

In the course of the project, an application development paradigm was also defined that enables software developers to easily design their distributed application by composing small *building blocks* (the Docker containers, indeed), and to deploy them along the Cloud-Edge continuum based on specific requirements and needs. The development process defined by the paradigm is called "Toskerization", as it recalls the two steps that needs to be taken in order to shape an application to fit the orchestrator's provisioning process: 1) dockerization of the application's building components; 2) definition of TOSCA blueprints representing the deployment instruction for the components.

According to step 1) of the paradigm, application developers need to shape their own application as a "chain" of docker-ized components and configure adequately each docker's I/O parameters. To ease the chain build step, a public repository of already available and Toskerized Docker components can also be accessed by developers for reuse in their application. Also, developers are allowed and encouraged to share newly developed components with the community via the same repository. With regards to step 2) of the paradigm, for all components developers must craft TOSCA blueprints that specify in a standardized way the instructions to enforce the component provisioning.

While the former version of orchestrator supports the provisioning of a multi-component application over a pool of computing resources belonging to the same domain (e.g., that of a Cloud infrastructure), a new component was developed to enforce the cross-domain provisioning, i.e., the provisioning of components across different administrative domains (e.g., a public Cloud infrastructure and an on-premise Edge machine). The *Pipeliner* Docker component is charged with the execution of cross-domain provisioning tasks. The Pipeliner is a simple web server coded in Python that interacts with the orchestrator via REST API for the achievement of the provisioning task. It is Toskerized and is available in the repository of publicly accessible components. In Listing 2 an exerpt of the Pipeliner's TOSCA blueprint is shown. When a cross-domain application provisioning need to be executed, the Pipeliner will be deployed by the orchestrator as the first link of the components' chain, then it will collaborate with the orchestrator to deploy the rest of the components in the chain.

IV. DEPLOYMENT USE CASE IN A SMART-CITY SCENARIO

During the IoTwins project, the orchestrator was heavily tested in I4.0 use cases proposed by the industrial partners. To prove that the the orchestrator can be effective in different settings, we tested its capabilities in a smart-city use case scenario. In particular, we considered a scenario where the the local police needs to localize in-motion vehicles by their license plate number. The reason to track such vehicles could be several: identify the position of a vehicle that has just been stolen, identify vehicles that run away after causing an accident, verify that a specific license plate number is used by just one vehicle, etc.

The web is full of open-source and commercial tools that run video/images analytics and detect the license plate numbers of vehicles. We will refer to such type of applications as automatic license plate recognition (ALPR). ALPR tools are emplyed, e.g., in parking assistance systems, automated toll booths, admittance control in restricted traffic areas, etc. Almost all ALPR systems leverage trained AI models to accomplish license plate recognition tasks. While most of ALPRs run in the Cloud, such models have proven capable to execute also on low computing power hardware, thus fostering their deployment also in small computing nodes located in the urban infrastructure (e.g., WiFi access points, base stations, or

```
topology_template:
  inputs:
    token:
      type: string
      description: token for authorizing deployments
      required: yes
    urls:
      type: string
      description: tosca templates urls
      required: yes
    parameters:
      type: string
      description: input parameters for the templates
      required: yes
    groups:
      type: string
      description: user groups for each deployment
      required: yes

  node_templates:
    pipeline:
      type: tosca.nodes.indigo.Container.
                    Application.Docker.Marathon
      properties:
        environment_variables:
          OIDC_TOKEN : { get_input: token }
          PARAMETERS : { get_input: parameters }
          TOSCA_URLS : { get_input: urls }
          USER_GROUPS : { get_input: groups }
        artifacts:
          image:
          file: iotwins-harbor.cloud.
                    cnaf.infn.it/infn/infn-pipeline-edge:v1.0
          type: tosca.artifacts.
                    Deployment.Image.Container.Docker
        requirements:
        - host: pipelinedockerruntime

    pipelinedockerruntime:
      type: tosca.nodes.indigo.Container.Runtime.Docker
      capabilities:
        host:
          properties:
            num_cpus: 1.0
            mem_size: 1 GB
            publish_ports:
                - protocol: tcp
                  source: 2546
```

Listing 2: TOSCA blueprint for the provisioning of the Pipeliner docker container

even dashboard cameras mounted on vehicles), which we refer to as Edge devices.

Coming back to the need of the police to identify a vehicle that is moving in a wide area, the solution we propose aims to exploit the capability of multiple urban Edge devices to capture images from the road in real time and to also run light elaboration on these images. Let us analyze some possible options. Letting Edge devices to stream all captured data to the Cloud where an ALPR system runs is doable but expensive from the bandwidth occupation point of view. On the contrary, letting the Edge devices do both image capturing and license plate recognition might not be viable, as it fails to guarantee service quality - some devices might not be able to bear the computing burden - and would still require that evidences (captured images containing license plates) be streamed to a central storage owned by the administration. Analyzing the image processing done by ALPR systems, a pipeline of smaller and lighter tasks is clearly identifiable: 1) frame wise, identification of a vehicle using an car detection model; 2) on the frame where vehicles have been identified, identification of

the license plate using a license plate detection model; 3) on frames containing license plates, recognition of the displayed numbers.

Therefore, we propose that Edge devices capture and elaborate video streams in real-time, seeking for images that contains license plates (step 1 and 2). Such images would then be sent over to a remote computing infrastructure (e.g., the Cloud) owned by the city administration, where the final step of numbers recognition (step 3) is run. With this approach, data are therefore sent to the Cloud only when a license plate is detected, limiting network congestion and the transfer of sensitive data that are not useful for the use case.

For the test purpose, we will make use of an existing ALPR service publicly available on the web[7], which we split in two smaller services and dockerized to fit out Edge/Cloud setting. We remark that the objective of the test is not to analyze the performance of the ALPR solution, but rather to prove the capability of the orchestrator to easily provision a multi-component application in the cloud continuum.

The license plate detection model service (LPD) is built on top of a writer-reader concurrency problem. The writer is a RESTful server implemented in Python 3.10 using the Flask web framework. The server has a frame upload POST API that stores the image in the file system for further elaboration. The reader accesses the file system and gives the frame in input to a YOLO model able to detect and square a license plate. The output of the model is sent over a message broker (i.e., RabbitMQ). The license plate number recognition service (LPNR) subscribes to the message broker for receiving the frame and starting a thread that performs optical character recognition (OCR). This is implemented in Python 3.10 and easyocr, which supports more than 80 languages.

We used our orchestrator to enforce the provisioning of the just described ALPR application in a computing continuum environment that includes one Edge machine and one private Cloud Infrastructure. We assume that: i) the Cloud infrastructure runs the Openstack framework; ii) the orchestrator is deployed in a VM instantiated in the Cloud; iii) an Edge machine is available in an administrative domain different than the Cloud and is equipped with the Mesos/Marathon runtime. There follows a description of all steps we took for packing up the application and provisioning it through the orchestrator.

Preliminarily to the provisioning step, we prepared the required orchestrator input. First, we backed the two docker images representing the LPD and the LPNR respectively. To support the data communication between the two Docker containers (i.e., stream of images flowing from LPD to LPNR), we lean on a RabbitMQ message broker, whose docker implementation is already available in the public repository (Software Artifact Repository shown in Figure 1) that is co-located on the orchestrator premises. Such a choice enables an easy scale of the testbed in the case that more Edge instances join the scenario. The reader may refer to this git repo[8] to

---

[7]https://github.com/mftnakrsu/Automatic_Number_Plate_Recognition_YOLO_OCR

[8]https://github.com/ahmadalkhansa/license-plate-detection-microservice.git

```yaml
topology_template:
  inputs:
#....omitted code.....#
    rabbitmq-username:
      type: string
      description: RabbitMQ username
      required: yes
    rabbitmq-password:
      type: string
      description: RabbitMQ password
      required: yes
    rabbitmq-queue:
      type: string
      description: RabbitMQ queue
      required: yes
    rabbitmq-host:
      type: string
      description: RabbitMQ broker
      required: yes
#....omitted code.....#
  node_templates:
    lpd:
      type: tosca.nodes.indigo.Container.
                    Application.Docker.Marathon
      properties:
        force_pull_image: no
        environment_variables:
          RUSER: { get_input: rabbitmq-username }
          RPASSWORD: { get_input: rabbitmq-password }
          RHOST: { get_input: rabbitmq-host }
          RPORT: { get_input: rabbitmq-port }
          RQUEUE: { get_input: rabbitmq-queue }
        uris: []
        artifacts:
          image:
            file: iotwins-harbor.cloud.cnaf.infn.it/
                        infn/infn-lpd-rabbitmq:1.1
            type: tosca.artifacts.Deployment.
                        Image.Container.Docker
      requirements:
      - host: lpddockerruntime

    lpddockerruntime:
      type: tosca.nodes.indigo.Container.
                    Runtime.Docker
```

Listing 3: TOSCA blueprint for the provisioning of the LPD docker component

check out how the LPD docker image was configured to interface with the message broker.

Once backed up, the two images were uploaded to the public repository. Next, we defined the TOSCA blueprints of the two components. In Listing 3, we show an excerpt of the LPD's TOSCA blueprint.

We remark that the repository already contains the *Pipeliner*, a Docker component that supports the deployment and pipelining of components when there is the need of provisioning distributed components across different domains (which is the case we are addressing, indeed). To trigger the application provisioning, we just needed to submit a request to orchestrator's Front end. We opted to submit the request via the *orchent* client, and provided it with proper parameters and the TOSCA blueprints of the application components. In listing 4 we report the shell script to invoke the provisioning request via the orchent tool with the mentioned input. The script is requesting the provisioning of the LPD-RabbitMQ-LNPR pipeline. The *urls* set of parameters specify the TOSCA blueprints of components to be deployed as well as the deployment order. The *groups* parameters set indicate the deployment location of components listed in the urls. According to these

```
orchent depcreate pipeliner-v-1-0.yaml '{
    "params": "{
        "rabbitmq-username":"test",
        "rabbitmq-password":"test",
        "rabbitmq-queue":"test"}",
    "urls":"{
        "rabbitmq-v-1-7.yaml",
        "lpd-rabbitmq-v-1-1.yaml",
        "lnpr-rabbitmq-v-1-1.yaml"}",
    "groups":"{
        "TB08/cloud",
        "TB08/edge",
        "TB08/cloud"}"
}'
-g TB08/cloud
```

Listing 4: TOSCA blueprint for the provisioning of the LPD docker component

provisioning configurations, RabbitMQ will be deployed in the Cloud, afterwards LPD will be deployed in the Edge and finally LNPR will be deployed in the Cloud. Parameters in the *params* sections are used set up the RabbitMQ instance and to configure LPD and LNPR to exchange data via the RabbitMQ. The provisioning strategy can be easily tuned by playing with the mentioned parameters. For instance, if the user wanted to deploy all components in the Edge, they would just need to change the *groups* parameters configuration into the following:

"*groups*" : {"*TB*08/*edge*","*TB*08/*edge*","*TB*08/*edge*"}

In a similar way, they could instruct the Pipeliner to deploy all components in the Cloud. In the Figure 2, we depicted a UML sequence diagram that captures the provisioning steps triggered by the script shown in listing 4. Upon the reception of the provisioning request, the orchestrator executes the deployment of the Pipeliner service which, in its turn, issues as many deployment requests as the number of input TOSCA blueprints. We remark that requests issued by the Pipeliner are submitted to the orchestrator via the REST API offered by the Front End.

To test that that the provisioning terminates successfully, i.e., all components were deployed and configured in a correct way, we stimulated the ALPR application with a stream of sample images. The LPD correctly identified the images containing license plate objects and published them to the RabbitMQ broker. The latter delivered the images to the LNPR that eventually managed to read the license plate numbers depicted within.

V. CONCLUSIONS AND FUTURE WORK

The Cloud continuum is a complex computing context which includes computing resources from the Cloud, Edge and IoT environments and is built on top of an aggregation of network infrastructures and services that such environments can offer. In spite of the many opportunities and advantages, there are several challenges posed by the continuum paradigm, of which the operation cost seems to be the toughest one. In this paper we have proposed a service orchestrator that takes advantage of the cloud continuum features and tries
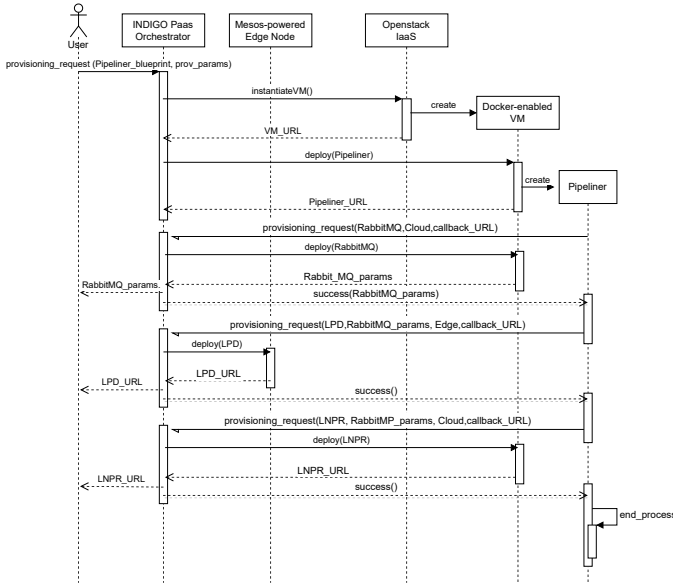
Fig. 2. UML sequence diagram of the application provisioning

to cut the costs for the provisioning of composite applications. In the future, we will provide the orchestrator with the intelligence to sense the available computing environment and to autonomously pick the best provisioning scheme to enforce according to the level of service quality claimed by the application owner.

REFERENCES

[1] The IoTwins consortium, "The IoTwins project." https://www.iotwins.eu/, 2019 - 2022. Last accessed in Dec 2022.
[2] OASIS, "TOSCA Simple Profile in YAML Version 1.3." http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html, Feb. 2020. Last accessed in Dec 2022.
[3] O. Tomarchio, D. Calcaterra, and G. D. Modica, "Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks," *Journal of Cloud Computing*, vol. 9, no. 1, 2020.
[4] T. Kiss, P. Kacsuk, J. Kovacs, B. Rakoczi, A. Hajnal, A. Farkas, G. Gesmier, and G. Terstyanszky, "MiCADO—Microservice-based Cloud Application-level Dynamic Orchestrator," *Future Generation Computer Systems*, vol. 94, pp. 937 – 946, 2019.
[5] E. D. Nitto, P. Matthews, D. Petcu, and A. Solberg, *Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach*. 2017.
[6] D. Petcu, B. D. Martino, S. Venticinque, M. Rak, T. Máhr, G. E. Lopez, F. Brito, R. Cossu, M. Stopar, S. Šperka, and V. Stankovski, "Experiences in building a mOSAIC of clouds," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, May 2013.
[7] E. Kamateri, N. Loutas, D. Zeginis, J. Ahtes, F. D'Andria, S. Bocconi, P. Gouvas, G. Ledakis, F. Ravagli, O. Lobunets, and K. A. Tarabanis, "Cloud4SOA: A Semantic-Interoperability PaaS Solution for Multi-cloud Platform Management and Portability," in *Service-Oriented and Cloud Computing, ESOCC 2013* (K.-K. Lau, W. Lamersdorf, and E. Pimentel, eds.), pp. 64–78, 2013.
[8] O. Tomarchio, D. Calcaterra, G. Di Modica, and P. Mazzaglia, "Torch: a tosca-based orchestrator of multi-cloud containerised applications," *Journal of Grid Computing*, vol. 19, no. 1, 2021.
[9] A. Al-Dulaimy, C. Sicari, A. V. Papadopoulos, A. Galletta, M. Villari, and M. Ashjaei, "Tolerancer: A fault tolerance approach for cloud manufacturing environments," in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, 2022.
[10] Cloudify, "Cloudify." http://cloudify.co/, 2019. Last accessed on Dec 2022.
[11] OpenStack, "OpenStack Heat." https://wiki.openstack.org/wiki/Heat, 2016. Last accessed on Dec 2022.
[12] HashiCorp, "HashiCorp Terraform." https://www.terraform.io/, 2019. Last accessed on Dec 2022.
[13] N. Saif, N. Ahmmed, S. Pasha, M. S. K. Shahrin, M. M. Hasan, S. Islam, and A. S. M. M. Jameel, "Automatic license plate recognition system for bangla license plates using convolutional neural network," in *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, pp. 925–930, 2019.
[14] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December, pp. 779–788, 2016.
[15] R. Chowdhury, F. Rabby, M. S. Rahman, and M. A. Razzak, "Identification of unauthorized vehicles by license plate recognition through image processing," in *2021 5th International Conference on Electronics, Materials Engineering & Nano-Technology (IEMENTech)*, pp. 1–4, 2021.
[16] A. Galletta, A. Ruggeri, M. Fazio, G. Dini, and M. Villari, "Mesmartpro: Advanced processing at the edge for smart urban monitoring and reconfigurable services," *Journal of Sensor and Actuator Networks*, vol. 9, no. 4, 2020.
[17] A. Firasanti, T. E. Ramadhani, M. A. Bakri, and E. A. Zaki Hamidi, "License plate detection using ocr method with raspberry pi," in *2021 15th International Conference on Telecommunication Systems, Services, and Applications (TSSA)*, pp. 1–5, 2021.
[18] A. Costantini, G. Di Modica, J. C. Ahouangonou, D. C. Duma, B. Martelli, M. Galletti, M. Antonacci, D. Nehls, P. Bellavista, C. Delamarre, and D. Cesini, "Iotwins: Toward implementation of distributed digital twins in industry 4.0 settings," *Computers*, vol. 11, no. 5, 2022.
[19] A. Borghesi, G. Di Modica, P. Bellavista, V. Gowtham, A. Willner, D. Nehls, F. Kintzler, S. Cejka, S. R. Tisbeni, A. Costantini, M. Galletti, M. Antonacci, and J. C. Ahouangonou, "Iotwins: Design and implementation of a platform for the management of digital twins in industrial scenarios," in *Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021*, pp. 625–633, 2021.
[20] OMG, "Business Process Model and Notation (BPMN 2.0)." http://www.omg.org/spec/BPMN/2.0/, Jan. 2011. Last accessed on Dec 2022.
[21] The OpenInfra Foundation, "Openstack." https://www.openstack.org/, 2010. Last accessed in Dec 2022.
[22] The Apache Software Foundation, "Apache Mesos." https://mesos.apache.org/, 2009. Last accessed in Dec 2022.
[23] The Apache Software Foundation, "Apache Marathon." https://mesosphere.github.io/marathon/, 2009. Last accessed in Dec 2022.
[24] The Apache Software Foundation, "Apache Chronos." https://mesos.github.io/chronos/, 2009. Last accessed in Dec 2022.