Project Acronym: **COLA**

Project Number: **731574**

Programme: **Information and Communication Technologies**
Advanced Computing and Cloud Computing

Topic: **ICT-06-2016 Cloud Computing**

Call Identifier: **H2020-ICT-2016-1**
Funding Scheme: **Innovation Action**

Start date of project: 01/01/2017 Duration: 30 months

Deliverable:

D6.3 Prototype and documentation of the scalability decision service

Due date of deliverable: 30/06/2018 Actual submission date: 04/07/2018

WPL: Jozsef Kovacs

Dissemination Level: PU

Version: V0.19

1. Table of Contents

2. List of Figures and Tables

Figures

Tables

3. Status, Change History and Glossary

Status:	Name:	Date:	Signature:
Draft:	Jozsef Kovacs	02/07/18	Jozsef Kovacs
Reviewed:	Antonis Michalas	03/07/18	Antonis Michalas
Approved:	Tamas Kiss	04/07/18	Tamas Kiss

Table 1 Status Change History

Version	Date	Pages	Author(s)	Modification
V0.1	05/06	ALL	Jozsef Kovacs	Empty Skeleton
V0.2	19/06	Section 9	Osama Abu Oun	Implementation of MiCADO V4
V0.3	22/06	Section 10	Jozsef Kovacs	Implementation of MiCADO V5
V0.4	25/06	Section 10.3 and 10.3.4.1	Gregoire Gesmier, James Deslauriers	TOSCA Submitter and Docker adaptor
V0.5	25/06	Section 10.3.4.2	Mark Emodi	Occopus adaptor
V0.6	25/06	Section 10.3.4.3	Attila Farkas	Policy Keeper adaptor
V0.7	26/06	Section 11.1	James Deslauriers	Consumption based policy
V0.8	26/06	Section 10.1	Attila Farkas, Mark Emodi	Ansible Playbook
V0.9	26/06	Section 10.4	Attila Marosi, Jozsef Kovacs	Dashboard
V0.10	27/06	Section 8	Attila Farkas, Jozsef Kovacs	Implementation of MiCADO V3.1
V0.11	27/06	Section 7	Jozsef Kovacs	Designing the MiCADO Scalability Decision Maker Microsercvice
V0.12	28/06	Section 11.2	Jozsef Kovacs	Deadline based policy
V0.13	29/06	Sections 8.3, 9.7, 10.5	Osama Abu Oun, Attila Farkas	Availability for MiCADO V3.1, V4, V5

V0.14	29/06	Section 6	Jozsef Kovacs	Designing the MiCADO Orchestration Layer
V0.15	29/06	Section 12	Jozsef Kovacs	Current Status and Conclusion
V0.16	29/06	Section 4 and 5	Jozsef Kovacs	Introduction, MiCADO Generic Architecture Framework
V0.17	01/07	ALL	Tamas Kiss	Improvements, fixes
V0.18	02/07	ALL	Jozsef Kovacs	Improvements, fixes based on Tamas Kiss suggestions
V0.19	04/07	ALL	Jozsef Kovacs	Improvements based on the review by Antonis Michalas

Table 2 Deliverable Change History

Glossary

API	Application Programming Interface
MiCADO	Microservices-based Cloud Application-level Dynamic Orchestrator
COLA	Cloud Orchestration at the level of Application
REST	Representational State Transfer (service interface)
CLI	Command Line Interface
TOSCA	Topology Orchestration Specification for Cloud Application
VM	Virtual Machine
IaaS	Infrastructure-as-a-Service
JSON	JavaScript Object Notation

Table 3 Glossary

4. Introduction

This deliverable, describes the design and implementation of the MiCADO (Microservices based Cloud Application-level Dynamic Orchestrator) scaling decision maker (SDM) subsystem. SDM is focusing on scaling virtual machines and container services. MiCADO is a compound service providing automatic scaling and orchestration of microservices, as well as of cloud resources (virtual machines) required for executing the services. The aim of MiCADO is to implement this double orchestration in an intelligent way following the scaling policies specified by the user together with the infrastructure description. This scaling functionality, realized by the scalability decision service (called Policy Keeper) is described in the following sections.

This deliverable, reports the work performed by WP6 as a continuation of work described in deliverables D6.1 and D6.2. The work described in this deliverable also utilizes the results reported in deliverables produced by WP5 detailing the design of Application Description Template with TOSCA representing the description for applications and policies to be handled by MiCADO. The deliverables produced by WP8 specifying the application requirements are also considered as important input in this deliverable. Moreover, the work in this deliverable represents an important input for the deliverables produced by WP7.

The rest of this deliverable is organized as follows: In Section 5 we overview the generic architecture of the MiCADO framework, while in Section 6 the design plan of the MiCADO scalability decision service is introduced. The next three sections detail the versions of MiCADO implemented during the reporting period. Improvements of MiCADO v3.1 are shortly introduced in Section 7, while MiCADO v4 integrating a job execution framework is detailed in Section 8. MiCADO v5 is introduced in Section 9 as the first version including the newly designed Scalability Decision Maker component. Scaling examples are introduced in Section 10 to demonstrate the operation of the Scalability Decision Maker component. Finally, current status and conclusion closes the deliverable.

5. The MiCADO generic architecture framework

The layers of MiCADO supporting the dynamic application level orchestration of cloud applications are illustrated in Figure 1. This generic framework is based on the concept of microservices, as defined for example by Balalaie [1]. Cloud computing is a natural platform for microservices that provide decoupling of independent components from a monolithic application. Cloud enables execution and resource allocation of these independent components based on their specific needs. One microservice might require significant storage resources while another could be CPU intensive. Cloud execution offers the possibility to optimize resource allocation and resource cost dynamically. The alternative would be to allocate a monolithic infrastructure, the size of which is large enough to be sufficient to cover peak performance as well. The requirement for peak performance happens rarely, therefore allocated resources of the monolithic infrastructures remain unused in most of the time.

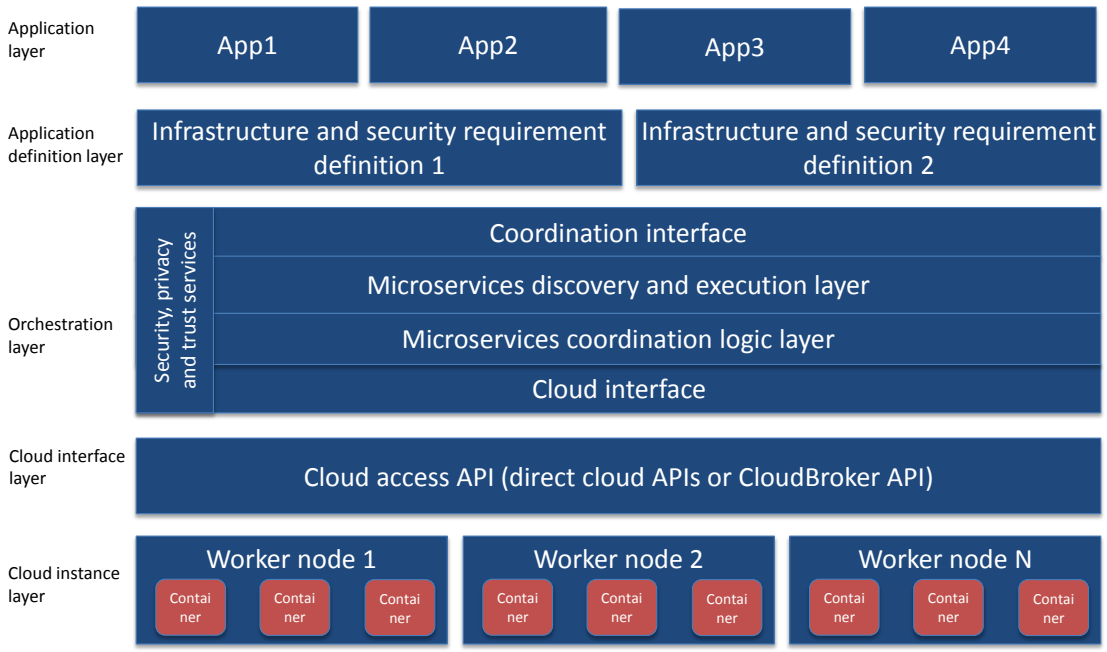

Figure 1 MiCADO generic architecture framework

The layers of the MiCADO generic architecture (from top to bottom), based on the above described microservices-based concept are as follows:

1. **Application layer**. Application layer contains actual application code and data described by application definition (layer 2) to function in such a way that a desired functionality is reached. For example, this layer could populate database with initial data, and configure HTTP server with look and feel and application logic.
2. **Application definition layer**. This layer allows definition of the functional architecture of applications using application templates. At this level, software components and their requirements (both infrastructure and security specifications) as well as their interconnectivity are defined using application descriptions uploaded to a public repository. As the infrastructure is agnostic to the actually executed

application, the application template can be shared with any application that requires such an environment.

3. **Orchestration layer.** This layer is divided into four horizontal and one vertical sub-layers. The horizontal sub-layers are:
   a. **Coordination interface API.** This sub-layer provides access to the orchestration control and decouples the orchestration layer from the application definition layer. This set of APIs enables application developers to utilize the dynamic orchestration capabilities of the underlying layer and supports the convenient development of dynamically and automatically scalable cloud-based applications by embedding these API calls into application code.
   b. **Microservices discovery and execution.** This sub-layer manages the execution of microservices and keeps track of services running. Execution management combines both start-up and shut down of microservices. Service management gathers information about currently running services, such as service name, IP address and port where the service is reachable and optional service tags to help service coordination.
   c. **Microservices coordination logic.** To reap the benefits from cloud-based execution, it becomes necessary to understand how the current execution environment is performing. Information needs to be gathered and processed. If bottlenecks are detected or the currently running infrastructure appears underutilized, it may be necessary to either launch or shut down cloud instances, and possibly move microservices from one physical worker node to another.
   d. **Cloud interface API.** It is responsible for abstracting cloud access from layers above. Cloud access APIs can be complex interfaces, as they typically cater for a large number of services provided by the cloud provider. On the other hand, the microservices execution and coordination logic layers (see b and c) only need to shut down and start instances. Abstracting this to a cloud interface API simplifies the implementation of the aforementioned layers, and if new Cloud access APIs are implemented, only this layer needs to change.
   e. **Security, privacy and trust services.** The orchestration layer also includes a vertical sub-layer that offers services related to security, privacy and trust. These services span among multiple levels of the orchestration layer, as it is illustrated in Figure 1. The main aim is to save the application developers from detailed security management. To achieve this, the security, privacy and trust services of the orchestration layer take the general security policies defined at the Application definition layer, as well as security credentials for the application domain. These inputs are used by the special purpose security policy enforcement services to enforce the security policies at orchestration level.
4. **Cloud interface layer.** This layer provides functions to launch and shut down cloud instances. There can be one or more cloud interfaces to support multiple clouds. Besides directly accessing cloud APIs, generic cloud access services, such as the CloudBroker platform [2] can be also used at this layer to support accessing multiple, heterogeneous and distributed clouds via its uniform access layer.
5. **Cloud instance layer.** This layer contains cloud instances provided by Infrastructure-as-a-Service (IaaS) cloud providers. These instances can run various containers that execute actual microservices. The layer typically represents state-of-the-art of cloud technology provided by various public or private cloud providers.

6. Designing the MiCADO Orchestration Layer

6.1 General design principles

In this section, we are giving an overview of the MiCADO Orchestration Layer outlining its architecture and basic functionalities in order to let the reader understand how the work described in this deliverable fits into the architecture of MiCADO.

It is important to mention that at this level of abstraction; each component is named after its functionality. In this section we introduce the overall high-level design where no concrete tool is assigned for implementing a particular functionality, to make this layer independent from technologies. This architecture has been designed taking COLA deliverable D8.1 - "Business and technical requirements of COLA use cases" as input, which specifies the requirements of the COLA use cases.

The MiCADO Orchestration Layer is responsible for deploying, executing, scaling and managing microservices or network of microservices, and for maintaining the allocation of resources required for the microservices. The overall architecture of the MiCADO Orchestration Layer (MiCADO for short in the rest of this section) can be seen in Figure 2.

MiCADO essentially forms a cluster which is able to dynamically allocate, attach, or detach and release cloud resources for optimizing the resource usage during executing the submitted microservices. MiCADO consists of two main logical components: **Master node** and **Worker nodes**. **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 of resources and scheduling of microservices. **Worker nodes** are volatile components, representing execution environments for the microservices, i.e. they are executing the actual microservices. **Worker 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 to it.

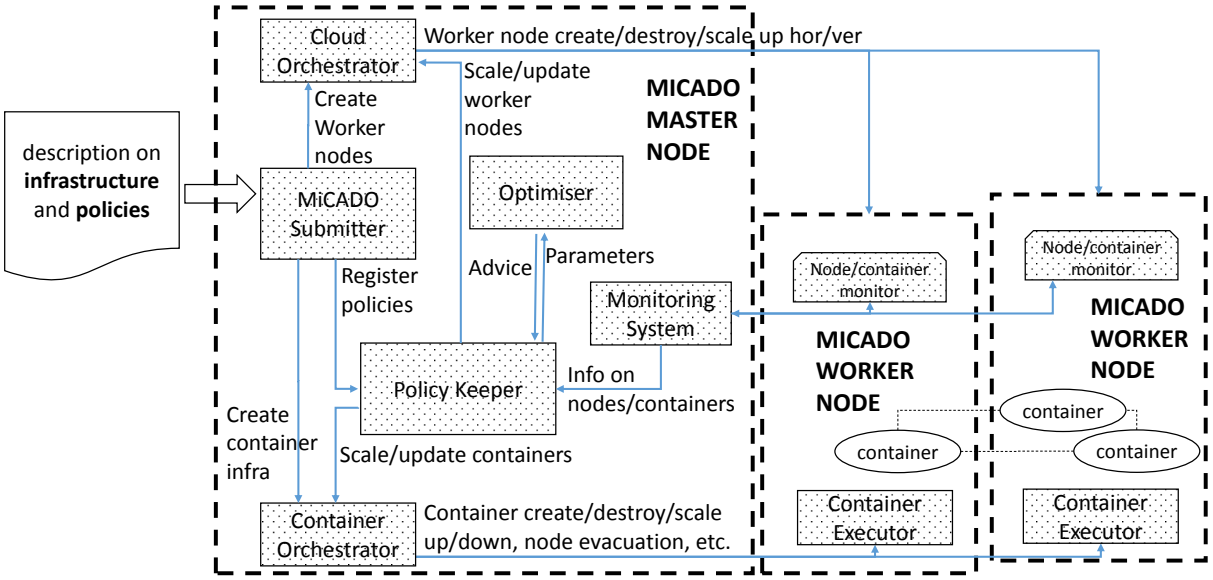

Figure 2 Architecture of the MiCADO Orchestration Layer

MiCADO Master Node (box with dashed line on the left side of Figure 2) contains the following key components:

- **MiCADO Submitter** is the primary service request endpoint for creating an infrastructure to run an application, managing this infrastructure and the application itself. Submitted infrastructures are received by this component. The incoming description (e.g. in TOSCA format) is interpreted and the different sections of the description are forwarded to the underlying components.
- **Cloud orchestrator** is responsible for communicating to the Cloud API in relation to allocating and releasing resources, and for building up/shutting down new MiCADO worker nodes whenever required.
- **Container orchestrator** is responsible for allocating new microservices (realized by containers) on the worker nodes, to keep track of their execution, and to destroy them if necessary. This component must also realize the scale up and down functionality on container services upon request.
- **Monitoring system** is responsible for collecting information on load of the resources and on resource usage of the container services, and to provide this information for the other components on the MiCADO master node. Alternatively, it may provide alerting functionality in relation to the measured attributes to detect values that require reaction.
- **Policy keeper** is the key component that implements policies and makes decisions related to allocating/releasing cloud resources and scheduling container services among worker nodes. Moreover, this component assures that the cloud and container orchestrators are instructed in a synchronized way during the operation of the entire system.
- **Optimizer** is a background (micro)service performing long-running calculations on demand for finding optimized setup of both resources and container infrastructure. An optimization calculation can be initiated with the required parameters on resources and containers. Following this, the result of optimization is forwarded to the Policy Keeper component for consideration and execution.

MiCADO Worker Nodes (boxes with dashed line on the right side of Figure 2) contain the following components:

- **Node/container monitor** component 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.
- **Container executor** is responsible for starting, executing and destroying containers upon requests from the Container Orchestrator on the Master node.
- **Container** components are realizing the user services defined in the (container) infrastructure description submitted through the MiCADO submitter on the Master node.

The basic operation of the architecture above can be summarized in the following way: a new application and infrastructure description is submitted through the MiCADO submitter. Based on this description, the initial number of MiCADO worker nodes is created by the Cloud Orchestrator. Once the MiCADO worker nodes are up and running, the Container infrastructure is submitted to the Container orchestrator component which realizes the container services on the worker nodes. Once the initial deployment has been done, policies related to the application are registered in the Policy Keeper component. The Monitoring

system starts collecting information on the nodes and containers, and the Policy Keeper starts updating the deployment (including both the worker nodes and the containers) when necessary. The Optimizer performs calculation in the background and provides advice for the Policy Keeper after a certain time interval.

In this architecture, the Cloud Orchestrator and Container Orchestrator components together with the Submitter realize the initial deployment of the resources and containers. In case there are any policies defined in relation to controlling the resource consumption of the container infrastructure, the Policy Keeper, Optimizer and Monitoring system components together form a controlling loop implementing the predefined policy. Once the initial deployment has been done, updates can be only confirmed by the Policy Keeper component.

This architecture is built by loosely coupled functionalities like resource allocation/release, container allocation/deallocation, initial deployment, monitoring and decisions on scalability. For example, the controlling components (Policy Keeper, Optimizer, Monitoring) can be detached from the architecture and it is still operational for realizing the initial deployment of the submitted infrastructure.

One of the most important aim of this architecture is to provide a modular and pluggable framework where different functionalities can be delivered by different components on-demand, and where these components can be easily substituted. The resulting solution will be agnostic to the underlying component implementation.

6.2 Specific design principles related to COLA use cases

When designing the MiCADO architecture, specific requirements of the COLA project use cases have also been considered. There are five categories of requirements defined in D8.1 by the COLA use cases: system requirements, data requirements, performance requirements, security requirements, and other requirements.

- *System requirements* relate to the underlying operating system, which is Ubuntu in most of the use cases, except for the Saker Solutions use case, where Windows is the base operating system.
- *Data requirements* for the use cases are relatively low. However, using an external database is a good alternative for data intensive applications, if it is required.
- *Performance requirements* are planned to be fulfilled by applying the policies and utilizing the auto-scaling mechanism. Container applications will be automatically scaled together with worker nodes to deliver additional computing resources.
- *Security requirements* will be mainly addressed by WP7. However, a set of default security mechanisms, such as VPN, encrypted channels and certain firewall settings, are going to be supported. Moreover, both private and public clouds will be supported.
- *Other requirements* in D8.1 include *data protection*, *robustness*, and *quality of service.* Even though these extra requirements are not considered as part of the main MiCADO architecture they will be investigated together with the application owners.

When implementing the architecture shown in Figure 2, the tools realizing the different components must be integrated together keeping in mind the possibility to replace them with alternate solution if it is required later.

6.3 Overview of MiCADO implementation

In the first phase of implementation reported in Deliverable D6.1, Cloud orchestration and Container orchestration (depicted by red boxes in Figure 3) have been realized. As deliverable D6.1 details, Occopus [3][4] implements the cloud orchestration, while Docker Swarm [5] implements the Container orchestration subsystem.

In the second phase of implementation, a Monitoring system (depicted by green boxes in Figure 3) has been integrated and documented in Deliverable D6.2. As deliverable details, the Prometheus monitoring subsystem [6] with Node exporter [7] and CAdvisor [8] components (as data sources) on the Worker nodes, has been added to the MiCADO implementation.

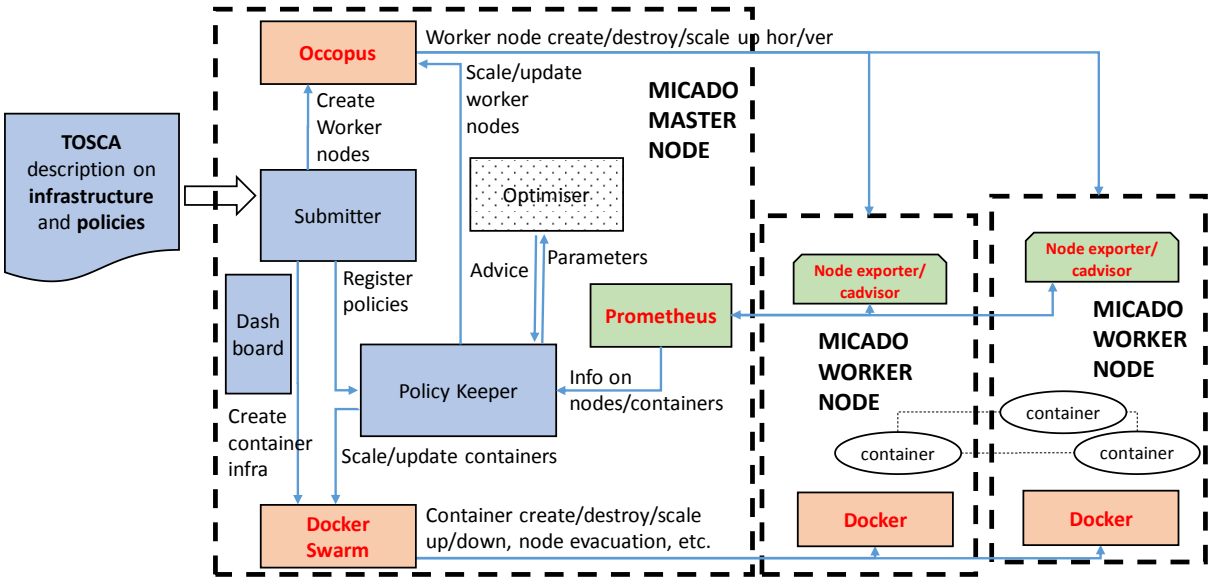

Figure 3 MiCADO implementation after stage 2

In the current phase of implementation, the Policy Keeper (to support automatic decision making), Dashboard and Submitter with TOSCA support (depicted by blue boxes in Figure 3) is going to be introduced. The Optimiser component (depicted by uncolored, dotted box in Figure 3) will be the target of the next phase in the implementation work.

7. Designing the MiCADO Scalability Decision Maker microservice

The MiCADO Scalability Decision Maker is intended to perform a decision on scaling by calculating the optimal number of instances needed to be created. The scaling decision must be made on two levels:

1. **Virtual machine level:** Scaling up or down the number of virtual machines (on which Docker Swarm cluster is realized and the container application is running) realizes adding or removing resources from the cluster. Whenever virtual machine level upscaling happens, a new Docker node is attached and the Docker cluster grows. Downscaling at virtual machine level means removing Docker nodes from the Docker Swarm cluster i.e. Docker cluster loses resource.
2. **Container level:** In Docker Swarm a (micro)service is realized by containers running on the nodes of Docker Swarm in a distributed way. In order to add more resources to a particular microservice, the number of instances of the containers (realizing the Docker service) must be increased. Docker Swarm makes sure the containers are executed in parallel on the nodes of the cluster and the user requests arriving to the service is distributed among the containers to be handled. Scaling up and down the number of containers of a given service increases the parallelism of the request handling at containers level i.e. increases the resources associated to the given service.

In the next sections, the main concept, architecture, operation and low-level policy format will be detailed to introduce how the Scalability Decision Maker microservice has been realized inside MiCADO.

7.1 Concept

In MiCADO there are control loops realized on virtual machine and container levels. Control loops are depicted in Figure 4.

The virtual machines (i.e. nodes) are represented by boxes entitled Node1 and Node2. First, information is collected on the nodes by the Prometheus monitoring system. As a decision maker service, the Policy Keeper component holds the list of monitored parameters (extracted from Prometheus) and the scaling rules describing the decision on scaling. Once a decision is made, Occopus performs the scaling of the nodes, i.e. launches or destroys virtual machines and attaches them as MiCADO workers to the Docker Swarm cluster. This mechanism realizes the virtual machine level control loop.

The containers labelled by Cont A, B, C, D are forming a container infrastructure and realizing services for the users. Various types of parameters are monitored and collected by Prometheus which can be used for decision making. Policy Keeper holds the scaling rules for each service and performs the decision in function of the value of the incoming parameters. The decision of container scaling is finally realized by Docker Swarm.

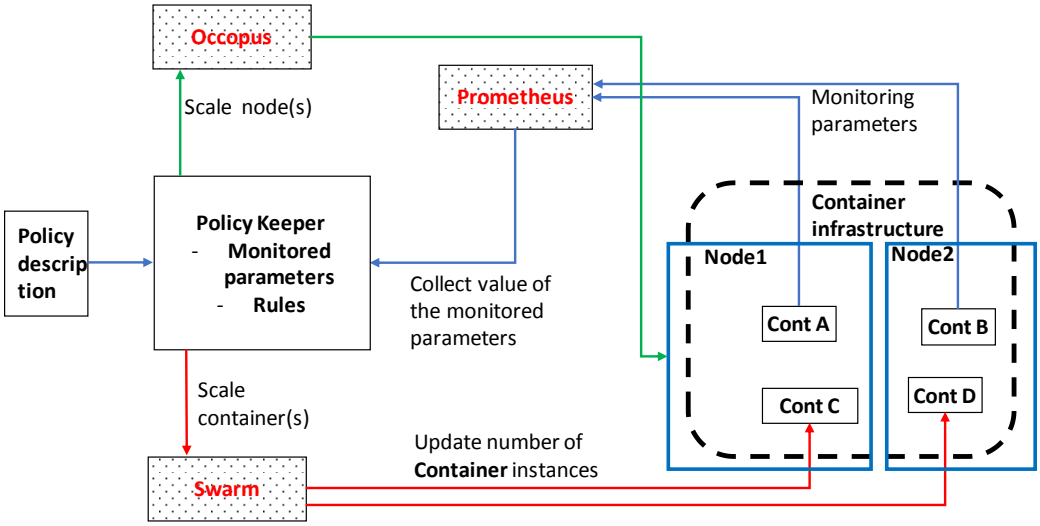

Figure 4 Control loops in MiCADO

In the latest MiCADO implementation, the monitoring system is already realized by Prometheus, while the orchestration on both levels are implemented by Occopus (on cloud level) and Swarm (on container level). To complete the control loop either at the level of nodes or at the level of containers, design decision must be made on the following:

- what are the parameters to be monitored on the observed object (node or service);
- what is the scaling rule which provides a decision based on the actual value of the monitoring parameters.

Major design principles

To avoid limiting the monitoring parameters in MiCADO, the first design principle is to let the monitoring parameters be defined dynamically for each submitted application. This must be true for parameters not supported by the current monitoring setup. *The key principle is to make the monitoring system dynamically extendable in terms of data sources and monitoring parameters*.

To a make the scaling rule (which defines the behavior of MiCADO in terms of scaling) as flexible as possible, it must be an input parameter (together with the application description) instead of making fixed and selected from a predefined list of scaling rules. *The key design principle is to make the scaling rule specifiable by the user in a flexible way.*

Monitoring parameters

Most typical scaling rules require load-related parameters of the nodes at virtual machine level scaling. Monitoring the cpu-, memory- and network load of the nodes provides most of the parameters for a typical scaling rule. However, there are situations where the application (realized by a container infrastructure) requires scaling based on parameters that are not on the list of predefined monitored parameters. For example, the application may require significant disk capacity on the node to cache some data, or may require other types of resources inside the virtual machine that are not among the list of predefined monitored parameters to ensure proper scaling.

The main goal of MiCADO is to provide maximum flexibility in terms of monitored parameters. To do so, MiCADO aims to support dynamically configured list of monitored parameters instead of a selection from already configured parameters. To do that, Policy Keeper allows the user to dynamically specify new parameters that will be monitored, even if the current monitoring setup is not able to gather the value of these new parameters. This dynamic extension of the monitoring system is supported by Prometheus through its query language, query API and dynamically configurable exporters realizing the data extraction.

Scaling rules on the level of application may require the implementation of more complex scenarios. These scenarios may rely on monitoring parameters which are not predefined and provided by the default built-in monitoring system. Moreover, if an application scaling rule requires some information which exists inside the application's internal state, special data collection component is required to be attached to the monitoring system as data source.

Scaling rules

The scaling rule is intended to calculate the required number of replicas of containers for a certain service or the required number of instances of virtual machines. The scaling rule should express the direction (up/down) and quantity (instance number) of scaling. A scaling rule may be reutilized by different applications provided that the application characteristics are similar and the business policy needed by the operator/user of MiCADO is similar. A complex scaling rule has the task of coordinating the resource capacity available for the application (virtual machine level scaling) and the resource usage by the application (container level scaling). For both, the aim of the Policy Keeper is to provide maximum flexibility, configurability. The complexity of the scaling rules, and the variety of user requirements may easily result in insufficient support from scaling rules in case Policy Keeper tries to provide a predefined set of scaling rules.

Using predefined scaling rules may perfectly support some groups of applications. However, the variety of requirements will always result in more complex rules to be implemented. To support scaling rules and policies for diversity of applications and requirements, Policy Keeper supports scaling rules to be defined as user inputs. Handling the scaling rules as inputs provides maximum flexibility for the user and removes limitations of MiCADO in relation to supported types of applications and scaling logic.

The scaling rule for the Policy Keeper must be an expression that can be automatically evaluated with the monitoring parameters as input, and the output of the evaluation is the decision on scaling i.e. the number of instances. To give the user as much freedom as possible, the scaling rule should be able to formalize arithmetic, logic and control expressions.

In Policy Keeper, the scaling policies contains the list of monitoring parameters together with their definition and the scaling rule. The policy is described in YAML and the language for expressing the scaling rule is selected to be Python, since the Policy Keeper itself is designed to be implemented by a microservice in Python language. The simplicity of the language and the easy evaluation resulted in introducing the support of Python language in the scaling rule definition.

7.2 Operation

The overall architecture of MiCADO has been initially designed in COLA deliverable D6.1. The main components are Prometheus for monitoring, Docker Swarm for container execution, Occopus for virtual machine orchestration, Submitter to handle TOSCA-based descriptions and finally the Policy Keeper to perform decision on scaling. This section, focuses on the implementation of Policy Keeper and the surrounding components connected to it. A detailed architecture of the Policy Keeper and its environment can be seen in Figure 5.

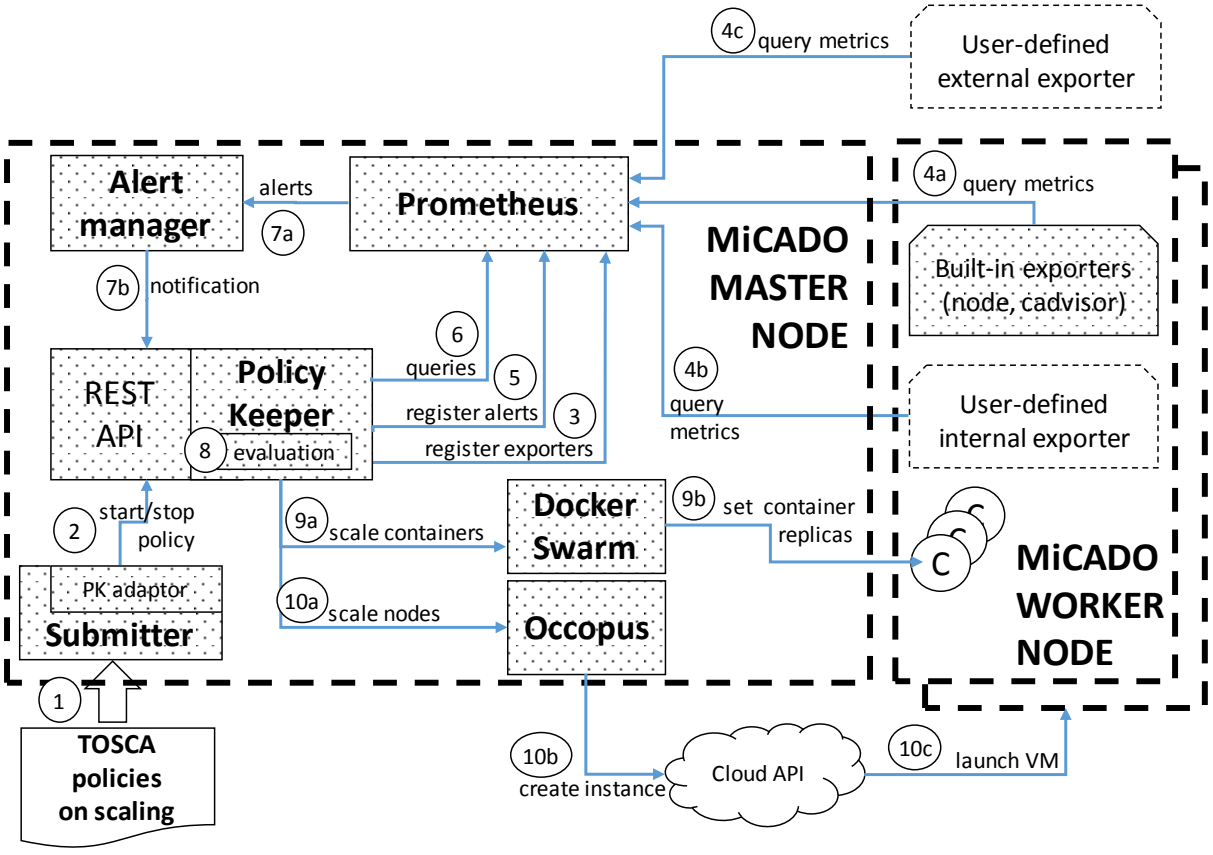

Figure 5 Policy Keeper and its environment in MiCADO

MiCADO integrates Prometheus as a monitoring tool on the MiCADO master node and has two exporters running on each worker nodes to collect information on the node and on the containers running on a given node. With the support of these two built-in exporters (node exporter and cadvisor) a long list of parameters (metrics) can be monitored and queried from Prometheus.

To monitor a parameter that is not supported by the node exporter or the cAdvisor, Policy Keeper provides a mechanism to dynamically attach i.e. register new user-defined exporters. By defining the location of a new exporter, Policy Keeper can configure Prometheus to collect metrics from a user-defined exporter. The new exporter can be either executed by MiCADO (internal) or can be executed outside and independently from MiCADO (external). Deployment of the exporter is not performed by the Policy Keeper. Once the exporter is attached, the metrics become available in Prometheus. Policy Keeper uses the query

interface of Prometheus to collect the value of the parameters originating from the exporters and are specified in the scaling policy.

Policy Keeper also supports alerting with the help of Prometheus. When the scaling policy contains definition of alerts, they are registered in Prometheus to be maintained. Alert manager is a component part of the Prometheus software package and handles alerts sent by Prometheus. Alert manager organizes the alerts and notifies Policy Keeper through its REST interface when an alert fires. Upon scaling decision Docker Swarm and Occopus realizes creation or removal of instances if necessary.

The overall flow of operation focusing on realizing scaling is as follows (see Figure 5):

- The submitter receives a TOSCA-based description of the scaling policy as part of the overall TOSCA description on the container infrastructure (Step 1).
- The submitter uses its Policy Keeper adaptor to convert the TOSCA based scaling policy format of MiCADO to the native policy format (see Section 7.3) of the Policy Keeper. After the conversion, the policy is sent to the Policy Keeper through its REST interface for realization (Step 2).
  To keep the implementation of MiCADO components interchangeable (one of the design principles in MiCADO), the handling of TOSCA representation (for the infrastructure and policies) is kept in one single component, i.e. in the submitter. In case TOSCA or the entire application specification language changes, only the Submitter is affected. Following the design, the MiCADO components interfacing with the submitter receive information in their own specified format after conversion.
- As a first step, the Policy Keeper registers the exporters specified in the policy (Step 3).
- Prometheus immediately starts pulling the metrics from the exporters regardless they are built-in (Step 4a), user-defined internal (Step 4b) or user-defined external (Step 4c).
- In case the policy contains definition of alerts, Policy Keeper registers them in Prometheus as well (Step 5).
- At this point, Prometheus is ready to deliver value of metrics from its exporters. Policy Keeper periodically issues queries towards Prometheus to evaluate the expressions for the variables (referred by the scaling rule) (Step 6).
- Whenever an alert is firing, Prometheus through Alert manager (Step 7a) notifies Policy Keeper (Step 7b) which registers the event.
- Policy Keeper periodically evaluates the scaling rules (Step 8) which may refer to variables and alerts.
- As a result of the evaluation of the scaling rules, Policy Keeper may instruct Docker Swarm (Step 9a) to scale up/down a given container (Step 9b).
- The evaluation of the scaling rule may also result in scaling at virtual machine level. In this case, Policy Keeper instructs Occopus (Step 10a) for scaling which in turn asks the target cloud API to create instances (Step 10b). Finally, a new VM is launched (Step 10c) on which a new MiCADO worker is built up and attached to MiCADO master.

This step-by-step operation of Policy Keeper and its environment ensures the realization of two control loops on virtual machine and container levels.

7.3 Policy format

The Policy Keeper component takes a policy description as input in order to implement handling Prometheus exporters, Prometheus expressions, Prometheus alerts and scaling rules. The policy description is structured to address sections for each of these topics. Policy description uses yaml syntax and has the following structure:

```
stack: <name of docker stack>
data:
  sources:
    - '<ip>:<port>'
  constants:
    <name of constant>: '<value of the constant>'
  queries:
    <name of parameter>: '<prometheus query expression>'
  alerts:
    - alert: <name of alert>
      expr: '<prometheus logical expression>'
      for: <time period: 1s, 1m, 1h, etc.>
scaling:
  nodes:
    min: <minimum number of nodes>
    max: <maximum number of nodes>
    target: |
      <python code to realize scaling rule>
  services:
    - name: "<name of docker service to scale"
      min: <minimum number of containers>
      max: <maximum number of containers>
      target: |
      <python code to realize scaling rule>
```

The variable called `'stack'` is required to identify the docker stack to be manipulated through docker swarm. Under the section named `'data'`, all Prometheus query and alert related settings can be specified. The section called `'scaling'` contains the scaling related specification, both for `'nodes'` i.e. to scale at virtual machine level and for Docker `'services'` i.e. to scale at container level. A more detailed description of the policy, will be provided in the next sections.

7.3.1 Data sources

Dynamic attachment of an external exporter can be performed under the `'source'` subsection by adding a list item with the ip address and port number of the exporter. The following YAML structure shows an example:

```
data:
  sources:
    - '192.168.154.116:8090'
    - 'rabbitmq_exporter:8090'
    - 'myexporter.mydomain.com:6000'
```

Each item found under the 'data'/'sources' subsection is configured under Prometheus to start collecting the information provided/exported by the exporters. Once done, the values of the parameters provided by the exporters become available.

7.3.2 Metrics

To utilize one of the exporters i.e. to query a metric collected by the newly configured exporter, a Prometheus query expression must be defined. Prometheus queries must be listed under the 'queries' subsection under the 'data' section of scalability policy. An example is shown below:

```
data:
  queries:
    REMAININGTIME: '{{DEADLINE}}-time()'
    ITEMS: 'rabbitmq_queue_messages_persistent
                              {queue="machinery_tasks"}'
```

In this example, two variables called 'REMAININGTIME' and 'ITEMS' have been defined with their corresponding Prometheus query expression. Each time the Policy Keeper evaluates the queries by Prometheus, it returns a value which is then associated to the variable name and can be referred in the scaling rule.

7.3.3 Constants

As can be seen by the previous example, for 'REMAININGTIME' variable, a predefined constant has been referred. Each referred constant, specified under the 'constants', subsection is replaced by its associated value. The following YAML structure shows an example:

```
data:
  constants:
    DEADLINE: 1529499571
```

To refer to a constant, Jinja2 [9] type syntax (i.e. using double brackets around the name of the constant) must be used. Here is an example to refer to the value of a constant:

```
{{DEADLINE}}
```

7.3.4 Alerting

Prometheus supports alerting mechanism. Alerts can be considered as notifications over an event which is important in relation to scaling. For example, the event that a service becomes overloaded can be considered important provided that scaling up service can reduce load on the actual containers.

To utilize Prometheus alerting system, alerts can be defined in the MiCADO scaling policy description under the 'alerts' subsection of 'data' in scalability policy with a dictionary

of three pieces of key-value (‘alert’, ’expr’, ’for’) organized as an item in the list of alerts. The following YAML structure shows an example (together with constants to make it clear) where an alert is configured to fire whenever the average cpu usage for all the containers belonging to the given service is above a certain threshold for at least 30 seconds.

```
data:
  constants:
    SERVICE_NAME: 'stressng'
    SERVICE_FULL_NAME: '{{stack}}_stressng'
    SERVICE_TH_MAX: '60'
    SERVICE_TH_MIN: '20'
  alerts:
  - alert: service_overloaded
    expr: 'avg(rate(container_cpu_usage_seconds_total
            {container_label_com_docker_swarm_service_
            name="{{SERVICE_FULL_NAME}}"}[30s]))*100 >
            {{SERVICE_TH_MAX}}'
    for: 30s
```

A named alert (‘alert’) is a logical expression (‘expr’) which is evaluated by Prometheus and the alert is fired when the expression is continuously evaluated to true for a predefined period of time (‘for’).

The alert firing is detected by the Policy Keeper and for fired alerts a variable with the name of the alert is generated and set to true. Similarly to the queries expression, the alert definition may also refer to constants using {{ }} brackets. To check if an alert is firing, the scaling rule simply refers to the name of the alert as a Boolean variable. The following YAML code shows an example:

```
scaling:
  ...
  services:
    - name: ‘stressng’
      ...
      target: |
        if service_overloaded:
          m_container_count+=1
```

A scaling rule (which is detailed in the next section) may contain the reference to the alert name to apply it in the decision process.

7.3.5 Scaling rules

Scaling rule in the policy description expresses the decision on scaling i.e. it is realized by a code snippet. Scaling rule must be defined for nodes (i.e. to scale at virtual machine level) and for services (i.e. to scale at container level). The following YAML code shows the structure of the scaling section inside the policy description:

```
scaling:
  nodes:
    min: <minimum number of nodes>
    max: <maximum number of nodes>
    target: |
      <python code to realize scaling rule>
  services:
    - name: "<name of docker service to scale"
      min: <minimum number of containers>
      max: <maximum number of containers>
      target: |
     <python code to realize scaling rule>
```

Policy Keeper supports the specification of the scaling rule by a Python expression under the `'target'` keyword. The Python expression must be formalized with the following conditions:

- Each constant defined under the 'constants' section can be referred; its value is the one defined by the user
- Each variable defined under the 'queries' section can be referred; its value is the result returned by Prometheus in response to the query string
- Each alert name defined under the 'alerts' section can be referred, its value is a logical True in case the alert is firing, False otherwise
- Expression must follow the syntax of the python language
- Expression can be multiline
- The following predefined variables can be referred; their values are defined and updated by Policy Keeper:
    - m_nodes: python list of nodes belonging to the docker swarm cluster
    - m_node_count: the target number of nodes
    - m_container_count: the target number of containers for the service the evaluation belongs to
    - m_time_since_node_count_changed: time in seconds elapsed since the number of nodes changed
- In node level scaling rule, the name of the variable to be set is 'm_node_count'; as an effect the number stored in this variable will be set as target instance number for the virtual machines.
- In container level scaling rule, the name of the variable to be set is 'm_container_count'; as an effect the number stored in this variable will be set as target instance number for the given container service.

Specifying a scaling rule with Python for the Policy Keeper to scale up and down based on the events 'service_overloaded' and 'service_underloaded' can be done simply as shown in the next YAML code:

```
scaling:
  services:
  - name: myservice
    min: 1
```

```
max: 5
target: |
  if service_overloaded:
     m_container_count+=1
  if service_underloaded:
     m_container_count-=1
```

The scaling rule (specified under the `'target'` keyword) is evaluated periodically. Before each evaluation, the values of the variables and alerts are updated based on Prometheus queries. The Python expression is expected to update the necessary variables ('m_container_count' in this case) to express the need for scaling.

When the expression is evaluated and the target number of containers or nodes are available, the returned value is updated if necessary to be higher or equal to the `'min'` value and to be lower or equal to the `'max'` value. As a consequence, Policy Keeper always keeps the target number between the minimum and maximum regardless of the value returned by the scaling expression.

8. Implementation of MiCADO V3.1

During the reporting period, MiCADO V3 has been further extended with some features and bugfixes and has been released as MiCADO V3.1. The architecture of V3.1 is the same as of MiCADO V3 defined in deliverable D6.2.

8.1 Features, bugfixes, limitations

The most important improvement in this version is to introduce configurability in regards to containers and their thresholds, In the previous version, MiCADO automatically assigned a scaling policy to each of the submitted containers. This is a serious limitation for those infrastructures where certain containers are not scalable by design (e.g. databases). To overcome this limitation, v3.1 introduced the possibility to define the list of containers to be scaled.

Another limitation in V3.0 was that whenever a container was submitted the thresholds at which the scaling must happen to increase or to decrease the replicas were fixed, predefined/hardwired in the MiCADO deployment files. To make it configurable, MiCADO V3.1 provides the possibility to specify the thresholds for each container separately.

Beyond these improvements, the alert generator component has been re-implemented to fit to the new requirements and features more easily.

Here is a summary of the improvements in MiCADO V3.1 compared to MiCADO V3:

Features

- list of individual Docker services to be scaled can be defined by the user;
- upscale and downscale thresholds of each Docker services can be defined by the user;
- default threshold settings can be applied for services not defined individually.

Bugfixes

- applying fixed versions instead of “latest” versions of the Docker images;
- undefined Docker service resource requirement is now considered unlimited instead of a fixed default value;
- fixing too frequent upscaling, which caused unnecessary resource allocation
- code refactoring.

Container scaling policy

Setting the scaling policy for each Docker container can be done in the “scaling_policy.yaml” file. This file specifies which Docker services will be auto-scaled and their scaling thresholds. One must specify the Docker service name (like service_name1) and scaling thresholds (scale down and scale up parameters). The format of the scaling policy file is as follows:

```
services:
  service_name1:
    scaledown: 20
    scaleup: 60
```

```
service_name2:
  scaledown: 20
  scaleup: 80
```

The name of the service is the same as defined in the compose file submitted to the MiCADO framework.

Limitations of MiCADO V 3.1

- scaling decisions are performed based on CPU load;
- new virtual machines can be launched every 5 minutes and new containers every minute;
- Docker containers will utilize newly launched Swarm node when a Docker service is scaled up;
- Docker service upscaling happens only in case of free resources; new VM allocation is applied when average CPU load of the VMs reaches a certain threshold.

8.2 Tutorial

MiCADO V3.1 has been released and the user guide (see Figure 6) for it has been published on the COLA website.

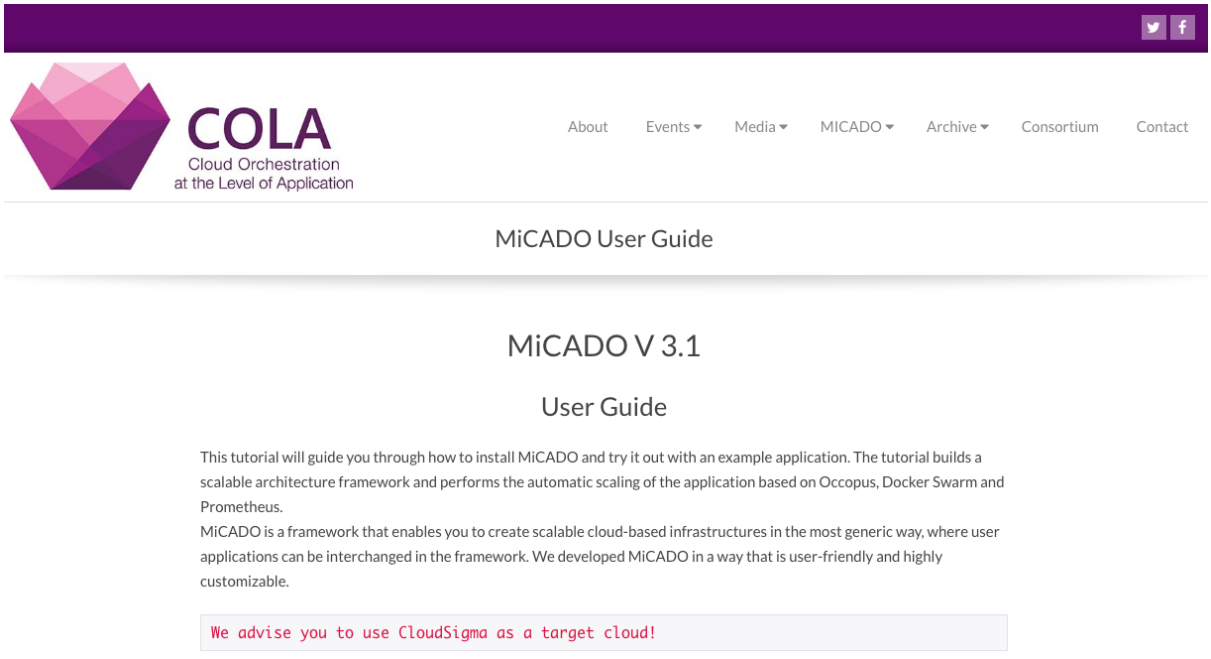

Figure 6 Screenshot of COLA webpage containing MiCADO V3.1 User Guide

8.3 Availability

Source code:
https://github.com/micado-scale/micado/tree/0.3.1

User guide for MiCADO V3.1:

https://project-cola.eu/micado-tutorial

Source code of Alert generator component:
https://github.com/micado-scale/component-alert-generator

Docker container of Alert generator component:
https://hub.docker.com/r/micado/alert-generator

9. Implementation of MiCADO V4

MiCADO V4 has been designed and implemented in order to support Batch Processing applications. This requirement was specifically raised by one of the COLA use case scenarios entitled Scalable Evacuation Planning Service, implemented by Saker Solutions and Brunel University. Details of this use-case can be found in the COLA deliverables D8.1 and D8.2. The use case is based on a discrete event simulation scenario where a large number of jobs is submitted to be processed by a user-defined deadline, by a dynamically changing number of cloud workers. MiCADO V3.1, as described in the previous section, could only support the scaling of cloud-based services. In order to support batch processing applications and deadline-based poilicies, additional components have been designed and implemented. This section, describes these additional components and explains how they have been integrated into MiCADO V4.

MiCADO V4 presents two components for *Batch Processing applications.* JQueuer and CAutoScaler take a list of jobs and an auto-scaling policy, start the application containers in the cloud, dispatch the jobs to the containers, and auto-scale up or down the number of containers in order to finish the jobs according to the given policy. JQueuer and CAutoScaler are designed to be platform-independent and cloud-independent. In addition to that they can work with any Container Orchestration Engine.

Please note that MiCADO V4 was developed in parallel to MiCADO V5 (described in Section 10). The rationale behind this parallel development was that we wanted to provide support for batch processing applications in a relatively early stage of the project to support the above-mentioned COLA use-case. Both MiCADO V4 and MiCADO V5 were based on MiCADO V3 (and V3.1). As a consequence, the MiCADO V4 components described in this section have not been integrated into the first release of MiCADO V5. However, the respective developer teams have investigated the integration of deadline-based batch processing scenarios into V5 and this integration is currently ongoing. MiCADO V4 components will be added in V5 as external components to support the desired use-case scenarios.

In the following, we present the structure of an experiment which is used in our design, the detailed design of JQueuer and CAutoScaler, respectively, the implementation of these components followed by testing and performance results, and finally the integration of these components into MiCADO.

9.1 Experiment Structure

In job submission type applications, for example simulations or image/video processing, there are always numerous scenarios that need to be completed on large computational resources. However, as these application areas evolved independently, the vocabulary used to identify the different units of execution is rather different. To avoid any confusion or misunderstanding, in this section we define and present these units of execution as experiment, job and task, as it is illustrated in Figure 7. These terms are described as follows:

A. **Experiment**

An experiment consists of two parts. The first part is a set of global parameters (upper part of Figure 7) while the second part consists of a list of jobs (lower part of Figure 7). Global parameters are, for example, the application container's image, the auto-scaling policy (e.g. the deadline by which the experiment should be finished), the minimum and maximum

resources for each container, and an estimated length of each task. An experiment is considered to be “accomplished” when all jobs are executed successfully.

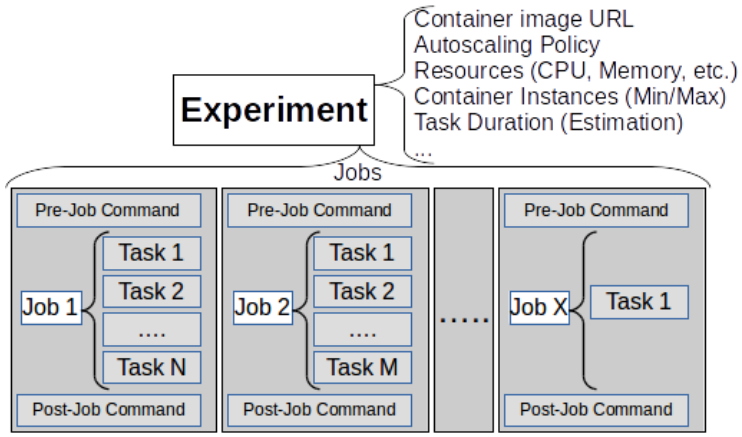

Figure 7 Experiment Structure

Experiment’s Description Language: An experiment (including jobs and tasks) might be stored using various formats, such as XML (Extensible Markup Language), JSON (JavaScript Object Notation), or YAML (Yet Another Markup Language).
There are two options to describe an auto-scaling policy. In the first option, the auto-scaling policy can be described using the same format that describes the experiment. In this case, the implementation of the auto-scaling policy is part of the main system. The other option is to consider the auto-scaling policy as an external module/service (module-based policy) which could be called (along with the arguments) using a RESTful (Representational State Transfer) API. In this case, the auto-scaler calculates the number of containers and sends it back to the system to take the appropriate action. In this second option, the experiment should have a field that contains the URL of the auto-scaling policy service. This option could be used in science gateways since it separates the development of the auto-scaling policies from the main system and gives the possibility to application developers to add their own Policies.

B. **Job**

A job consists of three parts: 1) Pre-job Command, 2) Tasks and 3) Post-job Command. While the first and third parts are optional, the second part is required. Pre-Job, Post-Job and task commands will be invoked within the container so as to launch an application, execute a system call, etc.
(1) Pre-Job Command (Optional): It is the command that should be invoked in the container at the beginning of each job and before running the tasks. The command might be used to initialize the parameters or to reserve the resources which are needed to execute a task.
(2) Tasks (Required): It is a list of tasks that should all be executed sequentially in the same container. If any task failed for any reason, the whole job will be considered as “failed” and it will be re-queued or cancelled, depending on the configuration of the system. Most of the time, each job consists of one single task. However, in some experiments tasks are depending on each other and need to be executed in a certain order inside a job (e.g. the first task would parse the argument and download files from a server, the second task would run the application, while the third task will upload the results to a server). Another

motivation to put multiple tasks inside one job is to enhance network utilization and reduce overhead by fetching and executing multiple inputs in a batch (e.g. fetching of multiple images at once in order to be analyzed sequentially instead of fetching one image at a time).
(3) Post-Job Command (Optional): This command should be executed after finishing all the tasks of the job and before getting a new job from the job queue. It might be used to free the resources, reset the parameters, etc. A job is considered to be "accomplished" when all of its tasks are executed successfully.

C. **Task**

A task is the smallest unit in this structure. It contains the command line that should be called in the container and the parameters (arguments) which should be passed along with this command. An example of the above structure is a simulation experiment. The experiment has global parameters including the container's image. Let us consider an experiment with thousand jobs where each job consists of one task. The task in this case will contain the command line of the simulation application that needs to be executed inside a container, and different sets of parameters that this command line requires.

9.2 JQueuer Design

JQueuer is a queuing system that can be used in conjunction with container technologies to support the execution of a large number of jobs and the enforcement of certain up or down scaling policies. JQueuer is a distributed system that is composed of two independent components: JQueuer Manager and JQueuer Agent (Figure 8). In the following, we are going to discuss the structure and the functionality of each of these. Communication methods between the JQueuer Manager and the JQueuer Agent were left out so as to be defined in the implementation according to the technologies used.

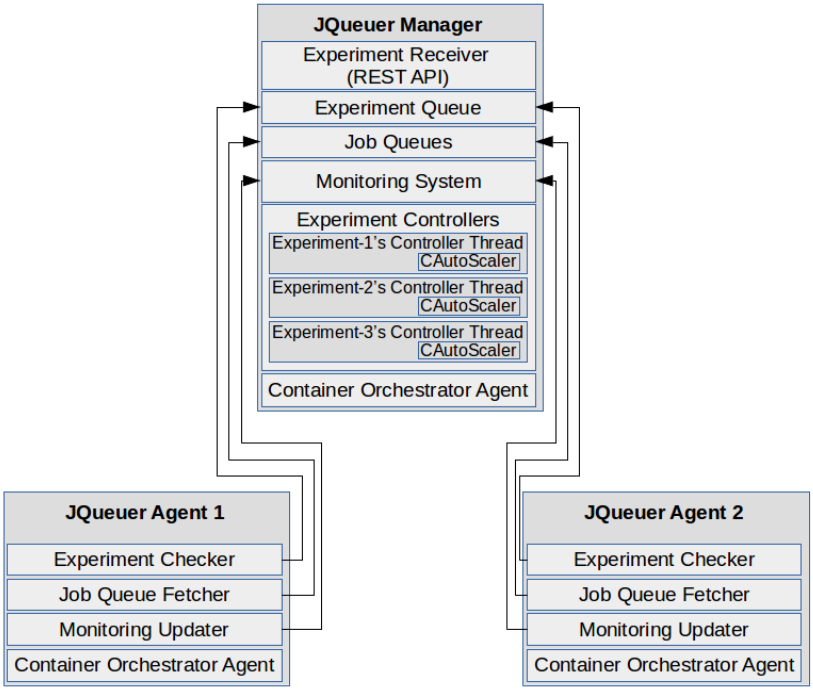

Figure 8 JQueuer Structure

9.2.1 JQueuer Manager

JQueuer Manager is the main component of the JQueuer system which should be running on a Container Orchestration Manager (e.g. Docker Swarm) where it can control the containers and the services. The JQueuer Manager consists of several sub-components. Each sub-component has different sets of tasks. The sub-components and their tasks are described as follows:

1) Experiment Receiver: Is a RESTful web service which provides a standard API to submit the experiment file/object to the JQueuer system via HTTP Request. When an experiment is received, the “Experiment Receiver” will generate an “Experiment ID” which will be used to identify this experiment in the system. The experiment sender will receive this ID as HTTP response.

2) Experiment Queue: It is a list of the experiment IDs which have been submitted. Each experiment has two important items in this queue: the Experiment Service Name and the Job Queue ID. JQueuer Agents use this list to recognize whether the containers running (on their virtual machines) should be controlled or not.

3) Experiment Controllers: It is an array of threads in which each thread controls a single experiment. A controller will be instantiated directly after receiving the experiment and it will keep running as long as the experiment is not accomplished. A controller is in charge of the following tasks:

   a. Job Parsing: It parses the jobs from the experiment file/object and adds these jobs in a job queue dedicated to this experiment.

   b. Monitoring (Data Analysis): It analyses the experiment execution data that was received from the Monitoring Database. The resulted information will be used for deciding whether the system should scale up, scale down or continue with the current number of containers.

   c. Decision of auto-scaling: This is done using the CAutoScaler component which calculates the number of containers needed to accomplish the experiment according to the auto-scaling policy. The CAutoScaler is discussed in detail in Section 9.3.

4) Job Queues: Each experiment depends on a dedicated job queue which has a unique ID. The mechanism used to dispatch jobs from job queues is discussed in Section 9.2.2.

5) Monitoring System: The monitoring system contains the monitoring data related to all experiments. The system will be accessed from the Experiment Controllers so as to gather the monitoring data related to their experiments.

6) Container Orchestrator Agent: The agent works as a bridge between the Experiment Controllers and the Container Orchestrator Daemon that is used in the cloud. The agent receives the commands from the controllers, forwards them to the Daemon and waits for the results. The supported commands by the agent are:

   a. Create Service: This command is used to create the Experiment’s Service which requires two parameters: container’s image URL, and initial number of containers in the service. Before sending the command to the Daemon, the agent will form a third

parameter (Service Name) using the image URL parameter. Service Name will be used as an ID during the execution.

b. Get Service's Status: To get the status of its Experiment's Service, the controller issues this command along with its Service Name. The result, consists of the data related to the service including the number of containers running under this service and their status.

c. Scale Up/Down: The parameters required for this command are: Service Name and number of containers needed to run under this service which is calculated by the CAutoScaler.

d. Destroy Service: This command will be issued by the controller when all the underlying jobs are executed successfully.

9.2.2 JQueuer Agent

An instance of JQueuer Agent component should be running on each Container Orchestration Node. An instance should exist in the Container Orchestration Manager if an Experiment Service is running one or more of its containers in this Manager. The JQueuer Agent is responsible of controlling the services' containers of the experiments, fetching jobs from the job queues, monitoring the execution and sending data to the JQueuer Manager. From functional point of view, this component can be divided into sub-components as follows:

1) Experiment Checker: This sub-component monitors the Experiment's Queue in the JQueuer Manager. When a new experiment is added, the Experiment Checker will fetch the Experiment Service ID and the Job Queue ID items.

2) Job Queue Fetcher: It uses the Job Queue ID which has been obtained from the Experiment Checker so as to fetch the jobs from an experiment job queue and execute them on the containers of the corresponding Experiment Service.

3) Monitoring Updater: It monitors job execution on local containers and sends data and statistics to the Monitoring system in the JQueuer Manager.

4) Container Orchestrator Agent: The agent acts as bridge between the JQueuer Agent and the Container Orchestration Daemon on the local machine. The first item is used by Job Queue Fetcher so as to recognize the containers that belong to a certain experiment.

9.3 CAutoScaler Design

CAutoScaler is the second main part in the system. It is responsible for taking and executing the decision of scaling up/down an experiment by calculating the number of containers which should be running according to the scaling policy. The CAutoScaler works as a sub-component of the Experiment Controller. As there is a controller for each experiment, these will all have their own CAutoScaler instances. The current design focuses on one single policy, the deadline policy. In this policy the experiment should be accomplished before a given deadline by using at least the minimum number of containers and without surpassing the maximum number of containers. Each container will be allocated with minimum number of resources (memory and processing). Please note that the system is not limited to this one

policy, and it is very much possible to extend it to process different policies by implementing them in the CAutoScaler. The CAutoScaler has two main components: Container Calculator and Container AutoScaler. These two functionalities are explained in detail in the following two subsections.

9.3.1 Container Calculator

This functionality focuses on the process of calculating the number of containers needed at any moment. The process (Figure 9) starts as soon as the experiment is received and it calculates the number of containers needed to finish the experiment according to the scaling policy.

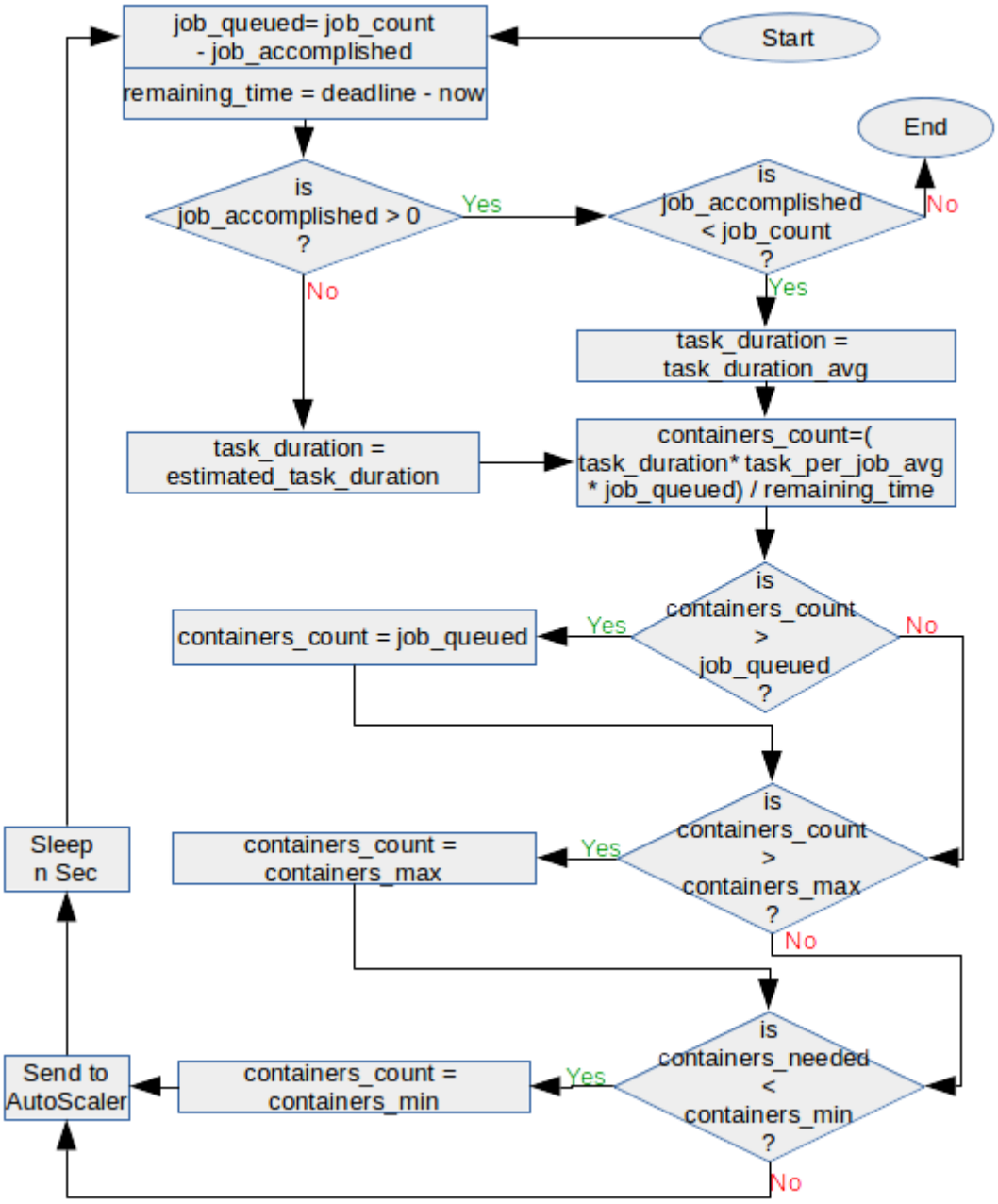

Figure 9 Container Count Calculator

The algorithm applied to calculate the number of containers depends on the actual phase of the experiment. The first phase is called the “Initial Phase” which covers the period between receiving the experiment and the successful execution of the first job. The second phase is called the “Monitored Phase” which starts as soon as a job is successfully executed, and it finishes when the experiment is finished which means all jobs are executed.

The main difference between the two phases is the way of calculating the average execution time of a task. Since there is still no monitoring data in the initial phase, the calculation is based on the estimation of duration needed to execute a single task as provided by the user. In the Monitored Phase, the calculation is based on the average of duration of all tasks (of

one particular experiment) which have been executed so far. The number of containers ("containers count") will be equal to the multiple of the duration needed to execute a single task, the average number of tasks per job, and by number of jobs in the queue, and the result is divided by remaining time until the deadline. The resulted value will also be checked against the minimum and maximum number of containers. At the beginning, the controller will start the Experiment Service using the "containers count" as an initial number of containers. When the Experiment Service is started and the number of containers which are in "running" state is greater than zero, the Container AutoScaler will process the number resulted from the Container Calculator and scale up/down the service as explained in Section 9.3.2.

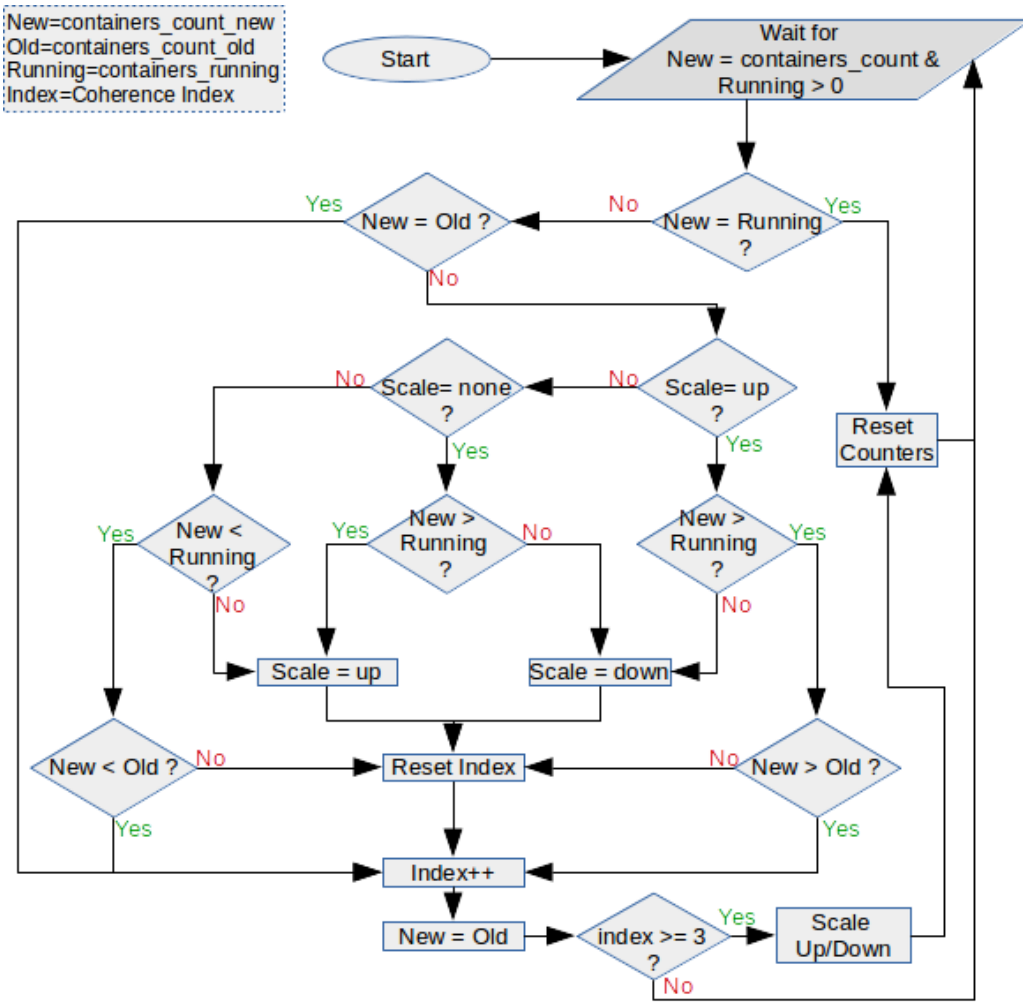

Figure 10 Container AutoScaler

9.3.2 Container AutoScaler

The Container AutoScaler (Figure 10) starts processing the number of required containers (received from the Container Calculator) as soon as there is at least one container (of the Experiment Service) is in "running" state.

The process uses coherence index so as to delay the scaling up/down in order to make sure that the number of required containers is coherent with the last two results. The coherence index is increased by one every time the function receives a new calculation suggestion. For example, if the last calculation suggested a scaling up while the current one is suggesting a scaling down, the function will reset the counters and start over without performing any scaling. If the last calculation and the current one are both suggesting the same thing (scaling up or scaling down), the function will take the lesser number of containers between the two calculations in the case of scaling up and the greater number in case of scaling down.

The idea behind selecting the lesser and greater number is to scale up and down gradually as much as possible which gives the system the possibility to re-evaluate the number of containers needed to finish the experiment. In both cases, the function will not scale up or down until the coherence index is at least three which means the last three calculations are suggesting the same action.

9.4 System Implementation

In this section, technologies and tools which have been used in the first implementation of JQueuer and CAutoScaler (Figure 11) are described. The two main components of the designed architecture, JQueuer Manager and JQueuer Agent have been developed using Python 3 and were prepared as Docker images. The MiCADO master which is the Docker Swarm Manager node in this solution (left hand side of Figure 11) hosts JQueuer Manager, with additional components for queuing, scheduling and data storage.

These components include Celery, an asynchronous distributed task/job queuing system that was used together with Rabbitmq (message broker) [10] and Redis (an in-memory database) [11] for capturing results. Redis was also applied for experiment queuing, simplifying data exchange between the manager and the agents. For monitoring, statsd [12] is used for saving statistics and events of the JQueuer Agents so that they can be accessed from Prometheus [6], and Grafana [13] as data visualization tool. The Experiment is described in JSON format. Each MiCADO worker hosts an instance of JQueuer Agent (right hand side of Figure 11) which has two main components: Container Updater and Container Manager.

A. **Container Updater**: The main function of this sub-component is to monitor the containers on the local machine so as to distinguish containers which belong to a particular experiment. Each Docker container shows in its information the name of its Docker Swarm Service. The Container Updater will check the container services against the experiment list in the Redis server. If the container is new and belongs to one of the experiments, a new Container Manager will be forked so as to manage this container and it will be added to its Manager List.

B. **Container Manager**: It is responsible for managing and controlling an experiment container. The life cycle of a Container Manager starts by fetching a job from the job queue belonging to the container. It then executes the *pre*-job script in the container and goes through the list of tasks. Tasks are executed sequentially, and after finishing them successfully, the Container Manager will run the *post*-job script. It sends the statistics to the statsd server including: job starting/finishing time and task starting/finishing time. If

the job failed for any reason, it sends the time spent before the job has failed to statsd, and it signals this failure to the Celery server. After finishing the job, it fetches another job and starts executing it. Container Manager keeps working until the job queue of this experiment becomes empty.

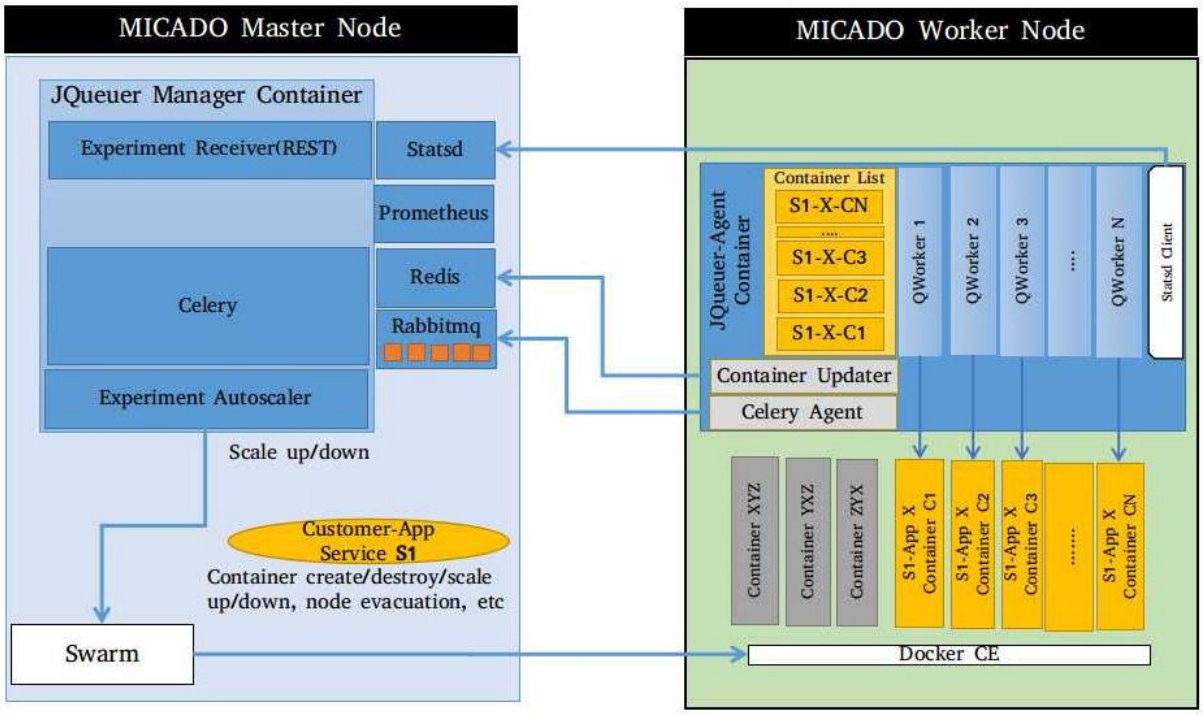

Figure 11 System Implementation

Containers from different experiments can coexist on the same machine, as it is shown in (Figure 11). JQueuer Agent will provide each Container Manager with a Container ID and a Job Queue ID. The Container List contains only those containers that belong to an experiment and have been assigned to managers. This is why we do not see Container S2-C3 on the list as this should be added only when the Container Updater checks the containers in its next round. The containers JQueuer Agent, XYZ, YXZ and ZXY have been ignored since they do not belong to any experiment.

9.5 Testing & Results

JQueuer was tested with Repast Simphony (RepastS), an open source agent-based modeling and simulation system using a simplified infectious disease simulation model. Modeling and simulations are often based on scientific work and involve interdisciplinary research teams which can be supported using science gateways.

Jobs were added to the experiment's JSON file along with the following parameters: RepastS Docker image URL, estimated execution time of a single task, minimum memory and CPU required by each RepastS container, minimum and maximum number of containers, and deadline. RepastS was deployed in a Docker container, also including a script that takes the HTTP URL of a simulation scenario and the FTP URL of the results

server. The script fetches the simulation scenario from the HTTP server, processes it, and then transfers the output file to the FTP server.
Several scenarios have been tested with varying parameters. The number of jobs varied between 250 and 1000 jobs per scenario, the deadline was between one and two hours, and the maximum number of containers was between 10 and 20.

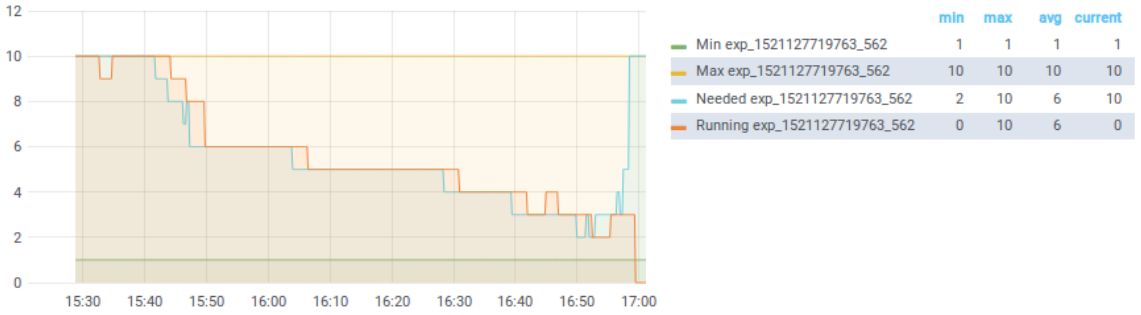

Figure 12 Repast - Container Autoscaling

The results shown in Figure 12 are for the following scenario: one task per job, 500 jobs, deadline is 90 minutes, estimated task duration is two minutes, the minimum number of containers is one while the maximum is ten, ten Docker Swarm Nodes.

The duration statistics were as follows: Job Execution Duration (Min: 52 sec, Max: 80 sec, Avg: 59 sec), Failure Duration (Min: 28 sec, Max: 47 sec, Avg: 38 sec) and eight jobs have failed (during scaling down as their containers have been terminated), but these have all been resubmitted.

Finally, Figure 12 shows how the CAutoScaler was scaling up/down the containers according to the accomplishment of the experiment. The figure clearly indicates that the experiment started with the maximum number of containers based on the estimated execution time provided by the user. However, as the system realised that a smaller number of containers will also be enough to finish the experiment by the set deadline, it started to scale down. Figure 12 also shows that the experiment finished exactly at the given deadline.

9.6 JQueuer and CAutoScaler in MiCADO V4

JQueuer and CAutoScaler have been applied to extend MiCADO V3 (as presented in Deliverable D6.2) to support batch processing applications. This section describes the design of MiCADO V4. MiCADO V4 (Figure 13) keeps the major components of the generic MiCADO design as described in D6.2. However, the MiCADO master node is extended with JQueuer Manager, while the MiCADO workers are running JQueuer Agent.

In this integration, the JQueuer infrastructure can utilize the monitoring system (prometheus and grafana) from MiCADO. The JQueuer in this case only calculates the number of containers needed to finish the experiment according the policy (from the policy keeper) and to values stored in the monitoring system by the JQueuer agents. The autoscaler is in charge of scaling up/down the containers according the calculated number.

The JQueuer in this case might be launched only when there is a need to run batch processing jobs in order to save the resources on master and worker nodes.

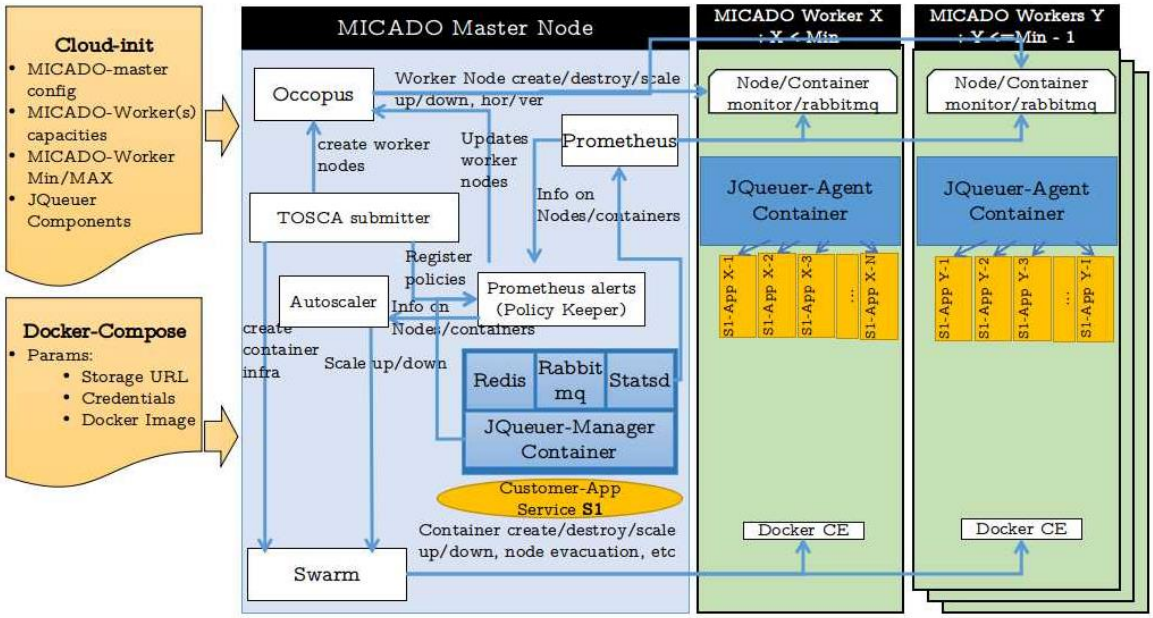

Figure 13 JQueuer & CAutoScaler Integration

9.7 Availability

Source code and a step by step guide on how to launch the infrastructure, submit the experiments and debug the JQueuer:

https://github.com/micado-scale/micado/tree/0.4.x

Even though the source code is documented, to understand the overall structure the reader needs to read the paper [14] which has been presented during IWSG2018.

JQueuer Manager and JQueuer Agent source codes:

https://github.com/micado-scale/component-jqueuer-manager/tree/0.1.x
https://github.com/micado-scale/component-jqueuer-agent/tree/0.1.x

Version 0.2.x of JQueuer Manager and JQueuer Agent are modified to work with MiCADO V5 which they separate the JQueuer from the autoscaling policy:

https://github.com/micado-scale/component-jqueuer-manager/tree/0.2.x
https://github.com/micado-scale/component-jqueuer-agent/tree/0.2.x

10. Implementation of MiCADO V5

The architecture of MiCADO V5 follows the architecture design plan introduced in Deliverable D6.1. During the reporting phases, the different components have been realized step-by-step. The architecture can be seen in Figure 14 (different colors indicate the different phases).

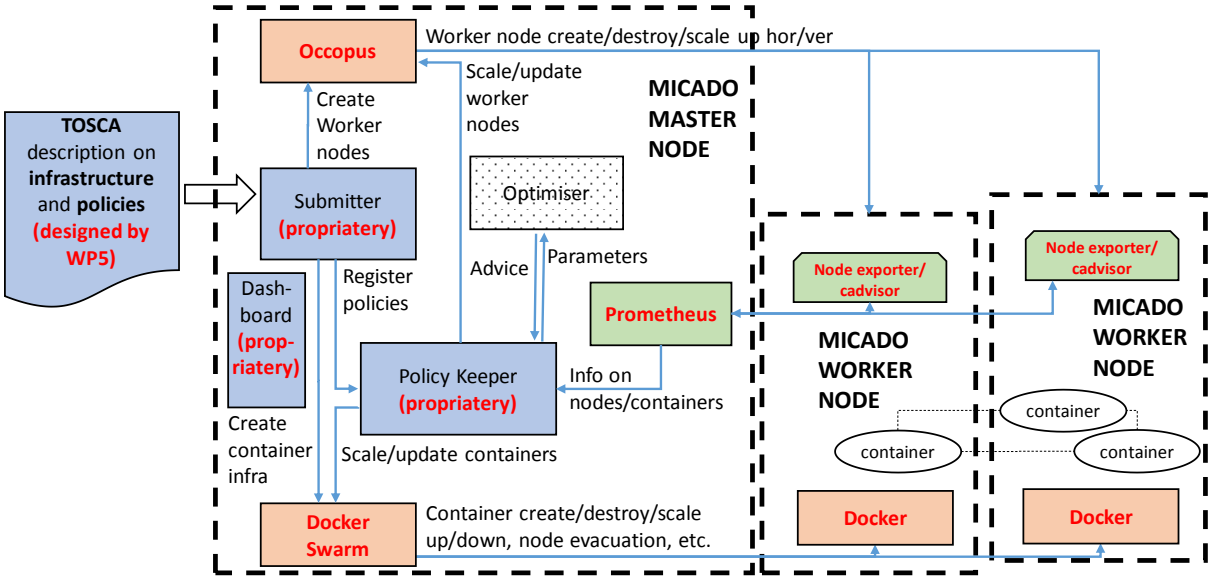

Figure 14 Architecture of MiCADO V5

In the first and second phase of the implementation the components depicted by red and green boxes in Figure 14 have been realized. In the current phase, the components (depicted by blue boxes in Figure 14) are implemented. These are the Policy Keeper (to support automatic decision making), the Dashboard and the Submitter with TOSCA support. The implementation of these components has been realized in MiCADO V5 and is documented in the next sections. Beyond these components, MiCADO V5 has also been extended to support deployment based on Ansible playbook.

MiCADO V5 is the first version which supports the entire life cycle of container applications including the submission, execution, automatic-scaling and shutdown. The TOSCA-based description containing the infrastructure and scaling policy specification is submitted through the REST API of the Submitter. The Submitter separates the information and communicates to the relevant components (Occopus,Docker and Policy Keeper) with its internal adaptors. First, Occopus allocates the necessary number of virtual machines to build MiCADO workers. Once the Workers are ready, Docker services are deployed and finally, Policy Keeper starts the automatic scaling of the nodes and containers based on the Scaling Policy. Policy Keeper uses the Monitoring system to collect metrics necessary for decision making. During the operation, internal status of the system can be inspected through the Dashboard of MiCADO.

In the next subsections, components and improvements compared to the status reported in COLA deliverable D6.2 are detailed. They are as follows: 1) Ansible deployment, 2) Policy Keeper, 3) TOSCA Submitter and 4) Dashboard.

10.1 Ansible playbook

Ansible-micado replaces MiCADO's cloud-init and shell script implementation (applied in MiCADO versions until V4) with an Ansible playbook-based deployment method.

Ansible

Ansible is an IT infrastructure automation tool that allows deploying and configuring infrastructure components using YAML-based description. The original MiCADO implementation used cloud-init and this MiCADO was only deployable on newly started VM's. Ansible allows deployment on existing resources and gives more refined control over the deployment procedure. Ansible is still invoke-able from cloud-init, thus the original deployment method still works.

Deployment methods

Ansible-MiCADO supports two different deployment methods: remote and local deployment. In the remote deployment, an empty virtual machine is contextualized from a controller node. This controller node contains the whole Ansible-MiCADO repository. The remote deployment is the recommended method.
In the local deployment case, the created virtual machine will be contextualized locally without a controller node. In that case, the virtual machine has to download the copy of the Ansible-MiCADO git repository. These two deployment methods are displayed in Figure 15.

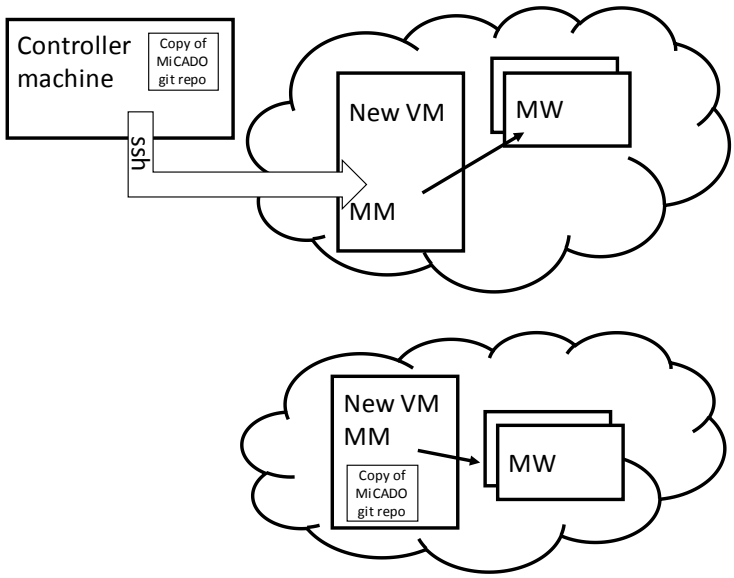

Figure 15 Deployment methods for MiCADO V5

Structure of Ansible playbook

Playbooks are Ansible's configuration, deployment and orchestration language. They can describe a set of steps in a general way to configure, build or modify applications. In the Ansible roles, there is commonly used directory layout, which contains the following building elements: files, handlers, templates, tasks and vars.

- **Files**
  Files contain the component's files that no longer needed to be modified on the fly and can be transferred to the system. These are often configuration files or files that are needed for the deployment.
- **Handlers**
  Modules should be idempotent which means that modules can relay when they have made a change on the system. Playbooks have a basic event system that can be used to respond changes. Handlers are defined by a list of tasks and the events can trigger the handlers.
- **Templates**
  Templates are files that can be changed on the fly. Templates are processed by the Jinja2 [9] templating language. It is a very powerful feature because you can dynamically change the content in your configuration files and that gives more flexibility in configuring systems.
- **Tasks**
  Each play contains a list of tasks. Tasks are executed in sequential order, one at a time before moving on to the next task. Modules should be idempotent, which means that if you are running a module multiple times you should get the same effect as running it just once.
- **Vars**
  You could define a specific variable in each role. You can highlight variables that can be later referenced.

Structure of MiCADO Ansible repository

Ansible-MiCADO playbook deploys the MiCADO master: configures the network, installs Docker and Docker Compose, downloads the required components images, configures Docker Swarm and starts the environment via Docker-Compose.

Under the files section (see Figure 16) in the MiCADO-master role there are the components config files. The misc file is a script, which waits the unattended upgrade in the cloud virtual machine to finish. The tasks folder contains the main parts of the whole task separated with logical separation. The templates folder contains the dynamically changing configuration. Usually, it depends on the IP or the hostname, but variables or gathering fact can be used here as well.

The order of the execution is the following: first the main task runs the wait-updates.sh to wait for the unattended upgrade to finish. After that, the role sets the DNS server (some provider does not have DNS server) hostname and includes the docker-install task. The Docker-install task installs the Docker and the required packages, initializes the Docker Swarm with setting the master availability to drain. Next, the files task creates the configuration and log folder and files for the different components. The docker-pull-images task pulls every required container image from Docker hub. The worker_node task creates the authentication file and generates an infrastructure template file. The start_micado task clones the source code from GitHub for the TOSCA submitter and Policy Keeper and starts the MiCADO master containers by using the Docker compose command.

```
ansible-micado
├── README.md
├── micado-master.yml
└── roles
    └── micado-master
        ├── README.md
        ├── files
        │   ├── alertmanager
        │   │   └── config.yml
        │   ├── consul
        │   │   └── config.json
        │   ├── grafana
        │   │   ├── data-dashboards-micado.json
        │   │   ├── provisioning-dashboards-provider.yaml
        │   │   └── provisioning-datasources-prometheus.yaml
        │   ├── misc
        │   │   └── wait-updates.sh
        │   ├── policykeeper
        │   │   └── prometheus_config_template.yaml
        │   ├── prometheus
        │   │   └── prometheus.yml
        │   └── toscasubmitter
        │       └── key_config.yml
        ├── meta
        │   └── main.yml
        ├── tasks
        │   ├── docker-install.yml
        │   ├── docker-login.yml
        │   ├── docker-pull-images.yml
        │   ├── files.yml
        │   ├── main.yml
        │   ├── setup-master-node.yml
        │   ├── start_micado.yml
        │   └── worker_node.yml
        └── templates
            ├── consul
            │   └── checks.json.j2
            ├── docker
            │   └── docker-compose.yml.j2
            ├── policykeeper
            │   └── policykeeper_config.yaml.j2
            └── worker_node
                ├── cloud_init_worker.yaml.j2
                └── temp_infrastructure_descriptor.yaml.j2
```

Figure 16 Structure of MiCADO Ansible playbook

10.2 Policy Keeper

Policy Keeper realizes the functionality of the scalability decision service. The design directions and goals have been summarized in Section 6. This section focuses on the implementation details.

10.2.1 Implementation

Policy Keeper implements the scalability decision service by monitoring, evaluating and instructing. The policy keeping functionality has been developed by scratch in Python using Flask for implementing the service endpoint.

The Flask based python code is running in a container on the MiCADO Master node and communicates to Prometheus (for evaluating queries and alerts), to Occopus to realize node scaling and to Docker to realize Docker service scaling.

The user-defined input for Policy Keeper is a YAML-based description specified in Section 7.3. The internal operation of Policy Keeper is illustrated in Figure 17. The following paragraphs details the internal operation.

The operation of Policy Keeper starts with the invocation of the start method of its REST API (Step1 in Figure 17). The parameter is a scaling policy description in which Policy Keeper first resolves the text where references are used. At this step Jinja2 [9] resolution is used.

The next phase (Step2 in Figure 17) configures Prometheus. Configuration involves the registration of the user-defined exporters. In case the exporter is an external entity no more to be done by the Policy Keeper. For internal exporters (running as a service under MiCADO) Policy Keeper instructs Docker to let the Prometheus service attach to the network of the exporter service. Otherwise, no communication is possible. The next step, is the generation of the rule files based on the alert definition specified in the policy file. At the end of this phase, Prometheus is notified to reload its configuration i.e. to activate the changes.

At this point, all preparation has been done, the periodic maintenance (evaluation and scaling) cycle can start. Each cycle starts with the node maintenance followed by the container maintenance. Between two consecutive cycles, a predefined period elapsed.

Node maintenance (Step3 in Figure 17) starts with collecting all the inputs necessary to evaluate a scaling rule (specified by the policy). The first step, is to evaluate the variables defined in the queries section of the policy. For each item a Prometheus query expression is defined which is sent to Prometheus for evaluation. When all variables are evaluated, it continues with collecting the state of alerts if specified any.

When an alert is fired, Policy Keeper is notified by Prometheus through the Alert Manager. These notifications are registered inside the Policy Keeper and evaluated when the status of alerts is prepared for the scaling rules. For each alert, a Boolean variable is generated (as specified in Section 7.3.4) which can be referred by the scaling rule.

The final step in collecting inputs for the evaluation is to update the values of the built-tin variables (specified in Section 7.3.5).

Evaluating the scaling rule for the node means the execution of the Python code specified in the scaling policy as scaling rule (specified in Section 7.3.5). The evaluation is done by a separate module in the Policy Keeper. The result of the evaluation is the number of instances to scale the nodes to. The final step is to notify Occopus about the scaling decision and set the number of MiCADO worker instances target number to the one returned by the evaluation.

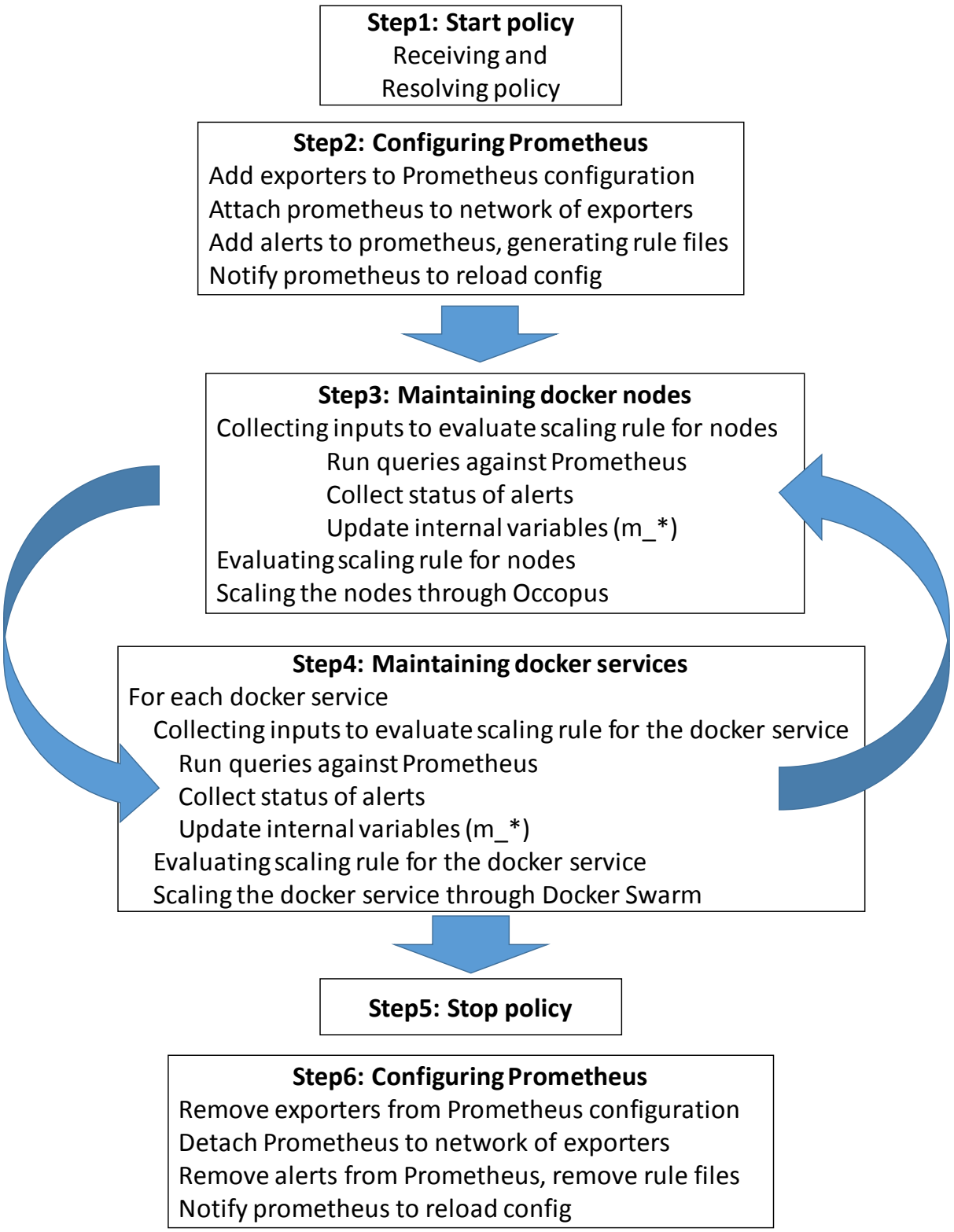

Figure 17 Internal high-level operation of Policy Keeper

The next stage during the operation is to perform the maintenance of the Docker Services (Step4 in Figure 17). The procedure described in the following paragraphs is performed for each individual Docker Service specified in the scaling policy. Practically, it has the same pattern as a for the node maintenance, since the same steps are performed for the Docker Service.

To scale a Docker Service, collecting the inputs to evaluate the scaling rule of the Docker Service starts, including the evaluation of the queries by Prometheus, collecting the status of the alerts and to update the internal variables. Following the pattern drawn by the node maintenance, evaluating the scaling rule happens for the Docker Service and the outcome of the evaluation realized by instructing Docker Swarm to scale the Docker Service to the calculated number of replicas.

When the Policy Keeper is instructed to stop the maintenance (Step5 in Figure 17), maintenance loop (Step3 and Step4 in Figure 17) is completed. As a consequence, Policy Keeper rolls back all the changes made in Step2, i.e. removes changes from the configuration file of Prometheus, detaches Prometheus from any network it was attached to, removes rule files containing the alerts and notifies Prometheus to reinitialize its configuration.

At this point, Policy Keeper becomes inactive and waits for further instruction through its REST API specified in detail in Section 10.2.2.

For the proper operation, it is necessary to configure Policy Keeper through its configuration file. The details on how to configure Policy Keeper can be found in Section 10.2.3.

10.2.2 REST API

Policy Keeper is a microservice with a service endpoint exposing several functionalities on different paths. This section details the REST API of Policy Keeper.

`POST /policy/set <policy description>`
: Invoking this call results in storing the policy description (passed as an argument) internally and permanently. Policy Keeper does not start maintaining the number of nodes and containers.

`POST /policy/start`
: Invoking this call results in starting the policy keeping procedure based on the policy description set previously by the /policy/set function.

`POST /policy/start <policy description>`
: This call is the combination of the previous two calls (/policy/set and policy/start). Invoking this call results in storing the policy description (passed as an argument) internally and permanently, then start maintaining the number of nodes and containers. Prometheus configuration is updated according to the specification found in the policy description.

`POST /policy/stop`

Invoking this call results that the Policy Keeper stops maintaining the number of nodes and containers. All previous updates on the Prometheus configuration is removed.

`POST /alerts/fire`
: This call is reserved for Prometheus Alert manager to notify Policy Keeper on the state of the alerts.

10.2.3 Configuration

The Policy Keeper microservice can be configured to fit to its environment. The configuration mainly includes Prometheus, Docker Swarm and Occopus related settings since Policy Keeper is communicating with these components.

The following list details the variables to be set in the Policy Keeper configuration file with an example value:

`prometheus_endpoint: 'http://prometheus:9090'`
: Endpoint of Prometheus in 'http://<ip>:<port>' format.

`prometheus_config_template: '/config/pk/pr_cfg_template.yaml'`
: Path of Prometheus configuration template file to be used when generating a new configuration file for Prometheus

`prometheus_config_target: '/config/prometheus/prometheus.yml'`
: Path of Prometheus configuration file to update

`prometheus_rules_directory: '/config/prometheus'`
: Path of directory for storing Prometheus rules

`swarm_endpoint: 'localhost:2375'`
: Endpoint of Docker Swarm in '<ip>:<port>' format.

`occopus_endpoint: 'http://occopus:5000'`
: Endpoint of Occopus in 'http://<ip>:<port>' format.

`occopus_infra_name: 'micado_worker_infra'`
: Name of infrastructure under Occopus containing the woker nodes

`occopus_worker_name: 'worker'`
: Name of the worker node under Occopus to be scaled up/down

`docker_node_unreachable_timeout: 60`
: Timeout for detaching unreachable docker nodes.

`logging:`
: Python logger configuration structure

10.3 TOSCA Submitter

10.3.1 General overview

The COLA project has selected OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications) [15] as the platform language specification for creating templates that describe the topology and policies of an application to be deployed in the cloud. The TOSCA Submitter is the main entry point for MiCADO and its main goal is to provide the user of the

system with an interface with which they may submit TOSCA templates in order to deploy applications in MiCADO. TOSCA is built on inheritance and typing, and supports customisation and extensibility within the bounds of the language. This permits the expression of platform-specific details when describing different parts of a deployment, making it easy to describe topologies and policies that can be understood by the varying components inside MiCADO.

The TOSCA Submitter runs in a Docker container on the MiCADO Master Node, and exposes a RESTful API to a user looking to deploy an application to the cloud. Through the API the user can submit a TOSCA template to deploy an application, can update an application using a modified version of the original TOSCA template, can bring the application down, and can retrieve data on one or all applications deployed in MiCADO.

The technology agnostic approaches of TOSCA and of MiCADO were strong motivations behind the design of the TOSCA submitter. Before understanding the TOSCA Submitter's functionality at a low-level, a general overview is necessary. A TOSCA template describes the properties and configuration relevant to the different aspects of an application's deployment in the cloud. The different sections that describe deployment include, but are not limited to:

- The hardware resources required for deployment (HDD, memory, CPU, network…);
- The software requirements (the application itself, dependencies…);
- Deployment policies (geographical restrictions, available connections, scaling rules…);
- Security policies (credential management, firewall configuration…).

These descriptions of the different aspects of deployment are required by specific end components, and the TOSCA Submitter, as the entry point, is the main conduit for this information. The TOSCA submitter is thusly tasked with feeding the relevant descriptions to their respective components, which should then process the information in order to deploy the application.

MiCADO, in turn, has a vendor-neutral interface that permits for interchangeability in the individual components that drive the different aspects of deployment. This drove another aspect of the design of the submitter, and to match the pluggable design of MiCADO with the technology agnostic approach taken by TOSCA, the Submitter uses small purpose-built adaptors to translate from the high-level TOSCA to a format understood by the chosen end component. These adaptors are also responsible for execution, or ensuring that the components use the translation to carry out the necessary tasks for the launch and tear down of the deployment. In the current implementation of MiCADO, adaptors have been written for the following components that relate to the aforementioned four aspects of deployment:

- Cloud Orchestration: Occopus [4] ;
- Container Orchestration: Docker [5];
- Deployment-Policy Management: PolicyKeeper (proprietary);
- Security-Policy Management: SecurityEnforcer (proprietary).

10.3.2 High-level functionality

The high-level architecture of the MiCADO TOSCA Submitter is illustrated in Figure 18. The most basic function of the TOSCA Submitter is the submission of a TOSCA template to deploy an application in the cloud. Upon submission of the template to the TOSCA Submitter, it first passes to the OpenStack TOSCA Parser [16] for validation and parsing. The TOSCA Parser validation checks not only for correctness in the YAML syntax of the file, but also for adherence to TOSCA naming conventions and to the rules of basic inheritance, a key concept in TOSCA. Successful validation returns a parsed *ToscaTemplate* object, with a variety of attributes and methods that facilitate future handling of the TOSCA template. The parsed template object then passes through a MiCADO validator that performs checks to ensure that the TOSCA template conforms with further syntactic conventions and custom type definitions described by MiCADO. These checks are performed by comparing the MiCADO-specific relationships and types used in the TOSCA template with their original definition.

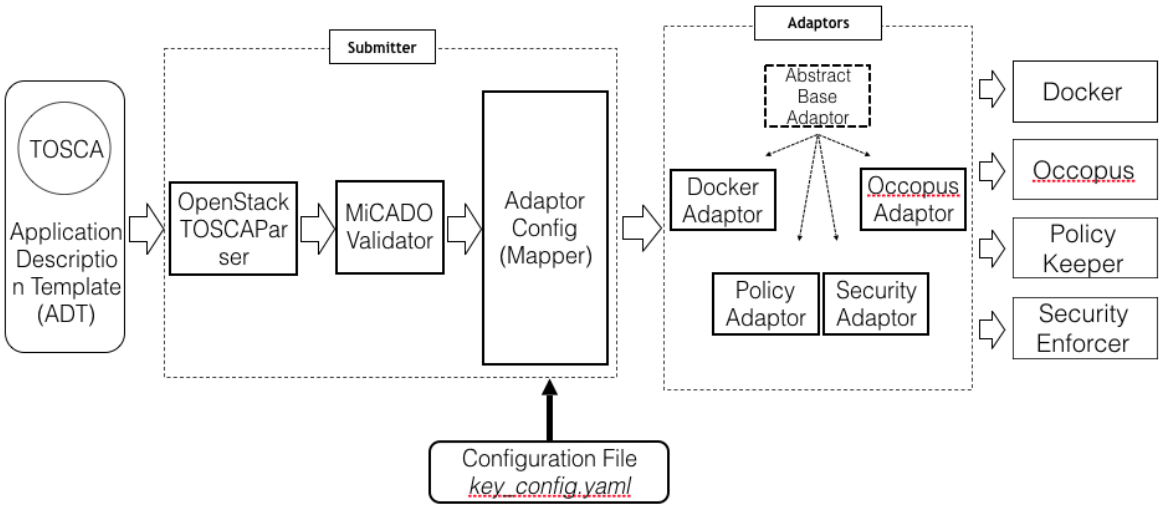

Figure 18 MiCADO Tosca Submitter architecture

After validation, the template object passes to the adaptor configuration component, which performs two tasks. First it configures the adaptors as specified in *key_config.yaml* by setting configuration parameters such as working directory and output paths. Then it performs a mapping step, which can separate out sections of the TOSCA template based on a configurable key list, and pass the resultant sections to the intended component adaptors. Sections relevant to security policies are a strong candidate for being mapped out and passed to only a relevant adaptor, but other configurations are possible too.

The template object then arrives, in part, or in whole at each of the adaptors that have been configured for MiCADO. The adaptors are developed to match the chosen end components responsible for deployment. Adaptors must extend an abstract base class that defines methods that relate to translation, execution, update and undeployment of the relevant part of deployment overseen by the respective component. Adaptor developers have freedom in development as long as these abstract methods achieve the deployment steps they are expected to.

In an order specified within the adaptors' configuration, the Submitter first calls the *translate* method of each of the adaptors to complete the translation. This step involves converting data of the TOSCA template object into a format that can be understood by the targeted component. As an example, the Docker adaptor in the current implementation of MiCADO produces a Docker Compose file. The second step is execution. Adaptors, in a configurable order, pass the data to their components in order to achieve their portion of deployment. For example, the Occopus cloud orchestrator makes the relevant API calls to launch the virtual machines required for the infrastructure. After translation and execution, application deployment is complete and a unique ID which refers to this deployment is returned to the user in the API response.

The *update* and *undeploy* methods of the adaptors are called when the relevant API calls are made to the TOSCA Submitter. An *update* requires a modified TOSCA template as well as the unique ID of the deployment in question. The *update* method of the adaptor should carry out the specific tasks necessary for updating their component's part of the deployment, though this often involves another translation and execution step, albeit slightly modified. The *undeploy* call takes the unique ID of the deployment and the adaptors, and in a specified order instructs their components to take down the resources allocated, or provisions made for this specific deployment.

10.3.3 Implementation details

The internal architecture of the TOSCA Submitter is demonstrated in Figure 19.

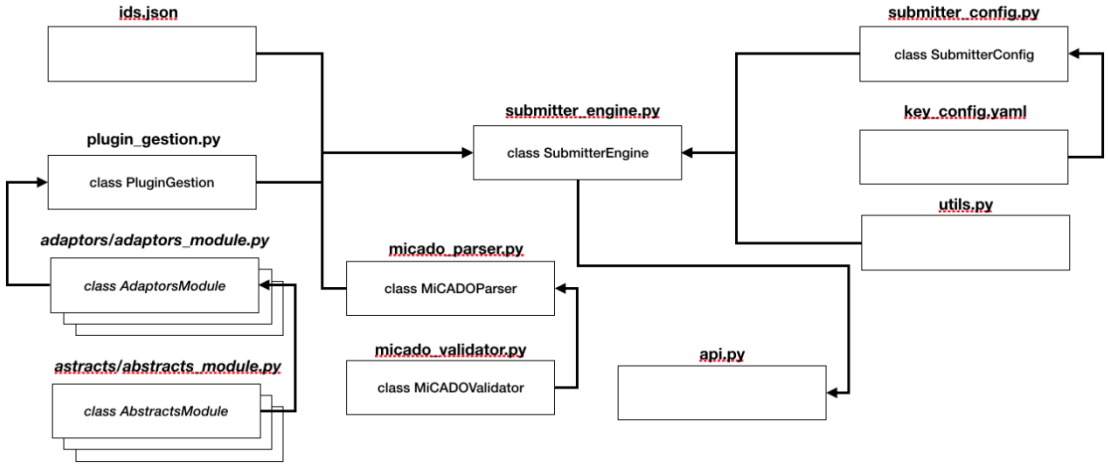

Figure 19 MiCADO Submitter internal architecture

The current MiCADO Submitter is composed of the following components:

- *submitter_engine.py*
- *api.py*
- *micado_parser.py*
- *micado_validator.py*
- *plugin_gestion.py*
- *submitter_config.py*

- *utils.py*
- *abstracts/[abstracts_module].py*
- *adaptors/[adaptors_module].py*
- *system/key_config.yaml*
- *system/ids.json*

The next part of the document describe these modules in detail.

submitter_engine

This module is the main component of the MiCADO submitter and it instructs the different other components to launch the application described in the provided template. Only one class is present in this module (*SubmitterEngine*) which has three main methods: *launch*, *update* and *undeploy*.

The constructor *(__init__)* instantiates the class *SubmitterConfig* for the whole instance of the submitter. This configuration object contains multiple dictionaries for the different configurations, as it is explained under the *submitter_config* module (section 7.3.3.6). Based on the adaptors set in the configuration file, also creates a list with the different adaptors' class names that will be available across the instance by calling the *plugin_gestion* module.

The *launch* method can take up to three parameters: *path_to_file,* which is required and is the path of the template to launch, *parsed_params,* which is an optional key:value dictionary for filling inputs defined in the template, and *app_id* which allows for user-defined IDs. This method will first parse the input TOSCA template using the *micado_parser* module, and retrieve the *TOSCAParser* object template. The OpenStack *TOSCAParser* project being still in development, there are current issues that required workarounds. One of these issues relates to policies: when *get_input* is used for properties, the *TOSCAParser* does not resolve it as a *get_input* object, but leaves it as a string (which will look like *{ get_input: name_of_input }*). To fix this we have to manually replace this field with the value linked to the name of the input. This step is achieved after the *micado_parsing*. If no *app_id* has been provided then an id is generated. The adaptors are instantiated providing them with *app_id,* its configuration and the template, and a dictionary is created whose keys contain the name of the adaptor class, and whose values contain the adaptor object just instantiated. The json containing the ids of the application is updated by adding the new *app_id*. Next, each adaptor calls its translate method. The order in which the adaptors are calling the translate method is defined in the *config* file. In the following step the *execute* method of each adaptor is called. The order of these calls is also set in the *config* file. If there is any issue during the execution of the *execute* method, e.g. *AdaptorCritical* error is caught, and the *undeploy* method is called on the already executed adaptors in reverse order of the original *execute* method call.

If no errors are detected following the repeated execution of the *execute* method, there is a check to see if some outputs need to be saved (the outputs described in the TOSCA template). If outputs do exist, the *id.json* file is updated under *app_id* of the current adaptor, and the output is saved (for more explanation see on the following *id.json* part). The *launch* method returns the id of the application.

The *update* method takes up to three parameters: *app_id,* which is required and is the id link to the deployment of the application we want to update, *path_to_file,* which is also required and is the location where we can find the template, and optionally, *parsed_params,* which is the same kind of dictionary described in the *launch* method. Similarly to the *launch* method, the same first few steps are completed, meaning that 1) the TOSCA template is parsed, 2) the *TOSCAParser* object is returned, 3) *get_input* in the template is replaced by the actual value of the input, 4) the adaptors are instantiated. Next, for each adaptor following the order set in the *config*, the update method is called. If everything went well, the *id.json* file is updated as previously, and nothing is returned.

The *undeploy* method takes the required *app_id*, which is the id of the target application to be undeployed. The first step of this method is to instantiate the adaptors. This time the adaptors are instantiated with only the *app_id* and their own *config*. Then the *undeploy* function of each adaptor, in the order set in the *config*, are executed. Finally, the cleanup method of all the adaptors are invoked, following the order setup in the *config* file.

api

The *api* module provides an interface for the user to submit, monitor and update the application through TOSCA templates via the previously described *submitter_engine* methods (*launch, update, undeploy*).

micado_parser

This module is composed of one class called *MiCADOParser* which has only one method called *set_template*. The *set_template* method takes as input up to two parameters: a *path* which is going to be the path of the desired template; and an *optional* parameter called *parsed_params* which is a dictionary with the keys being the inputs that the user wants to modify, and the values being the values of the input. The first step in this method is to check whether the path provided is an absolute path or a url. Depending whether it is an url or an absolute path, the *TOSCAParser* object called *ToscaTemplate* is instantiated with the parameters *path*, *parsed_params* and *isfile*. *isfile* is a Boolean variable set to *True* if the path is absolute, and *False* if it is an url, basically indicating whether the file is local or remote. Next the MiCADO validator is instantiated, and the validation method is called on the *TOSCAParser* object template to check whether it is compliant with the MiCADO submitter.

micado_validator

This module checks whether the input template is compliant with the MiCADO submitter. There is some specificity related to the submitter component that needs to be there to be able to launch an application. The *micado_validator* checks the template that received from the *micado_parser*. If everything is compliant with the MiCADO submitter, then it returns no error, and the process of launching the application can continue.

plugin_gestion

This module has one class called *PluginsGestion* and it implements one main method called *get_plugins*. *get_plugins* takes one required parameter called *plugin_name*. First, all the plugins which are located under the adaptor's directory are going to be loaded. Second, the method looks for the plugin with the required class name. If the plugin is found, then the method returns the requested adaptor class.

submitter_config

The *submitter_config* module has one class called *SubmitterConfig*, and it implements two main methods: *get_lists_adaptors* and *mapping*.

The constructor (__init__) first reads the config file (*key_config.yaml)* located under the system directory, and splits it into different dictionaries (*main_config, step_config and adaptor_config*).

The *mapping* method alters the *adaptor_config* part to add into each adaptor the *main_config dryrun* parameter that will be needed by the adaptor. This acts as a mapper step. If a template is passed as a parameter while calling the *mapping* method, the *adaptor_config* will also be altered and it will create a sub-dictionary. If one of the types for the adaptor is present, the *nodetemplate* object or the *policytemplate* object will be set as the value, otherwise if no object with the same type has been found a None type will be returned as value. Figure 20 illustrates how the *adaptor_config* dictionary is structured.

```
{
   'DockerAdaptor':{
      'types':[
         {
            'tosca.nodes.MiCADO.Container.Application.Docker':<toscaparser.nodetemplate.NodeTemplate
object at 0x7f41c31ed080>
         }
      ],
      'dry_run':False,
      'endoint':'endpoint',
      'volume':'/var/lib/submitter/files/output_configs/'
   },
   ...
   'PkAdaptor':{
       ...
   }
}
```

Figure 20 Adaptor_config dictionary

The *get_list_adaptor* method returns a list of all adaptors required in the *key_config.yaml* file.

utils

The *utils* module includes three methods used across the submitter. An *id_generator* that will generate a random eight-character long id, a *get_yaml_data* which will return a yaml object that is read from an unput path, and a *dump_order_yaml* which will dump into the path wanted a yaml object ordered.

abstracts

The *abstracts* directory contains multiple modules inside, all of which are describing the requirements for other modules that derive from it (e.g. *base_adaptor* and *exceptions*).

key_config

The *key_config.yaml* file is used to configure the whole submitter, as it is illustrated in the example shown in Figure 21. Each adaptor can decide what configuration it requires under its respective section in the *key_config* file. The *step* section defines for each action the order of adaptors to follow. The *config* section is the main configuration of the submitter, which describes the path of the log file and whether or not the adaptors should run in *dryrun* mode for development purposes.

```
config:
  dry_run: True
  path_log: "submitter.log"

step:
  translate:
    - DockerAdaptor
    - PkAdaptor
    - OccopusAdaptor
  execute:
    - DockerAdaptor
    - PkAdaptor
    - OccopusAdaptor
  update:
    - DockerAdaptor
    - PkAdaptor
    - OccopusAdaptor
  undeploy:
    - DockerAdaptor
    - PkAdaptor
    - OccopusAdaptor
  cleanup:
    - DockerAdaptor
    - PkAdaptor
    - OccopusAdaptor

adaptor_config:
  DockerAdaptor:
    types:
      - "tosca.nodes.MiCADO.Container.Application.Docker"
    endpoint: "endpoint"
    volume: "/var/lib/submitter/files/output_configs/"

  PkAdaptor:
    types:
      - "tosca.policies.Scaling.MiCADO"
    endpoint: "policykeeper:12345"
    volume: “/var/lib/submitter/files/output_configs/"

  OccopusAdaptor:
    types:
      - "tosca.nodes.MiCADO.Occopus.*"
    endoint: "endpoint"
    volume: "/var/lib/submitter/files/output_configs/"
```

Figure 21 Structure of the *key_config.yaml* file

ids

The *ids.json* file, generated/updated by the *SubmitterEngine*, is shown in Figure 22. This file is a dictionary that has been created by the *SubmitterEngine*. It includes the id of the application with a sub-dictionary that describes the possible outputs requested in the TOSCA template. In the example of Figure 22, *E1CVCI4Q* is the id of the application, and *ip_address* and *port* are the results of the output requested by the TOSCA template.


```
{
  "E1CVCI4Q":{
    "ip_address":{
      "service":"E1CVCI4Q_Inycom",
      "result":"120.12.12.12"
    },
    "port":{
      "service":"E1CVCI4Q_Inycom",
      "result":"120"
    }
  }
}
```


Figure 22 example of a ids.json created

Here **E1CVCI4Q** is the id of the application, and **ip_address** and **port** are the results of the output requested by the TOSCA template.

10.3.4 Implementing the Adaptors

As it was explained previously, the adaptors are in charge of communicating with the underlying service. Based on the relevant part of the TOSCA template, they transform/translate the input into the desired format that the targeted service can understand. To achieve this, each adaptor has to implement the abstract *base_adaptor* class. There are five main methods in this class: *translate* which takes care of translating the input template, *execute* which launches the execution of the service with the required information, *update* which is used when the template needs to be updated, *undeploy* which brings down the instance of the service for this particular application, and *cleanup* which is a method used to do some cleaning after undeployment. The constructor takes three parameters: the id of the application, the *config* that is a dictionary defined by the adaptor developer in the *key_config.yaml*, and an optional parameter being a TOSCA Template object.

10.3.4.1 Docker/Swarm Adaptor

The Docker/Swarm Adaptor is implementing the abstract class *base_adaptor*. The *__init__* method stores file paths and the unique ID, and creates placeholder variables for the data and eventual output.

When the ***translate*** method is invoked, it reads the template object and creates a new compose-formatted dictionary containing all the Docker-relevant TOSCA types and their properties. Once this is done, the dictionary will be dumped into the YAML format, and will

be readable as a Docker compose file. The file is named using the application ID and is saved to a location specified in the *key_config.yaml* file.

The ***execute*** method uses Python's *subprocess* module to run the Docker client's *docker stack deploy* command, pointed at the Docker-Compose file which was created during the translate step. If output has been requested in the TOSCA template, the *get_output* method will use *subprocess,* the Docker client's *inspect* command and store the requested data for the user.

The ***update*** method calls *translate* to create a temporary Docker compose file based on the newly provided TOSCA template. The new temporary compose file is compared to the original, and if there is any difference, the temporary compose file replaces the original, and *execute* is called again which forces an update on the running services. If there is no difference between the two files, then the temporary Docker compose file will be discarded.

The ***undeploy*** method uses *subprocess* to call the Docker client's *docker stack down* command to remove the service from Swarm.

The ***cleanup*** method will then remove the Docker compose file.

10.3.4.2 Occopus Adaptor

The Occopus adaptor is implemented from the base_adaptor class. The __init__ method defines the required variables that are necessary for processing. The adaptor is currently capable of handling four types of cloud interfaces, such as: CloudBroker, CloudSigma, EC2 and Nova. The adaptor uses Docker SDK 3.3.0 to execute the Occopus container and run the *occopus-import* and *occopus-build* commands. These calls will be replaced with REST API calls in the future when the functionality will be ready in Occopus. The other operations such as maintain, attach and undeploy are called via the Occopus REST API.

In the **translate** method the adaptor receives the application to be deployed in the TOSCA-based description, and also Occopus related information which defines the cloud resource and node information for Occopus that are required for the virtual machine orchestration. The adaptor translates the TOSCA file and creates 1) an infra definition and 2) a node definition file. These files are stored under a preconfigured storage volume specified in the key_konfig argument of the translate method. For both files, the names are generated to contain application ID to ease debugging.

In the **execute** method the adaptor executes the occopus-import command inside the Occopus container to import the node definition file through CLI. After the import succeeded, the adaptor executes the occopus-build command (with the infrastructure description as argument) inside the Occopus container to start the building process of the MiCADO Worker infrastructure based on virtual machines. Occopus performs the MiCADO Worker infrastructure deployment and when it is done, the adaptor finally instructs Occopus through its REST API to start the maintenance of the infrastructure.

In the **undeploy** method the adaptor receives the APPLICATION ID from the submitter engine. Based on this information the adaptor sends an infrastructure destroy request to Occopus through its REST API. Finally, Occopus performs a graceful removal of the all the

virtual machines (including preallocated temporary cloud storage volumes) implementing a MiCADO Worker.

In the **update** method first the adaptor generates the Occopus infrastructure and node definition files as described in the translate method. After the comparison of the newly generated files with the ones actually running under MiCADO the operation of the adaptor differs. If there is change only in the infrastructure definition, the adaptor starts the execute phase again. As a result, Occopus will update the stored infra definition without shutting down any virtual machines. In case the node definition file changes, the adaptor destroys and redeploys the virtual machines since the updates can only be ensured with these steps. The adaptor in this situation executes its own undeploy and start methods (defined above) to perform the changes.

During the **cleanup** phase the adaptor gets the APPLICATION ID from the Submitter engine and removes the associated infrastructure and node definition files generated by the translate method.

10.3.4.3 Policy Keeper Adaptor

The Policy Keeper Adaptor is implementing the abstract class base_adaptor. The __init__ method sets all the variables that will be available across the instance of the adaptor.

The **translate** method reads the TOSCA template and parses the policies section into a dictionary. It then dumps the dictionary into a YAML file that is named after the application id, and the file path is set in the *key_config.yaml* file. The created YAML file is compatible with the Policy Keeper.

The **execution** method invokes the start REST API call of the Policy Keeper to start the realization of the scaling policy passed as arguments. The argument is the scaling policy description in YAML translated by the translate method defined above.

The **undeploy** method invokes the stop REST API call of the Policy Keeper to finish the realization the scaling policy passed previously by the start method.

The **update** method calls the translate method and creates a temporary policy YAML file. Next, it compares the original and the temporary YAML files. If the two files are the same, the temporary YAML file will be deleted. If there is any difference between the two YAML files then the original policy YAML file is deleted and the undeploy and execute methods are invoked to update the scaling policy to be realized by the Policy Keeper.

The **cleanup** method removes the created YAML file from the defined location.

10.3.5 REST API definition

- To **launch an application from a url** you can use one of the following curl commands:
  - `curl -d input="[url to TOSCA Template]" -X POST http://[IP]:[Port]/v1.0/app/launch/url/`
  - `curl -d input="[url to TOSCA Template]" -d id=[ID] -X POST http://[IP]:[Port]/v1.0/app/launch/url/`

- `curl -d input="[url to TOSCA Template]" -d params='{"Input1": "value a", "Input2": "value b"}' -X POST http://[IP]:[Port]/v1.0/app/launch/url/`
- `curl -d input="[url to TOSCA Template]" -d id=[SOMEID] -d params='{"Input1": "value a", "Input2": "value b"}' -X POST http://[IP]:[Port]/v1.0/app/launch/url/`

- To **launch an application from a file** that you pass to the api you can use one of the following curl commands:
  - `curl -F file=@[Path to the File] -X POST http://[IP]:[Port]/v1.0/app/launch/file/`
  - `curl -F file=@[Path to the File] -F params='{"Input1": "value a", "Input2": "value b"}' -X POST http://[IP]:[Port]/v1.0/app/launch/file/`
  - `curl -F file=@[Path to the File] -F id=[SOMEID] -F params='{"Input1": "value a", "Input2": "value b"}' -X POST http://[IP]:[Port]/v1.0/app/launch/file/`
  - `curl -F file=@[Path to the File] -F id=[SOMEID] -X POST http://[IP]:[Port]/v1.0/app/launch/file/`
- To **update from a url** a wanted application you can use one of the following curl commands:
  - `curl -d input="[url to TOSCA template]" -d params='{"Input1": "value a", "Input2": "value b"}' -X PUT http://[IP]:[Port]/v1.0/app/udpate/file/[ID_APP]`
  - `curl -d input="[url to TOSCA template]" -X PUT http://[IP]:[Port]/v1.0/app/udpate/file/[ID_APP]`
- To **update from a file** a wanted application you can use one of the following curl commands:
  - `curl -F file=@"[Path to the file]" -d params='{"Input1": "value a", "Input2": "value b"}' -X PUT http://[IP]:[Port]/v1.0/app/udpate/file/[ID_APP]`
  - `curl -F file=@"[Path to the file]" -X PUT http://[IP]:[Port]/v1.0/app/udpate/file/[ID_APP]`
- To **undeploy** a wanted application you need to feed it the id:
  - `curl -X DELETE http://[IP]:[Port]/v1.0/app/undeploy/[ID_APP]`
- To **get the ids** of the application deployed and its information related:
  - `curl -X GET http://[IP]:[Port]/v1.0/list_app/`
- To **get the information** for an application:
  - `curl -X GET http://[IP]:[Port]/v1.0/app/[ID_APP]`

10.4 Dashboard

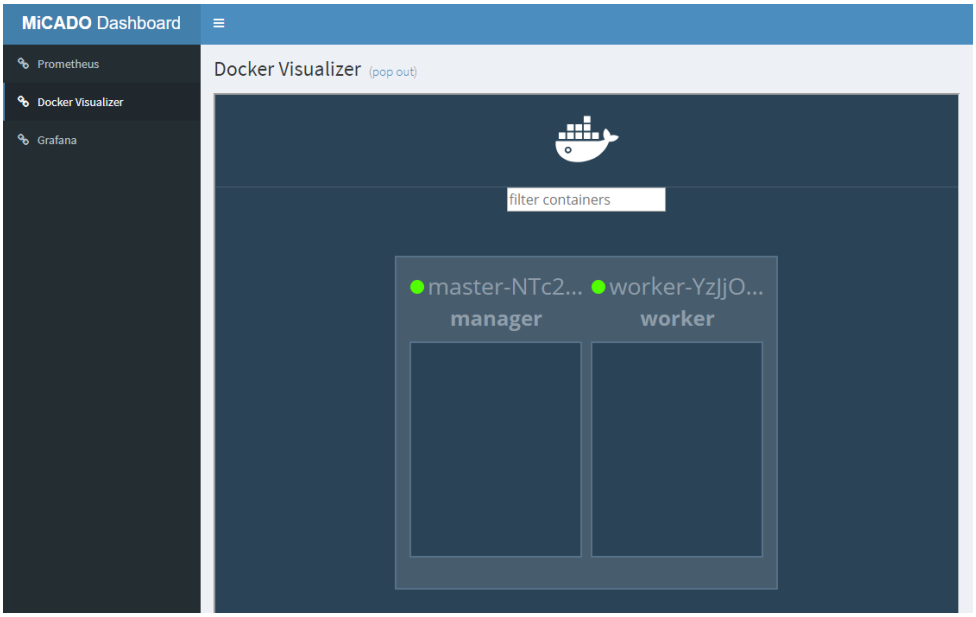

Figure 23 MiCADO Dashboard showing Docker Visualizer

MiCADO Dashboard collects web-based user interfaces into a single coherent Dashboard. It is a Python Flask [17] based application that uses a modified Bootstrap [18] based administrative template called AdminLTE v2 [19]. Bootstrap is an open source web frontend library for developing with CSS, JavaScript and HTML. It was originally developed at Twitter (and originally named Twitter Blueprint). AdminLTE is an open source admin dashboard and control panel theme based on Bootstrap v3. It provides a range of common reusable and responsible components.

The Dashboard integrates all interfaces via IFrames and custom CSS that provides responsive behavior (i.e., the IFrame resizes with the window). Additionally, all user interfaces are also available via pop out windows for convenience. New user interfaces can be easily configured and added later to the Dashboard.

Currently, the Dashboard contains (see Figure 23) a title bar and a left-side menubar containing three menu items. The three menu items are covering the web interface of three components to visualize the internal status of MiCADO. They are

- Docker Visualizer
- Prometheus
- Grafana

The MiCADO Dashboard is integrated via the Ansible playbook with the rest of the MiCADO infrastructure. It runs in a separate Docker container on the master node. Within the container it runs as a WSGI service using the Gunicorn Python WSGI HTTP Server [20].

10.4.1 Docker Visualizer

Docker Visualizer (is an open-source tool for visualizing the internal status of a Docker Swarm. The tool shows the Swarm Docker Nodes with vertically aligned rectangles with labels on top of them. Whenever a new MiCADO worker node is attached it immediately and automatically appears in Docker visualizer.

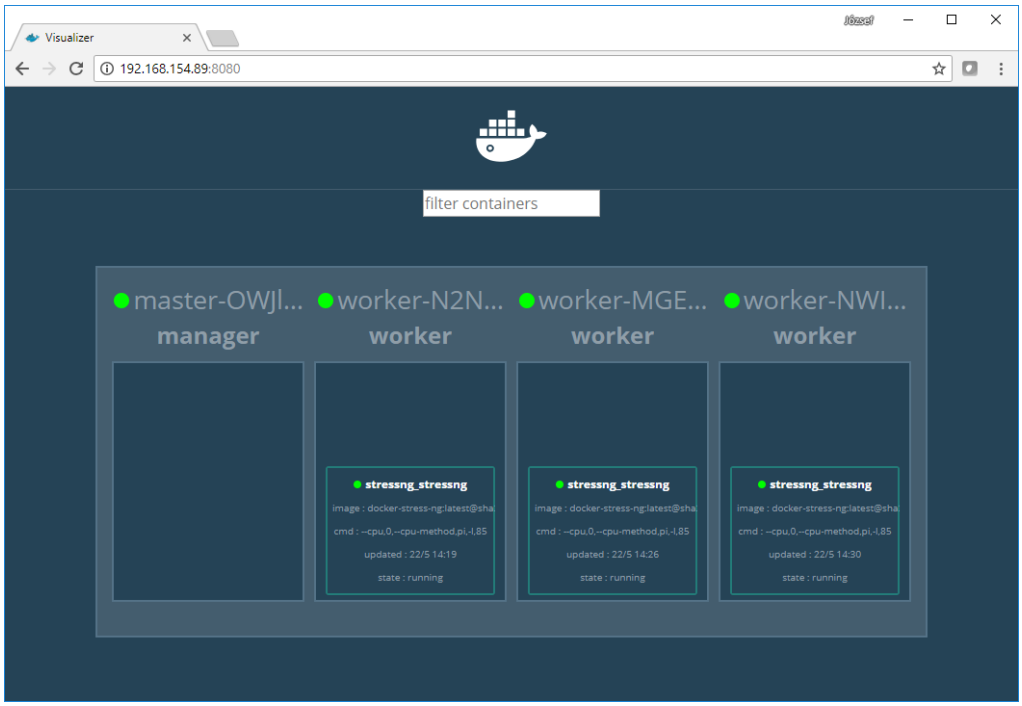

Figure 24 Docker Visualizer showing the overview of Docker Swam

Each rectangle representing a Docker node contains boxes representing the containers hosted by that particular Docker node. The Docker container box details the name, image, command and state of the container.

This tool has been selected to be part of the dashboard since it provides an easy overview of the docker nodes and containers executed by MiCADO.

10.4.2 Grafana

Grafana is an easy-to-use visualizer tool to display data in various formats like charts, diagrams, graphs, etc. Grafana has a configurable dashboard which has been customized under MiCADO (see Figure 25) to show various information, like

- Number of worker nodes;
- Number of containers;
- Time elapsed since MiCADO is deployed;
- Alerts configured under Prometheus.

Beyond several numbers listed above, a few time-based graphs have also been configured to continuously show resource usage related information in MiCADO:

- Average CPU usage of virtual machines (nodes) and containers;

- CPU usage per nodes and per containers;
- Memory usage per nodes and per containers;
- Network traffic on cluster;
- Network usage per container.

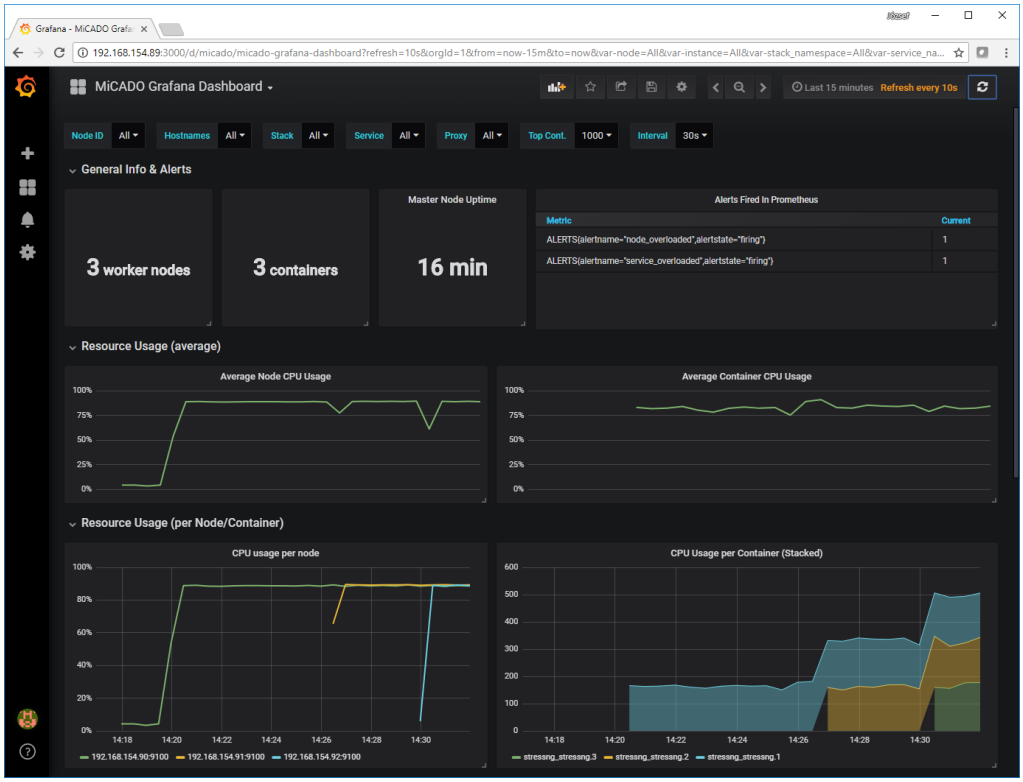

Figure 25 Grafana showing the resources usage diagrams under MiCADO

The dashboard is automatically deployed under Grafana at the deployment time of MiCADO master.

This tool has been selected to be part of the dashboard since it provides configurable and flexible graphical overview of any metrics e.g. resource usage in time under MiCADO.

10.4.3 Prometheus

The Prometheus monitoring system is part of MiCADO. Hence, making it accessible from the dashboard is natural. Prometheus has a simple, but useful web interface where any of the user defined metrics can be visualized.

One of its useful features is that the details of the configured alerts can be visualized (see Figure 26). Beyond simply inspecting the alerts, the user of MiCADO can control whether all alerts have been successfully configured by the Policy Keeper.

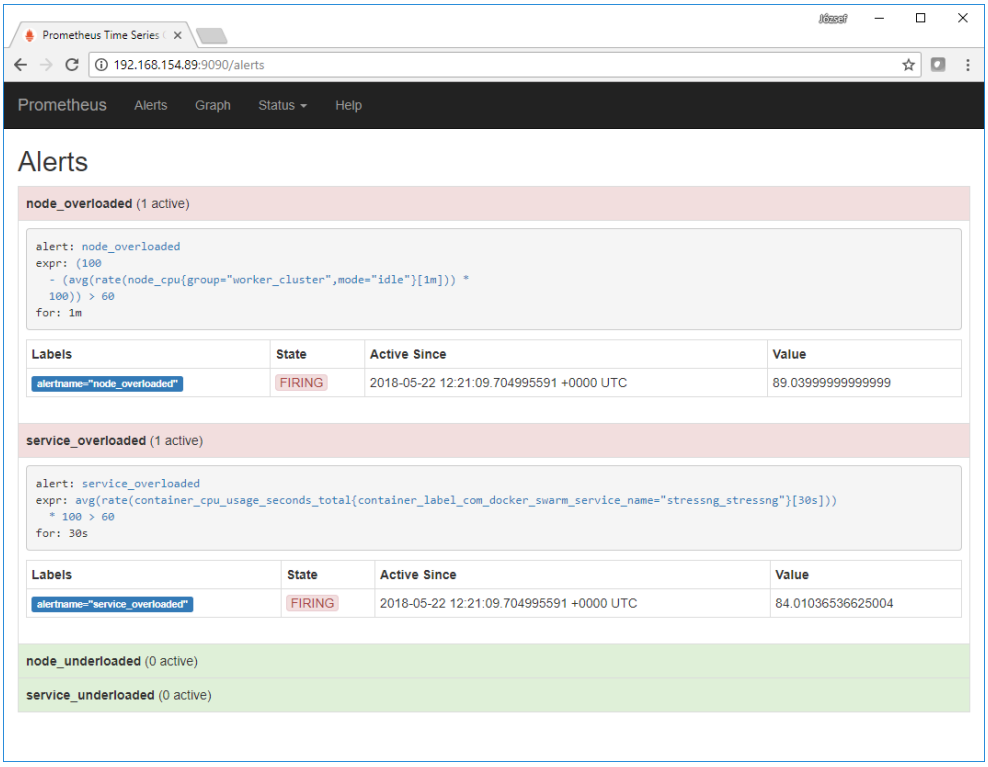

Figure 26 Prometheus showing the state of alerts under MiCADO

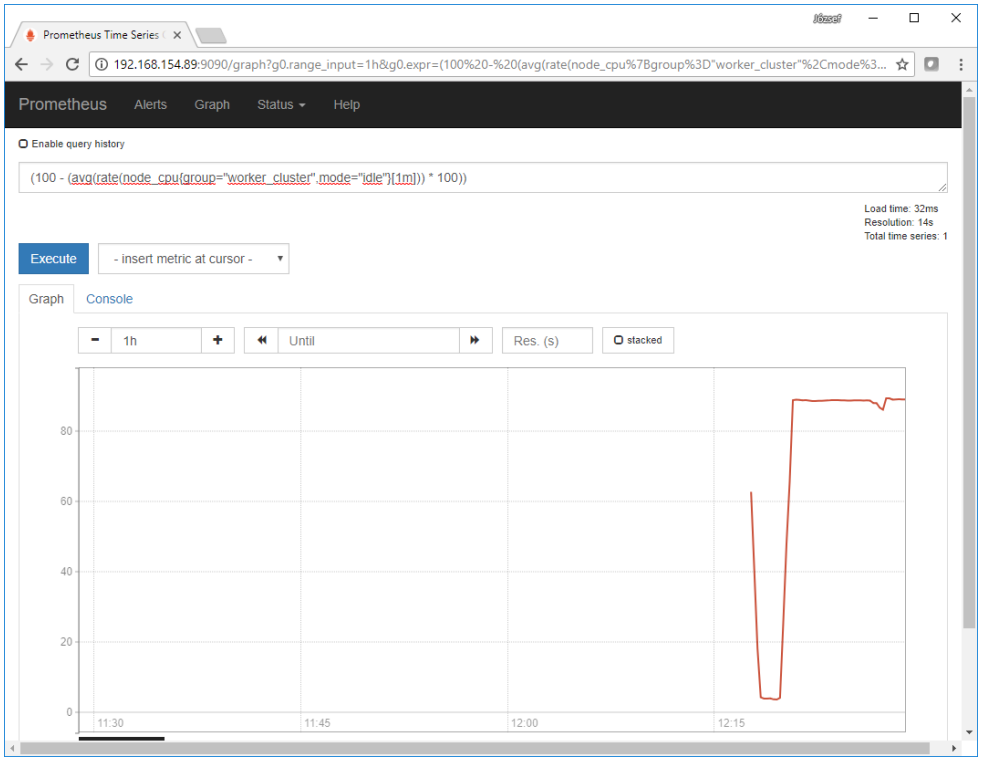

Figure 27 Prometheus showing the value of a metric in time under MiCADO

Another important and useful feature of Prometheus is to inspect the value of any metrics and its value in time (see Figure 27). The metric can be the one which has been configured under the scaling policy as a definition of variable or can be the value based on which an alert is configured.

Prometheus graphical user interface is made accessible primarily for the application developers due to the possibility to easily formalize a Prometheus query which then can be inserted in the scaling policy of a given application.

10.5 Availability

Ansible playbooks for deployment:
https://github.com/micado-scale/ansible-micado

Short user guide:
https://github.com/micado-scale/ansible-micado/blob/master/README.md

Source code of Policy Keeper component:
https://github.com/micado-scale/component-policy-keeper

Frame Docker container of Policy Keeper component:
https://hub.docker.com/r/micado/policykeeper/

Source code of Tosca Submitter component:
https://github.com/micado-scale/component_submitter

Docker container of Tosca Submitter component:
https://hub.docker.com/r/micado/toscasubmitter/

Documentation of Tosca Submitter:
https://rawgit.com/micado-scale/component_submitter/master/documentation/_build/html/index.html

MiCADO related TOSCA templates:
https://github.com/micado-scale/tosca

Source code of Dashboard component:
https://github.com/micado-scale/component-dashboard

Docker container of Dashboard component:
https://hub.docker.com/r/micado/dashboard/

11. Scaling examples in MiCADO V5

11.1 Consumption based policy

The consumption-based policy makes scaling decisions based on percent over time thresholds set against some hardware resources. For this example, the CPU load of containers and virtual machines are queried through Prometheus. The upper threshold for CPU load is set at 60% and the lower threshold is at 20%, with a minimum elapsed time for nodes set at 1minute and a minimum elapsed time for containers set at 30 seconds. When a Prometheus query fetching the CPU load of a container or node returns a value above the 60% threshold for longer than the minimum allowed time, an alert is generated for the container or node that is overloaded, and the infrastructure scales the container or node up by a count of one. Conversely, when Prometheus queries for CPU load return under the lower 20% threshold for the minimum allowed time, the infrastructure will scale the unused resource down by a count of one.

As a test application for the consumption-based policy, *stressng [21]* has been selected. The *stressng* tool is an extension of Linux *stress*, which is an intentional load generator for stressing hardware resources including CPU, memory and IO. The *stressng* extension offers more customization and consistency in load tests, which makes it ideal for the purposes of testing a policy based on CPU consumption. The tool is deployed inside a Docker container, and Docker Swarm settings are set such that a maximum of one instance of *stressng* is permitted to run on each virtual machine node. No minimum number of *stressng* containers is set.

Figure 28 MiCADO scale-up using a consumption-based policy on *stressng*

To create a test scenario for the consumption-based policy, a TOSCA template is written for the *stressng* application that describes the application itself (and the load it should produce), the cloud topology in which it is deployed, and the policies that govern its scaling decisions, including the upper and lower thresholds for scaling. This template is then submitted to the TOSCA Submitter and the appropriate data is translated by the configured adaptors before being passed to the end components for deployment. Occopus spins up a virtual machine with the resources described in TOSCA, and a container running *stressng* is launched on

Docker Swarm. The PolicyKeeper is given identification information of the virtual machine node and the application container so that it may build and enforce a consumption-based policy that uses the necessary Prometheus queries to target this particular deployment.

Upon successful deployment, *stressng* will begin to load the CPU to the specified percentage, 85% in the test described herein. After 30 seconds, a first alert, indicating an overloaded container, will fire. As there is no virtual machine node for Docker to use, nothing will happen until after one minute, when the second alert, indicating an overloaded virtual machine node, fires. When the overloaded node alert fires, a call from the PolicyKeeper to Occopus will scale up and provision a new virtual machine, which will take a short time to spin up and join Docker Swarm. Once it has, the earlier alert can be satisfied and the PolicyKeeper will call Docker to scale up the number of containers so that one exists on each of the virtual machines.

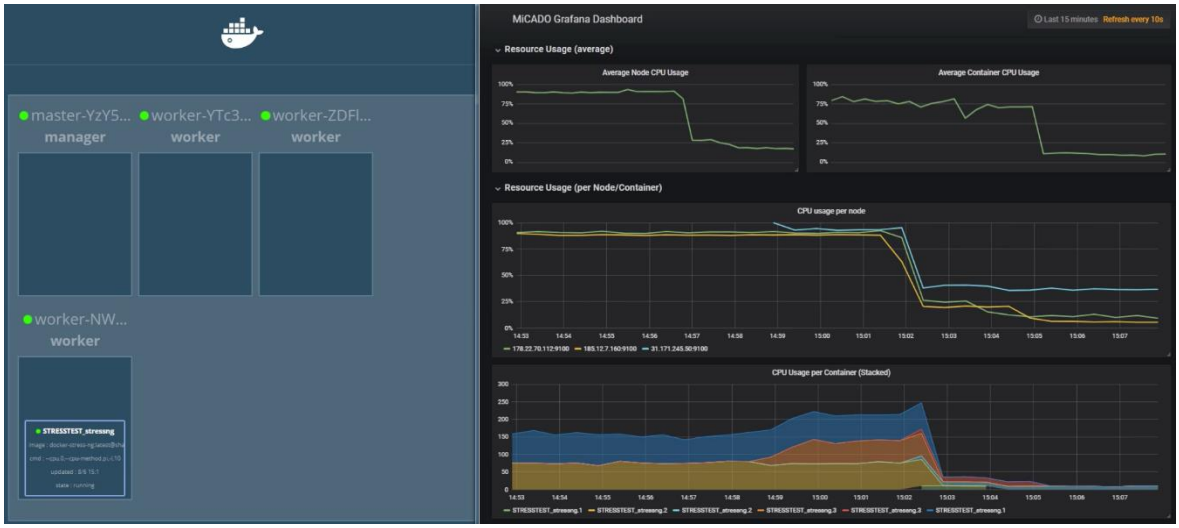

Figure 29 MiCADO scale-down using a consumption-based policy with *stressng*

The scale-up response will continue until the maximum number of instances as set in the TOSCA template is reached, or until the *stressng* service is manually stopped and removed. The entire scale-up response can be visualised in Figure 28. In order to force a scale-down response, testing continues and a modified TOSCA template is submitted to the TOSCA Submitter as an update. The only modification that the new template carries is a reduction in the load generated by *stressng* down from the earlier 85% to a very low 5%. As soon as the update is applied, Prometheus queries will return a figure under the lower threshold, and underloaded alerts will be generated. After 30 seconds the container alert will be generated, and the PolicyKeeper will instruct Docker to scale down containers by a count of one. After 1 minute the virtual machine underloaded alert will be generated, and Occopus will be called to tear down a node. This will continue until the minimum number of instances as set in the TOSCA template is reached, or until a greater CPU load is imposed on the infrastructure. After scaling down, the various parts of the deployment are brought down together with an API call to the TOSCA Submitter. The scale-down response can be seen in Figure 29.

Snippet 1 shows the complete TOSCA template which realizes the above deployment of *stressng*. The template describes three important sections – the application in container to run, the cloud and cloud resources to provision, and the consumption-based policy for scaling.

The first entry in the **node_templates** key defines the *stressng* container as being of the **type** *tosca.nodes.MiCADO.Container.Application.Docker*. **Properties** set the application's command at runtime, and set a restriction so a maximum of one container exists on each virtual machine. The **artifacts** section describes the Docker image to use, and the repository where it can be found.

Next, a *worker_node* virtual machine is defined with the TOSCA type *tosca.nodes.MiCADO.Occopus.CloudSigma.Compute*. **Properties** define endpoints and cloud names, and the **capabilities** define the hardware resources, disk images and security details of the virtual machine to be launched.

Lastly, the policies section defines two **scalability** policies of type *tosca.policies.Scaling.MiCADO*, one which targets the *worker_node* virtual machine, and the other for the *stressng* container. **Constants** define resource thresholds at which to scale up or down, and for containers, the name of the application stack. Named **alerts** are described using Prometheus queries, and specify a length of time after which the alert should be fired. The maximum and minimum number of instances to scale up and down to are also defined here in the **properties** section. The last property is the **scaling_rule**, which carries the Python code to perform the act of scaling based on the targets, constants and alerts defined just above.

```
imports:
  - https://raw.githubusercontent.com/micado-scale/tosca/master/micado_types.yaml
repositories:
  docker_hub: https://hub.docker.com/
topology_template:
  node_templates:
    stressng:
      type: tosca.nodes.MiCADO.Container.Application.Docker
      properties:
         command: "--cpu 0 --cpu-method pi -l 85"
         deploy:
           resources:
             reservations:
               cpus: '1.0'
      artifacts:
       image:
         type: tosca.artifacts.Deployment.Image.Container.Docker
         file: lorel/docker-stress-ng
         repository: docker_hub
    worker_node:
      type: tosca.nodes.MiCADO.Occopus.CloudSigma.Compute
      properties:
        cloud:
          interface_cloud: cloudsigma
```

```
          endpoint_cloud: https://zrh.cloudsigma.com/api/2.0
      capabilities:
        host:
          properties:
            num_cpus: 2000
            mem_size: 1073741824
            vnc_password: secret
            libdrive_id: 87ce928e-e0bc-4cab-9502-514e523783e3
            public_key_id: ADD_YOUR_ID_HERE
            firewall_policy: ADD_YOUR_ID_HERE
  policies:
    - scalability:
        type: tosca.policies.Scaling.MiCADO
        targets: [ worker_node ]
        properties:
          constants:
            NODE_TH_MAX: '60'
            NODE_TH_MIN: '20'
          alerts:
          - alert: node_overloaded
            expr: '(100-(avg(rate(node_cpu{group="worker_cluster",mode="idle"}[60s]))*100)) >
{{NODE_TH_MAX}}'
            for: 1m
          - alert: node_underloaded
            expr: '(100-(avg(rate(node_cpu{group="worker_cluster",mode="idle"}[60s]))*100)) <
{{NODE_TH_MIN}}'
            for: 1m
          min_instances: 1
          max_instances: 3
          scaling_rule: |
            if len(m_nodes) <= m_node_count and m_time_since_node_count_changed > 60:
              if node_overloaded:
                m_node_count+=1
              if node_underloaded:
                m_node_count-=1
            else:
              print('Transient phase, skipping update of nodes...')
    - scalability:
        type: tosca.policies.Scaling.MiCADO
        targets: [ stressng ]
        properties:
          constants:
```

```
            SERVICE_NAME: 'stressng'
            SERVICE_FULL_NAME: '{{stack}}_stressng'
            SERVICE_TH_MAX: '60'
            SERVICE_TH_MIN: '20'
          alerts:
          - alert: service_overloaded
            expr:
'avg(rate(container_cpu_usage_seconds_total{container_label_com_docker_swarm_service_name="{{SERVICE_F
ULL_NAME}}"}[30s]))*100 > {{SERVICE_TH_MAX}}'
            for: 30s
          - alert: service_underloaded
            expr:
'avg(rate(container_cpu_usage_seconds_total{container_label_com_docker_swarm_service_name="{{SERVICE_F
ULL_NAME}}"}[30s]))*100 < {{SERVICE_TH_MIN}}'
            for: 30s
          min_instances: 1
          max_instances: 3
          scaling_rule: |
            if len(m_nodes) == m_node_count:
              if service_overloaded and m_node_count > m_container_count:
                m_container_count+=1
              if service_underloaded:
                m_container_count-=1
            else:
              print('Transient phase, skipping update of containers...')
```

Snippet 1. TOSCA Template for stressng deployment with consumption based policies

11.2 Deadline based policy

In order to demonstrate the flexibility of the policy keeper, a deadline-based policy has been compiled. For the needs of the demonstration, we used CQueue [22], a container execution tool containing a master and a worker. The Master node implements a queue, where each item (called task in CQueue) represents the specification of a container execution (image, command, arguments, etc.). The Worker node fetch the tasks one after the other and execute the container specified by the task.

With this lightweight container queueing system we implemented a simple deadline based policy. In our example, the container to be executed stores Autodock Vina [23], a popular molecular docking simulation application. One container execution equals one job execution. The list of jobs has been inserted into the queue of CQueue.

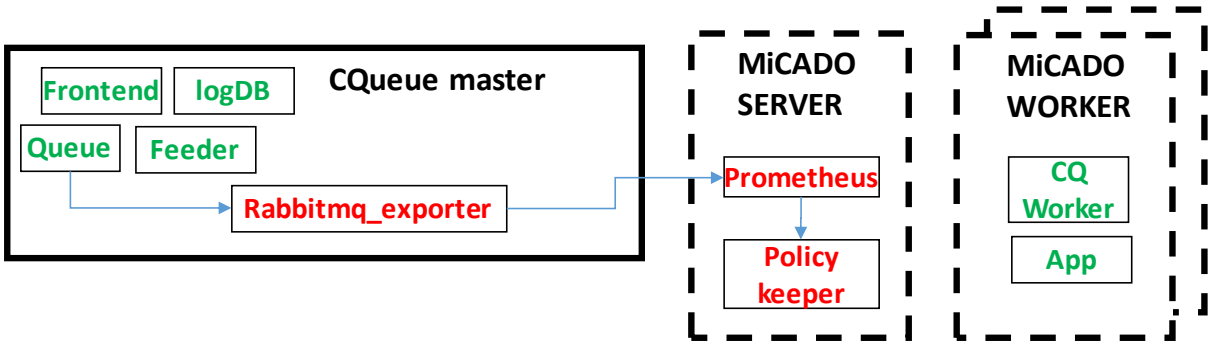

Figure 30 Architecture for deadline-based policy with CQueue in MiCADO V5

Figure 30 shows the high-level architecture of the demonstrated scenario. On the left hand side, a separate VM executes the Master node of CQueue, which implements a queue based on RabbitMQ to store the items representing the container executions. The CQWorker is submitted into MiCADO to be scaled up/down in order to reach a set deadline.

A scaling a policy has been defined which requires three parameters: deadline, actual number of items, and average execution time of a container (job). Average execution time and deadline are considered as fixed parameters in this demonstration. However, the number of items in the queue must be continuously monitored.

To monitor the number of items in the queue, a RabbitMQ exporter has been attached to RabbitMQ on the CQueue master node. The policy can continuously monitor the number of items with Prometheus utilizing the exporter.

The demonstration is executed as follows: 200 molecule docking simulation jobs have been submitted, with an average of 40 seconds execution time and with a deadline of 20 minutes from the time of submission. The policy has been implemented in a way that each node executes maximum two simulations (containers) in parallel.

After the submission, when the initial calculation happened, MiCADO started to scale the worker nodes up to four. After approximately one minute, there are already two nodes available, and therefore scaling still continues (see the log in Figure 31). After four minutes all four nodes are available. However, in order to meet the deadline, seven containers are necessary. The number of nodes and containers started to scale down after 15 minutes, and by 18 minutes all simulation jobs have been finished.

```
Number of required nodes: 4.0          =>after 1 minute
Number of requested nodes: 4
Length of queue: 196.0
Number of required containers: 7.0
Number of requested containers: 4
...
Number of required nodes: 4.0          =>after 4 minutes
Number of requested nodes: 4
Length of queue: 170.0
Number of required containers: 7.0
Number of requested containers: 7.0
...
Number of required nodes: 2.0          =>after 15 minutes
Number of requested nodes: 3
Length of queue: 30.0
Number of required containers: 3.0
Number of requested containers: 3.0
...
Number of required nodes: 0            =>after 18 minutes
Number of requested nodes: 0
Length of queue: 0.0
Number of required containers: 0
Number of requested containers: 0
```

Figure 31 Log messages (extraction) during deadline-based execution

On the MiCADO Dashboard, the number of nodes and containers are continuously shown. See Figure 32 for a screenshot showing the peak performance (4 nodes, 7 containers) during running the simulations.

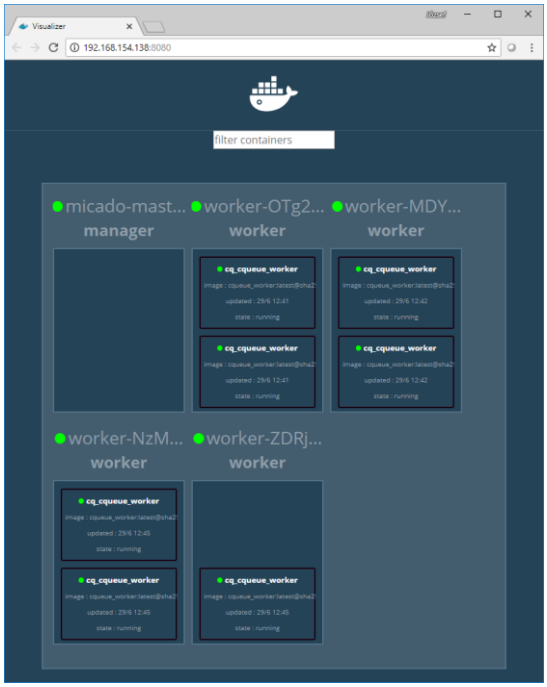

Figure 32 Nodes and container at peak during deadline-based execution

As the deadline is approaching, the simulations are continuously processed by the containers. To get the number of simulations remaining, the following Prometheus expression is used in the policy:

`"rabbitmq_queue_messages_persistent{queue="machinery_tasks"}"`

To inspect the number of simulation jobs remaining in a graphical way, MiCADO Dashboard can be used by simply specifying this Prometheus expression as an input on the query page of Prometheus under the 'Prometheus' menu. Figure 33 shows how the number of simulations remaining reaches zero by the deadline.

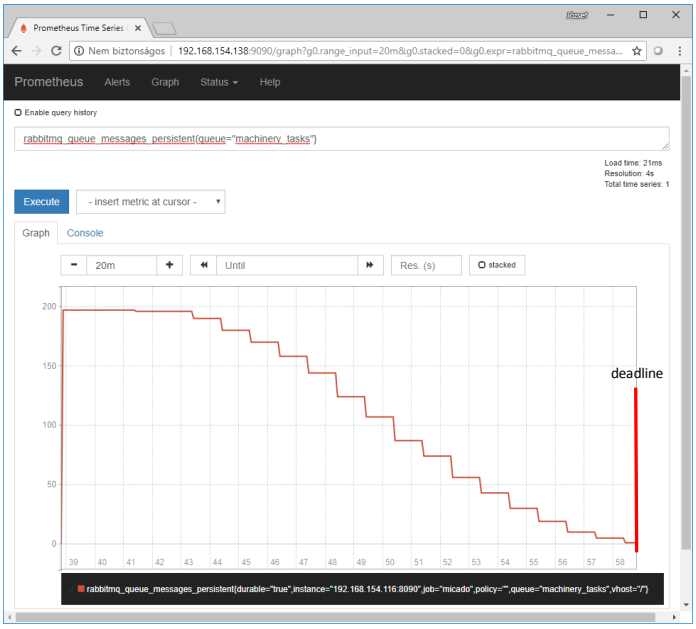

Figure 33 Number of jobs in time during deadline-based execution

To see the resource usage, Grafana (part of MiCADO Dashboard) can be used (Figure 34).

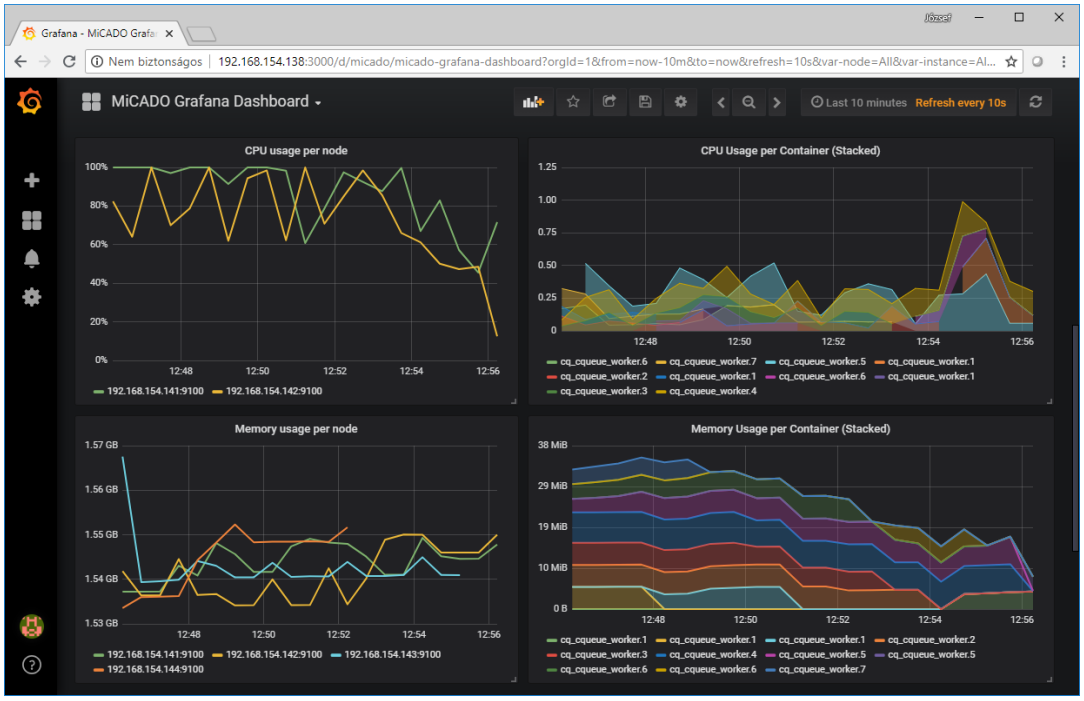

Figure 34 Resource usage during deadline-based execution

Grafana under MiCADO is configured to show the CPU, memory and network usage both for the virtual machines (left side) and for the containers (right side). Currently, Figure 34 shows the CPU (upper row) and Memory usage (lower row) per nodes and per containers.

To present all the details of the deadline-based policy implemented for the demonstration, the complete TOSCA description is also presented for full understanding. The TOSCA description has been cut into two parts. The first part (see Figure 35) contains the infrastructure specification, while the second part (see Figure 36) contains the scaling policy.

```
tosca_definitions_version: tosca_simple_yaml_1_0

imports:
  - https://raw.githubusercontent.com/micado-scale/tosca/master/micado_types.yaml

repositories:
  docker_hub: https://hub.docker.com/

topology_template:
  node_templates:
    cqueue_worker:
      type: tosca.nodes.MiCADO.Container.Application.Docker
      properties:
        environment:
          BROKER: amqp://guest:guest@192.168.154.116:5672
          RESULT_BACKEND: redis://192.168.154.116:6379
        deploy:
          restart_policy:
            condition: any
        volumes:
          - /usr/bin/docker:/usr/bin/docker
          - /var/run/docker.sock:/var/run/docker.sock
      artifacts:
       image:
         type: tosca.artifacts.Deployment.Image.Container.Docker
         file: sanyi86/cqueue_worker
         repository: docker_hub

    worker_node:
      type: tosca.nodes.MiCADO.Occopus.EC2.Compute
      properties:
        cloud:
          interface_cloud: ec2
          endpoint_cloud: https://opennebula.lpds.sztaki.hu:4567
      capabilities:
        host:
          properties:
            region_name: ROOT
            image_id: ami-00000371
            instance_type: m1.small
```

Figure 35 Deadline-based policy with CQueue under MiCADO V5, Part 1/2

The infrastructure description (in Figure 35) shows that the MiCADO worker nodes were launched on the SZTAKI Opennebula cloud. The CQueue worker container is defined under the cqueue_worker section with all the environment variables necessary for CQueue worker to build up connection to the CQueue master.
The scaling policy (in Figure 36) shows the main sections (sources, constants, queries) for both nodes and containers. Under sources, the RabbitMQ exporter location is defined. The constants section contains average execution time (AET) and deadline (DEADLINE) as most important parameters. The queries section specifies the time remaining (REMAININGTIME) and number of simulation job remaining (ITEMS) to be monitored. Finally, the scaling rules

for nodes and containers specify how many instances need to be launched based on the values of parameters specified under the constant and queries sections.

```
…
  policies:
    - scalability:
        type: tosca.policies.Scaling.MiCADO
        targets: [ worker_node ]
        properties:
          sources:
          - '192.168.154.116:8090'
          constants:
            AET: 40
            DEADLINE: 1530270216
            MAXNODES: 5
            MAXCONTAINERS: 10
          queries:
            REMAININGTIME: '{{DEADLINE}}-time()'
            ITEMS: 'rabbitmq_queue_messages_persistent{queue="machinery_tasks"}'
          min_instances: 1
          max_instances: '{{MAXNODES}}'
          scaling_rule: |
            reqnodes=0
            if ITEMS>0:
              reqconts = ceil(AET/(REMAININGTIME/ITEMS)) if REMAININGTIME>0 else
MAXCONTAINERS
              reqnodes = ceil(reqconts/2)
              if reqnodes<m_node_count-1:
                m_node_count-=1
              if reqnodes>m_node_count:
                m_node_count+=1
            else:
              m_node_count = 0
            print "Number of required nodes:",reqnodes
            print "Number of requested nodes:",m_node_count
    - scalability:
        type: tosca.policies.Scaling.MiCADO
        targets: [ cqueue_worker ]
        properties:
          sources:
          - '192.168.154.116:8090'
          queries:
            REMAININGTIME: '{{DEADLINE}}-time()'
            ITEMS: 'rabbitmq_queue_messages_persistent{queue="machinery_tasks"}'
          min_instances: 1
          max_instances: '{{MAXCONTAINERS}}'
          scaling_rule: |
            print "Length of queue:",ITEMS
            required_count = 0
            if ITEMS>0:
              required_count = ceil(AET/(REMAININGTIME/ITEMS)) if REMAININGTIME>0 else
MAXCONTAINERS
              m_container_count = min(required_count, len(m_nodes) * 2)
            else:
              m_container_count = 0
            print "Number of required containers:",required_count
            print "Number of requested containers:",m_container_count
```

Figure 36 Deadline-based policy with CQueue under MiCADO V5, Part 2/2

12. Current status and conclusion

In this deliverable we introduced the design and implementation of the Scalability Decision Maker component. A new component, called Policy Keeper realizes the scaling policies where the main design principle was to provide a flexible solution where both monitoring the parameters and specification of the scaling rules can be defined by MiCADO users. For monitoring parameters, on demand extension of monitoring data sources has been introduced by utilizing dynamically configurable Prometheus exporters. For flexible scaling rule definition, a scripting language, Python has been selected to formalize the scaling algorithm needed for the submitted application. Policy Keeper provides scaling decision functionality based on user's input both on virtual machine and container level.

Moreover, the deliverable introduced three versions of MiCADO that were developed during the reporting period. MiCADO V3.1 is an enhanced version of MiCADO V3 developed previously and reported in COLA deliverable D6.2. MiCADO V4 is developed to provide a flexible, easy-to-use job execution framework. MiCADO V5 aims to be a general scaling framework integrating the Policy Keeper to provide autoscaling functionality to various types of application. For each version, availability of the source code and documentation have been detailed.

MiCADO v5 has also been improved in various aspects such as deployment, visualization and TOSCA compliance, when compared to MiCADO V3. Ansible deployment and Dashboard has been introduced in MiCADO v5 to make it more user friendly. TOSCA compliance has been developed by integrating the new submitter component. TOSCA is supported at the level of infrastructure definition and at the level of scaling policy description in MiCADO v5.

To show the applicability of the Policy Keeper and Submitter in MiCADO v5, detailed description of two different scenarios (consumption-based and deadline-based) has been elaborated and demonstrated.

The next step in the MiCADO framework development is the design and development of the optimizer component to target cost-based optimization in relation to scaling which will be reported in deliverable D6.4.

13. References

[1] Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2015). Migrating to Cloud-Native Architectures Using Microservices: An Experience Report

[2] CloudBroker GmbH. “CloudBroker Platform”. [Online]. Available: http://cloudbroker.com/platform/. [Accessed: 7 Mar 2017]

[3] József Kovács and Péter Kacsuk. 2018. Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures. J. Grid Comput. 16, 1 (March 2018), 19-37. DOI: https://doi.org/10.1007/s10723-017-9421-3

[4] Occopus website, http://occopus.lpds.sztaki.hu

[5] Docker, http://www.docker.com

[6] Prometheus website, https://prometheus.io/docs/introduction/overview/

[7] Node exporter website, https://github.com/prometheus/node_exporter

[8] Cadvisor website, https://github.com/google/cadvisor

[9] Jinja2, http://jinja.pocoo.org/docs/2.10

[10] RabbitMQ, https://www.rabbitmq.com

[11] Redis, https://redis.io/

[12] Statsd, https://github.com/etsy/statsd

[13] Grafana, https://grafana.com/

[14] Osama Abu Oun, Tamas Kiss: Job-queuing and Auto-scaling in Container-based Cloud Environments, IWSG 2018, 10th International Workshop on Science Gateways, 13-15 June 2018, Edinburgh, UK.

[15] OASIS Topology and Orchestration Specification for Cloud Applications Version 1.0, [online] Available from: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html

[16] TOSCAParser, https://wiki.openstack.org/wiki/TOSCA-Parser

[17] Flask – A Python Micro Framework. http://flask.pocoo.org/

[18] Bootstrap. https://getbootstrap.com/

[19] AdminLTE Control Panel Theme. https://adminlte.io/

[20] Gunicorn ‘Green Unicorn’. http://gunicorn.org/

[21] stress-ng, http://kernel.ubuntu.com/~cking/stress-ng

[22] CQueue, https://gitlab.com/lpds-public/documents/tree/master/COLA/cqueue

[23] Autodock VINA, http://vina.scripps.edu/