Enabling Modular Design of an Application-Level Auto-Scaling and Orchestration Framework using TOSCA-based Application Description Templates

James DesLauriers, Tamas Kiss, Gabriele Pierantoni, Gregoire Gesmier, Gabor Terstyanszky
Centre for Parallel Computing, University of Westminster, London, UK
j.deslauriers@westminster.ac.uk, {t.kiss, pierang, g.gesmier, g.z.terstyanszky}@westminster.ac.uk

***Abstract*—This paper presents a novel approach to writing TOSCA templates for application reusability and portability in a modular auto-scaling and orchestration framework (MiCADO). The approach defines cloud resources as well as application containers in a flexible and generic way, and allows for those definitions to be extended with specific properties related to a desired container orchestrator chosen at deployment time. The approach is demonstrated in a proof-of-concept where only a minor change was required to a previously used application template in order to achieve the successful deployment and lifecycle management of the popular web authoring tool Wordpress on a new realization of the MiCADO framework featuring a different container orchestrator.**

Keywords—TOSCA, MiCADO, container orchestration, kubernetes, docker swarm

I. Introduction

Cloud adoption by research, public sector and enterprise organizations continues to grow. Having flexible, on-demand access to computing resources and services can result in significant cost and time savings. Moreover, large, upfront capital investments can be replaced by day-to-day operational costs over a longer period of time. There are, however, definite barriers to entry for the scientific research community and smaller institutions that lack the cloud-specific skills and knowledge necessary for shifting to the cloud. Additionally, organizations may struggle with achieving maximum savings due to a lack of flexibility and scalability at the level of the application.

To support these groups, there is the need for a generic framework which provides support for launching and managing a variety of applications in the cloud. The framework should be tied to no specific cloud service provider and should support a mix of public, private and community clouds. It should also provide flexibility at the application level by providing optimized deployment and runtime orchestration with features such as automated scaling and enhanced security. This flexibility should be provided in the form of a single interface which describes the topology of cloud resources and governs the application with user-defined policies specifying performance, cost, security and other requirements necessary to see the application through its lifecycle.

The European funded COLA [1] (Cloud Orchestration at the Level of Application) project set out to address these issues, and design and develop a generic framework outlined above. The proposed framework is called MiCADO [2] (Microservices-based Cloud Application-level Dynamic Orchestrator), a platform for the deployment and dynamic automated scaling of applications in the cloud. MiCADO is entirely open source and implements a microservices architecture in a modular way. The modular design supports varied implementations where components can easily be replaced with a different realization of the same functionality. At the time of writing, the current implementation of MiCADO uses widely applied technologies such as Kubernetes [3] (container orchestrator), Occopus [4] (cloud orchestrator) and Prometheus [5] (monitoring), and some additional custom implemented components. Based on the modular design principles, replacing any of these building blocks is kept as simple as possible.

The user-facing interface for defining the required cloud topology (containers and virtual machines) and the policies which regulate the application and its infrastructure is an Application Description Template [26] (ADT). The ADT is a YAML [6] (YAML Ain't Markup Language) template written in the Oasis Standard TOSCA [7] (Topology and Orchestration Specification for Cloud Applications) language specification and it is designed specifically for portability across this modular framework. Since the MiCADO platform sees a range of users authoring templates for their applications, the ADT has high readability and provides flexible levels of customization and abstraction.

There are two sections to an ADT – one to describe the cloud topology of the application and infrastructure, and one to describe the Quality of Service (QoS) parameters. The policies which describe these non-functional requirements, such as placement, data locality, security, scalability, and performance, were discussed last year in the IWSG 2018 Conference Proceedings [26]. This paper covers the first of the two ADT sections: the section related to the description of the containers and virtual machines which make up the application.

In order to support MiCADO as a truly modular framework, the ADT interface has to be supportive of modularity as well. This modularity presented several challenges for designing such a template, especially insofar as describing the cloud resources so that they could be reused across different implementations of the framework. In following with other projects which have implemented TOSCA as a language, the early ADTs of MiCADO featured topology

types for cloud resources (virtual machines, containers and volumes) which were all strongly related to their respective end components. This meant a loss of portability, since by changing an end component in the implementation, we now required new TOSCA types to be written and then referenced in new ADTs.

This paper presents a novel approach to writing cloud resource types for TOSCA, which offers reusability of that resource by different orchestration tools. The ADT format is compliant with our understanding and interpretation of TOSCA Simple Profile in YAML v1.0 [8], and follows the core values espoused by the newer Oasis Standards, TOSCA v1.1 and v1.2. The format provides more flexibility and control for those ADT authors who understand the underlying technologies of the respective components they are describing cloud resources for. At the same time, the ADT structure still features variable levels of abstraction, which make it possible for users without component-specific knowledge to author ADTs and deploy applications in MiCADO. We present the steps taken to build reusable types for containerized applications in TOSCA and the way to extend them for specific orchestrators as desired. As a proof-of-concept, we refer to a case study where a MiCADO demonstrator application had requirements exceeding the capabilities of the container orchestrator inside MiCADO at the time. We demonstrate a solution where the modularity of MiCADO is leveraged, and the existing ADT is easily reused to deploy the application under a new MiCADO implementation featuring a different container orchestrator.

The rest of the paper is structured as follows. First, the State of the Art is described, including an introduction to TOSCA, and a look at related work covering the adoption and interpretation of TOSCA in industry and academia. In section III a short introduction to MiCADO and its modular design is described. In section IV, we build the base TOSCA types for containerized applications and then demonstrate how those types can be extended for a specific container orchestrator. We conclude with a proof-of-concept demonstration where a single ADT is used to deploy a microservices application to a different implementation of MiCADO, featuring a different container orchestrator.

II. State of the Art

Containerization refers to a more lightweight virtualization approach than that offered by virtual machines, whereby an application is packaged with its specific dependencies on a minimal virtualization layer, usually sharing the host kernel. The most popular container technology of the moment is Docker [9], but there are others gaining popularity such as: cri-o [10], rkt [11], and Mesos Containerizer [12]. Container orchestration refers to the deployment and management of application containers across a cluster of connected servers, or nodes. Popular solutions for the orchestration of containers include: Docker Swarm [13], Kubernetes [14], and Mesos Marathon [15].

The Topology and Orchestration Specification for Cloud Applications is an OASIS Standard for describing the full topology and operational behavior of an application running in the cloud. The topology is defined as building-blocks called *nodes,* which represent components such as the software, virtual machines, storage volumes, and networks which make up the application. The operational behavior is managed by defined *relationships* between the above components and through lifecycle management *interfaces* in the form of scripts, configurations, or API invocations. There are many good resources for TOSCA, ranging from journal publications [16] [17] to the current specification itself - TOSCA Simple Profile in YAML 1.2 [18].

There is an increasing number of industry products and amount of academic research coming out around the topic of TOSCA and descriptive cloud languages. While most TOSCA adopters claim to solve vendor lock-in while offering high-levels of flexibility and portability, not all of them leverage containers, and few describe cloud resources in a way that is portable across a modular framework.

One of the earliest TOSCA runtimes is OpenTOSCA [19], which orchestrates templates written in the original TOSCA XML format. Providing relief for the low readability of XML is an integrated modelling tool called Winery [20], which permits the visual creation of a TOSCA template using a graphical user interface. OpenTOSCA and Winery adhere to the TOSCA v1.0 normative XML specification and have not been extended to supporting containers or container orchestration.

Cloudify [21] is one of the early commercial adopters of TOSCA, and, to support their orchestration framework, have created their own unique Domain Specific Language (DSL) with TOSCA as a base specification. Cloudify is modular insofar as it has been extended with plugins to provide support for different cloud service providers, container platforms, as well as a variety of automation tools. The Cloudify DSL uses strict types: within container orchestration, for example, there is one type defined for creating a non-orchestrated Docker container, another type for a Docker container orchestrated by Docker Swarm, and a third type for a Docker container orchestrated by Kubernetes. Each different type requires key/value pairs or properties specific to the orchestrator acting on it, making reuse of that container definition with a different orchestrator unlikely, or impossible.

ARIA [22] is the now retired open-source project that was born out of Cloudify. The ARIA project was built on the Cloudify code base, and supported its plugins, but diverged with regards to language, keeping strict adherence to the normative DSL, TOSCA Simple Profile in YAML v1.0. The aim of ARIA was to provide a set of TOSCA-inclined tools to support the uptake of a normative TOSCA by other organizations, one such contribution being a Cloudify plugin to support the orchestration of TOSCA normative templates. No specific examples of orchestrating containers were found in ARIA templates.

From an ARIA contributor, came the open-source Puccini [23], a frontend which can currently translate an extended TOSCA v1.1/v1.2 template into a middle-language called Clout, then again into an orchestrator specific language before being fed to that orchestrator. One example involves a TOSCA template translated to Clout, then into Kubernetes manifests before being piped into the Kubernetes command-line tool. There is very strict typing here, with descriptions of applications to be orchestrated with Kubernetes being abstracted at a high level. Applications, along with all of their properties and requirements are first fully defined as new TOSCA types. These types are then imported to and referenced in the TOSCA template to be used at deployment time. As

opposed to extending a generic type with specific properties and requirements for an application, this approach adds complexity, introducing another layer when creating templates to deploy an application.

The open-source Alien4Cloud [24] (Application Lifecycle Enablement for Cloud) is an application management platform which leverages the portability of TOSCA to encourage uptake of the cloud by enterprise organizations. It offers a custom DSL with strict, but not total adherence to TOSCA Simple Profile in YAML v1.0. Plugins and a graphical interface offer support for orchestrating and designing these TOSCA templates using various tools, including Cloudify, Mesos, Kubernetes and Puccini.The Alien4Cloud approach to defining types is less strict, but more complexly layered. The Docker container image itself is defined as a generic type for all Docker images, ignorant of orchestrator. Then, a container runtime must be defined, giving flexibility to authors desiring an alternative runtime. Finally, a container deployment unit is defined, which instructs the framework which container orchestrator should be used. This three-layered approach does offer ease of portability across different orchestration tools, but complicates the initial authoring of a TOSCA template.

Another approach is used in the orchestration management engine TosKer [25]. This approach to defining applications separates the application from the container, defining one type to represent the Docker container, and another to represent the software which may (or may not) run inside that container. This approach provides flexibility for an orchestration engine that seeks to combine containers and traditionally run applications, but adds another layer of complexity at the same time.

As early TOSCA did not inherently support use at run-time, another solution, CAMEL [35] (Cloud Application Modelling and Execution Language), was developed as part of the European-funded PaaSage [36] project and has been used in other large projects such as MELODIC [37] and Cactos [38]. CAMEL provides a dynamic representation of the running instance as a model (models@run-time), where changes to the system are reflected in the model and changes to the model are reflected in the system. An OASIS working group has been investigating and implementing plans to integrate CAMEL-style instance modelling into the TOSCA specification.

The current wide range of approaches to TOSCA were found to be either too complexly layered, or not generic enough to serve the modularity of MiCADO. A previous publication focusing on MiCADO Application Description Templates in TOSCA [26] offers more information on our approach to describing Quality of Service and Non-Functional Requirements and provides the base for the extended approach to describing cloud resources taken in this paper.

III. MiCADO

MiCADO is an application-level multi-cloud orchestration and auto-scaling framework that is currently being developed in the European H2020 COLA (Cloud Orchestration at the Level of Application) project. The concept of MiCADO is described in detail in [2]. In this section a high-level overview of the framework is provided to explain its architecture, building blocks and implementation.

The generic, technology independent architecture of MiCADO is presented in Figure 1. MiCADO consists of two main logical components: Master Node and Worker Node. Master Node is the head of the cluster performing the collection of information on microservices, the calculation of optimized resource usage, the decision making, and the realization of decisions related to handling resources and to scheduling microservices. Worker Nodes are volatile components representing execution environments for the microservices. These nodes are continuously allocated/released based on the dynamically changing requirements of the running microservices. Once a new Worker Node is allocated and attached to the cluster, the Master Node utilizes its resources by allocating microservices on it. The input to MiCADO is a TOSCA-based ADT that will be the focus of this paper.

The MiCADO Master Node (box with dashed line on the left in Figure 1) includes six components. MiCADO Submitter is the primary service endpoint for creating an infrastructure to run an application, and managing this infrastructure and the application itself. The incoming ADT is interpreted by the MiCADO Submitter and related parts are forwarded to other key components. Creating new MiCADO Worker Nodes and deploying application containers on these Worker Nodes are the responsibility of the Cloud Orchestrator and Container Orchestrator components, respectively. The Cloud Orchestrator is responsible for communication with the Cloud API to allocate and release resources, and build up and shut down new MiCADO Worker Nodes when necessary. The Container

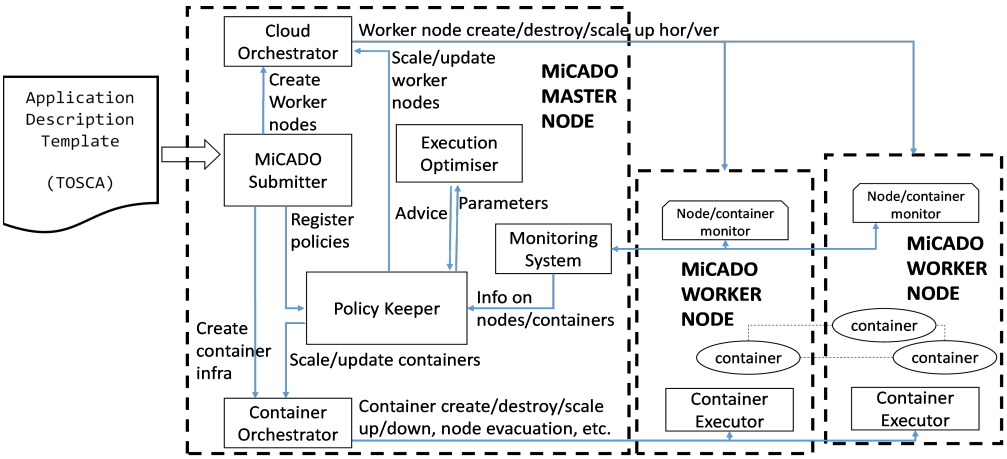

Figure 1. Generic architecture of MiCADO

Orchestrator allocates new microservices (realized by containers) on the Worker Nodes, keeps track of their execution and destroys them if necessary. The Monitoring System collects metrics on worker node resources and on resource usage of the container services, and makes this information available for the Policy Keeper component on the Master Node. It also provides alerting functionality in relation to the measured attributes to detect values that require reaction; these alerts will also be consumed by the Policy Keeper. Based on the metrics and alerts provided by the Monitoring System, the Policy Keeper applies implemented scaling policies to make scaling decisions and call the components responsible for allocating/releasing cloud resources and scheduling container services among the Worker Nodes. Moreover, this component makes sure that the Cloud and the Container Orchestrator are instructed in a synchronized way during the operation of the entire system. Lastly, the Execution Optimizer is a background microservice performing long-running calculations on demand for finding optimized setup of both cloud resources and container infrastructures.

MiCADO Worker Nodes (boxes with dashed line on the right in Figure 1) contain the Node/container monitor that is responsible for measuring the load of the resources and the resource usage of the container services. The measured attributes are then provided to the Monitoring System running on the Master Node. The Container Executor starts, executes and destroys containers upon requests from the Container Orchestrator. Container components are realizing the user services defined in the (container) infrastructure description submitted through the MiCADO Submitter on the Master Node.

The current implementation of MiCADO utilizes Occopus [4], an open source multi-cloud orchestration solution as Cloud Orchestrator that is capable of launching virtual machines on various private (e.g. OpenStack or OpenNebula-based) or public (e.g. Amazon Web Services or CloudSigma [27]) cloud infrastructures, and also via the CloudBroker Platform [28]. For Container Orchestration, the MiCADO versions mentioned in this paper use either Docker Swarm [13], or Kubernetes [14]. The monitoring component is based on Prometheus [5], a lightweight, low resource consuming, but powerful monitoring tool. The MiCADO Submitter and Policy Keeper components were custom implemented during the COLA Project. The current MiCADO prototype does not include the Optimiser component, its design and development is ongoing at the time of writing this paper.

IV. Building an ADT

The approach taken to adopting TOSCA into the COLA Project for use with MiCADO is inherently different than the adoption approach by other frameworks and research activities described in the related work. MiCADO orchestrates at the level of the application. This primarily refers to support for container orchestration, where the assumption is that the application or its microservices have already been packed into one or more container images which are all in a ready-state. MiCADO also supports a so-called VM-only deployment, again with the assumption that the virtual machine images to be deployed contain the necessary libraries and the application is ready to accept a command or input.

For TOSCA, this meant that MiCADO could very simply define two broad types of nodes covering the two major cloud resources – one for virtual machines, and one for containers. This gave us a base node type for each, which could be extended to support a variety of cloud service providers or container runtimes.

- ***tosca.nodes.MiCADO.Compute***
  - *tosca.nodes.MiCADO.Compute.EC2*
  - *tosca.nodes.MiCADO.Compute.OpenStack*
- ***tosca.nodes.MiCADO.Container.Application***
  - *tosca.nodes.MiCADO.Container.Application.Docker*
  - *tosca.nodes.MiCADO.Container.Application.rkt*

The next step was to define the orchestrator within the MiCADO framework that would act on these resources in order to begin and manage their lifecycle. To this end, we leveraged the *interface* type defined within TOSCA. In the TOSCA specification, an *interface* must be defined for each node, to take responsibility for managing the lifecycle of that node. The so-called Standard interface of TOSCA uses deployment and implementation artifacts (typically in the form of shell scripts or Chef or Puppet configurations) to manage that lifecycle through four main stages: create, configure, start, and stop. These script artifacts allow for extra inputs to be fed in at deployment time, defined directly in the interfaces section of the TOSCA template.

In MiCADO, these lifecycle stages are handled by whatever respective orchestrator is responsible for that node. There is no requirement to associate those stages with a script or piece of automation code, but it is still necessary to pass information to said orchestrator, so it can correctly handle each stage of the lifecycle. Deployment artifacts become the required container or virtual machine image. Implementation artifacts are inferred to be the native configuration format belonging to that orchestrator, ready to accept custom inputs just as in the Standard TOSCA interface. To fulfil this requirement, the base interface type is extended for specific MiCADO orchestrators as below.

- ***tosca.interfaces.MiCADO***
  - *tosca.interfaces.MiCADO.Occopus*
  - *tosca.interfaces.MiCADO.Swarm*
  - *tosca.interfaces.MiCADO.Kubernetes*

```
simple-python-app:
  type: tosca.nodes.MiCADO.Container.Application.Docker
  properties:
     entrypoint: "python3 app.py"
     container_name: my-app
     environment:
      PYTHON_PATH: /usr/lib/python3
     tty: true
     stdin_open: true
  artifacts:
   image:
     type: tosca.artifacts.Deployment.Image.Container.Docker
     file: python:3-slim
     repository: docker_hub
  interfaces:
    Kubernetes:
      create:
        inputs:
          strategy:
            type: Recreate
```

Figure 2. Application Description Template for a generic single-container application. Described using the Docker Compose naming conventions, but orchestrated with Kubernetes.

The node and interface types described above are the base for describing cloud resources in all MiCADO ADTs, and by the end of project will be used in over 20 different use case application demonstrators, each with a variety of different requirements and all being able to benefit from a choice of orchestrator. The next section describes the approach taken to define a generic node type for applications in Docker containers, which can easily be reused for defining a generic node type for other resources such as virtual machine instances or other container runtimes.

A. *Defining a generic container type*

To define the generic set of options which the user could set in the properties section of a Docker container node type, we made a quick review of the options available and inputs required when orchestrating a Docker container with each of Swarm, Kubernetes and Mesos. Any options which were clearly related to orchestration, such as scheduling or update strategies, were dismissed and would become the available inputs for the interfaces section of each orchestrator, to further control lifecycle stages. From the remaining options, though naming and grammar varied slightly, a set of base properties was apparent and became the properties section of the Docker container node type.

To support portability, the generic properties aim to support the naming and grammar of all major orchestration platforms, so moving from one to the other requires no changes to them. The remaining options, tightly related to orchestration can still be set and modified as inputs in the interface section of the definition. To offer full support for the specific settings offered by each orchestrator, the naming and grammar of these options are still orchestrator-specific. Figure 2 provides an example of the portability and extended support offered by this approach. Here, a single simple container is defined in an ADT using Swarm-style naming, then orchestrated by Kubernetes. The generic container properties (in the upper portion of the definitions) are flexible in that they can be expressed using any supported orchestrator's nomenclature, and then scheduled by any supported orchestrator. When selecting the orchestrator (in the lower portion of the definition), other orchestrator-specific options can be specified under inputs, so long as they match the naming and grammar of that specific orchestrator. Here, an update strategy is defined, not for the container, but for the Kubernetes workload, so orchestrator-specific grammar is required.

```
wordpress:
  type: tosca.nodes.MiCADO.Container.Application.Docker
  properties:
     labels:
       tier: frontend
     environment:
      WORDPRESS_DB_HOST: wordpress-mysql
      WORDPRESS_DB_PASSWORD: admin
     resources:
       reservations:
         cpus: "0.6"
     ports:
     - target: 80
       published: 30010
  artifacts:
   image:
     type: tosca.artifacts.Deployment.Image.Container.Docker
     file: wordpress:5.0.3-apache
     repository: docker_hub
  requirements:
    - volume:
        node: nfs-volume
        relationship:
          type: tosca.relationships.AttachesTo
          properties:
            location: /var/www/html
  interfaces:
    Swarm:
      create:
        inputs:
          update_config:
            parallelism: 0

wordpress-mysql:
  type: tosca.nodes.MiCADO.Container.Application.Docker
  # description of this service omitted for brevity

nfs-server-pod:
  type: tosca.nodes.MiCADO.Container.Application.Docker
  properties:
    privileged: True
  # other properties & artifacts omitted for brevity
  interfaces:
    Swarm:
      create:
        inputs:
          networks:
            word_net:
              ipv4_address: 10.96.0.240

nfs-volume:
  type: tosca.nodes.MiCADO.Container.Volume
  interfaces:
    Swarm:
      create:
        inputs:
          driver_opts:
            type: nfs
            o: "addr=10.96.0.240,rw"
            device: ":/"

word_net:
  type: tosca.nodes.MiCADO.network.Network.Swarm
```

Figure 3. Application Description Template describing the container topology of a Wordpress microservices architecture for deployment with Docker Swarm

V. Proof of Concept

One application demonstrator of the MiCADO project planned for the deployment of WordPress [29], a popular content management framework as a microservices architecture. The demonstrator sees frontend (WordPress), backend (MySQL [30] database) and shared storage server (NFS [31], (Network File System)) components deployed in containers to MiCADO worker nodes. The NFS server

```
wordpress:
  type: tosca.nodes.MiCADO.Container.Application.Docker
  # properties, artifacts, requirements unchanged
  interfaces:
    Kubernetes:
      create:
        inputs:
          strategy:
            type: Recreate

wordpress-mysql:
  type: tosca.nodes.MiCADO.Container.Application.Docker
  # description of this service omitted for brevity

nfs-server-pod:
  type: tosca.nodes.MiCADO.Container.Application.Docker
  properties:
    privileged: True
  # other properties & artifacts omitted for brevity
  interfaces:
    Kubernetes:
      create:
        inputs:
          clusterIP: 10.96.0.240

nfs-volume:
  type: tosca.nodes.MiCADO.Container.Volume
  interfaces:
    Kubernetes:
      create:
        inputs:
          nfs:
            server: 10.96.0.240
            path: /
```

Figure 4. Modified ADT from Figure 3 for deployment with Kubernetes

container is linked to a storage volume and the MySQL and WordPress containers mount that volume for persistent and shared storage. MySQL credentials are passed to the WordPress frontend to connect it with the database container. Lastly, scaling policies are attached to the frontend and virtual machine worker nodes so they scale up and down to meet a variable network load under a benchmarking test. A single ADT was written to describe and deploy the virtual machine infrastructure, the three Docker containers and their network, the volume linking to the NFS server and the set of policies which would regulate the automated scaling. The container, network and volume descriptions can be seen in the snippet in Figure 3.

When implementation of this demonstrator began, MiCADO featured Docker Swarm as the container orchestrator. Running the NFS server in a Docker container requires elevated privileges not provided in containers by default. The Docker runtime permits elevating these privileges with the --privileged or --cap-add flags, however, as recently as the current stable release at the time of writing (18.09), Docker Swarm orchestration does not offer support for either of these options. As it was implemented, MiCADO was unable to support this particular application demonstrator. However, because of its modular design and supportive TOSCA interface, it was possible to swap out Docker Swarm for another container orchestrator to offer a solution which could support running the NFS share as part of a WordPress deployment. A review of other widely used container orchestration tools showed that both Kubernetes and Mesos Marathon supported elevating privileges in orchestrated containers. Because of its popularity and the range of other features it supports, Kubernetes won the candidature to replace Swarm as the container orchestrator.

On the implementation side, leveraging the modularity of MiCADO was straightforward. The configuration of Docker Swarm and its visualizer component were removed from the Ansible [32] playbook responsible for the installation of MiCADO, and the installations of the Kubernetes core-components and dashboard were added in their place. MiCADO worker nodes were instructed to join a Kubernetes cluster instead of a Swarm cluster as they had done previously. Port forwarding rules were changed and security enablers were rewritten to support the Kubernetes networking approach. Lastly, a new adaptor was written for the MiCADO Submitter component to support translation to and execution of Kubernetes manifests.

Because of the design and modular support built into the TOSCA ADT, only orchestrator specific options had to be changed. The policies and virtual machine definitions remained identical, as their respective components were not changed for this implementation. Figure 4 shows the necessary changes to the container and volume descriptions of the ADT. The core definition of Docker containers, as expressed by their properties, did not change. The definition of the volume providing the link to the NFS share saw a change in the interfaces section, as attaching NFS shares is handled differently by the two orchestrators. The interface section of the container interfaces also saw a change. In the case of the Wordpress container, the update policy was rewritten in the style of a Kubernetes manifest. For the NFS-server container, the assignment of the IP address was rewritten for Kubernetes. The definition of the network is implicit in Kubernetes, so this was removed entirely.

Submitting this now changed ADT to a MiCADO implementation featuring Kubernetes in place of Swarm sees a successful deployment of the three aforementioned components in containers on the public CloudSigma cloud, as seen in the Kubernetes dashboard capture shown in the top half of Figure 5. The automated scaling functionality of MiCADO is also preserved, as can be seen in the Grafana [33] graph capture in the lower portion of Figure 5, where the WordPress frontend scales up in response to high network traffic generated by an HTTP load testing tool called wrk [34]. New nodes joining the cluster immediately receive traffic thanks to the default round-robin routing mesh of the container orchestrator, though a load balancer could also be implemented. The modularity of MiCADO allowed us to extend support to a specific use case which otherwise would have required architectural and design changes. The flexibility of TOSCA and our Application Description Templates gave us the power to do so without having to significantly change the user-facing interface which described the application and all of its dependencies and requirements.

VI. Conclusion & Future Work

This paper presented a novel approach to authoring TOSCA templates for the reuse of generic cloud components across different orchestration tools. A proof-of-concept demonstrated that, when requirements of an application exceeded the capabilities of a modular framework, the definition of that application could be reused and successfully deployed on the same framework featuring a different, capable component.

As the development of MiCADO continues, so does the work in testing and supporting its modularity with TOSCA and Application Description Templates. Supporting modular cloud orchestration using TOSCA ADTs is on the horizon, with the aim to provide high levels of portability of a containerized application across a wide variety of cloud service providers.

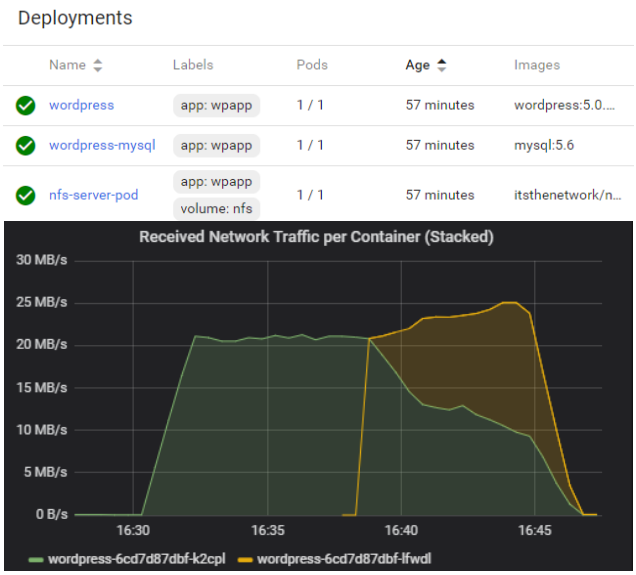

Figure 5. WordPress application successfully deployed to MiCADO and scaling automatically in response to an HTTP load test

ACKNOWLEDGMENT

This work was funded by the COLA Cloud Orchestration at the level of Applications Project No. 731574.

REFERENCES

[1] "COLA – Cloud Orchestration at the Level of Application." [Online]. Available: https://project-cola.eu/. [Accessed: 1-Mar-2019].

[2] T. Kiss, et al., "MiCADO - Microservices-based Cloud Application-level Dynamic Orchestrator", Future Generation Computer Systems, Vol 95, pp 937 – 946, May 2019. DOI: https://doi.org/10.1016/j.future.2017.09.050.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, "Borg, Omega, and Kubernetes". Queue 14, 1, Pages 10 (January 2016), 24 pages. DOI: https://doi.org/10.1145/2898442.2898444.

[4] J. Kovacs, P. Kacsuk, "Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures", Journal of Grid Computing, vol 16, issue 1, pp 19-37, 2018.

[5] "Prometheus," [Online]. Available: https://prometheus.io/. [Accessed: 1-Mar-2019].

[6] O. Ben-Kiki, C. Evans, I. döt Net,. "YAML ain't markup language version 1.2," 2009 [Online] Available: http://yaml.org/spec/1.2/spec.html. [Accessed: 1-Mar-2019].

[7] Oasis, "Oasis Topology and Orchestration Specification for Cloud Applications (TOSCA)." [Online]. Available: https://www.oasisopen.org/committees/tc_home.php?wg_abbrev=tosca. [Accessed: 1-Mar-2019].

[8] Oasis, "TOSCA Simple Profile in YAML Version 1.0." [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html. [Accessed: 1-Mar-2019].

[9] Docker, "Enterprise Application Container Platform." [Online]. Available: https://www.docker.com/. [Accessed: 1-Mar-2019].

[10] "cri-o Lightweight Container Runtime for Kubernetes." [Online]. Available: https://cri-o.io/. [Accessed: 1-Mar-2019].

[11] CoreOS, "rkt – A security-minded, standards-based container engine." [Online]. Available: https://coreos.com/rkt/. [Accessed: 1-Mar-2019].

[12] Apache Mesos, "Mesos Containerizer." [Online]. Available: http://mesos.apache.org/documentation/latest/mesos-containerizer/. [Accessed: 1-Mar-2019].

[13] Docker, "Swarm mode overview." [Online]. Available: https://docs.docker.com/engine/swarm/. [Accessed: 1-Mar-2019].

[14] Kubernetes, "Production-Grade Container Orchestration." [Online]. Available: https://kubernetes.io/. [Accessed: 1-Mar-2019].

[15] Apache Mesos, "Marathon: A container orchestration platform for Mesos and DC/OS." [Online]. Available: https://mesosphere.github.io/marathon/. [Accessed: 1-Mar-2019].

[16] P. Lipton, D. Palma, M. Rutkowski, and D.A Tamburri. "Tosca solves big problems in the cloud and beyond!." IEEE Cloud Computing (2018).

[17] B. Antonio, J. Soldani, and P. Wang. "TOSCA in a Nutshell: Promises and Perspectives." In European Conference on Service-Oriented and Cloud Computing, pp. 171-186. Springer, Berlin, Heidelberg, 2014.

[18] Oasis, "TOSCA Simple Profile in YAML Version 1.2." [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html. [Accessed: 5-Mar-2019].

[19] T. Binz, et al., "OpenTOSCA – a runtime for TOSCA-based cloud applications." In International Conference on Service-Oriented Computing, pp. 692-695. Springer, Berlin, Heidelberg, 2013.

[20] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "Winery–a modeling tool for TOSCA-based cloud applications." In International Conference on Service-Oriented Computing, pp. 700-704. Springer, Berlin, Heidelberg, 2013.

[21] Cloudify, "Cutting Edge Orchestration." [Online]. Available: https://cloudify.co/. [Accessed: 5-Mar-2019].

[22] Apache, "About ARIA TOSCA." [Online]. Available: http://ariatosca.incubator.apache.org/. [Accessed: 5-Mar-2019].

[23] "Puccini - Deliberately stateless cloud topology management and deployment tools based on TOSCA." [Online]. Available: https://github.com/tliron/puccini. [Accessed: 5-Mar-2019].

[24] "ALIEN 4 Cloud." [Online]. Available: http://alien4cloud.github.io/. [Accessed: 5-Mar-2019].

[25] A. Brogi, L. Rinaldi, J. Soldani. "TosKer: Orchestrating applications with TOSCA and Docker." In European Conference on Service-Oriented and Cloud Computing, pp. 130-144. Springer, Cham, 2017.

[26] G. Pierantoni, T. Kiss, G. Gesmier, J. DesLauriers, G. Terstyanszky, JMM Rapún, "Flexible Deployment of Social Media Analysis Tools", International Workshop on Science Gateways, 13-15 June 2018, Edinburgh, UK.

[27] Cloudsigma Holding AG. "Cloud servers & Hosting". [Online]. Available: https://www.cloudsigma.com/. [Accessed: 5-Mar-2019].

[28] CloudBroker GmbH., "Compute-intensive applications in the cloud." [Online]. Available: http://cloudbroker.com/. [Accessed: 5-Mar-2019].

[29] WordPress, "Features." [Online]. Available: https://wordpress.com/. [Accessed: 5-Mar-2019].

[30] "MySQL." [Online]. Available: https://www.mysql.com/. [Accessed: 5-Mar-2019].

[31] Microsoft, "Network File System overview." [Online]. Available: https://docs.microsoft.com/en-us/windows-server/storage/nfs/nfs-overview. [Accessed: 5-Mar-2019].

[32] Red Hat Ansible, "Ansible is Simple IT Automation." [Online]. Available: https://www.ansible.com/. [Accessed: 5-Mar-2019].

[33] Grafana Labs, "Grafana – The open platform for analytics and monitoring." [Online]. Available: https://grafana.com/. [Accessed: 5-Mar-2019].

[34] "wrk – Modern HTTP benchmarking tool." [Online]. Available: https://github.com/wg/wrk. [Accessed: 5-Mar-2019].

[35] A. Rossini, et al., "The cloud application modelling and execution language (CAMEL)," Open Access Repositorium der Universität Ulm, 2017. DOI:http://dx.doi.org/10.18725/OPARU-4339

[36] A. Rossini, "Cloud application modelling and execution language (CAMEL) and the PaaSage workflow." In Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC, vol. 567, pp. 437-439. 2015.

[37] G. Horn, and P. Skrzypek. "MELODIC: utility based cross cloud deployment optimisation." In 2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 360-367. IEEE, 2018.

[38] G. Henning, et al., "CACTOS toolkit version 2: accompanying document for prototype deliverable D5. 2.2." Open Access Repositorium der Universität Ulm, 2017. DOI:http://dx.doi.org/10.18725/OPARU-4319