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: 33 months

Deliverable:

D6.4 Prototype and documentation of the price/performance optimization service

Due date of deliverable: 30/09/2019 Actual submission date: 27/09/2019

WPL: Jozsef Kovacs

Dissemination Level: PU

Version: V0.22

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	24/09/19	Jozsef Kovacs
Reviewed:	Gabor Terstyanszky	25/09/19	Gabor Terstyanszky
Approved:	Tamas Kiss	27/09/19	Tamas Kiss

Table 1 Status Change History

Version	**Date**	**Page/Section**	**Author(s)**	**Modification**
V0.1	23/08	ALL	Jozsef Kovacs	Empty Skeleton
V0.2	26/09	Section 7	Jozsef Kovacs, Eszter Kail	Preword for Optimizer
V0.3	28/09	Section 7.3	Jozsef Kovacs	Add implementation details on Optimizer
V0.4	04/09	Section 7.4	Jozsef Kovacs, Reka Kosa	Add section on User manual
V0.5	06/09	Section 9	Balint Kovacs	Adding Security Enablers Implementation Reference Guide
V0.6	10/09	Section 5	Jozsef Kovacs	Writing section
V0.7	12/09	Section 6	Jozsef Kovacs	Writing section
V0.8	13/09	Section 4	Jozsef Kovacs	Writing section
V0.9	16/09	Section 8	James Deslauriers	Description of WMIN developments
V0.10	17/09	Section 8	Jozsef Kovacs	Updating with SZTAKI developments
V0.11	18/09	Section 7.1	Eszter Kail, Istvan Pintye	Add background section for Optimizer
V0.12	18/09	Section 7.5	Eszter Kail, Istvan Pintye	Add section on testing results
V0.13	19/09	Section 7.2	Istvan Pintye, Eszter Kail, Jozsef Kovacs	Add concept on Optimizer
V0.14	19/09	Section 7.5	Jozsef Kovacs	Rewriting, corrections
V0.15	20/09	Section 8.8	Resmi Ariyattu	Adding description on Terraform

V0.16	20/09	Section 10	Jozsef Kovacs	Writing section
V0.17	20/09	Section 1,2,3	Jozsef Kovacs	Updating sections
V0.18	20/09	ALL	Jozsef Kovacs	Formatting
V0.19	23/09	Sections 7,8,9,10 and Appendix1	Jozsef Kovacs	Updates based on Tamas Kiss recommendation
V0.20	24/09	Section 8.10	Vitalii Zakharenko, Jozsef Kovacs	Writing and integrating section on CloudBroker integration
V0.21	25/09	ALL	Gabor Terstyanszky	Review of D6.4
V0.22	26/09	ALL	Jozsef Kovacs	Addressing review suggestions

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) price/performance optimization service, called MiCADO Optimizer. This service was the last component to be developed in the framework of the project. The work described in this document was completed in Task 6.4 in M16-M33. During this period there were several other developments performed by WP6, which are also outlined in this report. Since the deliverable is the last one produced by WP6, it also summarizes the security features developed by WP7 and integrated by WP6 into the MiCADO framework.

This deliverable reports the work performed by WP6 as a continuation of work described in deliverables D6.1, D6.2 and D6.3. The work described in this deliverable also utilizes the results reported in WP5 and WP8 deliverables such as the design of Application Description Template that describes applications and policies to be handled by the MiCADO framework and the application requirements. Moreover, the work presented in this deliverable is an important input for the deliverables produced by WP7.

The rest of this deliverable is organized as follows: In order to make this document self-contained and to let the reader understand the background a short summary of the generic architecture of the MiCADO framework in Section 5 and of the design and implementation of the current MiCADO Orchestration Layer in Section 6. Next, Section 7 focuses on introducing the main result of Task 6.4, namely the newly designed and implemented MiCADO Optimizer service. Beyond this service, WP6 has developed further features and functionalities for the MiCADO framework in M16-M33, These developments have been summarized in Section 8. Finally, current status and conclusion in Section 0 closes the deliverable. Appendix contains a reference guide about the security functions integrated into MiCADO.

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.

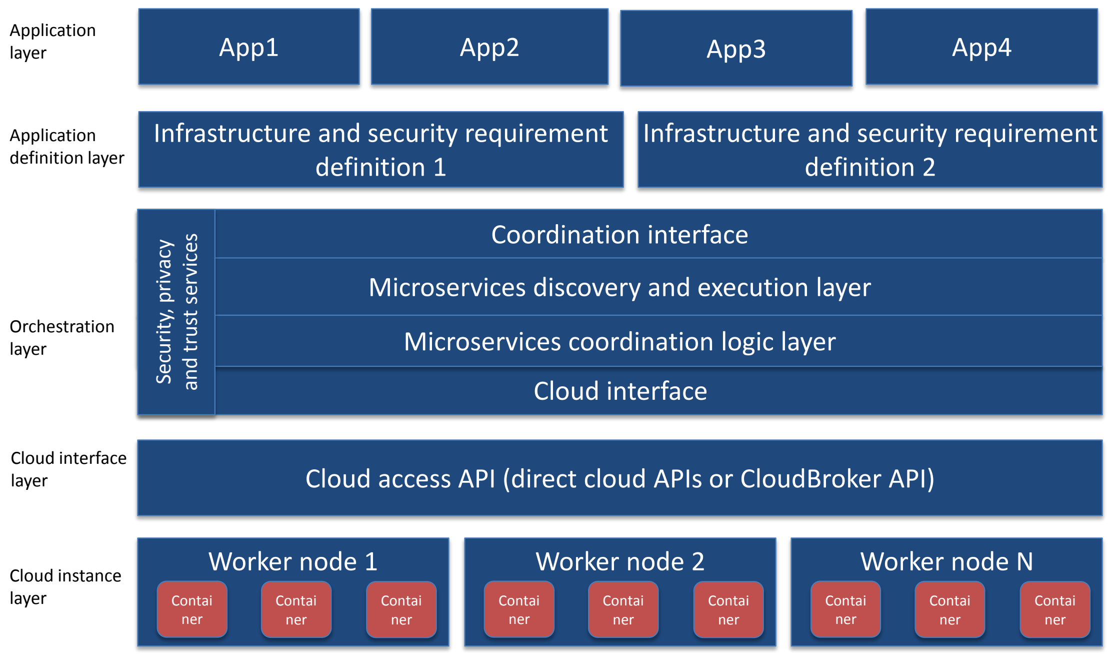

Application layer
App1
App2
App3
App4
Application definition layer
Infrastructure and security requirement definition 1
Infrastructure and security requirement definition 2
Orchestration layer
Security, privacy and trust services
Coordination interface
Microservices discovery and execution layer
Microservices coordination logic layer
Cloud interface
Cloud interface layer
Cloud access API (direct cloud APIs or CloudBroker API)
Cloud instance layer
Worker node 1
Worker node 2
Worker node N
Container

Figure 1 MiCADO generic architecture framework

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

1. **Application layer**. Application layer contains the actual application code and data described by application definition (layer 2) to work 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 layer and decouples it from the application definition layer. This set of APIs enables application developers to utilize the dynamic orchestration capabilities of the underlying layers 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 layer.** This sub-layer manages the execution of microservices and keeps track of services running. Execution management combines both start-up, running 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 discovery and 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.

   The vertical sub-layers are:
   e. **Security, privacy and trust services.** 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. MiCADO Orchestration Layer

6.1 Overview of the design

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 the MiCADO framework.

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 into consideration 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 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.

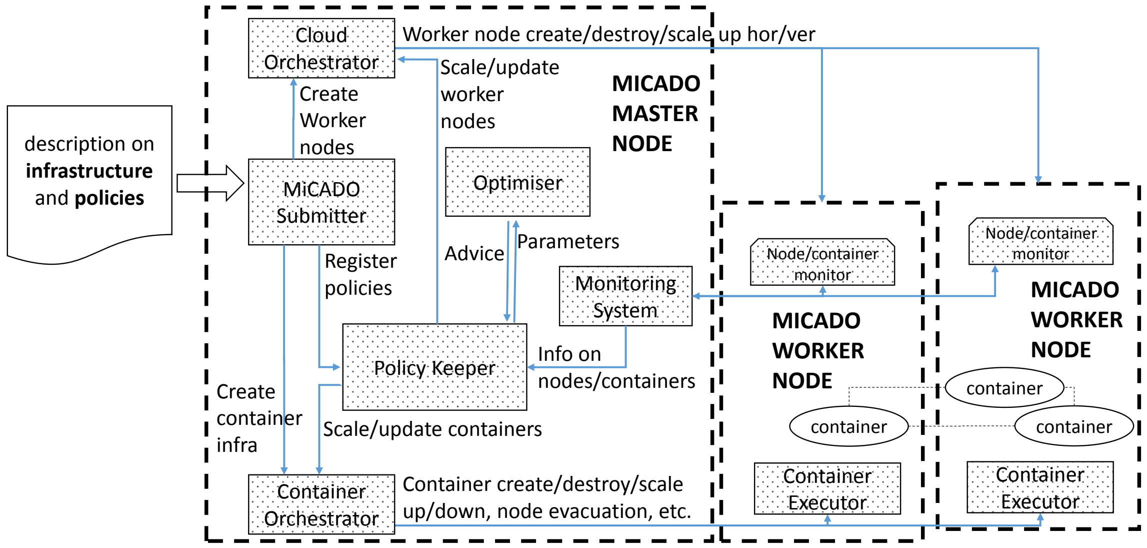


Figure 2 Architecture of the MiCADO Orchestration Layer

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. Description of submitted infrastructures is received by this component. The 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 for 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, keeping track of their execution, and destroying them if necessary. This component must also manage the scaling 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 providing 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 action(s).
- **Policy Keeper** is the key component that implements policies and makes decisions for 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 microservice performing calculations on demand for finding optimized setup of cloud resources of the infrastructure. An optimization calculation can be initiated with the required parameters on resources. 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 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 running on the Master node.
- **Container** components realize 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, 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. 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 to improve the execution.

In this architecture, the Cloud Orchestrator and Container Orchestrator components together with the MiCADO Submitter realize the initial deployment of the resources and containers. In case there are any policies defined for 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.

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 System) 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 Overview of the 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. However, later during the project at M24 the consortium decided to replace Swarm with Kubernetes [6].

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 [7] with Node exporter [8] and CAdvisor [9] components (as data sources) on the Worker nodes, has been added to the MiCADO implementation.

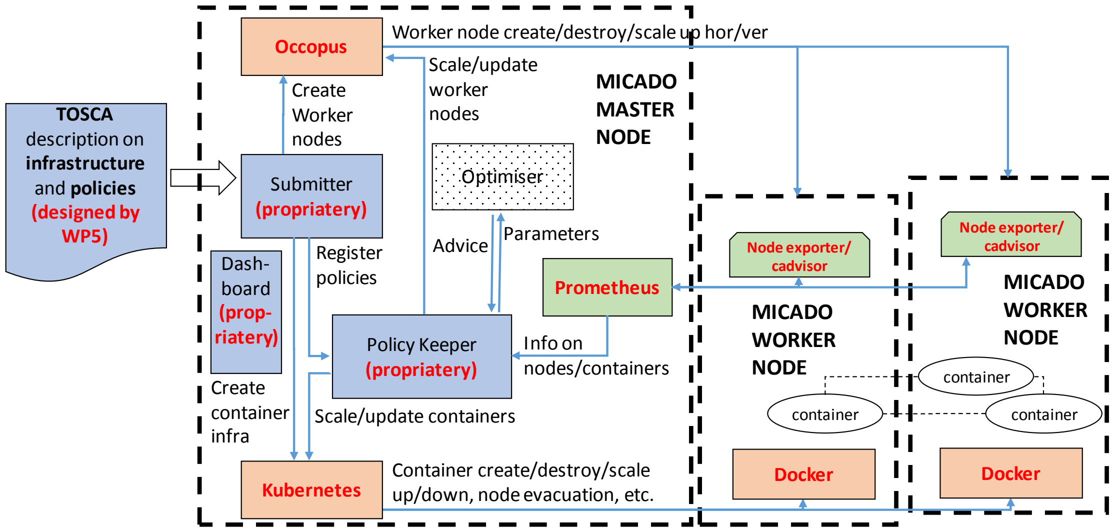


Figure 3 MiCADO implementation stage by stage

In the fourth phase of implementation, the Policy Keeper (to support automatic decision making), Dashboard and MiCADO Submitter with TOSCA support (depicted by blue boxes in Figure 3) has been introduced and reported in Deliverable D6.3.

Finally, in the recent project period, the Optimizer (depicted by uncolored, dotted box in Figure 3) has been implemented and reported in the current D6.4 deliverable.

7. Designing the MiCADO Price/Performance Optimization Service

The task of the MiCADO Price/Performance Optimization service (further called Optimizer) is to provide scaling decision in a way that the application running under MiCADO is optimized for costs and performance. This means that the application should provide the best/required performance for as low cost as possible. Increasing the performance usually results in increased costs, while lowering the costs usually decreases the performance for a given set of resources.

Calculating and detecting the best performance can be a complicated task. It is much simpler for the optimizer to let the user describe the expected performance, instead of detecting it by the optimizer. Even for the same application, best performance can be defined in different ways depending on what key feature(s) the application must provide to achieve the best performance where performance may focus on user experience, processing speed or memory usage, it is the task of the user to decide on.

In order to optimize the execution of an application towards costs, one of the possibilities for cost tracking is to continuously monitor the actual costs for each resource consumed by the application. Some clouds may provide it (e.g. Amazon Web Services), but most of them do not, i.e. cost tracking cannot be applied in every cloud. Moreover, even if costs can be extracted through the cloud API, for optimization costs must be discovered for each scaling action in order to provide good decision, i.e. how a certain scaling command would affect the overall costs. As a consequence, cost tracking can be a very complicated task when the expectation is to gather cost details for each scaling action and for each resource individually and on a per minute basis.

One possibility to avoid cost tracking (i.e. continuously calculating or monitoring the cost for each monitoring resource) is to calculate/support the decision on scaling in a way, which ensures that the resource usage is always kept to the minimum but still providing the expected performance defined by the user. The basic assumption is with the lowest number of resources used for the application the lowest cost can be reached for the same application provided that the internal structure and operation of the application is already optimal and cannot be affected by the optimizer.

To simplify the optimization, the optimizer implements virtual machine level horizontal autoscaling, so the resources are represented by homogeneous virtual machines, while the expected performance can be expressed by one measurable metric provided by the Prometheus monitoring system integrated into MiCADO. Beyond the metric itself, threshold levels are also required to let the optimizer know what values of the metric representing the expected performance of the application are appropriate.

In order for the optimizer to find the lowest number of resources in every situation as quickly as possible, it must learn the consequences generated by the scaling commands; increasing or decreasing the number of resources assigned to the application. Learning the scaling effects can be done by continuously monitoring certain metrics and variables of the application and finding the relation among them. For this purpose, machine learning algorithms and techniques have been applied.

The following sections will explain the details of the concept and operation.

7.1 Background

Based on exhaustive literature review we found that possible solutions for optimized scaling were categorized into five different categories. These are threshold-based, queuing theory based, control theory based, time-series analysis based and machine learning-based solutions [10][11].

One of the most widely used scaling techniques is the **threshold-based approach**. Its main advantage is simplicity. However, as it is based on statistical analysis of cost efficiency it is not as optimal as it could be. **Queuing theory and control theory-based solutions** require a system model before they can be applied, which is not possible in every situation or requires lot of efforts to create this model. Classic **time-series methods** aim to forecast future behavior from a huge amount of historical data to forecast future workload or resource usage. The historical data on customers' usage patterns is usually collected by cloud providers. In case of MiCADO, this huge amount of historical data is not available since, MiCADO collects information for one single application used by one single user.

The above-mentioned approaches for optimized auto-scaling rely on a deep understanding of the application, the underlying infrastructure and their dynamics to accurately scale resources. In absence of such information, exhaustive instrumentation and experiments are required from the user or developer to study the system. The main disadvantage of these solutions is that an experienced application developer is needed, however the dynamics of the application can vary from application to application, and from infrastructure to infrastructure. Therefore, these solutions require significant human efforts.

There are many popular techniques used to carry out learning the relationship between variables. "**Machine learning**" solutions such as regression, neural networks and reinforcement learning techniques can be considered in auto-scaling. **Regression** is a statistical method used to determine the polynomial function that is the closest to a set of points (in this case, the w values of the history window). The objective is to find a polynomial for which the distance from each of the points to the polynomial curve is as small as possible and therefore fits the data best. **Neural networks** consist of an interconnected group of artificial neurons, arranged in several layers. It has an input layer with several input neurons, an output layer with one or more output neurons, and one or more hidden layers in between. During the training phase, neural networks are fed with input vectors and random weights. Those weights are adapted until it shows the desired output for a given input. Recently an emerging new method, the **Reinforcement Learning** (RL) [12][13], concerning auto-scaling has appeared. RL approach will adapt to suit the environment based on its own experience. It offers the potential to develop optimal allocation policies without requiring explicit system models and knowledge extracted from historical usage of data. RL also provides additional advantages when it comes to delayed effects of a configuration change on both applications and infrastructure performance metrics and also on future decisions on reconfiguration. Their main drawback is that to find the optimal policy the system should scale the resources randomly during the learning phase. During the learning phase this method discovers decisions causing bad results, too. Due to characteristic, a very detailed large state space is explored, so the learning period takes very long time. Therefore, this approach can be very cost intensive.

The proposed solution for the Optimizer is to apply **Machine learning method combining Regression and Neural networks**. This solution enables a guided state space discovery which is less cost intensive and takes only a fraction of time compared to the RL solution.

7.2 Concept

The goal of the Optimizer is to give advice on the optimal number of virtual machines that should be allocated to provide cost efficient resource usage from the perspective of the user. In order to realize this, the Optimizer is designed along the following aspects. These are as follows.

1. **Expected performance: target variable**
   The Optimizer needs to know what the expected performance is that the application should deliver for the users. The performance must be expressed with a variable, which can be continuously monitored and tracked. This variable is called target variable for which the Optimizer requires minimum and maximum values. These min and max values define a range in which the Optimizer must keep the value of the target variable.

2. **Effect of scaling events on system variables**
   The Optimizer must learn the effect of the scaling events, i.e. how the different system related parameters/variables/metrics change when a virtual machine is added or removed from the set of resources the application is using. The assumption is that when a new virtual machine is added the load is distributed among the nodes and the monitored system parameters (expressing load-related values) will definitely change after the scaling happened. Based on this assumption, the Optimizer must continuously track the system parameters and extract (and register) their values before and after the scaling events. Based on these values, the Optimizer can learn and predict how the variables will change after scaling. The system variables in this context express load-related parameters, like CPU usage, memory usage, number of interrupts, size of incoming packets and so on. From the Optimizer point of view, we call these variables as system (or input) variables.

3. **Effect of scaling events on target variable**
   Once the relation among the pre and post values of each system variables is learned, the Optimizer needs to learn the relation between the system variables and the target variable. Based on this knowledge, when the Optimizer has the actual values for each system variables the target variable can be predicted as well. Once the target variable can be predicted based on the system variables, the Optimizer will be able to predict the target variable for each investigated (hypothetical) scaling event and as a result, it will be able to select the best alternative i.e. to suggest scaling (or virtual machine number) which is predicted to be the best or most optimal for the running application.

4. **Lowest number of resources**
   The Optimizer must keep the number of virtual machines (resources) as low as possible while keeping the target variable in the specified range. Since price-performance optimization in the current solution does not involve the tracking of the cost of the individual resources, the goal is to keep the resources as low as possible in order to keep the cost (i.e. price) of operating the application as low as possible.

There are two learning mechanisms distinguished and designed for optimization as it has been described above. They are as follows:

1. **Learning the relation between values of system metrics before and after scaling**
   The goal of this learning model is to learn the linear relation between the values of the system metrics before the scaling action is executed and the values after the scaling action was performed. For this purpose, the machine learning method applied in the Optimizer is a collection of linear regression models. Several investigations have been performed and linear regression model [14] has been chosen due to the fact that it gives the best prediction results for this case [15].

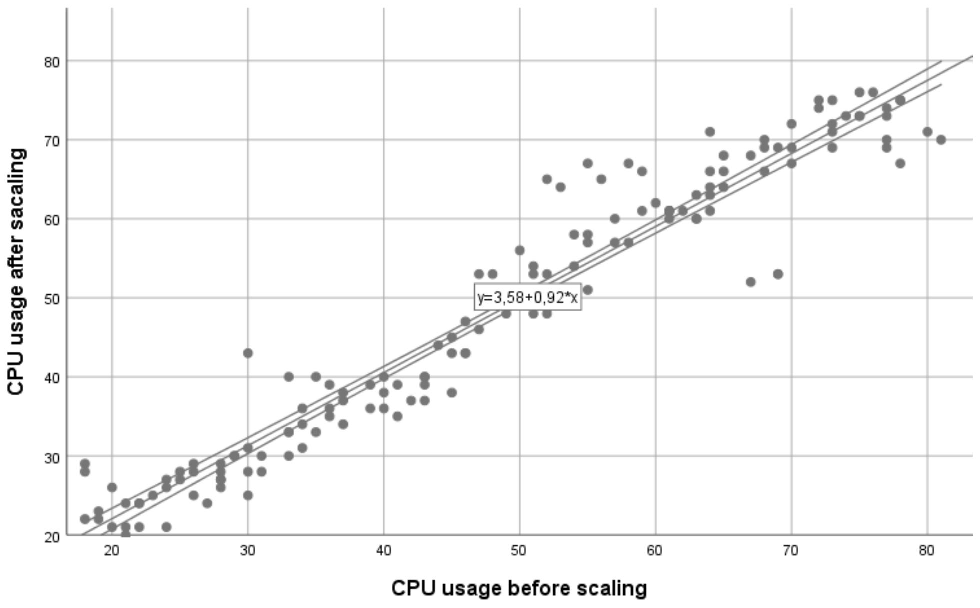


Figure 4 Linear regression model for CPU load

These models are created for each system variable and the inputs are value pairs of a variable before and after the scaling action. With many value pairs registered in a linear regression model of a system variable (e.g. for CPU load see Figure 4), the relation will be automatically calculated, and this calculated relation can be used later for predictions. One regression model is able to predict the new value of a system variable (metric) for a given VM number and its change. The calculation is based on the following equation:

$$metric_{1_{est}} = \omega_0 + \omega_1 * metric_{1_{act}} * \frac{VM_{act}}{VM_{act} + VM_{new}} + \omega_{2*} metric_{1_{act}} * \frac{VM_{new}}{VM_{act} + VM_{new}}$$

where $metric_{1_{act}}$ refers to the actual measured value of the first system metric, $metric_{1_{est}}$ refers to the estimated value of this system metric, VM_{act} is the number of VMs that are already allocated by the auto-scaling module, the VM_{new} is the number of VMs after a hypothetical scaling. Solving the regression, the optimizer will know the w_0, w_1, w_2 weights. Once these weight are calculated, the LR can be used to predict the values of system variables (metrics) if VM number changes from a given value to a new value, i.e. scaling would happen.

2. **Learning the relation between the system variables and target variable**
   The goal of this learning model is to discover and learn the non-trivial and non-linear relation among the system variables and the target variable. Based on our experiences this relation cannot be represented in a linear fashion. For modeling non-linear relation, Neural Network called Multi-Layer Perceptron Neural Network [12][15] have been chosen (see Figure 5). It has three layers altogether, where the hidden layer (in the middle) has its activation functions configured as hyperbolic tangent. The input layer (on the left side) receives the values of the system variables, while the output layer (on the right side) represents the value of the target variable. The neural network has two operational phases. During training phase, many sets of input and output variables are fed to the network, which as a result learns their relation automatically by adjusting the weight values of each neurons in the network. The more set of variables are shown to the network the more precise prediction can be expected. Once the weights are configured to provide relation between the input and output values with an appropriate error rate, the neural network is considered as trained and the phase turns to production. In production phase, the trained neural network then can calculate the target variable for a set of input variables based on its knowledge.

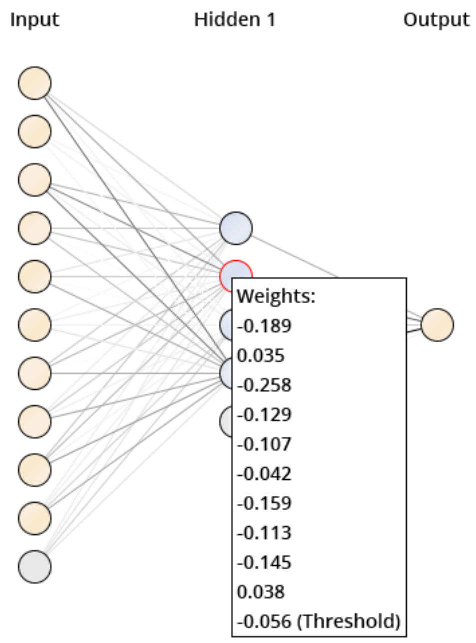


Figure 5 Neural Network in the Optimizer

The knowledge is represented by the weight values of the network, which are periodically updated. The network can further train itself during production phase and is able to calculate its error rate. Error rate is based on the fact that for each set of input variables the actual value of the target variable is also shown to the network, so the difference can be measured.

Advice generation mechanism in the Optimizer

In order to get a proper advice from the Optimizer, it needs to have properly built knowledge. This knowledge is represented as:

- several trained linear regression (LR) models for each system variable
- a trained neural network (NN) model for the target variable

The Optimizer performs the following steps when calculating an advice for scaling:

1. the Optimizer registers the actual number of virtual machines (N) and the actual values of the system variables ($I_1 \ldots I_M$)
2. The Optimizer elaborates several hypothetical scaling alternatives. These alternatives represent scaling down and up events i.e. the number of virtual machines can be decreased and increased: $N_{ALT}=\{N+\Delta S_{up}, N+\Delta S_{up}-1, \ldots, N, \ldots, N-\Delta S_{down}+1, N-\Delta S_{down}\}$ where ΔS_{up} and ΔS_{down} are parameters specifying the maximum allowed change in number of virtual machines in scaling down and scaling up events.
3. For each hypothetical number of virtual machines, N_{ALT} the Optimizer performs checks whether the modified number of virtual machines offers better performance for the application.
4. The evaluation starts with calculating the new system variables ($I_1 \ldots I_M$) using the LR models. Each system variable is submitted to the LR model belonging to the given system variable and LR returns the new value of the system variable for the specified number of virtual machines. The result is the set of new system variables for a given VM number.
5. The newly calculated values for the set of system variables ($I_1 \ldots I_M$) then fed to the NN model, which returns the predicted value of the target variable (T_x).
6. When the value of target variables is calculated for each investigated number of virtual machines ($T_1 \ldots T_X$) a filtering happens. The filtering criteria for the modified number of virtual machines is that the corresponding value of target variable must be within the expected range. As a result, only those VM numbers remain in the list for which the prediction shows proper value of the target variable. If there is no virtual machine number that can keep the target variable in the expected range, the virtual machine number, with target variable closest to the range, will be selected. This means that the most cost effective VM number is selected.
7. Beyond the VM number selection, reliability is also calculated to give feedback on the goodness of the suggested VM number for consideration. The reliability is based on the Pearson's Correlation coefficients returned by the models. This value is normalized to the 0..100 domain to express percentage.

The description above gives a simple, but understandable explanation of the optimal virtual machine number selection mechanism of the Optimizer. With this mechanism, the Optimizer recommends to use the smallest VM number keeping the performance (i.e. the target variable) still in the expected range.

7.3 Architecture, implementation and integration

The Optimizer is implemented as a service with a REST API and is running on the MiCADO master node. The MiCADO architecture with the Optimizer can be seen on Figure 6.

The Optimizer service is linked and cooperating with the Policy Keeper (PK). As it has been described in D6.3, Policy Keeper is responsible for executing the user-defined scaling rules (which are realized by short python code) considering the advice coming from the Optimizer as input to the user-defined scaling rule.

The communication is initiated by PK in every situation. At startup, PK forwards the Optimizer a set of user defined parameters and variables. These are the settings, system and target variable. PK periodically collects the actual values of the system and target variables from Prometheus monitoring service and reports them to the Optimizer. Based on these reported values the Optimizer learns the relations and develops its knowledge. Having this knowledge, the user-defined scaling rule may request an advice from the Optimizer at any time. When a scaling rule requires an advice, PK contacts the Optimizer and ask for an advice. As a response, the Optimizer sends an advice on the number of virtual machines, which is the most optimal in the current situation. The user-defined scaling rule receives the advice in the form of a dictionary. This dictionary contains the number of virtual machines where validity, reliability, production phase and error messages if any are also part of the advice dictionary. Having this information PK makes the final decision, i.e. either to launch extra VMs or destroy some VMs, or to override the advice and scale VMs to a different value.

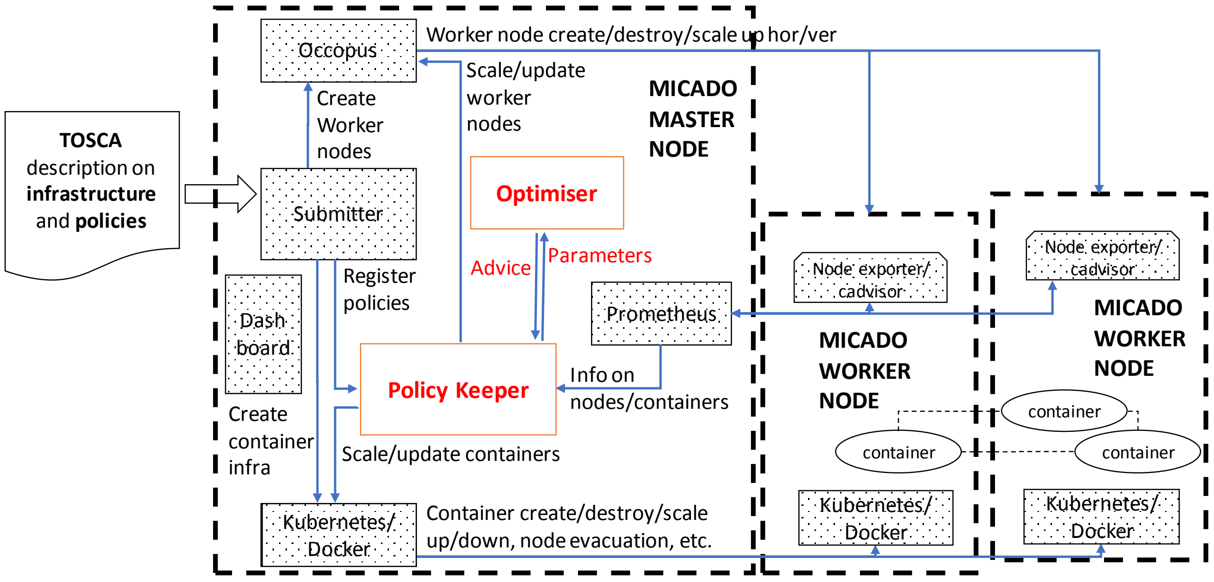


Figure 6 Optimizer and its environment

The Optimizer is implemented as a service, written in Python and implemented using Flask [16] library for the REST API. For implementing learning, neural network and linear regression mathematical models are used for which scikit-learn[17], pandas[18] and numpy[19] libraries are integrated. The Optimizer, similarly to the other components in MiCADO master, is running in a container, in which it maintains its own internal knowledge base.

The Optimizer has a REST API, which is as follows:

- POST /init <settings>
  Initializes the Optimizer. Resets (or keep) the knowledge based and registers the new set of input variables and the target variable. The <settings> may contain parameters with values associated affecting the operation of the Optimizer. This method is invoked once by Policy Keeper when starting a new policy.
- POST /sample <variables>
  Invoking this method with a data structure containing the value of each input variable and the target variable, trains the optimizer and helps further developing the knowledge base. It is periodically invoked by the Policy Keeper.
- GET /advice
  This method returns an advice for the number of virtual machines in the form of a data structure with a few additional fields like validity, reliability, error if any, phase

and so on. This method is invoked by the Policy Keeper once the user code requests an advice.

- GET /report
  This method returns a webpage containing diagrams, statistics on the current input and target variables and on their relations gained by the knowledge base built.

The invocation of the methods of the Optimizer by the user is not recommended. This is the reason it is not specified in details. The REST API of the Optimizer is used directly by the Policy Keeper, which exposes a different user interface for the developers. This user interface can be found in section 7.4.

7.4 User manual

For implementing more advanced scaling policies, it is possible to utilize the built-in Optimizer in MiCADO. The role of the Optimizer is to support decision making in calculating the number of worker nodes (virtual machines) i.e. to scale the nodes to the optimal level. Optimizer is implemented using machine learning algorithm aiming to learn the relation between various metrics and the effect of scaling events. Based on this learning, the Optimizer is able to calculate and advise on the necessary number of virtual machines.

7.4.1 Current limitations

Optimizer supports

- only web-based applications
  The current version of the Optimizer requires the average and sum request rate of the web server (Apache, Nginx, etc.) where the web-based application is running. This limitation is planned to be eliminated by the next version.

- only one of the node sets
  MiCADO supports multiple set of nodes scaling independently. The current Optimizer can be configured to provide advice for only one of the node sets in the application. No plan to remove this barrier at the moment.

- no container scaling
  MiCADO supports virtual machine (node) and container scaling independently. However, the Optimizer currently supports only VM scaling. To override this, container scaling rule is recommended to follow the number of nodes. There is no plan to remove this barrier at the moment.

7.4.2 Basic principles

- User specifies a so-called target metric with its associated minimum and maximum thresholds. The target metric is a monitored Prometheus expression for which the value is tried to be kept between the two thresholds by the Optimizer giving scaling advices.
- User specifies several so-called system metrics which represent the state of the system correlating to the target variable
- User specifies several initial settings (see later) for the Optimizer
- User submits the application through the ADT activating the Optimizer

- Optimizer starts with the 'training' phase during which the correlations are learned. During the training phase artificial load must be generated for the web application and scaling activities must be performed (including extreme values) in order to present all possible situations for the Optimizer. During this phase, Optimizer continuously monitors the system/target metrics and learns the correlations.
- When correlations are learnt, Optimizer turns to 'production' phase during which advice can be requested from the Optimizer. In this phase, Optimizer returns advice on request, where the advice contains the number of virtual machines (nodes) to be scaled to. In the production phase, the Optimizer continues its learning activity to adapt to the new situations.

7.4.3 Activation of the Optimizer

The Optimizer can be activated, configured and utilized through the ADT of MiCADO under the scaling policy (see example in Code 2). The Optimizer-related parameters must be inserted into subsections "constants" and "queries". Each parameter relating to the Optimizer must start with the "m_opt_" string. In case no variable name with this prefix is found in any sections, Optimizer is not activated.

7.4.4 Initial settings for the Optimizer

Parameters for initial settings are defined under the "constants" section and their name must start with the "m_opt_init_" prefix. These parameters are as follows:

- **m_opt_init_knowledge_base** is a parameter which specifies the way how the knowledge base must be built under the Optimizer. When defined as "build_new", Optimizer empties its knowledge base and starts building a new knowledge i.e. starts learning the correlations. When using the "use_existing" value, the knowledge is kept and continued building further. Default is "use_existing".
- **m_opt_init_training_samples_required** defines how many samples of the metrics must be collected by the Optimizer before it starts learning the correlations. Default is 300.
- **m_opt_init_max_upscale_delta** - maximum change in number of nodes for an upscaling advice. Default is 6.
- **m_opt_init_max_downscale_delta -** maximum change in number of nodes for a downscaling advice. Default is 6.
- **m_opt_init_advice_freeze_interval** - how many seconds must elapse before the Optimizer advises a different number of nodes. Can be used to mitigate the frequency of scaling. Defaults to 0.

7.4.5 Definition of system metrics for the Optimizer

System metrics must be specified for the Optimizer under the "queries" subsection to perform the training i.e. learning the correlations. Each parameter must start with the "m_opt_input_" prefix, e.g. m_opt_input_CPU. The following two pieces of variable must be specified for the web application:

- **m_opt_input_AVG_RR** - the average request rate of the web server.
- **m_opt_input_SUM_RR** - the summary of request rate of the web server.

7.4.6 Definition of the target metric for the Optimizer

Target metric is a continuously monitored parameter that must be kept between thresholds. To specify it, together with the thresholds, “m_opt_target_” prefix must be used. These three parameter must be defined under the “queries” sections. They are as follows:

- **m_opt_target_query_MYTARGET** - prometheus query for the target metric called MYTARGET.
- **m_opt_target_minth_MYTARGET** - value above which the target metric must be kept.
- **m_opt_target_maxth_MYTARGET** - value below which the target metric must be kept.

7.4.7 Requesting scaling advice from the Optimizer

In order to receive a scaling advice from the Optimizer, the method **m_opt_advice()** must be invoked in the scaling_rule section under the scaling policy section of the ADT.

IMPORTANT! Minimum and maximum one of the node scaling policy, must contain this method invocation in its scaling_rule section for proper operation!

The **m_opt_advice()** method returns a python dictionary containing the following fields:

- **valid** stores True/False value indicating whether the advice can be considered or not.
- **phase** indicates whether the Optimizer is in “training” or “production” phase.
- **vm_number** represents the advice for the target number of nodes to scale to.
- **reliability** represents the goodness of the advice with a number between 0 and 100. The bigger the number is the better/more reliable the advice is.
- **error_msg** contains the error occurred in the Optimizer. Filled when valid is False.

7.5 Testing results

The Optimizer has been tested and evaluated. For this purpose, we have used an existing web server-based application with some changes. The Wordpress application is already part of the demo applications of MiCADO stored in the GitHub repository. Moreover, detailed description on how the Wordpress demo application can be deployed to MiCADO can be found in the documentation of MiCADO [20].

7.5.1 Architecture of the application

The Wordpress application is an Apache server with a Wordpress blog on top it inside a container. In order to perform the measurements and monitoring some metrics related to Apache, an Apache exporter has been integrated into the same container. So, the container contains the Apache webserver and the Apache exporter (see Figure 7) denoted with dashed line on the MiCADO worker node. Beyond this container, each worker node contains a Node exporter in order to monitor the system variables required by the Optimizer. Both exporters are linked with Prometheus, which collects the values of the variables periodically.

Beyond these components an NFS volume is also hosted on the MiCADO worker node as a Kubernetes pod. This has the functionality to share a file system between the Apache webserver instances. Whenever a new Apache instance is created on a new MiCADO worker node the same file system with the same content is visible for the Apache webserver. This functionality is realized by Kubernetes volumes.

The Wordpress application requires a database in order to persist data stored on the webpages. For this purpose, a database has been attached and has been deployed in the Amazon cloud. The database in the current use case is realized by the RDS service of AWS to provide flexibility and significant capacity for handling a huge number of requests during the testing.

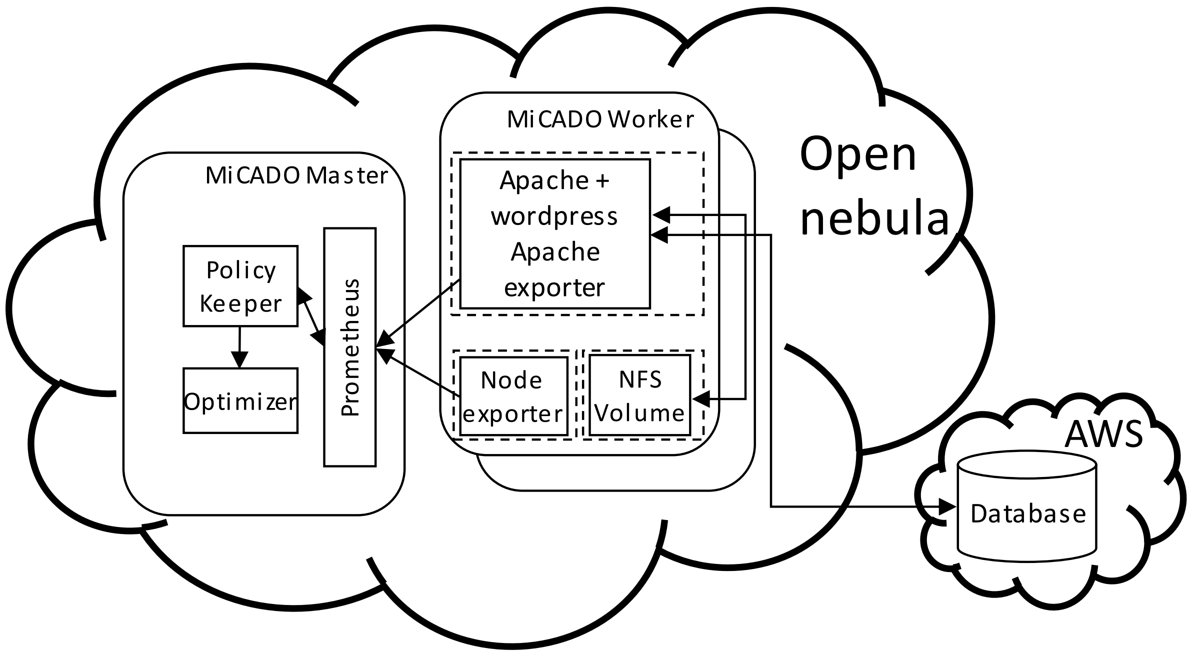


Figure 7 The architecture and environment of the Wordpress application during testing

The MiCADO master node contains quite a few components, however for the current use case only the Optimizer and its environment are represented, the rest is hidden in Figure 7. Prometheus is collecting the values of the variables periodically, while Policy Keeper requests the values whenever the scaling rule is evaluated. During the operation the Policy Keeper reports the values for the Optimizer, which then builds its knowledge.

7.5.2 ADT of the application

During the testing, a MiCADO 0.8.0 pre-release version was used and executed on the OpenNebula cloud of MTA SZTAKI. After the deployment of MiCADO, the Wordpress application was deployed using the ADT where topology is shown in Code 1 and scaling is shown in Code 2.

The topology has been shortened with removing the details in some of the ‘interface’ and ‘capabilities’ sections to fit into one page. 3 pods/containers (nfs-server-pod, nfs-volume, Wordpress) and 1 worker node (worker-node) is created (see Code 1). It can be seen how the Wordpress container utilizes the Amazon RDS (‘WORDPRESS_DB_HOST’ environment variable) and how it utilizes the shared file system (‘requirements’ section of ‘wordpress’ pod). The ‘endpoint_cloud’ at the end of ‘worker-node’ points to Opennebula.

```
tosca_definitions_version: tosca_simple_yaml_1_0
imports:
  - https://raw.githubusercontent.com/micado-scale/tosca/develop/micado_types.yaml
repositories:
  docker_hub: https://hub.docker.com/
topology_template:
  node_templates:
    nfs-server-pod:
      type: tosca.nodes.MiCADO.Container.Application.Docker
      properties:
         name: nfs-server-container
         securityContext:
           privileged: True
         args: ['/exports']
         ports:
           - target: 2049
             clusterIP: 10.96.0.240
           - target: 111
             protocol: udp
      requirements:
      artifacts:
       image:
         type: tosca.artifacts.Deployment.Image.Container.Docker
         file: itsthenetwork/nfs-server-ubuntu
         repository: docker_hub
      interfaces:
          …
    nfs-volume:
      type: tosca.nodes.MiCADO.Container.Volume
      properties:
        name: nfs-volume
      interfaces:
        Kubernetes:
          create:
            inputs:
              nfs:
                server: 10.96.0.240
                path: /
    wordpress:
      type: tosca.nodes.MiCADO.Container.Application.Docker
      properties:
         name: wordpress
         env:
         - name: WORDPRESS_DB_HOST
           value: database-1-instance-1.cvu3fktdwzal.eu-central-1.rds.amazonaws.com:3306
         - name: WORDPRESS_DB_USER
           value: root
         - name: WORDPRESS_DB_PASSWORD
           value: Passw0rd
         resources:
           requests:
             cpu: '900m'
         ports:
         - target: 80
           nodePort: 30010
           type: NodePort
         - containerPort: 80
           name: wordpress
         - containerPort: 9117
           name: wp-apache-exp
      artifacts:
       image:
         type: tosca.artifacts.Deployment.Image.Container.Docker
         file: emodimark/wordpress:5.1-metrics
         repository: docker_hub
      requirements:
        - volume:
            node: nfs-volume
            relationship:
              type: tosca.relationships.AttachesTo
              properties:
                location: /var/www/html
      interfaces:
          …
    worker-node:
      type: tosca.nodes.MiCADO.EC2.Compute
      properties:
        region_name: ROOT
        image_id: ami-00000526
        instance_type: t2.medium
        context:
          append: yes
          cloud_config: |
            runcmd:
            - apt-get install -y nfs-kernel-server nfs-common
      interfaces:
        Occopus:
          create:
            inputs:
              interface_cloud: ec2
              endpoint_cloud: https://opennebula.lpds.sztaki.hu:4567
      capabilities:
          …
```

Code 1 Topology part of ADT for Wordpress tested with the Optimizer

```
 policies:
- monitoring:
    type: tosca.policies.Monitoring.MiCADO
    properties:
      enable_container_metrics: false
      enable_node_metrics: true
- scalability:
    type: tosca.policies.Scaling.MiCADO
    targets: [ worker-node ]
    properties:
      sources:
        - 'wordpress:9117'
      constants:
        MINNODES: 1
        MAXNODES: 12
        m_opt_init_knowledge_base: "use_existing"
        m_opt_init_training_samples_required: 300
      queries:
        m_opt_input_AVG_RR: 'avg(rate(apache_accesses_total[2m]))'
        m_opt_input_SUM_RR: 'sum(rate(apache_accesses_total[2m]))'
        m_opt_input_CPU: 'avg(100 - (avg by (instance) (irate(node_cpu_seconds_total{mode="idle"}[2m])) * 100))'
        m_opt_input_Inter: 'avg(irate(node_intr_total[1m]))'
        m_opt_input_CTXSW: 'avg(irate(node_context_switches_total[1m]))'
        m_opt_input_KBIn: 'avg(irate(node_network_receive_bytes_total{device="eth0"}[1m]))/1024'
        m_opt_input_PktIn: 'avg(irate(node_network_receive_packets_total{device="eth0"}[1m]))'
        m_opt_input_KBOut: 'avg(irate(node_network_transmit_bytes_total{device="eth0"}[1m]))/1024'
        m_opt_input_PktOut: 'avg(irate(node_network_transmit_packets_total{device="eth0"}[1m]))'
        m_opt_target_query_AVG_LAT_05: 'avg(http_request_duration_microseconds{quantile="0.9",instance=~"10.*:9117"})'
        m_opt_target_minth_AVG_LAT_05: 9000
        m_opt_target_maxth_AVG_LAT_05: 15000
      min_instances: 1
      max_instances: 9
      scaling_rule: |
        adv = m_opt_advice()
        print('  ADVICE.valid: '+str(adv['valid']))
        print('  ADVICE.phase: '+str(adv['phase']))
        print('  ADVICE.vm_number: '+str(adv['vm_number']))
        print('  ADVICE.reliability: '+str(adv['reliability']))
        print('  ADVICE.error_msg: '+str(adv['error_msg']))
        if adv['valid']:
          m_node_count = adv['vm_number']
- scalability:
    type: tosca.policies.Scaling.MiCADO
    targets: [ wordpress ]
    properties:
      min_instances: '{{ MINNODES }}'
      max_instances: '{{ MAXNODES }}'
      scaling_rule: |
        m_container_count = len(m_nodes)
        print 'REQ CONT:',m_container_count
```

Code 2 Scaling policy part of ADT for Wordpress tested with the Optimizer

The scaling part of the ADT is shown in Code 2. It realizes a scaling with the help of the Optimizer. The Optimizer settings, system and target variables are specified under the ‘constants’ and ‘queries’ sections with m_opt_init_, m_opt_input_ and m_opt_target_ prefixes.

There are two settings in the current ADT. The parameter ‘m_opt_init_knowledge_base’ tells the Optimizer to use its existing knowledge (from previous runs) to perform the calculations. If a new application and new knowledge base must be built, the value of this parameter must be set to ‘build_new’.

The system variables are listed with ‘m_opt_input_’ prefixes where the value is the query expression that can be evaluated by Prometheus using the built-in exporters or the ones listed under the ‘sources’ section. The following system variables have been defined for the Optimizer:

- AVG_RR - average request rate of the last two minutes
- SUM_RR - summary of request rate of the last two minutes
- CPU - average cpu load of the nodes of the last two minutes
- Inter - average number of system interrupts happened in the last minute
- CTXSW - average number of context switch happened the last two minutes
- KBIn is average number of KBs received by the network of the nodes
- PktIn - average number of packets received by the network of the nodes

- KBOut - average number of KBs sent by the network of the nodes
- PktOut - average number of packets sent by the network of the nodes

For the target variable expressing the expected performance, the following variables are defined in the ADT (see Code 2):

- AVG_LAT_05 - response time i.e. the latency of the http requests
- 'minth' and 'maxth' - minimum and maximum thresholds for a range in which the latency must be kept by the Optimizer.

Scaling rule for the Virtual machines becomes simple by using the Optimizer since the returned/suggested advice is simply returned for the Policy Keeper for scaling. The scaling rule for the containers is implemented in a way to place one container of Wordpress for each node/virtual machine.

7.5.3 Evaluation of results

The ADT above was used for running some experiments with the Optimizer.

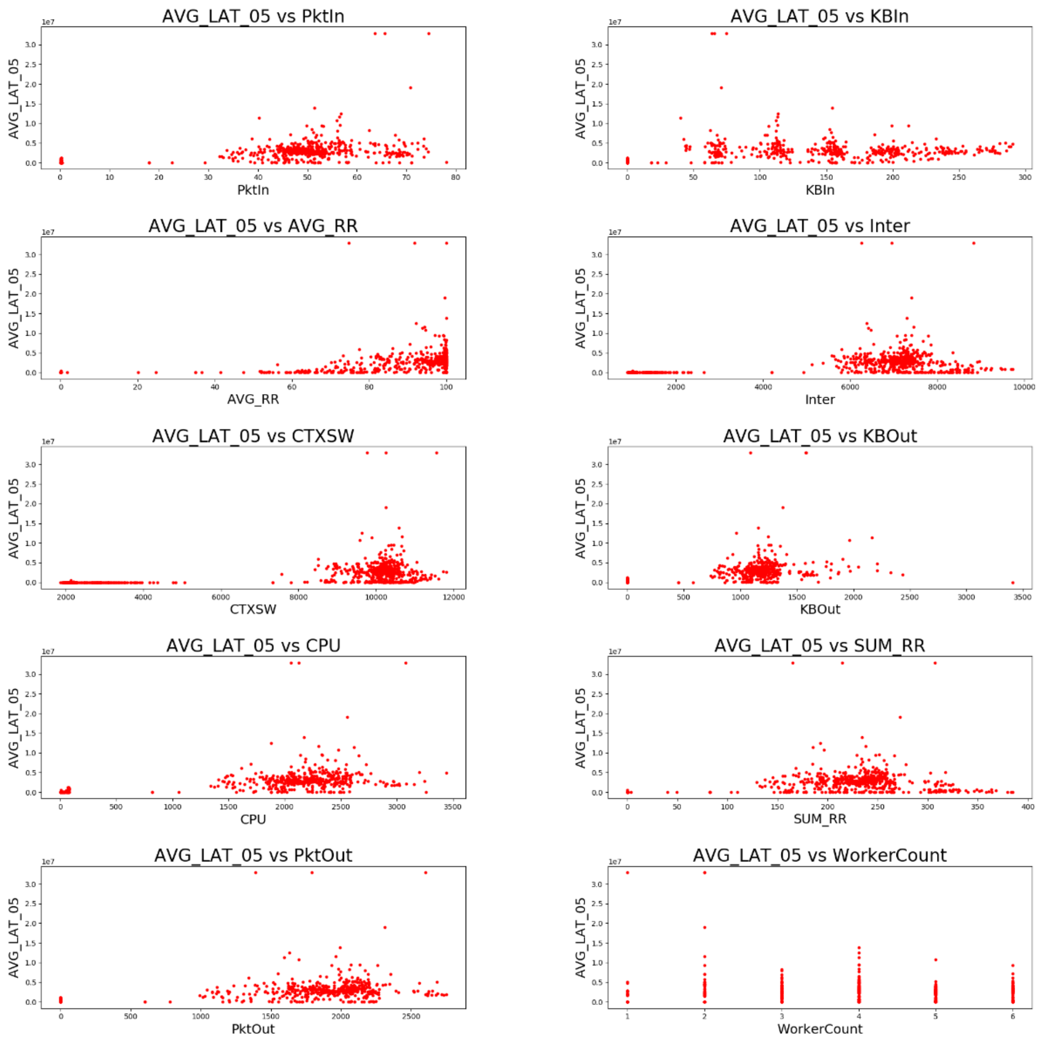


Figure 8 The correlation of the system metrics and the average response time (latency)

The graphs (see Figure 8) show the correlation of the different, monitored system variable and the user defined target variable (AVG_LAT_05) representing that it is an average value over the monitoring interval (15s). The variables assigned to the horizontal axis are the ones listed as input variables plus an extra diagram shows the same for the Worker count.

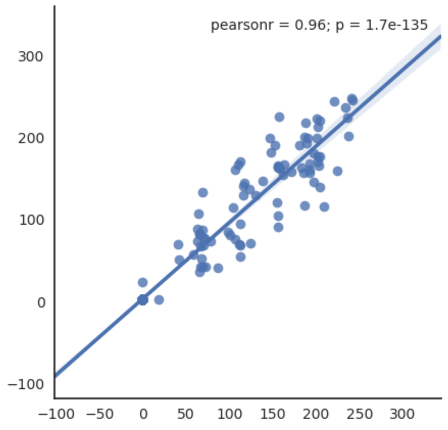


Figure 9 Evaluation of the linear regression model concerning the Kilobyte in per sec metric

Figure 9 gives a special representation, on how close real measure values of the Kilobyte in per sec system metrics to the estimated value of these metrics are. The x axis represents the real measured while the y axis represents the estimated Kilobyte in per sec metric. Each point on the graph represents a pair of the before mentioned values. The blue line in the middle represents the linear regression function. The closer the dots to the line are the better the model is. In this case, we can see that this model is not perfect, but it can grab the relation between the KBOut metric values after a scaling action. In the top right corner of the figure, the ‘pearsonr’ value is 0.96. This refers to the Pearson’s correlation coefficient which shows how strong correlation can be observed between the two variables. The closer this coefficient to 1, the stronger the correlation is. During the execution of the application similar diagrams are generated for each system variables (metrics).

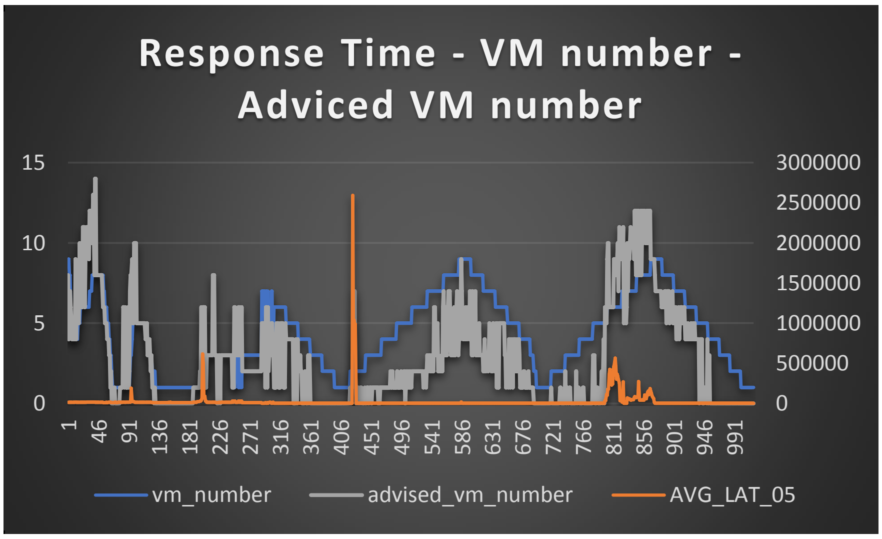


Figure 10 Training the Optimizer

We set the minimum number of samples required for proper operation, i.e.: estimation to 2750 with 15-sec sampling interval which means that the training phase lasted 687.5 minutes (11 hour and 45 minutes). During this time, we load the system with a quite ad-hoc and random way. The load was generated by Apache JMeter [21] with requests sent to the system. The number of the VMs allocated to the application was also continuously changed during this period. With this action our regression model was able to discover the linear relationship between the system metrics before and after a scaling action. The number of Virtual Machines was increased from the minimum value to the maximum value than reverse from the maximum to the minimum value. This scaling up and down process was repeated until the end of the training phase.

Figure 10 was captured during the training phase. The horizontal line represents the number of samples. The observations were taken periodically in every 15 sec. The blue line shows the real number of Virtual Machines which were artificially increased and decreased during the training phase. The orange line represents the average response time of the requests sent by the Apache JMeter. The grey line represents the given advice by the Optimizer module. It is observable that regardless of the current number of Virtual Machines, when the Response Time increased above a given limit, the Optimizer advised a higher number of Virtual Machines than the current number of Virtual Machines i.e. a scaling up action was suggested. For example, observing the samples from 800 to 860 when the response time increased above this limit, the Optimizer advised an increase in the number of VMs even if the current number of VMs were between 6 and 9. The same effect can be observed when the response time is lower than a certain limit the, e.g.: from samples 571 to 691 the Optimizer tried to decrease the number of Virtual Machines. The results can be comprehensively understood together with Figure 11.

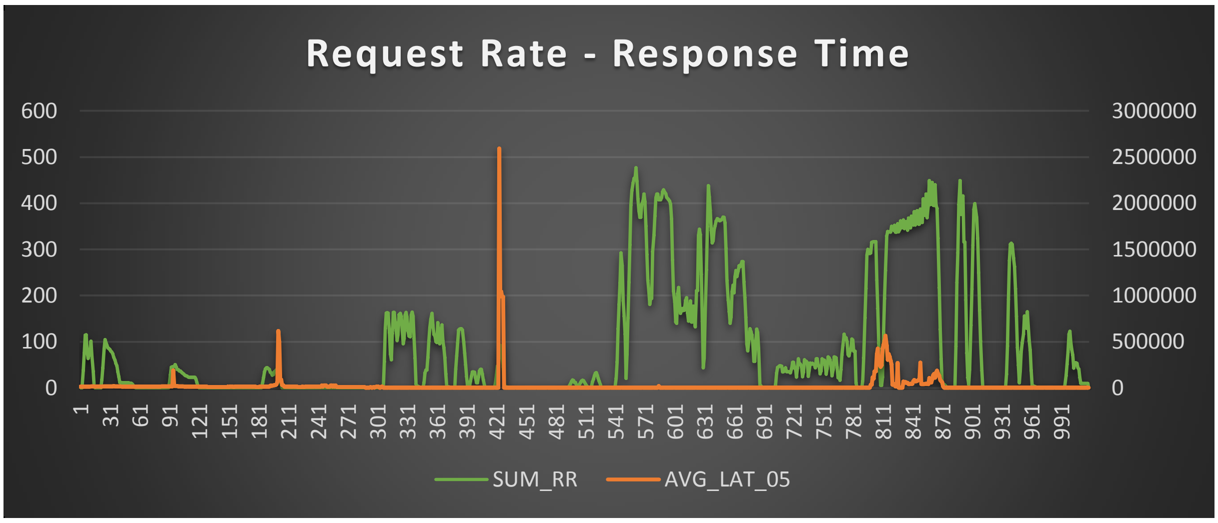


Figure 11 Request rate and response time in a function of time during the training phase

On Figure 11 the green line shows the incoming request rate, while the orange line remains the Response Time. Of course, the Request Rate, the current number of VMs and the Response time correlate to each other, but there are a lot of other factors that can have effect on the Response Time as well. From observation number 541 the request rate started to increase but the current number of VMs (see Figure 10) were enough to keep the response time at the desired level. Moreover, based on the Optimizer “knowledge” it advised a little bit lower number of VMs to keep the response time in the desired range.

While if we have a look at observation 841 beside the same request rate with the same number of VMs the system was not able to hold a response time under a certain level. In such a circumstance the Optimizer advised increasing the number of VMs (see Figure 10).

After collecting enough number of samples and the machine learning modules were trained the production phase may start, which means that auto-scaling will happen based on the advices of the Optimizer module. Figure 12 presents the operation of the Optimizer module during the production phase.

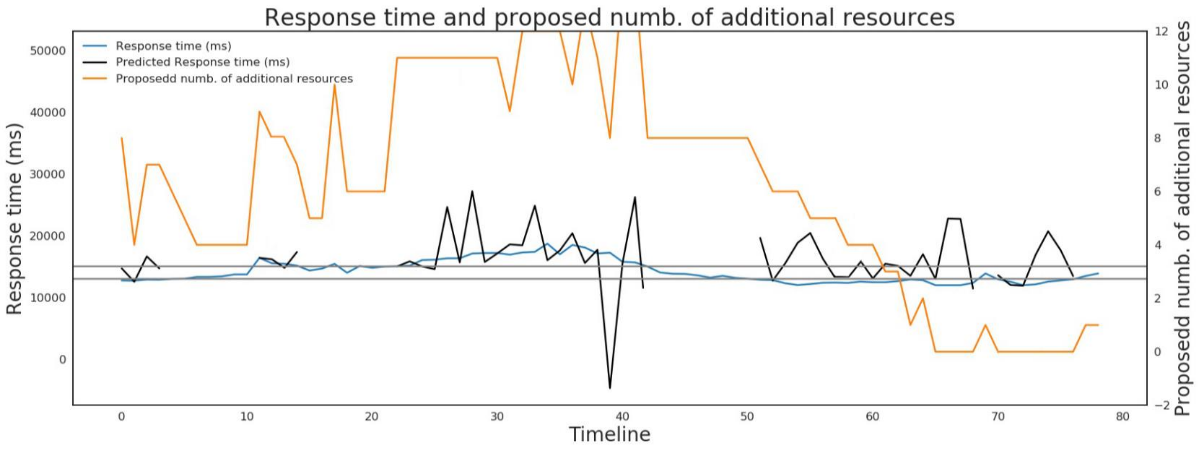


Figure 12 Operation during production phase

The horizontal axis represents the time, where each point stands for one observation. There can be seen 78 observations. There is a double vertical axis chart in the figure. The values on the left vertical axis represent the performance, in our case the response time in sec. The values on the right axis refer to the number of Virtual Machines from 0 to 12. During this measurement we fixed the maximum number of available VMs in 12. The blue line indicates the real – measured – response time. The black line shows the estimated response time and the orange values show the number of VMs proposed by the Optimizer.

After the Optimizer was trained, its task was to keep the response time within the interval represented with the parallel lines (upper 13000 ms, lower 12000 ms) under varying load. It can be seen that when the Response time was above the upper limit the Optimizer tried to define the optimal number of VMs what would result in getting back the response time into the predefined range.

7.6 Benefits and limits

The following list summarizes the most important benefits of the Optimizer:

- It realizes a so-called black-box modelling technique because the complex non-linear correlation between the system metrics and the target performance is revealed by the neural network.
- It is application agnostic, i.e.: the operation of the Optimizer is independent of the actual application. The only necessary condition is to ensure enough number and diverse samples for the machine learning module.
- When the characteristic of the application changes the machine learning module will detect it since it continuously learns or adjusts the weights upon receiving new samples.
- There is no need for the user to thoroughly know its application. The users only determine the metric which best characterizes the required performance or target SLA.

- It uses machine learning. Therefore, it inherits the advantages of machine learning techniques, i.e. it is not needed to reveal complex dependencies between system metrics and target performance metric, this has to be done by the neural network which is hidden from the user.

The following list summarizes the limits of the Optimizer concept:

- The training phase is an important and unavoidable step to train the Optimizer to build its knowledge. This training phase may generate extra costs. However, the relations can be discovered after a very short time and the training can continue during the production phase. Theoretically, it is possible to start the application in a production mode and leave the Optimizer to train itself during this period, however the scaling policy must be extended with the capability of scaling the application without Optimizer advices.
- The Optimizer is sensitive on the collected information i.e. how deeply is the state space of the application has been discovered during the training phase. The optimization will produce better advices when the state space is discovered in more details.
- Defining the thresholds for the target variable requires some experience and measurement. Moreover, the range between the thresholds must be specified carefully to avoid too narrow or too wide ranges.

7.7 Summary and future plans

The Optimizer has been designed (Section 7.2) and implemented (Section 7.3) according to the report above. The Optimizer has been integrated into MiCADO and user interface (Section 7.4) has been added. A sample application has been designed and executed to show the results (Section 7.5) achieved with the Optimizer. During the development and testing, several benefits and limits (Section 7.6) have been discovered.

We identified the following functionality-related improvements:

- support other application types compared to web-based applications
- support for up/downscaling during training with the Optimizer
- support for calculating the intensity of load generation towards the application
- alleviate the intensity of up/downscaling proposed by the Optimizer
- user-defined time interval for sampling
- exporting the knowledge base of Optimizer

There are also several future works related to the Optimizer implementation:

- further develop the neural network
- integrating autogrid search i.e. test and probe new NN layout after every n sample
- instead of continuous learning, learning should be done when the error rate is above certain level
- the algorithm for generating advice can be further improved
- improvements on the field of fault-detection and handling, logging

The integrated Optimizer with its features and capabilities is first released and accessible in MiCADO version 0.8.0. The source code of the Optimizer is stored on GitHub under the micado-scale organization at https://github.com/micado-scale/component-optimizer.

8. Overview of MiCADO developments

During the reporting period i.e. since deliverable D6.3, the project continued the development of MiCADO in every aspect. In this section, we give a short overview of the MiCADO releases, which were produced since D6.3 with a short summary of the most important changes and features.

8.1 MiCADO 0.6.0

This release was a milestone for the project, since this was the first version with full functionalities like scaling, dashboard, submission and monitoring including all the components except the Optimizer.

Several improvements were made to the API of the TOSCA submitter to improve the user experience, including:

- Adjusting all server responses so they contained a uniform set of messages and correct status code, in line with the standard set of HTTP status codes[1].
- Allowing the user to query the submitter for more information on a deployed application. Operations to query available worker nodes and available Docker services were implemented.
- Extending the translation of TOSCA to Docker-Compose to support the complete set of available Docker options
- Improvements to the set of included “helper scripts” which provided the user with examples by way of automating communication with the TOSCA Submitter API
- Bug fixes to the translation of TOSCA to an Occopus resource which previously affected OpenStack and EC2 worker node deployments

A significant update in this version was the appearance of the first security related functionalities like firewall, proxy, authentication and authorization management. For more detailed description on security related components, see Section 8.9.

For the code, please visit
https://github.com/micado-scale/ansible-micado/releases/tag/v0.6.0
For the manual, please visit
https://micado-scale.readthedocs.io/en/0.6.0

8.2 MiCADO 0.6.1

The goal of this release was to create a version for beta testing based on the feedbacks coming from project partners after using MiCADO 0.6.0. As a consequence, most of the updates were bug fixes in every aspect, however some notable new features also appeared:

- Removing checks for containerized applications during TOSCA translation, enabling deployments of standalone virtual machines
- Extending the set of ADT templates included as sample test applications to include a full example of the required TOSCA description for each supported cloud service provider.

[1] https://restfulapi.net/http-status-codes/

- Several smaller bug fixes and stability improvements including a health check on the TOSCA Submitter Docker container, better clean-up of resources by the Occopus adaptor, and the option to disable automatic updates on MiCADO worker nodes.

For the code, please visit
https://github.com/micado-scale/ansible-micado/releases/tag/v0.6.1
For the manual, please visit
https://micado-scale.readthedocs.io/en/0.6.1

8.3 MiCADO 0.7.0

The major version shift to MiCADO v0.7.0 marked a change in the container orchestrator component, from Docker Swarm to Kubernetes[2]. We decided to change the container orchestrator because we found that Swarm was not suitable for a planned application demonstrator. Kubernetes was selected because it met the demonstrator requirements and because of its soaring popularity in the market and uptake by the community. Initially envisaged as a proof-of-concept for demonstrating the modularity of MiCADO, the Kubernetes implementation proved popular and powerful and would become the main upstream of MiCADO.

The goal of the MiCADO v0.7.0 release was to maintain the same functionality, stability and feature-set of the v0.6.1 release, but with the different container orchestrator component – Kubernetes in place of the previously supported Docker Swarm. The focus was on swapping components with a minimal impact to the codebase of other components. Four main areas of MiCADO were identified as targets, which had to change in order to support Kubernetes:

- **Deployment** of Kubernetes to the MiCADO Master and Worker nodes
- Translation to and execution of Kubernetes templates by the **TOSCA Submitter**
- Support scaling of Kubernetes workloads by the **Policy Keeper**
- Replacing the Docker Swarm visualizer on the **MiCADO Dashboard** with a Kubernetes tool

Deployment
Ansible[3], the deployment tool used to deploy the MiCADO Master node, describes deployment and provisioning steps in an Ansible Playbook. Additional steps were added to the playbook, which would install and configure the master node of a basic Kubernetes cluster. MiCADO Worker nodes are configured via cloud-init[4] scripts, to which the appropriate steps were added to ensure these nodes installed Kubernetes and joined the Kubernetes cluster as workers.

TOSCA Submitter
The TOSCA Submitter was designed to support the modularity of MiCADO with pluggable adaptors which would handle the translation steps and execution calls of any desired component. Adding the functionality to translate to and execute Kubernetes templates was achieved by writing a new Kubernetes Adaptor to replace the existing Swarm Adaptor. Inside the new adaptor, calls to the Kubernetes control module, kubectl, managed the creation,

2 https://kubernetes.io
3 https://www.ansible.com/
4 https://cloudinit.readthedocs.io/en/latest/

management and removal of Kubernetes applications. An open-source tool called Kompose[5] was used together with the original translation code of the Docker Swarm adaptor to produce Docker-Compose files as before, and then translate them to Kubernetes templates. While this meant that several native features of Kubernetes were not supported in this version of MiCADO, it fit with our goal of maintaining the feature set of MiCADO v0.6.1.

Policy Keeper
The Policy Keeper was also designed with modularity in mind, and features handlers for interacting with the various pluggable components of MiCADO. A new Kubernetes handler was written to replace the Swarm handler and was responsible for querying and scaling the Kubernetes workloads running on MiCADO.

MiCADO Dashboard
The Docker Swarm visualizer which previously featured as a component of MiCADO accessible via the MiCADO Dashboard was removed and was replaced with the Kubernetes Dashboard. The Kubernetes Dashboard was deployed in a Kubernetes Pod and its endpoint configured in the Zorp firewall and MiCADO Dashboard components.

For the code, please visit
https://github.com/micado-scale/ansible-micado/releases/tag/v0.7.0
For the manual, please visit
https://micado-scale.readthedocs.io/en/0.7.0

8.4 MiCADO 0.7.1

MiCADO v0.7.1 addressed a number of bug fixes and began to extend the functionality of MiCADO to benefit from some of the functionalities of Kubernetes. The minor version upgrade to Kubernetes v1.13.1 introduced incompatibilities with MiCADO v0.7.0, which were critical fixes in the v0.7.1 release. To prevent further such cases and ensure a stable release, Kubernetes versions and component descriptions were locked to a minor version, which would be upgraded as required at each MiCADO release.

MiCADO v0.7.1 also included a new test demonstrator application which provided an example for ingress into an NGINX web server with the Kubernetes NodePort feature. The NGINX demonstrator also served as the first example of an application scaling based on a network-based scaling policy. This policy monitors the number of incoming requests as reported by the web server and scaled the application up or down accordingly to meet the load. Finally, this demonstrator provided an example for the Kubernetes service discovery feature of Prometheus for which support was introduced in this version of MiCADO. This made the configuration of custom metric exporters dynamic, allowing them to be identified and scraped automatically by Prometheus.

For the code, please visit
https://github.com/micado-scale/ansible-micado/releases/tag/v0.7.1
For the manual, please visit
https://micado-scale.readthedocs.io/en/0.7.1

[5] http://kompose.io/

8.5 MiCADO 0.7.2

The MiCADO v0.7.2 release continued the work, which began in v0.7.1 to extend and benefit from a fuller set of features offered by Kubernetes. To this end, the Kompose tool within the Kubernetes Adaptor, which supported a translation from TOSCA to Docker-Compose to Kubernetes Manifest was removed. The translation method in the Kubernetes Adaptor was re-written to support a direct translation from TOSCA to Kubernetes Manifest. This immediately added support for other Kubernetes workloads such as DaemonSets and Jobs, better configuration for exposing Pods through Kubernetes Services such as NodePort and ClusterIP, and support for a range of volume storage options such as HostPath and NFS.

Not long after the release of v0.7.2, the Kubernetes version, which MiCADO was locked to, became misconfigured during the Kubernetes release process. A hotfix was released as MiCADO v0.7.2-rev1, which pointed to a fixed Kubernetes version and introduced a user-option for setting Kubernetes versions in the future.

In this version, several new security components were also introduced like credential Store, Security Policy Manager, Crypto Engine, and so on. Further details can be found in Section 8.9.

For the code, please visit
https://github.com/micado-scale/ansible-micado/releases/tag/v0.7.2
For the manual, please visit
https://micado-scale.readthedocs.io/en/0.7.2

8.6 MiCADO 0.7.3

The MiCADO v0.7.3 release completed the integration with Kubernetes by moving all of the core MiCADO services from static Docker containers to Kubernetes Workloads. This meant that core MiCADO services could now take advantage of Kubernetes scheduling, self-healing and networking benefits. Core MiCADO services could also take advantage of the Kubernetes service discovery feature of Prometheus in the same way user applications had done since v0.7.2. This meant the removal of Consul (previously used for service discovery) from the MiCADO core – one less component and significantly fewer exposed ports on the Master node.

Also included in the v0.7.3 release was the support for multiple sets of worker nodes. This gave users the option to define separate worker nodes in an ADT and lock specific containers to specific worker nodes. Examples of this feature were added to the WordPress and cQueue test demonstrator applications, with each hosting their backend and frontend components on separate sets of worker nodes.

Starting in v0.7.3 it also became possible to build prepared images for the MiCADO Master and Worker nodes. The build process would pull and install all of the dependencies, static configurations and Docker images required for a MiCADO Master or Worker node. The drive image, which had been built, could be saved to the drive library or image repository of the user’s cloud account and then referenced later. Tags were added to the Ansible Playbook in order to select the build or start operations, which would speed up the launch of a MiCADO Master node. Prepared MiCADO Worker images, on the other hand, could be referenced directly in an ADT to greatly increase the deployment and scale-up speed of that application.

The API and engine of the TOSCA Submitter were also a focus of development in MiCADO v0.7.3. Error and log reporting in the Submitter were improved by making log verbosity more appropriate for end users, and by improving the overall uniformity and detail of error messages. Validation was added to the engine to provide users with faster, more detailed information about the validity of an ADT and dry-run methods were added to support developers with the setup and functionality of adaptors.

For the code, please visit
https://github.com/micado-scale/ansible-micado/releases/tag/v0.7.3
For the manual, please visit
https://micado-scale.readthedocs.io/en/0.7.3

8.7 MiCADO 0.8.0

The major version shift to MiCADO v0.8.0 marked a significant change in scaling. On one hand the Optimizer has been introduced to improve the quality and efficiency of scaling. This Optimizer is described in details in Section 7. On the other hand, a new downscaling feature arrived. From this version downscaling can be performed by selecting the virtual machine (i.e. the node) to be downscaled or shutdown.

In this version logging facility operating on the MiCADO master node has been reorganized to support log rotation in order to avoid logging information eating up the available disk space.

With regards to the user-interface of MiCADO, v0.8.0 set out to simplify the ADT. This was accomplished through abstraction in TOSCA. Specific parent types were created, which set default parameters for some common deployments in MiCADO, for example a basic Kubernetes deployment. When an ADT is being written, these parent types can be inherited and the author need only overwrite the subset of parameters specific to their application. These parent types can hide some of the complexity of TOSCA elements such as artifacts and interfaces and reduce the overall length of an ADT. Finally, support has been added to define a custom Kubernetes resource, in-line, in an ADT. These custom resources cannot be supported by MiCADO features such as auto-scaling, optimization, application-level firewalls or Prometheus service discovery, but do add the ability to support applications which are deeply rooted within the Kubernetes ecosystem and require such custom resources.

Also within the ADT, work continues towards adding support for the full set of Kubernetes resources. It is now possible to define multiple containers in a single Pod, as required by applications deployed in the Sidecar pattern. Kubernetes ConfigMaps, which support the definition and assignment of larger variable sets, can also now be defined in the ADT. To decrease the Worker node resource footprint, metric collection has been disabled by default in this version, and is now a user option to be set inside a TOSCA policy within the ADT. When these collectors are deployed, their priority has been increased, protecting them against eviction in the case of resource strain on a node.

For the code, please visit
https://github.com/micado-scale/ansible-micado/releases/tag/v0.8.0
For the manual, please visit
https://micado-scale.readthedocs.io/en/0.8.0

8.8 Terraform integration

8.8.1 Developing a Terraform Adaptor for MiCADO

The original implementation of MiCADO incorporates Occopus as Cloud Orchestrator. However, the modular design principles applied in MiCADO enable changing and replacing any component with a different tool with reasonable effort. Therefore, to demonstrate this feature of MiCADO in case of the Cloud Orchestrator component, a proof of concept prototype has been implemented using Terraform. Besides proving the principles of modularity, the new Terraform adaptor also offers several advantages and added features when compared to the current Occopus based solution. Most importantly, Terraform supports Microsoft Azure that is the desired cloud platform for one of the large scale demonstrators developed by Saker Solutions. Additionally, the Terraform plug-in can also better support multi-application features in future MiCADO releases. This section gives a short overview of Terraform and describes how the Terraform adaptor was developed.

8.8.2 Short overview of Terraform

Terraform is an open-source Infrastructure as a Code (IaaC) software tool developed by Hashicorp. It is written in GO language and is built on a plugin-based architecture. Because of its pluggable architecture, Terraform can be easily extended to support a new cloud provider, network or database. All the major cloud providers like AWS, Openstack, Google cloud, Azure, Alibaba cloud, Oracle cloud, and VMware cloud are supported by Terraform. It also supports configuration management tools (Docker, Kubernetes, Chef, Consul), network providers (Akamai, Cisco, Cloudflare, HTTP, DNS), version control providers (GitHub), monitoring and system management providers and database providers.

The main features of Terraform are:

- tool to build, change or version control infrastructure,
- it is cloud agnostic,
- talks to multiple cloud/infrastructure providers,
- ensures creation and consistency of infrastructure,
- can easily apply incremental changes,
- can preview the changes,
- scales easily,
- helps to destroy infrastructures when needed,
- has a state file for version control.

```
  provider "aws" {
    region     = "us-east-2"
    access_key = "x"
    secret_key = "y"
  }

  resource "aws_instance" "example" {
    ami           = "ami-0d8f6eb4f641ef691"
    instance_type = "t2.micro"
    vpc_security_group_ids = ["sg-9182b4f1",]
    user_data = "${file("${path.module}/template.yaml")}"
    count = "1"
..}
```

Code 3 A sample Terraform configuration file

There are several advantages in using Terraform. Since it uses Infrastructure as a Code we can easily change, share and reuse the infrastructure. The features of Terraform like Execution plans (dry run the configuration) and Resource graph (graph of all resources and their dependencies) help to minimize the errors that can occur. Terraform is well documented and is easily deployable as a container. Multiple applications are also supported by Terraform.

Terraform uses a custom declarative language known as HCL. It can load configuration files with .tf extension or .tf.json extension. A sample configuration file is given in Code 3.

The weak point of Terraform is that it is having only command-line interface. Moreover, it requires a specially formatted configuration file, which is of .tf extension compared to the TOSCA files used by MiCADO. To overcome this, the MiCADO Submitter component needs to interpret the TOSCA ADT that the user submits and pass the necessary parameters needed for Terraform to the terraform_adaptor. The adaptor helps to translate all the details it receives from the submitter into the terraform configuration file, which is then executed in the terraform container.

8.8.3 The Adaptor

Figure 13 demonstrates how Terraform as Cloud Orchestrator fits into the overall MiCADO architecture. The task of Terraform is to create and orchestrate virtual machines. As such, Terraform simply substitutes Occopus in the MiCADO architecture without significantly influencing or requiring major changes from other components.

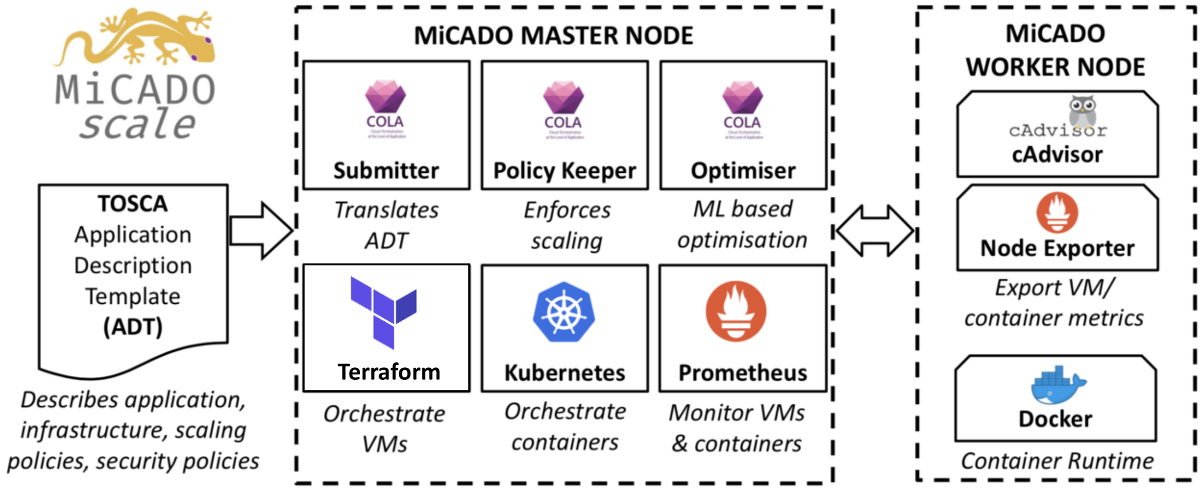


Figure 13 Overall MiCADO architecture and Terraform

When incorporating Terraform in MiCADO, the MiCADO Submitter component needed to be extended with a new Terraform adaptor. The overall architecture of the MiCADO Submitter and the place of the Terraform Adaptor in this architecture are illustrated in Figure 14.

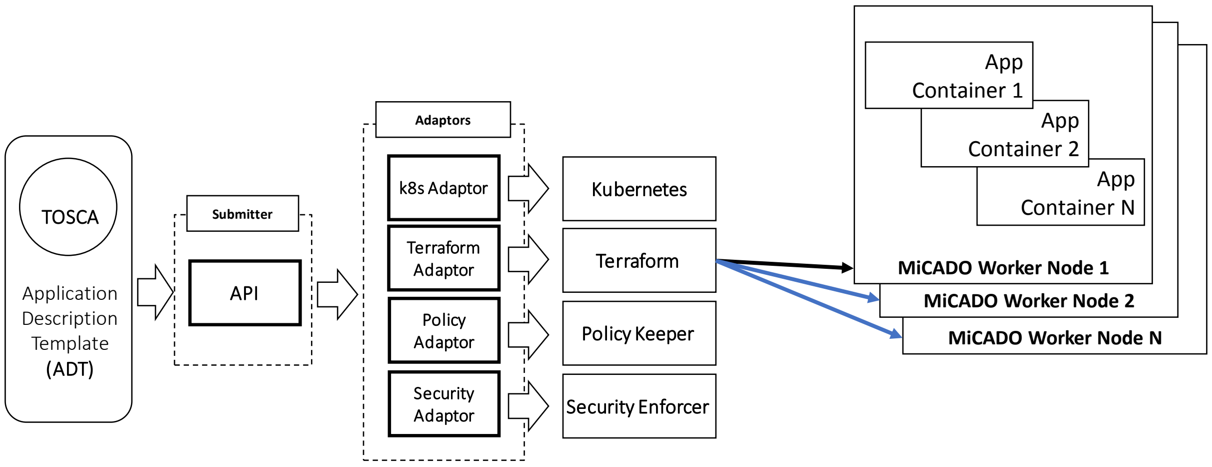


Figure 14 MiCADO Submitter with the new Terraform Adaptor

The newly developed Terraform adaptor helps us to make use of Terraform for cloud orchestration in MiCADO. The infrastructure deployment is controlled conveniently via CLI. Terraform is a single command-line application and Terraform adaptor helps to create a terraform configuration file from all the details provided to it and pass this file to Terraform container in MiCADO master for execution.

The Terraform adaptor inherits from the base_adaptor class. The __init__ method defines the variables that we use during the conversion process. The adaptor is currently capable of handling two types of cloud interfaces: EC2 and OpenStack Nova. Steps are taken to provide support for Azure cloud also. The adaptor uses Kubernetes to execute the Terraform container and run the *terraform init*, *terraform apply* as well as *terraform destroy* commands. In terraform_adaptor class, there are five methods that we are redefining.

- translate() - the adaptor receives the application related information from the TOSCA based ADT, the authentication details corresponding to the cloud provider and the cloud_init config file. Figure 15 shows the data flow during the translation process. The adaptor reads three input files (TOSCA ADT file, auth_data and cloud_init) and then creates a configuration file for terraform in the .tf format. Each cloud provider requires a separate function for writing out the corresponding configuration. These files are stored under a preconfigured storage volume specified in the key_config argument of the translate method.

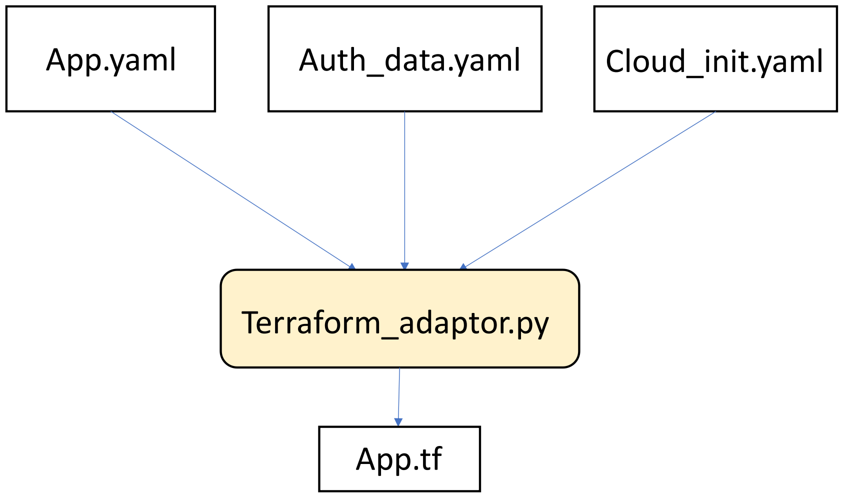


Figure 15 Translation done by Terraform_adaptor.py

- execute() - the adaptor executes the *terraform init* and *terraform apply* command inside the Terraform container. *Terraform init* is used to initialize the module and download all the required plugins. After the import succeeds, the adaptor executes the *terraform apply* command inside the Terraform container to start the building process of the MiCADO Worker infrastructure. Terraform builds the MiCADO Worker infrastructure based on the configuration files in the module.
- undeploy() - the adaptor performs an infrastructure destroy operation using the *terraform destroy* command. Terraform then performs a graceful removal of the all the virtual machines (including pre allocated temporary cloud storage volumes) implementing a MiCADO Worker.
- update() - the adaptor first generates a new configuration file as described in the translate method. It compares the newly generated files with the ones currently running under MiCADO. If there is any change in the configuration, the container executes the new configuration using the *terraform apply* command. Terraform decides whether to update the virtual machines without shutting down any of them or to destroy and redeploy the virtual machines based on the changes to be made to the infrastructure.
- clean_up() - the adaptor gets the APPLICATION ID from the Submitter engine and removes the associated configuration files generated during the translate method.

8.8.4 Future work

The Terraform adaptor and its integration with MiCADO is a proof of concept at the time of writing this report. However, it is now part of the official MiCADO development roadmap and it is planned to be fully incorporated in a release (anticipated for October/November 2019).

In order to achieve this full integration, the following tasks need to be completed. First, the Azure connection will need to be implemented enabling MiCADO to directly utilise MS Azure resources via Terraform. As it was mentioned before, this connection is required by one of the COLA demonstrators, SAKER. Secondly, the developed Terraform Adaptor will be fully integrated with the MiCADO Architecture.

This integration is planned in a way that the current Occopus-based cloud orchestration in MiCADO can still be selected at deployment time. The system administrator can decide which cloud orchestrator is to be deployed before deployment happens. This decision can be made based on the targeted clouds for the application. For example, if CloudSigma or CloudBroker are required as target clouds then Occopus is needed. However, if the target cloud is Azure then the recommended cloud orchestrator will be Terraform.

8.9 Security related functionalities

There are 7 user-visible security functions implemented by WP7 within the scope of the MiCADO framework to enhance the overall security of the product. These provide services to the end user and developers that implement industry-standard best practices, while minimizing the need of user-supplied configuration. The security user experience is crucial in terms of the actual security level of MiCADO deployments, as end users will only use functions, that are easily enabled and configured.

Besides the features that provide direct user value, there are also 3 support functions that aid to implement internal security and maintainability of the MiCADO platform. It is a design decision in the COLA project to build every aspect of the software in a pluggable architecture. By the nature of security enables, they tightly integrate with the actual implementations of the various MiCADO functions. The high-level support functions have been implemented in a standalone component, that is responsible for executing security workflows while hiding the actual implementation from other MiCADO components. This way the security enablers may be changed or replaced without touching other pieces of the framework relying on implementation details.

The deliverables “D7.2 MiCADO security architecture specification”, “D7.3 Design of application level security classification formats and principles” and “D7.4 Security policy formats specification” describe the designed security enablers in detail, also providing an open specification for implementing the various security functions.

Out of the identified and designed security features, the enablers that provide the most efficient security functions – based on feedback from WP8 use-case partners – have been selected for implementation.

The following user-visible functions have been implemented within the scope of the COLA project:

- Verifying OS images that are running the application containers
- Storing cloud credentials in an encrypted way
- Packet filtering firewall for the MiCADO master node
- Encryption, authentication and authorization on web access to the MiCADO master node
- Secure communication between MiCADO master and worker nodes
- Application-level firewalling for MiCADO applications
- Application-related secret handling and distribution

The above functions are addressing security problems, that are present in all current infrastructure, container and application orchestration solutions and have not been solved in any of the mainstream cloud solutions. While some cloud providers supply tools for workarounds, these all present an administrative overhead and are often neglected by cloud administrators.

8.9.1 Verifying OS images that are running the application

containers

This security enabler (see A1 of Appendix 1) aims to provide means for administrators to verify if their deployed operating system image has been modified by 3rd parties. As administrators are handing off responsibility to cloud providers for running their images, MiCADO would like to provide a way to enhance this trust by implementing detective controls. The current implementation is included in version 0.7.2 but needs to be simplified to be of use to end-users.

8.9.2 Storing cloud credentials in an encrypted way

This security enabler (see A2 of Appendix 1) aims to provide a means for administrators to store sensitive cloud credential information an encrypted way and audit access to the credentials. While the information to access the credentials needs to be available for MiCADO to drive automatic workflow execution, “data-at-rest” encryption and audit logging increase the security of credentials storage and provide detective controls for unauthorized access. The current implementation is based on the industry-standard open source credential store Hashicorp Vault and included in version 0.7.4. Integration with other MiCADO component needs to be improved.

8.9.3 Packet filtering firewall for the MiCADO master node

This security enabler (see A3 of Appendix 1) aims to provide a means for administrators to restrict network-level access to the MiCADO master node in a cloud-agnostic way. This area is painfully neglected in most mainstream cloud solutions and while there are proprietary tools on each cloud platform, they cannot be maintained in a uniform way. This enabler provides both preventive and detective controls by implementing network access control and audit logging for incoming connections. The current implementation is based on Netfilter and included in version 0.6.0.

8.9.4 Encryption, authentication and authorization on web access to the MiCADO master node

This security enabler (see A4 of Appendix 1) aims to provide a means for administrators to ensure that all web management connections to the master node are properly encrypted in transit and make sure that only authorized users have access to MiCADO resources, while providing easy access for machine accounts to perform automated tasks. This enabler is implemented as proprietary tools in the portfolio of some cloud providers, sometimes also dubbed as “Identity-aware proxy”, but no cloud-agnostic solution is included in mainstream cloud solutions. The current implementation is based on Balasys Zorp and included in version 0.6.0.

8.9.5 Secure communication between MiCADO master and worker nodes

This security enabler (see A5 of Appendix 1) aims to provide a means for administrators to ensure that all network traffic between the master and worker nodes is encrypted in-transit and access to sensitive master node resources is restricted. This enabler makes it possible to separate management and production functions into separate runtime environments, while providing the same level of security for transit connections as if they were running in an environment controlled entirely by the end user. This removes the need of trusting the cloud provider not to listen in on internal connections. The current implementation is based in the StrongSWAN IPSEC daemon and included in version 0.7.2.

8.9.6 Application-level firewalling for MiCADO applications

This security enabler (see A6 of Appendix 1) aims to provide a means for administrators to provide application-level filtering and enforcement capabilities for exposed MiCADO applications based on the NetworkSecurityPolicy descriptions in the ADT. This enabler brings in the networking and network security aspects of the application integrated into the same application descriptor while providing a cloud-agnostic implementation. The current implementation is based on Balasys Zorp and the Kubernetes Ingress Controller framework and is included in version 0.8.0.

8.9.7 Application-related secret handling and distribution

This security enabler (see A7 of Appendix 1) aims to provide a means for administrators to ensure that all application-related sensitive information is stored and transmitted in a secure way when being distributed within the MiCADO framework. Current mainstream solutions do not use encryption by default. By implementing encryption not only application-related information, but also systems secrets are protected against eavesdropping and modification. The current implementation is based on etcd and included in version 0.7.4.

As stated before, several support functions have also been implemented to facilitate pluggability and overall systems security:

- Security workflow director and common interface to security components (B1 of Appendix 1)
- Safely store and verify user accounts for accessing the MiCADO framework (B2 of Appendix 1)
- Providing cryptographic functions to other components within MiCADO (B3 of Appendix 1)

These functions are highly technical, and their specification can be found in deliverable “D7.3 Design of application level security classification formats and principles”.

The current status and technical information of the enablers is described in Appendix 1.

8.10 Integration with CloudBroker

To ensure the sustainability of MiCADO, the project integrated it with CloudBroker Platform (CBP).

Having analyzed the demand coming from the use-cases and experiments of the COLA project, at the beginning of 2019 CloudBroker introduced Infrastructure Visual Screen (IVS) that allows a user to manage and launch infrastructures of different complexities, assemble components and connections between them. The IVS is described in detail in D4.4. The core idea was to allow a user to benefit from both UX-friendly setup of components of IVS and performance optimization mechanics of MiCADO by making IVS components MiCADO-driven. This integration was a cooperation involving SZTAKI, CB, and UoW teams. The overall integration concept is presented on Figure 16.

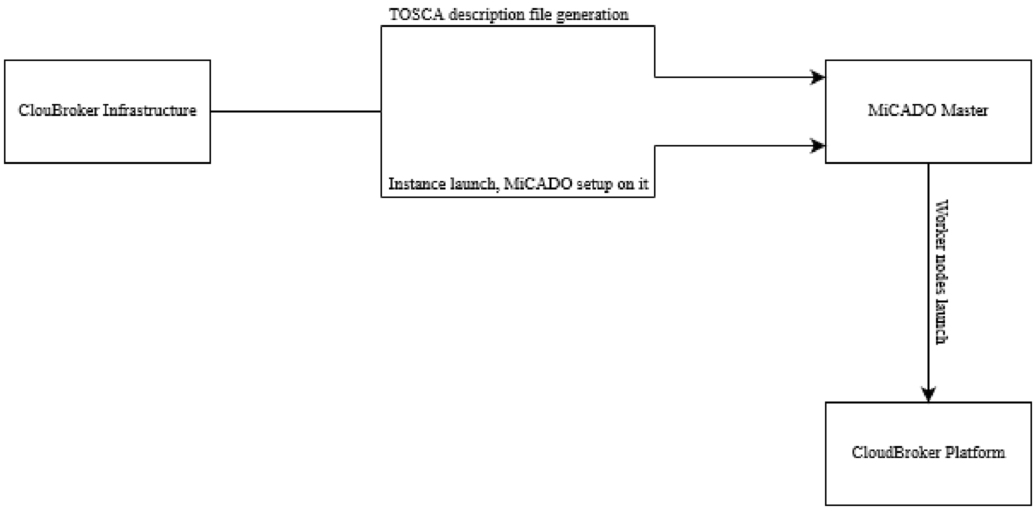
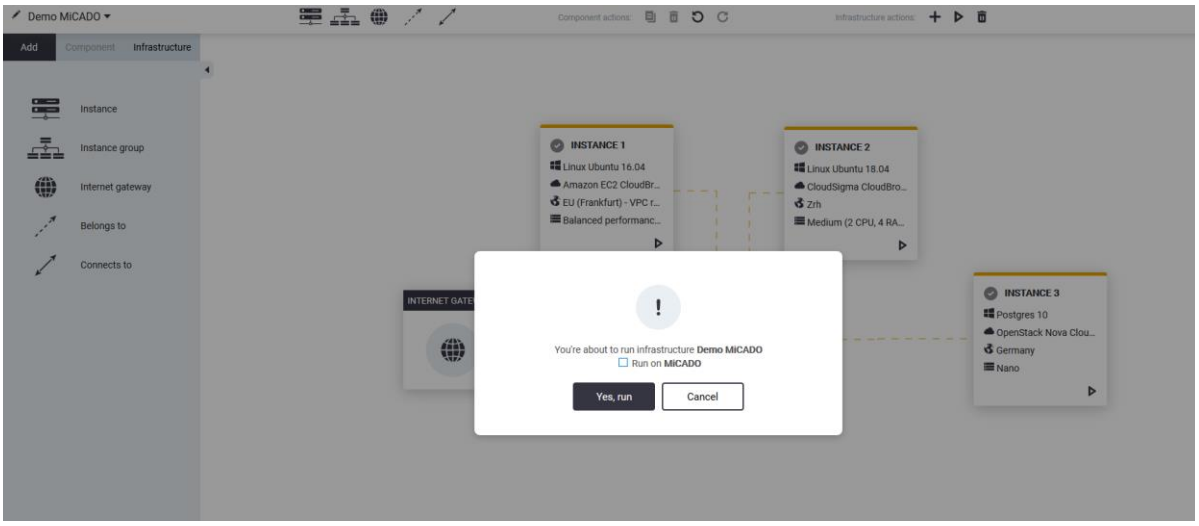

TOSCA description file generation
ClouBroker Infrastructure
MiCADO Master
Instance launch, MiCADO setup on it
Worker nodes launch
CloudBroker Platform

Figure 16 Concept of CBP and MiCADO integration

It was agreed that a separate deployment with the latest MiCADO version would be created on CBP. Another action was an update of the application description file to make it compatible with the CB infrastructures description (i.e. a corresponding field for infrastructure component ID was added). Based on these steps, infrastructure created with IVS could be run on MiCADO. When a user starts the infrastructure, two actions are initiated. First, CB starts an instance and installs MiCADO on it to launch a MiCADO master node. Second, CB generates a TOSCA description file and submits it to the MiCADO master node, which launches worker nodes for the specified infrastructure's instances.

By M31 the basic integration has been completed. As a result, it is possible to launch infrastructure created with CB IVS tools on MiCADO. This option is presented on Figure 17 below.



You're about to run infrastructure **Demo MiCADO**
☐ Run on **MiCADO**

Figure 17 Running the infrastructure on MiCADO scale using CB IVS

In the latest stage of integration, CB and UoW teams provided a functionality to add/edit master nodes on the fly in order to avoid any interruptions in the performance of the infrastructures.

By the end of the project it is expected to provide advanced policies to manage infrastructure's behavior (e.g. to define preconditions for instance scaling) and to develop a functionality to launch infrastructure with instances based on Docker images.

An Ansible playbook for MiCADO installation causes a need to start a new instance with MiCADO deployment for every launch, which affects the deployment time. Optimization of this process is scheduled for the future versions, together with automating MiCADO's deployment update whenever a new version is released.

9. Current status and conclusion

The main goal of this deliverable was to report on the achievements in task T6.4 in WP6 of the COLA project. The task defined the prototype and documentation of the price/performance optimization.

The results related to the Optimization service has been described in Section 7. The Optimizer has been designed, a prototype has been developed and integrated to MiCADO. Finally, a detailed documentation has been provided in this report.

Beyond the Optimizer service, numerous developments have been performed by the WP6 team, where the results (with one exception) have been released during the period of T6.4. The developments focused on improvements of MiCADO both on the area of functionality and user experience therefore they are considered as important steps ahead in the life of MiCADO.

With Task 6.4, WP6 finishes as well as the project. By the end of the project, MiCADO became a fully functional scaling framework with pluggable architecture. During the project we demonstrated the pluggable architecture by releasing MiCADO with Kubernetes instead of Swarm, and by implementing a version (released soon) with Terraform replacing Occopus.

During the developments, the communication of the team was realized with Slack at micado-scale.slack.com, for the source code management GitHub was used at github.com/micado-scale, and documentation was hosted and maintained at micado-scale.readthedocs.io.

The latest released version of MiCADO at the end of the project is MiCADO 0.8.0. However, there are plans for continuing the maintenance and development of MiCADO after the project.

10. 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] Kubernetes, https://kubernetes.io/

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

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

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

[10] T. Lorido-botr, “Auto-scaling Techniques for Elastic Applications in Cloud Environments,” pp. 1–44, 2012.

[11] T. L. J. Miguel-alonso and J. A. Lozano, “A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments,” pp. 559–592, 2014.

[12] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, I. Truck, Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow, in: ICAS 2011, Venice, Italy, 2011, pp. 67–74.

[13] R. Bahati, M. Bauer, Towards adaptive policy-based management, in: NOMS 2010, Osaka, Japan, 2010, pp. 511–518.

[14] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut, “Using machine learning for black-box autoscaling,” *2016 7th Int. Green Sustain. Comput. Conf. IGSC 2016*, 2017.

[15] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi, “MLscale: A machine learning based application-agnostic autoscaler,” *Sustain. Comput. Informatics Syst.*, 2017.

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

[17] SCIKIT-learn. https://scikit-learn.org/stable/

[18] Pandas, https://pandas.pydata.org/

[19] NumPy, https://numpy.org/

[20] MiCADO documentation site, https://micado-scale.readthedocs.io/en/latest/

[21] Apache JMeter, https://jmeter.apache.org/

Appendix 1: MiCADO Security Enablers Implementation Reference Guide

A. User-visible Features

A1. Verifying OS images that are running the application containers

- Technical name: Image Integrity Verifier
- Rationale: ensure that the image running in the cloud infrastructure has not been modified by 3rd parties
- Current status: integrated and running in MiCADO, but no mechanism to call the verification method, only works on Intel CPUs
- Initial availability: v0.7.2
- 3rd Party: no
- Improvement plans:
  - provide an automated mechanism for image or file verification of the underlying operating system (WP6 assistance needed)
  - simplify verification
  - make implementation CPU-agnostic
  - research whether cloud providers expose a way of retrieving image hashes, VM orchestrator should provide this function to the IIVR
  - integrate the verifier with the VM orchestrator
  - provide CLI tool (micadoctl) to add new hashes to the image database
- GitHub repository: https://github.com/micado-scale/component-iivr
- Docker Hub repository: https://hub.docker.com/r/micado/iivr/
- Docker Hub autobuild: yes
- Travis: no
- Configuration: none
- Relation to other components:
  - Deployment via ansible-micado, affected files:
    - https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/tasks/pull-docker-images.yml
    - https://github.com/micado-scale/ansible-micado/blob/master/roles/start-micado-master/templates/micado/micado-manifest.yml.j2
  - CLI configuration via security-policy-manager:
    - https://github.com/micado-scale/component-security-policy-manager/blob/master/bin/micadoctl

A2. Storing cloud credentials in an encrypted way

- Technical name: Credential Store
- Rationale: ensure that cloud credentials are not accessible in plain text when ‘at rest’ and access to the credentials are audited
- Current status: integrated and running in MiCADO, but cloud orchestrator still uses plain text to cache the credentials
- Initial availability: v0.7.2, reimplemented in v0.7.4
- 3rd Party: yes, Hashicorp Vault
- Improvement plans:
  - extend VM orchestrator to retrieve the credentials on a per-use bases and discard safely after use (WP6 assistance needed)

        - provide CLI tool to change credentials after deployment
    - GitHub repository: https://github.com/hashicorp/vault
    - Docker Hub repository: https://hub.docker.com/_/vault
    - Docker Hub autobuild: yes
    - Travis: CircleCI
    - Configuration: https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/files/vault/vault.hcl
    - Relation to other components:
        - Deployment via ansible-micado, affected files:
            - https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/tasks/pull-docker-images.yml
            - https://github.com/micado-scale/ansible-micado/blob/master/roles/start-micado-master/templates/micado/micado-manifest.yml.j2
        - The API functions are called by the Security Policy Manager component that hides the actual implementation of storing the cloud credentials, details in https://github.com/micado-scale/component-security-policy-manager/blob/master/app/secrets.py

A3. Packet filtering firewall for the MiCADO master node

- Technical name: iptables
- Rationale: ensure that only those services are exposed to the outside world that are needed for the operation of MiCADO
- Current status: integrated and running
- Initial availability: v0.6.0
- 3rd Party: yes, iptables
- Improvement plans:
    - limit the accessibility of MiCADO management ports to a set of pre-defined IP addresses / subnets
- GitHub repository: https://github.com/micado-scale/ansible-micado
- Docker Hub repository: N/A
- Docker Hub autobuild: N/A
- Travis: N/A
- Configuration:
    - https://github.com/micado-scale/ansible-micado/blob/master/micado-master.yml#L98
    - SystemD service files: https://github.com/micado-scale/ansible-micado/tree/master/roles/build-micado-master/files/iptables
    - config templates: https://github.com/micado-scale/ansible-micado/tree/master/roles/start-micado-master/templates/iptables
- Relation to other components:
    - Deployment via ansible-micado, see above
    - The set of rules needs to be re-evaluated every time a component is upgraded or replaced in MiCADO to ensure that their operation is not affected by firewalling

A4. Encryption, authentication and authorization on web access to the MiCADO master node

- Technical name: Zorp Master Node

- Rationale: ensure that all web management connections to the master node are properly encrypted in transit and make sure that only authorized users have access to MiCADO resources. Supports form-based authentication for users and HTTP Basic authentication for machine accounts.
- Current status: integrated and running in MiCADO
- Initial availability: v0.6.0
- 3rd Party: yes, Zorp GPL
- Improvement plans:
  - Certificate generation via letsencrypt
- GitHub repository: https://github.com/Balasys/zorp
- Docker Hub repository: https://hub.docker.com/r/micado/zorpgpl
- Docker Hub autobuild: no
- Travis: https://travis-ci.org/Balasys/zorp
- Configuration:
  - Web listening port configuration:
    - https://github.com/micado-scale/ansible-micado/blob/develop/micado-master.yml#L73
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/start-micado-master/templates/micado/micado-manifest.yml.j2#L923
  - Certificate generation:
    - https://github.com/micado-scale/ansible-micado/blob/master/sample-credentials-micado.yml
    - https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/files/zorp/zorp-entrypoint.sh
  - Authentication web form:
    - https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/files/zorp/authform.html
  - Policy template:
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/start-micado-master/templates/zorp/policy.py.j2
- Relation to other components:
  - Deployment via ansible-micado, affected files:
    - https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/tasks/pull-docker-images.yml
    - https://github.com/micado-scale/ansible-micado/blob/master/roles/start-micado-master/templates/micado/micado-manifest.yml.j2
  - Performs authorization by checking users against the Credential Manager and fetches group membership as well

A5. Secure communication between MiCADO master and worker nodes

- Technical name: IPSEC
- Rationale: ensure that all network traffic between the MiCADO master and worker are encrypted authenticated.
- Current status: integrated and running in MiCADO
- Initial availability: 0.7.2
- 3rd Party: yes, StrongSwan
- Improvement plans:
  - Encrypted overlay network to protect worker-worker communication
- GitHub repository: https://git.strongswan.org/?p=strongswan.git;a=summary
- Docker Hub repository: N/A

- Docker Hub autobuild: N/A
- Travis: https://travis-ci.org/strongswan/strongswan/
- Configuration:
  - Master node certificate generation:
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/start-micado-master/tasks/micado-start.yml
  - Master node configuration:
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/start-micado-master/templates/ipsec/ipsec.conf
  - Installation on master node:
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/build-micado-master/tasks/other-install.yml
  - Worker node certificate generation:
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/start-micado-master/templates/worker_node/cloud_init_worker.yaml.j2#L69
  - Worker node configuration:
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/start-micado-master/templates/worker_node/cloud_init_worker.yaml.j2#L45
  - Installation on worker node:
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/start-micado-master/templates/worker_node/cloud_init_worker.yaml.j2#L117
- Relation to other components:
  - Deployment via ansible-micado, see above
  - Certificate generation is done via the Security Policy Manager
  - If not operational, worker node cannot join the k8s cluster and/or provide metrics to Prometheus

A6. Application-level firewalling for MiCADO applications

- Technical name: Zorp Ingress Director
- Rationale: provide application-level filtering and enforcement capabilities for exposed MiCADO applications based on the NetworkSecurityPolicy descriptions in the ADT
- Current status: under development
- Initial availability: v0.7.4 / v0.7.5
- 3rd Party: yes, Zorp Ingress Director
- GitHub repository: https://github.com/Balasys/zorp-ingress-controller
- Docker Hub repository: https://hub.docker.com/r/balasys/zorp-ingress
- Docker Hub autobuild: via Travis
- Travis: https://travis-ci.org/Balasys/zorp-ingress-controller
- Configuration:
  - Worker node configuration:
    - https://github.com/micado-scale/ansible-micado/blob/develop/roles/start-micado-master/templates/worker_node/cloud_init_worker.yaml.j2#L45
  - TOSCA Submitter adaptor:
    - https://github.com/micado-scale/component_submitter/pull/186/commits/55a2505737d7b53525af9b50e55901de9b6afcac#diff-716a6c6ec30ef95b7f0423a09c495c88
- Relation to other components:
  - Part of the TOSCA Submitter as a Security Enforcer Adaptor
  - Part of the TOSCA Submitter as part of the K8s Adaptor
  - Applies configuration through the K8s API

- Reads configuration from the K8s API
- Configuration is generated based on the TOSCA descriptor available at: https://github.com/micado-scale/tosca/

A7. Application-related secret handling and distribution

- Technical name: K8s Secret
- Rationale: provide a way for TOSCA users to distribute secrets in a secure way to only affected application containers
- Current status: integrated and running in MiCADO, no mechanism is provided to read the secrets on the application container side
- Initial availability: v0.7.4
- 3rd Party: yes, k8s Secret API
- Improvement plans:
  - provide a mechanism to read the secrets on the application container side (WP6 assistance required)
  - avoid storing the secret in plain text in the ADT (symmetric encryption?)
- GitHub repository:
- Docker Hub repository:
- Docker Hub autobuild:
- Travis:
- Configuration:
  - TOSCA Submitter adaptor:
    - https://github.com/micado-scale/component_submitter/pull/186/commits/55a2505737d7b53525af9b50e55901de9b6afcac#diff-716a6c6ec30ef95b7f0423a09c495c88
- Relation to other components:
  - Part of the TOSCA Submitter as a Security Enforcer Adaptor
  - Stores secrets within the K8s Secret API
  - Configuration is generated based on the TOSCA descriptor available at: https://github.com/micado-scale/tosca/

B. Support functions

B1. Providing cryptographic functions to other components within MiCADO

- Technical name: Crypto Engine
- Rationale: provide abstraction for crypto functions so that the underlying implementation can be changed easily
- Current status: integrated and running in MiCADO, but no other components use it
- Initial availability: v0.7.2
- 3rd Party: no
- Improvement plans:
  - convert the component into a python library instead of the current REST-based application
  - identify MiCADO components that should use the CryptoEngine for cryptography, currently none identified
- GitHub repository: https://github.com/micado-scale/component-crypto-engine
- Docker Hub repository: https://hub.docker.com/r/micado/crypto-engine
- Docker Hub autobuild: no
- Travis: no
- Configuration: none
- Relation to other components:
  - Deployment via ansible-micado, affected files:
    - ➢ https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/tasks/pull-docker-images.yml
    - ➢ https://github.com/micado-scale/ansible-micado/blob/master/roles/start-micado-master/templates/micado/micado-manifest.yml.j2

B2. Security workflow director and common interface to security components

- Technical name: Security Policy Manager
- Rationale: provide a common interface to all MiCADO components to access security-related components and functions and run security-related workflows
- Current status: integrated and running in MiCADO, used for several infrastructure-related functions
- Initial availability: v0.7.2
- 3rd Party: no
- Improvement plans:
  - upgrade library for interacting with Hashicorp Vault (https://github.com/hvac/hvac) a version that is compatible with the latest Vault API
  - provide and interface to Credential Manager
- GitHub repository: https://github.com/micado-scale/component-security-policy-manager
- Docker Hub repository: https://hub.docker.com/r/micado/security-policy-manager
- Docker Hub autobuild: yes
- Travis: https://travis-ci.org/micado-scale/component-security-policy-manager
- Configuration: N/A
- Relation to other components:

- Deployment via ansible-micado, affected files:
  - https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/tasks/pull-docker-images.yml
  - https://github.com/micado-scale/ansible-micado/blob/master/roles/start-micado-master/templates/micado/micado-manifest.yml.j2
- Provides implementation for storing cloud credentials
- Provides implementation for storing application secrets
- Provides a PKI to be used by worker-master secure communication
- Provides an interface to CryptoEngine
- Provides an interface to Image Integrity Verifier

B3. Safely store and verify user accounts for accessing the MiCADO framework

- Technical name: Credential Manager
- Rationale: provide a lightweight REST-based interface for storing and verifying user accounts to use with the MiCADO web interface
- Current status: integrated and running in MiCADO, used by Zorp
- Initial availability: v0.6.0
- 3rd Party: no
- Improvement plans:
  - incorporate the flask-users library for simplification of the implementation
  - support multiple roles besides the currently supported user/admin distinction
  - avoid using plain-text credentials for initial provisioning (use an API call from the ansible playbook to add credentials via the API without saving them to file)
  - run automated tests on Travis
- GitHub repository: https://github.com/micado-scale/component-credential-manager
- Docker Hub repository: https://hub.docker.com/r/micado/credential-manager
- Docker Hub autobuild: yes
- Travis: no
- Configuration: https://github.com/micado-scale/ansible-micado/blob/master/sample-credentials-micado.yml
- Relation to other components:
  - Deployment via ansible-micado, affected files:
    - https://github.com/micado-scale/ansible-micado/blob/master/roles/build-micado-master/tasks/pull-docker-images.yml
    - https://github.com/micado-scale/ansible-micado/blob/master/roles/start-micado-master/templates/micado/micado-manifest.yml.j2
  - The Security Policy Manager contains the micadoctl CLI tool, that can change users and passwords in Credential Manager
  - The Zorp component uses Credential Manager directly to authenticate and authorize web access to MiCADO. A direct connection is used to avoid DoS attacks on the Security Policy Manager