

D6.2 Prototype and documentation of the monitoring service



Project Acronym: COLA

Project Number: 731574

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

Topic: ICT-06-2016 Cloud Computing

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

Start date of project: 01/01/2017

Duration: 30 months

Deliverable:

D6.2 Prototype and documentation of the monitoring service

Due date of deliverable: 30/09/2017

Actual submission date: 30/09/2017

WPL: Peter Kacsuk

Dissemination Level: PU

Version: V1.7

1. Table of Contents

1. Table of Contents	2
2. List of Figures and Tables	3
3. Status, Change History and Glossary.....	4
4. Introduction	6
5. The MiCADO generic architecture framework	7
6. Designing the MiCADO Orchestration Layer	9
6.1 General design principles	9
6.2 Specific design principles related to COLA use cases	11
7. System and service monitoring tools	13
7.1 Investigated tools.....	13
7.2 Selection process	14
8. Prometheus as the selected monitoring tool of MiCADO	16
9. Implementation of MiCADO and its monitoring subsystem	22
9.1 Implementation of the monitoring subsystem	22
9.2 Implementation of MiCADO V3	25
10. Performance evaluation	29
11. Current status and conclusion.....	32
12. References	33

2. List of Figures and Tables

Figures

Figure 1 MiCADO generic architecture framework	7
Figure 2 Architecture of the MiCADO Orchestration Layer	9
Figure 3 Main Prometheus configuration file	16
Figure 4 Built in graph page in Prometheus showing CPU usage on nodes	17
Figure 5 Target nodes in Prometheus	18
Figure 6 Alert definition configuration (Prometheus.rules)	19
Figure 7 Alert templating example used in MICADO	20
Figure 8 Alert manager conf. and alert query that gives back the firing alerts	21
Figure 9 Modular design of the monitoring subsystem	22
Figure 10. Architecture of MiCADO V3	26
Figure 11 Example configuration for MiCADO worker infrastructure in case of CloudSigma	27
Figure 12 Infrastructure and node creation/destroy	29
Figure 13 Resource optimization, on both scale up and down events with MiCADO	30

Tables

Table 1 Status Change History	4
Table 2 Deliverable Change History	5
Table 3 Glossary	5
Table 4 Comparison of the investigated monitoring tools	15

3. Status, Change History and Glossary

Status:	Name:	Date:	Signature:
Draft:	Botond Rakoczi	12/09/17	Rakoczi Botond
Reviewed:	Antonis Michalas	25/09/17	Antonis Michalas
Approved:	Tamas Kiss	30/09/17	Tamas Kiss

Table 1 Status Change History

Version	Date	Pages	Author(s)	Modification
V0.1	03/09	ALL	Botond Rakoczi	Empty Skeleton
V0.2	04/09	Section 5	Tamas Kiss, Botond Rakoczi	MiCADO generic architecture
V0.3	04/09	Section 6	Jozsef Kovacs Botond Rakoczi	Design plan of MiCADO
V0.4	05/09	Section 6	Jozsef Kovacs Botond Rakoczi	Cloud Orchestration and Occopus
V0.5	05/09	Section 7	Botond Rakoczi	System and service Monitoring
V0.6	05/09	Section 8	Botond Rakoczi	Prometheus
V0.7	06/09	Section 9	Botond Rakoczi	Implementation of Prometheus in MiCADO
V0.8	09/09	Section 10,11	Botond Rakoczi Jozsef Kovacs	Current Status Conclusion
V0.9	11/09	Section 6, 7, 8, 10, 11	Tamas Kiss	Small corrections
V1.0	13/09	Section 5, 6, 7, 8, 10	Gabor Terstyanszky	Small corrections
V1.1	15/09	Section 8	Botond Rakoczi	Add more explanation for the selection
V1.2	16/09	Section 4	Tamas Kiss Botond Rakoczi	Introduction of WP6 deliverables
V1.3	17/09	Section 8	Botond Rakoczi	Going-over

D6.2 Prototype and documentation of the monitoring service

V1.4	19/09	Section 7	Jozsef Kovacs	Adding Section 10 on performance
V1.5	19/09	Section 10	Jozsef Kovacs	Improving Section 9.2
V1.6	20/09	Section 11	Jozsef Kovacs	Improving Section 11
V1.7	20/09	All	Jozsef Kovacs	Formatting the entire document

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
DNS	Doman Name Service
NAT	Network Address Translation
CA	Certificate Authority
TLS	Transport Layer Security
LDAP	Lightweight Directory Access Protocol
PC	Personal Computer
VM	Virtual Machine
RDBtool	Round Robin Database Tool
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) monitoring subsystem, focusing on monitoring both cloud resources and container services. MiCADO is a compound service providing automatic scaling and orchestration of microservices, as well as of cloud resources required for executing the services. The aim of MiCADO is to implement this double orchestration in an intelligent way following the policies specified by the user together with the infrastructure description. This overall functionality, realized by the MiCADO framework is described in the following chapters, specifically concentrating on its monitoring subsystem.

The rest of this deliverable is organized as follows: In Section 5 we overview the generic architecture of the MiCADO framework, while in Section 6 the design plan of the MiCADO Orchestration Layer is introduced. System and service monitoring tools are compared in Section 7, and the selected monitoring tool, Prometheus is detailed in Section 8. Implementation details on the monitoring integration as well as on the latest MiCADO version are written in Section 9. The deliverable also contains performance evaluation in Section 10, and concludes the results in Section 11.

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 a lot of storage while another could be CPU intensive. Cloud execution offers the possibility to optimize resource allocation and thus resource cost dynamically. The alternative would be to allocate a monolithic infrastructure, the size of which is large enough to be sufficient to cover peak performance as well. The requirement for peak performance happens rarely, therefore allocated resources of the monolithic infrastructures remain unused in most of the time.

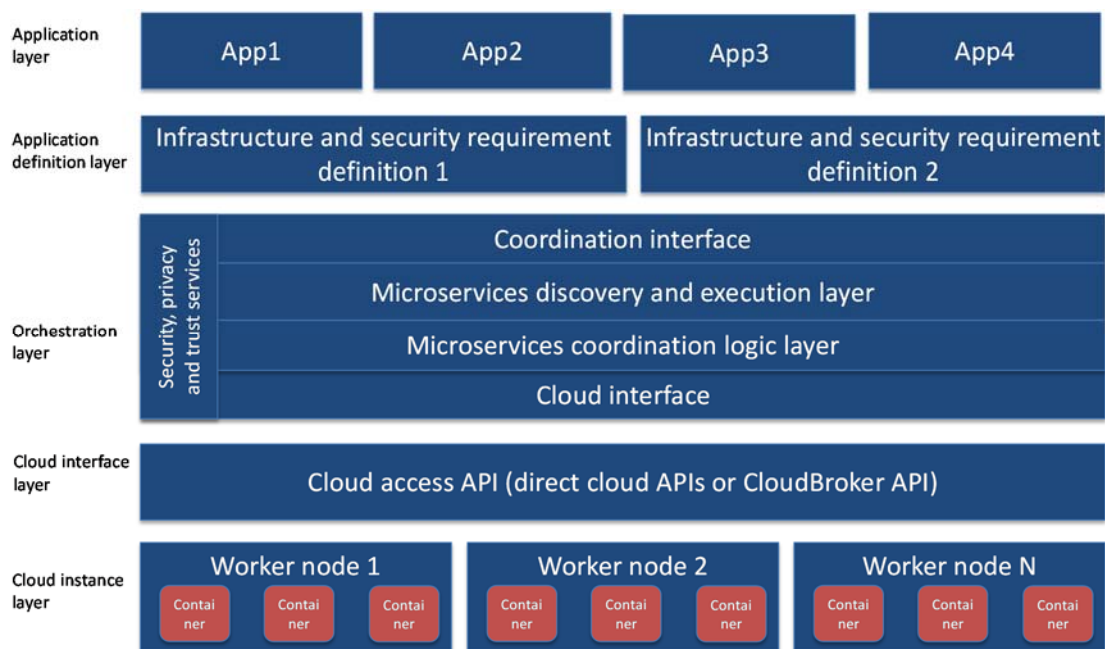


Figure 1 MiCADO generic architecture framework

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

1. **Application layer.** Application layer contains actual application code and data described by the application definition layer (layer 2) to function in such a way that a desired functionality is reached. For example, this layer could populate a 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.

D6.2 Prototype and documentation of the monitoring service

3. **Orchestration layer.** This layer is divided into four horizontal and one vertical sub-layers. The horizontal sub-layers are:
 - a. **Coordination interface API.** This sub-layer provides access to the orchestration control and decouples the orchestration layer from the application definition layer. This set of APIs enables application developers to utilize the dynamic orchestration capabilities of the underlying layer and supports the convenient development of dynamically and automatically scalable cloud-based applications by embedding these API calls into application code.
 - b. **Microservices discovery and execution layer.** This sub-layer manages the execution of microservices and keeps track of services running. Execution management combines both start-up and shut down of microservices. Service management gathers information about currently running services, such as service name, IP address and port where the service is reachable and optional service tags to help service coordination.
 - c. **Microservices coordination logic.** With large infrastructures and 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** is responsible for abstracting cloud access from layers above. Cloud access APIs can be complex interfaces, as they typically cater for a large number of services provided by the cloud provider. On the other hand, the microservices execution and coordination logic layers (see b and c) only need to shut down and start instances. Abstracting this to a cloud interface API simplifies implementation of aforementioned layers, and if new Cloud access APIs are implemented, only this layer needs to change.
 - e. **Security, privacy and trust services:** The orchestration layer also includes a vertical sub-layer that deals with security, privacy and trust related services for advanced security policy management. These services span multiple levels of the orchestration layer, as it is illustrated in Figure 1. The main aim is to shield 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 will be later used by the special purpose security policy enforcement services to enforce the security policies at orchestration level.
4. **Cloud interface layer.** This layer provides means 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. This layer typically represents state-of-the-art of cloud technology provided by various public or private cloud providers.

6. Designing the MiCADO Orchestration Layer

6.1 General design principles

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

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

MiCADO essentially forms a cluster which is able to dynamically allocate, attach, or detach and release cloud resources for optimizing the resource usage during executing the submitted microservices. MiCADO consists of two main logical components: **Master node** and **Worker nodes**. **Master node** is the head of the cluster performing the collection of information on microservices, the calculation of optimized resource usage, the decision making, and the realization of decisions related to handling of resources and scheduling of microservices. **Worker nodes** are volatile components, representing execution environments for the microservices, i.e. they are executing the actual microservices. **Worker nodes** are continuously allocated/released based on the dynamically changing requirements of the running microservices. Once a new worker node is allocated and attached to the cluster, the master node utilizes its resources by allocating microservices on it.

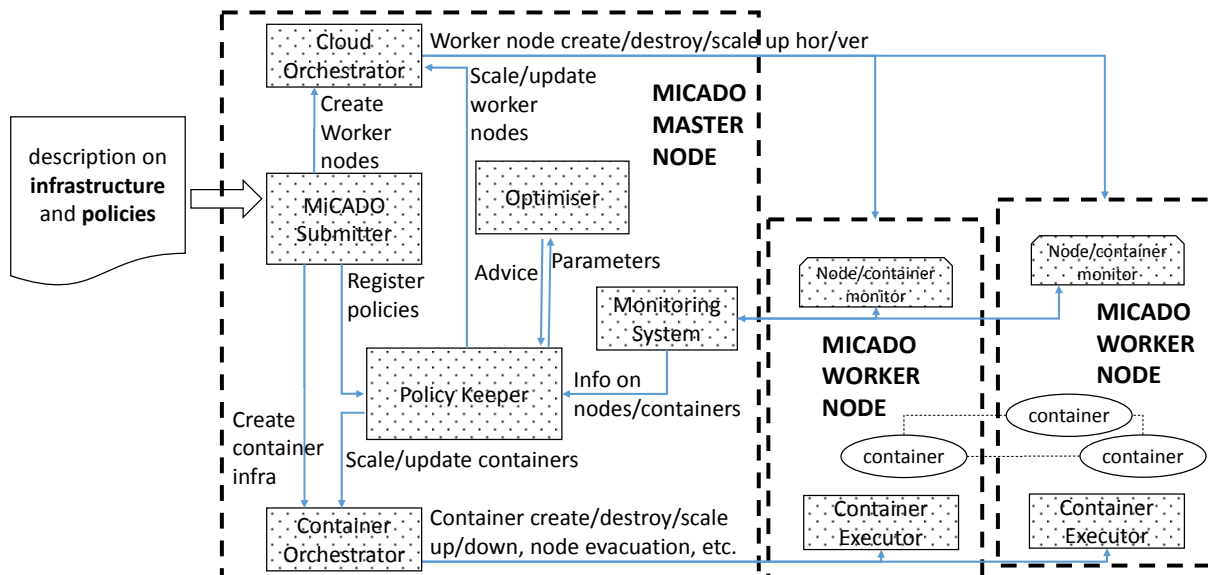


Figure 2 Architecture of the MiCADO Orchestration Layer

D6.2 Prototype and documentation of the monitoring service

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

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

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

- **Node/container monitor** component is responsible for measuring the load of the resources and the resource usage of the container services. The measured attributes are then provided to the Monitoring system running on the Master Node.
- **Container executor** is responsible for starting, executing and destroying containers upon requests from the Container Orchestrator on the Master node.
- **Container** components are realizing the user services defined in the (container) infrastructure description submitted through the MiCADO submitter on the Master node.

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

D6.2 Prototype and documentation of the monitoring service

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

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

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

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

6.2 Specific design principles related to COLA use cases

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

- *System requirements* relate to the underlying operating system, which is Ubuntu in most of the use cases, except for the Saker Solutions use case, where Windows is the base operating system. There are various alternatives in executing Windows applications in MiCADO which are currently being investigated. A promising alternative is using Windows emulator software on Linux. However, adopting a native Windows solution is also a possibility with or without containers.
- *Data requirements* for the use cases are relatively low. However, using an external database is a good alternative for data intensive applications, if it is required.
- *Performance requirements* are planned to be fulfilled by applying the policies and utilizing the auto-scaling mechanism. Container applications will be automatically scaled-up together with worker nodes to deliver additional computing resources.
- *Security requirements* will be mainly addressed by WP7. However, a set of default security mechanisms, such as VPN, encrypted channels and certain firewall settings, are going to be part of the MiCADO initial setup. Moreover, both private and public clouds will be supported.
- *Other requirements* in D8.1 include *data protection, robustness, and quality of service*. Even though these extra requirements are not considered as part of the main MiCADO architecture they will be investigated in a later state of the project.



D6.2 Prototype and documentation of the monitoring service

In order to implement the architecture on Figure 2, tools realizing the different components must be investigated carefully and must be integrated together, keeping in mind the possibility to replace them with an alternate solution later if it is required.

7. System and service monitoring tools

In the MiCADO framework, a monitoring subsystem is needed to perform system and service check, and provide valid information about their status to other components. The cloud orchestration in MiCADO will depend on, and execute scaling events based on the provided information. The major focus of this deliverable is the selection and integration of such monitoring tool to the generic MiCADO Orchestration Layer. In this chapter, we will investigate different monitoring tools and select one to be integrated into MiCADO. It must be emphasized that while this selection is important and directly influences the MiCADO implementation of the COLA project, due to the modular design the selected monitoring tool can be relatively easily replaced later with another tool, if such change is required.

7.1 Investigated tools

In this section, we give a short overview of relevant monitoring tools. We have investigated several commercial products (e.g. Zabbix, Ganglia, Nagios, Icinga, and Prometheus) in this field. We shortly describe each of them to clarify their advantages and/or disadvantages.

Zabbix [3]

Zabbix is a widely utilized monitoring tool that performs well especially when considering speed. It is well-optimized to provide fast and efficient monitoring. It is a traditional monitoring system (concerning its functionalities) and widely used for years now. Users can interact with Zabbix in a user-friendly web GUI. It uses a traditional relational database management system (RDBMS), which performs poorly compared to NOSQL databases and on time-series data. Zabbix can monitor HTTP, FTP, SSH, POP3, SMTP, SNMP, MySQL on a high abstraction level and even at the level of services. With third party applications even more options are available. On the one hand, Zabbix targets enterprise companies with large number of servers and scalability. On the other hand, it does not support basic/core features such as Oracle, Exchange, and Active Directory. There is a wide range of available configurations that could be applied to separate target servers. However, for MiCADO it is not necessary. As SZTAKI has been using Zabbix for years now, project partners have significant experience in relation to how much effort it takes to install and maintain it. It has an alerting subsystem as well for sending email notifications and trigger events such as server failure. It also features custom alert scripts and templating where you can add your own shell scripts to be executed in case of an event. Its powerful templating system is considered as an important feature that makes Zabbix a good candidate for MiCADO. However, configuration and maintenance is a very time-consuming task. Additionally, developers should be experts in regular expressions to write efficient alerting templates which is not the case in other approaches such as Nagios. Zabbix also lacks significant user community where you can ask questions, while for example Nagios and Ganglia offer a much larger community.

Ganglia [4]

Ganglia uses a Round Robin database (RDB). Instead of traditional RDBMS-based format where the size of the data is significantly compacted, RDB stores the data in a time series format. This is a useful feature for creating archives from the database when it gets unused. In Ganglia, instead of the server pulling the data, it uses a push model, which means that nodes send their data to the Ganglia server on a regular basis. Its user interface is difficult to use and the whole software lacks user friendly configurations, meaning configuration is done via text files.

D6.2 Prototype and documentation of the monitoring service

Nagios [5]

Nagios can be considered as a more advanced version of Zabbix (introduced earlier). It can monitor the same metrics as Zabbix and uses RDBMS. However, communication between the components is done by SSH. Installing Nagios is less complicated and faster than Zabbix. Compared to Zabbix, the major disadvantage of Nagios is that the user interface is read only and configuring the software is only possible via configuration files. This factor makes the interaction with the software hard and rigid. It features a large number of forks (OpsView, OP5, Centreon, Icinga, Naemon, Shinken), and numerous third-party apps as well. Out of the box the software itself offers a more simplified installation than in case of Zabbix or Prometheus, however with third-party extensions it can be configured for the user needs. In MiCADO, from the beginning we used cloud-init for installing software on the virtual machines. Deployment of third-party applications for Nagios through command-line instructions is not well-supported and difficult.

Prometheus [6]

Prometheus is a relatively new monitoring tool that uses the latest technologies. It uses its own database and querying language called PromQL. It stores data in a time series format, which saves space by compressing the data and allows sophisticated queries. Compared to traditional databases it is more efficient since it stores its database in the memory. It uses a pull model, which means that in certain time intervals it requests data from the monitored nodes. This process involves running a monitoring agent on every node, which calls exporters as well as a node discovery service that reveals the concrete list of participating nodes to Prometheus. Since Prometheus supports a large number of node discovery servers (e.g. consul, Azure, dns, ec2), it can be easily modified in case an administrator decides to switch to a different one. Prometheus can pull data in different forms, which makes it flexible, and it also supports user defined metrics and third-party apps. In addition, Prometheus GUI is considered as the most user-friendly GUI compared to the rest of the existing tools. Furthermore, it supports a wide variety of operating systems, services and micro-services. In MiCADO, the applications and services are executed as containers, which makes Prometheus a good candidate.

Icinga [7]

Icinga is a Nagios port that has two main differences compared to Nagios. The first is that it uses one global API instead of multiple ones. This makes it easier to implement in enterprise systems and third-party applications as well. The second advantage is that users can create their own rules for performing availability checks. This is a great feature for handling complex infrastructure components. It gives the advantage of flexibility for server OS diversity, and its configuration compared to Nagios is less complex. Nevertheless, updating Icinga following new releases proves to be a challenge, as it is not backward compatible.

7.2 Selection process

After studying the concurrent monitoring systems, a comparison of these was made. The monitoring tools have been compared based on eight criteria:

- Speed;
- Ease of use;
- Compatibility;
- Database type;

D6.2 Prototype and documentation of the monitoring service

- Alerting system;
- Community;
- Complexity;
- Third-party apps.

Based on these criteria and experiences introduced in the previous chapter, we have created a summary in Table 4.

	Speed	Database	Complexity	Ease of use	Compatibility	Alerting System	Community	Third-Party Apps
Zabbix					X	X		
Ganglia	X				X		X	
Nagios					X		X	X
Prometheus	X	X	X	X	X	X	X	
Icinga	X		X	X	X			X

Table 4 Comparison of the investigated monitoring tools

As it can be seen, none of the tools meets all the MiCADO criteria. In the one hand we can see that most of the investigated tools were created for a particular purpose and finding one that fits all the requirements is not easy. On the other hand, due to the modular design of MiCADO, it is assured that components, including the monitoring service, can be replaced with a different product relatively easily, if necessary. We also investigated the modularity of the tools above to find out which monitoring software gives the best opportunity to be changed/replaced in the future. In this aspect, Prometheus and Icinga are the best choices. Finally, Prometheus was selected for the current MiCADO implementation as it supports better alerting system. However, as it was emphasized above, this tool can be substituted in future MiCADO implementations if a better candidate emerges.

The current requirements raised by MiCADO (i.e. monitoring the capacity of a virtual machine, monitoring the load generated by container components, defining general rules based on the measured attributes, and generating user-defined notifications based on alerts) are all supported by Prometheus. Currently, Prometheus is widely accepted by infrastructure developers. Additionally, using Prometheus' inner codebase - written in GO [8] language, supporting web applications and the newest technologies - decreases latencies when using data from multiple sources. We also examined using GO for some parts of MiCADO and used it to implement one of the modules in the monitoring subsystem. In the next chapter, we will see that we use almost every feature of our selected monitoring tool, instead of using a portion of functionalities from a much larger and more complex tool. This simplifies the configuration and the development of MiCADO in advance. We also introduce a self-developed module to extend its execution functionality and alerting system.

D6.2 Prototype and documentation of the monitoring service

8. Prometheus as the selected monitoring tool of MiCADO

This chapter gives an overview of the most significant characteristics of Prometheus, including node discovery, node exporters, alerts, alert generator, alert manager and alert executor.

Prometheus is a cluster monitoring system which helps instrumenting, collecting, querying, and alerting. It nourishes a target database and gets the node list from the service discovery. After collecting the node list it uses a pull model to acquire information about the nodes (metrics) and stores this information in its database.

Prometheus stores information in a multi-dimensional data model, where time series are identified by a metric name connected to key-value pairs. This way, most of the metrics fit in the memory and it has the advantages that in-memory databases offer. Querying is done by a flexible language which allows slicing and dicing the metrics to generate tables, graphs and alerts. These queries can be executed easily by its built-in Graph Page (as shown in Figure 5) which creates graphs automatically for deeper understanding. Based on the alerts the communication to the Cloud Orchestrator component (depicted in Figure 2) can be implemented in order to perform the given scaling action. (Please, note that Cloud Orchestrator is currently realized by Occopus [9] as stated in Deliverable D6.1 of the COLA project).

In Prometheus, we can configure every parameter in a single configuration file making the deployment phase simpler. Figure 3 illustrates an example for configuring Prometheus inside the MiCADO framework. This file helps us to connect the subcomponents of the monitoring system. First, we specify which node discovery agent is used (Consul) and give its address to Prometheus. Next, we need to connect the alert manager service to Prometheus (port 9093). We can add our own alerts by specifying the "rule_files" parameter. Finally, tuning of the scrape configuration, which specifies the duration of how often we should pull metrics from the nodes is required.

```
rule_files:
  - 'prometheus.rules' # name of the rule file
scrape_configs:
  - job_name: cluster_monitoring
    scrape_interval: 10s #how often pull metrics
    consul_sd_configs: #node discovery config
      - server: '172.31.0.5:8500'
        datacenter: application
        services: ['lb_cluster', 'worker_cluster', 'app_docker_cluster']
    relabel_configs: #rewrite label before scrape
      - source_labels: ['__meta_consul_service']
        regex: '(.*)'
        target_label: 'job'
        replacement: '$1'
      - source_labels: ['__meta_consul_service']
        regex: '(.*)'
        target_label: 'group'
        replacement: '$1'
  alerting:
    alertmanagers: # target alert manager configuration
      - scheme: http
        static_configs:
          - targets:
              - "172.31.0.3:9093"
```

Figure 3 Main Prometheus configuration file

D6.2 Prototype and documentation of the monitoring service

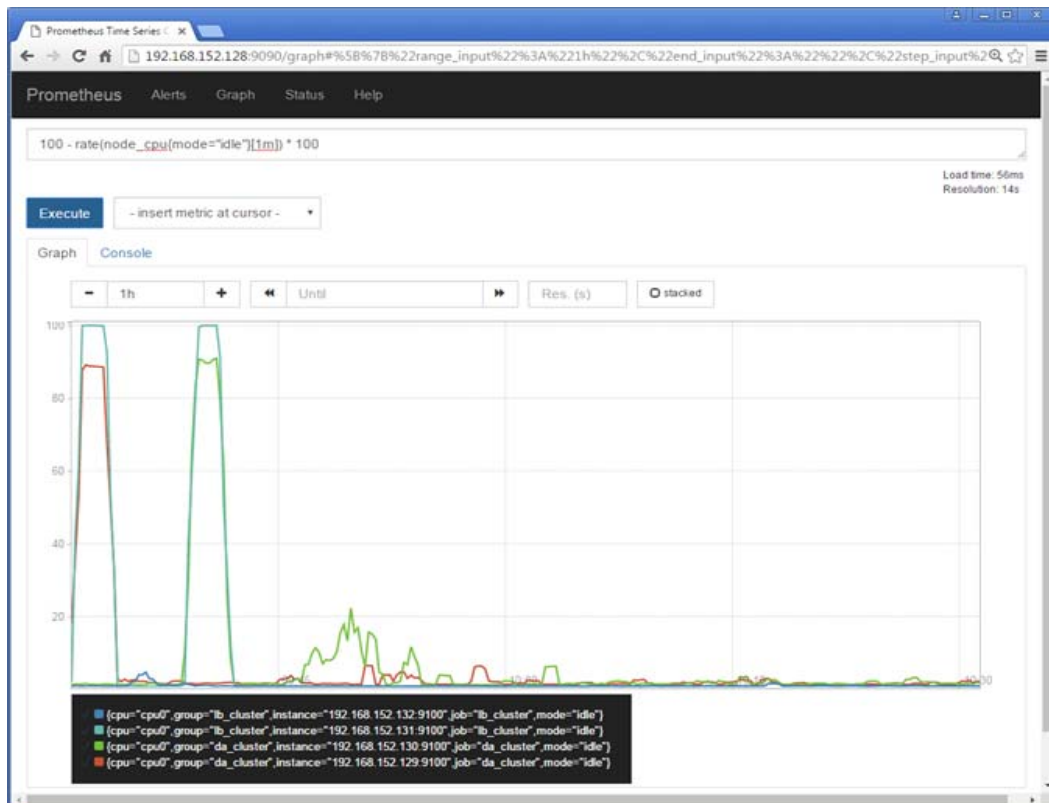


Figure 4 Built in graph page in Prometheus showing CPU usage on nodes

Node discovery

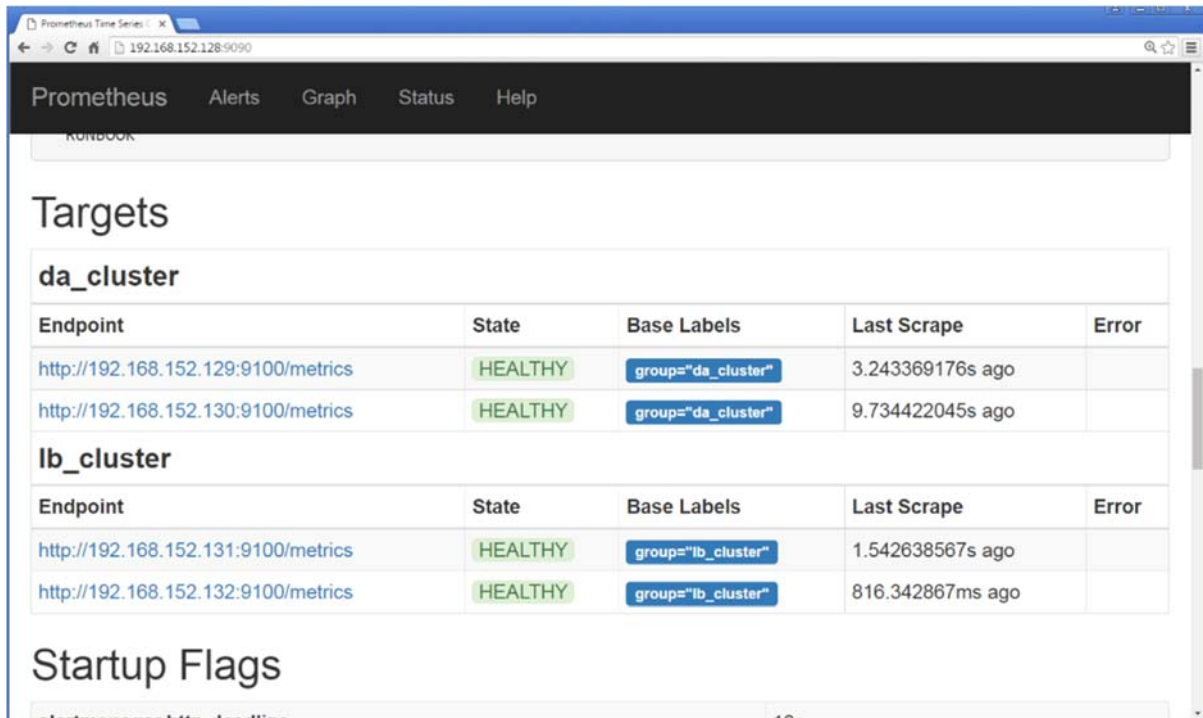
Prometheus uses a pull model and frequently asks the monitored nodes for their data. Since the location of the nodes is unknown, a node discovery service for Prometheus needs to be configured (Figure 4). A node discovery service is useful for basic health checks on target machines and for discovering new services in a specified IP range. In MiCADO we use Consul for node discovery. We install a Consul agent on every node of the worker node cluster and those will automatically connect to the Consul server after they booted up. On the Consul server we can see these nodes with their appropriate IP addresses. We also make some health checks on the nodes whether they are reachable, and also on the running MiCADO services (such as Prometheus, Consul, Occopus, alert generator, alert manager, alert executor, and Swarm) whether they are reachable on their GUI for the end user.

Configuring Prometheus node discovery to specify only the different clusters that we wish to monitor is considered as of paramount importance. We can see that three clusters will be monitored by Consul in the Prometheus configuration file. These are the following:

```
-services: ['lb_cluster', 'worker_cluster', 'app_docker_cluster']
```

If we check the web page of Prometheus on the “Target” page we will see the different nodes and their health status (Figure 5).

D6.2 Prototype and documentation of the monitoring service



The screenshot shows the Prometheus web interface with the 'Targets' tab selected. It displays two clusters of target nodes. The 'da_cluster' section contains two nodes, both in a 'HEALTHY' state. The 'lb_cluster' section contains two nodes, also in a 'HEALTHY' state. Each node entry includes its endpoint URL, state, base labels, last scrape time, and any error messages.

da_cluster				
Endpoint	State	Base Labels	Last Scrape	Error
http://192.168.152.129:9100/metrics	HEALTHY	group="da_cluster"	3.243369176s ago	
http://192.168.152.130:9100/metrics	HEALTHY	group="da_cluster"	9.734422045s ago	
lb_cluster				
Endpoint	State	Base Labels	Last Scrape	Error
http://192.168.152.131:9100/metrics	HEALTHY	group="lb_cluster"	1.542638567s ago	
http://192.168.152.132:9100/metrics	HEALTHY	group="lb_cluster"	816.342867ms ago	

Figure 5 Target nodes in Prometheus

Node exporters

In order to collect the metrics from the target machines we need an agent running on every node. There are different collectors or so called exporters for Prometheus for different purposes. In MiCADO we use two different types of collectors for monitoring two types of components: virtual machines and docker containers.

The first one is the “node_exporter” which collects monitoring data from the Virtual Machine itself. Natively, it supports Linux so a wide range of different metrics are offered. By default this agent monitors contrack, CPU, diskstats, entropy, filefd, filesystem, loadavg, mdadm, meminfo, netdev, netstat, sockstat, stat, textfile, time, uname, vmstat, but user defined metrics can be also defined (e.g. http calls/ service).

The second one we used is Cadvisor [10] – a third-party application developed by Google to support monitoring microservices and Docker containers. Currently, the collected data in this level is limited. However, simple network and system information such as CPU, RAM, process IDs are already accessible. We implemented container monitoring with the help of Cadvisor to allow scaling events in the microservice layer before we scale the number of virtual machines. By doing so, we managed to receive faster feedback in the control circle.

These two agents are running on every node and are listening on different port numbers to make them accessible from the web. Our Consul server registers these ports and passes it to Prometheus to pull data.

Alerts

Alerts in Prometheus fire every time when a given query is fulfilled in the monitored data. Alerts have different states as follows:

Work Package: WP6

D6.2 Prototype and documentation of the monitoring service

- Off: The query is not fulfilled
- Pending: The query is fulfilled and it waits for a given time
- Firing: Pending alerts after a given time become firing

First an alert enters into pending state and over time it becomes firing. For example, we can query the CPU load in a given cluster, then create two alerts for under and overloaded state. Prometheus includes well-defined aggregation functions to use (abs, absent, bottom, ceil, changes, clamp_max, clamp_min, count_scalar, delta, sqrt, time, topkvector, <aggregation> over_time).

The following alert shows how we can calculate the sum of the CPU loads in the worker node cluster in 60 seconds bases. Based on this information, we create an alert that fires after 2 minutes if the alert remains active, every time when the utilization of the CPU load exceeds 50%. On the label fields, we have to fill out at least the IDs that provide information to Occopus and Swarm for the orchestration.

```
worker_CPU_utilization=100-(avg (rate (node_CPU {group="worker_cluster",
mode="idle"}[60s])) * 100)

worker_ram_utilization=(sum(node_memory_MemFree{job="worker_cluster"})/sum(node_mem
ory_MemTotal{job="worker_cluster"})) * 100

worker_hdd_utilization=sum(node_filesystem_free{job="worker_cluster",mountpoint="/"
, device="rootfs"})/sum(node_filesystem_size{job="worker_cluster",mountpoint="/"
, device="rootfs"}) *100

ALERT worker_overloaded
  IF worker_CPU_utilization > 80
  FOR 2m
  LABELS {alert="overloaded", cluster="worker_cluster", node="{{node_id}}",
infra_id= " {{infra_id}}"}

  ANNOTATIONS {
    summary = "Worker node cluster overloaded", description = "cluster average CPU
utilization is above 80%"}
```

Figure 6 Alert definition configuration (Prometheus.rules)

Alerts can be easily templated and could be built up from more criteria. Automated replacement of the “infra_id” and “node_id” placeholders with real values is supported and performed by Occopus.

Alert generator

Using alerts in connection with the VM metrics is not enough in MiCADO. Our collected data from Cadvisor should provide enough information to make alerts based on the Docker container states as well. Also, when multiple applications are running next to each other, multiple alert definitions should exist. To support this methodology, we had to implement an alert generating service to MiCADO. The service generates application specific alerts in Prometheus and deletes them when the application is deleted. To get the running applications,

D6.2 Prototype and documentation of the monitoring service

it asks Docker Swarm for the list, parses the Json answer message, and generates the alerts. When we got the Json from that information and a template, we can create the new alerts at the end of the process. In Figure 7 we can see a part of the code and the template itself.

```
#create new rules
if [ -n "$name" ]; then
if grep -q $name "/etc/prometheus/prometheus.rules"; then
    echo "already exists"
else
    echo "create new rule named $name "

    echo "ALERT $name"_overloaded"
    IF
avg(rate(container_cpu_usage_seconds_total{container_label_com_docker_swarm_service_name=$namequotes }[30s]))*100 > $cpulimit
    FOR 30s
    LABELS {alert=""overloaded"", type=""docker"", application=$namequotes}
    ANNOTATIONS {
    summary = ""overloaded""}
fi
```

Figure 7 Alert templating example used in MICADO

Alert manager

Prometheus offers a tool to manage alerts, collect and assort them. When an alert fires, Prometheus sends out the alert to the Alert manager service which service then runs next to Prometheus and takes care of problems indicated by the alerts (e.g. fast scale up, slow scale down, and rapidly firing alerts). It has a tree structure where alerts are rooted to receivers (leaves) that specify what to do with a given alert. In MiCADO, we send out these alerts to an executor service (described later in the 'Alert Executor' section) which communicates with Occopus. To avoid rapidly firing alerts, the alert manager can wait some time before sending them out (group wait). It also repeats firing alerts toward the executor service. Firing alerts can be checked by querying them, as shown in Figure 8. Alert manager supports different notification methods like email, Pagerduty¹, pushover, Webhook. However, it does not support script execution and for this manner alerts has to be sent out to a Webhook receiver², and on the other side a service has to parse it and do the scripting part.

¹ Pagerduty: User defined web calls, which can be scheduled and can query system statuses.

² webhook: They are user-defined HTTP call backs providing information for 3rd party apps.

D6.2 Prototype and documentation of the monitoring service

```

route:
receiver: 'alert_executor'
  group_wait: 10s
  group_interval: 20s
  repeat_interval: 3m
  group_by: [cluster,
alertname]
  receivers:
  - name: 'shell_executor'
  webhook_configs:
  - url: http://localhost:9095

```



```

{"type": "vector", "value": [{"metric":
{"__name__": "ALERTS", "alert": "underloaded", "alertname": "lb_underl
oaded", "alertstate": "firing", "cluster": "lb_cluster"}, "value": "1",
"timestamp": "1464345376.312"}], "version": "1"}

```

Figure 8 Alert manager conf. and alert query that gives back the firing alerts

In traditional auto-scaling, multi-tier applications have a common problem [11]. The monitoring system with percentage based queries without the aggregation of scaling event, performs badly because it scales up too early [12] (decisions take place immediately after a given alert fires). Scalable infrastructures like this one help to solve this problem and stay as a base for extreme scaling situations for the future needs. To achieve this, alert manager first collects the firing alerts and only over a given time will it send them out in a summarized and sorted form to the alert executor.

Alert Executor

This service is not part of Prometheus so we had to develop it. This service makes Prometheus able to maintain scaling activities with REST calls. It runs next to Prometheus and gets the alerts from the alert manager (see Figure 8). It first parses the incoming Webhook call, makes labels accessible (AMX_ALERT_<n>_label), and then looks for the label called “alert_firing” and if it equals to “underloaded” or “overloaded” it calls Occopus or Swarm. For scaling, Occopus needs an infrastructure id and a node id, while Swarm needs the current application specification in a Json format. Requirements for Occopus can be templated into the alert definitions when we generate the alerts, but for application specification we need a much more complex logic for updating the configuration and increasing/decreasing the number of Docker containers/applications. The program code of the alert executor component can be found in the MiCADO code repository [13].

9. Implementation of MiCADO and its monitoring subsystem

In this chapter we will give an overview how the components have been implemented in the MiCADO framework. In Figure 9 the monitoring subsystem component's design plan and its connections are depicted. After discussing the implementation of the monitoring subsystem we also describe how the latest MiCADO (V3) works as one entity.

Please note that the implementation of MiCADO has already been outlined in D6.1 Section 10. The architecture and details of MiCADO versions V0, V1 and V2 have been described there, also referring to MiCADO V3 as work in progress. In this deliverable we focus on the detailed implementation of the monitoring subsystem of MiCADO, and also outline the completed implementation of MiCADO V3.

9.1 Implementation of the monitoring subsystem

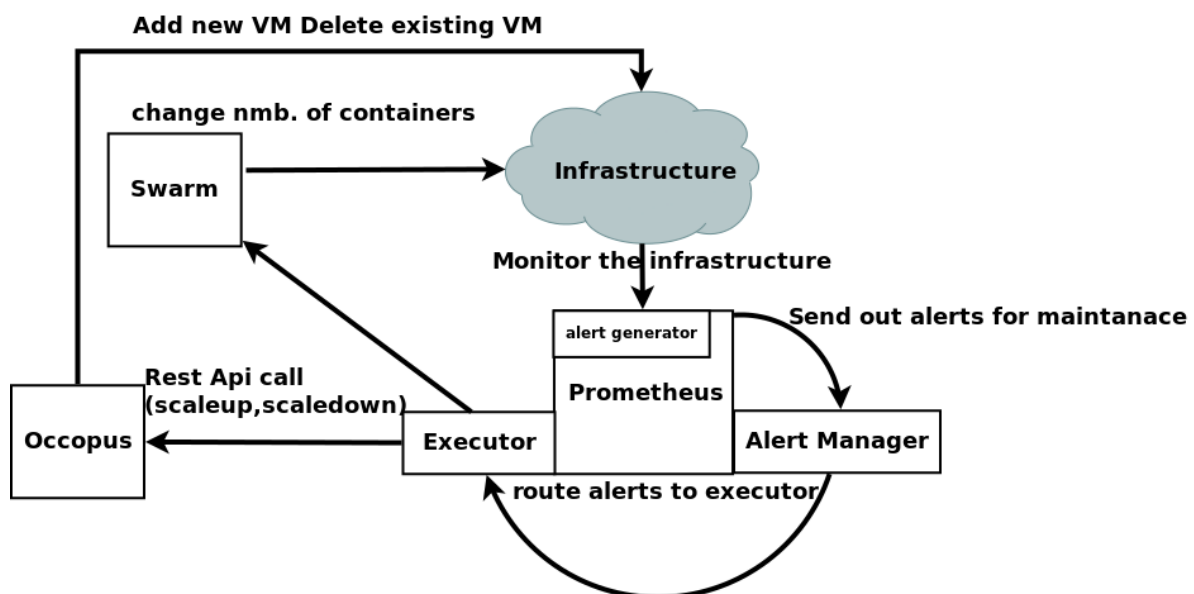


Figure 9 Modular design of the monitoring subsystem

As it was discussed in Chapter 8, MiCADO uses Prometheus as monitoring tool. On every worker node in the worker node cluster the monitoring agents collect runtime information of the system as well as the Docker containers. The monitored data first arrives at Prometheus' own database where it is stored as a time series data within memory. If the memory becomes full, Prometheus writes it out to the HDD into the /Prometheus/ folder. The infrastructure itself does not create any application on the Docker cluster, so the collected data will come from the host machine resources at the beginning. As the users start deploying their own applications to the worker cluster, new alerting rules are made by the alert generator component. This is an important step since we have to know:

- the applications,
- the details of the application (resource limitations, IDs, etc.),
- how many instances are running.

D6.2 Prototype and documentation of the monitoring service

The details of a running application such as resource limitations are queried through the Swarm API. We take advantage of the Docker Swarm API as we ask periodically if there are Docker services running in the Docker cluster for which Prometheus rules are not yet generated. If we find such services, then we create new alerting rules. At the same time alerting rules without existing applications in Swarm are deleted. Alerting rules with application specific details such as CPU limitations, custom names, etc. are pushed to Prometheus using Prometheus' API. When a new rule is added, the alert generator refreshes the list on the fly. When the user deletes an application, the rule is removed from the list, alert generator refreshes, and as a result, we have a clean, automated alerting system maintained by our own alert generator service. It is worth mentioning that the APIs mentioned here are relatively new at the time of writing deliverable. We have seen some improvements over the new versions of the APIs. However, since the APIs are changing with almost every release, we decided to stick with one particular version. We used Swarm API version 1.25 and V1 for Prometheus.

The next component in the monitoring subsystem is Prometheus' own alert manager service (port 9093). The service communicates over HTTP with Prometheus and gets firing alert definition in text format. While we have a few configurations available in Prometheus to set timings on the alerts, most of these configurations have been transferred to a separate component to encapsulate functionalities in an object oriented manner. The alert manager takes care of the alerts before routing them to the appropriate receiver or leaf in a tree structure. We configured Prometheus to send out firing alerts to the alert manager running on port 9093. In its configuration we can set timing, such as how often should it send out incoming alerts, grouping incoming alerts, or waiting after an alert is executed before it sends it out again. We configured the alert manager in a way that we averaged the time that it takes for a VM to finish its booting process and to connect to the monitoring system. It is important since we have to stop firing alerts before we execute a scaling event again. The new node will be connected to the cluster and since the load on the cluster will change, the alert statuses will change before the next execution in alert manager. Prometheus supports mostly all functionalities that other monitoring tools offer as well, such as sending email notifications or adding log messages to a central logging server. However, in the current MiCADO version, at the time of writing this deliverable, we only used its Webhook calls. Webhook means, it sends out the alerts in text format to an address. In this address our own alert executor service is receiving the request.

The next component is our alert executor service (on port 9095) that executes REST calls based on the incoming Prometheus alerts, towards Occopus and Docker Swarm. At the time of this deliverable we only used this component for executing alerting rules. However, we should be able to control scaling events with policies in COLA while also taking care of the optimization. These features will be implemented later, but we already started to make the first steps to implementation and thus we had to realize that this component is more closely connected to the other orchestration components in MiCADO than to the monitoring system. This is the reason why on Figure 2 you can see a "Policy maker" component, which is the first version of the implementation of the alert executor service. Also, we followed this design as modularity and interchangeability are key factors in MiCADO. While alerting rules are present in almost every monitoring tool investigated in Section 7.1 we have the opportunity to replace Prometheus without changing the alert executor service.

This Alert Executor module is built up from three parts:

- Service adapter part
- Routing part

D6.2 Prototype and documentation of the monitoring service

- Execution part

Service adapter part

We need to open the module towards the other components to receive alert definitions from the alert manager service. When we receive incoming alert definitions we create environment variables from them. The following variables provide a way to deal with the alerts:

- AMX_RECEIVER: name of receiver in the AM triggering the alert
- AMX_STATUS: alert status
- AMX_EXTERNAL_URL: URL to reach alert manager
- AMX_ALERT_LEN: Number of alerts; for iterating through AMX_ALERT_<n>.. vars
- AMX_LABEL_<label>: alert label pairs
- AMX_GLABEL_<label>: label pairs used to group alert
- AMX_ANNOTATION_<key>: alert annotation key/value pairs
- AMX_ALERT_<n>_STATUS: status of alert
- AMX_ALERT_<n>_START: start of alert in seconds since epoch
- AMX_ALERT_<n>_END: end of alert, 0 for firing alerts
- AMX_ALERT_<n>_URL: URL to metric in Prometheus
- AMX_ALERT_<n>_LABEL_<label>: alert label pairs
- AMX_ALERT_<n>_ANNOTATION_<key>: alert annotation key/value pairs

Routing part

The next thing we have to do with alerts processed by the adapter is to route them either to a scaling logic which works on Docker applications or to the one which works on virtual machines. The reason for this is that the scaling events will be divided and we will scale later in two levels (VM and Docker). When we route alerts we depend on the Label fields in the alert definition. We need to check whether the Label called "level" equals to VM or Docker and depending on it we route it to the appropriate logic behind it. Also, we have to do it for every alert which can cause some problems. The problem comes from the fact that alert manager sometimes sends out the alert definition as one entity and we have to make sure that we processed all of the alerts within the input file. However, this is feasible since we have access to the length of the alerts (AMX_ALERT_LEN) in the input and we can iterate over them by a simple loop. The routed alerts are then received by the execution part.

Execution part

The main scaling events are originated from the execution part. It has to connect to Occopus and Swarm components to realize scaling events. Figure 2 shows that two components are connected to the Policy keeper component. Communication in the direction of the Cloud orchestrator in our implementation means to call Occopus to scale the number of virtual machines in the cluster. For scaling, Occopus requires two parameters: infrastructure id and node id. We collect these inputs also from the label field of the alert definitions. In fact, the IDs are substituted with real values at deployment time and inserted into the alerts that check the resource usage of the VMs in the cluster. This way we can easily change the number of virtual machines and realize the basic auto scaling feature of MICADO at the level of VMs.

The other connection is towards the container orchestrator as it is shown on Figure 2. In our implementation it means that we have to connect to Docker Swarm and change the number of application containers. In fact, we have to do it first, if we have enough resource for a new container before changing the number of VMs, since it takes less time. It will be less time consuming because we do not have to create a new VM first, connect it to MICADO, and then

D6.2 Prototype and documentation of the monitoring service

download the application image and start it. Instead, if we already downloaded it we just run another instance of the same application image.

However, scaling the number of containers is not that easy. As it was mentioned before, we have to make sure that there is enough resource in the cluster to start a new container. To do so, we have to:

1. Query the resource usage in every VM monitored by Prometheus using its API.
2. Ask Docker Swarm about the application specification such as CPU limitation.
3. Go through the response we got from Prometheus and check whether there is at least one VM where the resources are enough to start a new container.

For example, Docker Swarm tells us that the application container needs at least 20% CPU. Now we can iterate over the nodes and check whether at least one VM has CPU usage under 80%. If the answer is “NO”, then it means that every VM in the cluster is overloaded, and we will have to scale up the number of nodes with the help of Occopus. If the answer is “YES” then we can go on and scale up the number of application containers.

To change the number of containers in a Docker service, we have to do the following steps:

1. Cut out some parts of the Docker Swarm response (Json) we got in step 2 previously.
2. Increment the label “ID” in the Json response.
3. Change the value of “.Spec.Mode.Replicated.Replicas” which specifies the number of containers.
4. If we scale down the check if the number of containers are ≥ 2
5. Send back the new application specification to Docker Swarm.

Steps 1 and 2 are needed because Docker Swarm is not accepting the response otherwise. Step 1 makes sure that the response is in a good format, and it does not contain anything which is not needed. In Step 2 we have to increment the version ID of the application, otherwise Docker Swarm will not know which specification is the new one and which is the old one. After these steps the new Json will be accepted by Docker Swarm and depending on the received response it will allocate a new container for VM which has enough resource, or it will delete one of the application containers.

When we realize scale down events from the incoming alert definition, we have to make sure that it does not influence the application reachability from the user's point of view. Occopus always leaves one virtual machine running and not scaling under one. However, Docker Swarm's operating principle differs from it. We have to make sure that if the number of containers of a given application is already one, we do not scale it down further. This is previously shown as step 4. This is the way how the alert execution process works in our implementation of MICADO V3.

9.2 Implementation of MiCADO V3

This section discusses the implementation of the latest MICADO release, MiCADO V3. The design plan of MiCADO V3 has already been shortly introduced in D6.1, while the actual implementation will be detailed here. MiCADO V2 has been improved covering several aspects described in the next paragraphs. During the description the term “MiCADO service” is used as a collective noun for the overall service the MiCADO Master node exposes towards the users. MiCADO V3 contains improvement in four aspects as follows:

D6.2 Prototype and documentation of the monitoring service

- **Hide Occopus from the end user**

In the previous versions of MiCADO, the deployment of the Master node together with the Worker nodes have been implemented using Occopus. Since MiCADO is considered as a service, the goal is to make this service independent from Occopus. In MiCADO V3 the Master node can be deployed by any cloud orchestration tool with the help of a deployment script (see “cloud-init config file” in Figure 10). This script can be passed as user-data to be executed on the newly created virtual machine. Moreover, deploying the MiCADO Master node does not strictly require launching a new virtual machine, since later we can provide other forms of MiCADO Master node deployment such as compose file for Docker or recipes for configuration managers, etc.

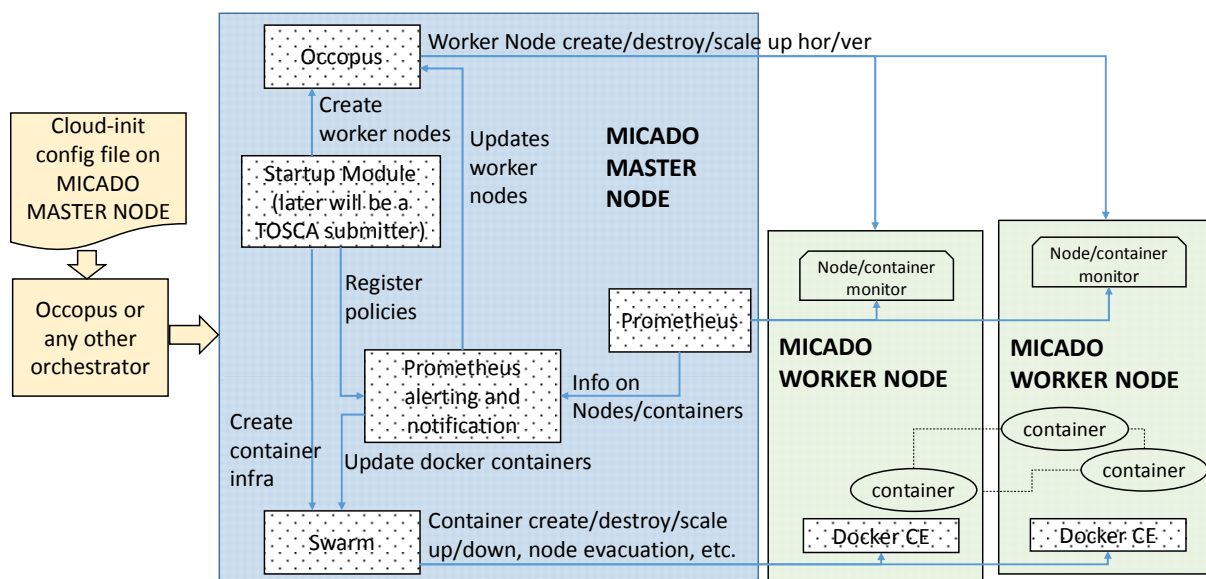


Figure 10. Architecture of MiCADO V3

- **Separate the deployment of MICADO master from the worker nodes**

In MiCADO V2, deployment of the worker nodes was strictly part of the deployment of the MiCADO framework as Occopus required at least one worker node to be instantiated at startup. Since the MiCADO service is considered to perform the deployment of the user defined container infrastructure once the user performs the submission, there is no need for worker nodes when the MiCADO master initializes. Therefore, we separated the deployment of the worker nodes from the deployment of the MiCADO master. In order to achieve this, the Master node performs the deployment of the worker nodes by using the internal Occopus service (see Startup Module in Figure 10).

- **Use Occopus as a service**

Occopus from MiCADO V3 is used as an orchestration service (see Occopus in Figure 10) by the other components of MiCADO to ease the communication among the components and for error detection of the worker nodes launched by Occopus. For the Occopus service, a new Docker image has been created and launched automatically at startup to run inside a container, similarly to other components of MiCADO. The main design goal during MiCADO development is to integrate the building blocks (components) as Docker services.

- **Introducing the initial version (placeholder) of the submitter**

The final version of MiCADO is designed to integrate a submitter component for the users

D6.2 Prototype and documentation of the monitoring service

to submit their container based infrastructure using TOSCA. In MiCADO V3 a new image has been created as a placeholder for the submitter service (see Startup Module in Figure 10), but the submitter has not been implemented yet. Temporarily, the submitter image contains some startup scripts performing the deployment of the MiCADO worker nodes on behalf of the user. In the final version, the MiCADO submitter will initiate the deployment of the worker nodes just before the deployment of a new container infrastructure is initiated (by the user).

MiCADO V3 is operating the following way. First, the user creates a new virtual machine in any cloud using the provided cloud-init configuration file as user data. As a result, the newly created virtual machine deploys all the services of the MiCADO master node (see Figure 10) such as Occopus, Docker Swarm, Prometheus, Prometheus alerting and notification service, and the Startup Module. Once the services come to live, the Startup Module initiates the creation of a new infrastructure containing the worker nodes through Occopus on a target cloud. Please, note that this step simulates the requirements for new cloud resources to host a container infrastructure submitted by the user.

```
user_data:
  auth_data:
    type: cloudsigma
    email: YOUR_EMAIL
    password: YOUR_PASSWORD

  resource:
    type: cloudsigma
    endpoint: https://zrh.cloudsigma.com/api/2.0
    libdrive_id: 74655f92-b7bb-4084-b4ec-fdbe46d576ca
    description:
      cpu: 1000
      mem: 1073741824
      pubkeys:
        -
          YOUR_PUBLIC_KEY_ID
    nics:
      -
        ip_v4_conf:
          conf: dhcp

  scaling:
    min: 1      #minimal number of MiCADO workers
    max: 10    #maximum number of MiCADO workers
```

Figure 11 Example configuration for MiCADO worker infrastructure in case of CloudSigma

The deployment of the MiCADO worker infrastructure is based on a predefined configuration passed as part of the cloud-init configuration file for the MiCADO master. The syntax of the predefined configuration for the MiCADO worker infrastructure follows the syntax of Occopus and is shown in Figure 11.

It contains

- authentication information for the target cloud (see 'auth_data' in Figure 11),
- virtual machine creation related details (see 'resource' in Figure 11),
- and scalability limitations (see 'scaling' in Figure 11)

This information is used by the Startup Module to create the authentication data, infrastructure description and node definition files for Occopus. Once the files have been created, the worker

D6.2 Prototype and documentation of the monitoring service

infrastructure deployment is initialized through the REST API of the Occopus service. Please note, that in the final version this step will be performed by the Submitter component of MiCADO master when a new container infrastructure is submitted. The newly created MiCADO worker node(s) then attaches to MiCADO master to form a cluster and become ready for executing containers.

At this point the user may submit a container infrastructure to Swarm, either remotely or locally after logging in to the MiCADO master node. When the container infrastructure has been submitted, the containers come to live and start executing on the worker nodes. The startup module recognizes the arrival of new containers and registers policies into Prometheus (see 'Startup Module' in Figure 10). From this point, Prometheus is monitoring the free capacity of the MiCADO worker nodes by the node exporter module and the load generated by each container. In both cases the decision on scaling is taken by the Prometheus' alerting and notification component (see Figure 10), and realized by Occopus for worker nodes or by Docker Swarm for containers.

10. Performance evaluation

In order to evaluate the performance of the MiCADO prototype implementation described in Section 9, a set of experiments have been designed and implemented on the CloudSigma public cloud. Different phases, including building up the MiCADO infrastructure, and also scaling up/down the application nodes have been measured using the Data Avenue (DA) [18] application. These experiments provided evidence for the automated scalability features provided by MiCADO.

The first task in preparing for the measurements is to create the MiCADO infrastructure. For this task, we used Occopus. Overall, it took 320 seconds on average, on our target cloud to build the infrastructure. Once the components are successfully connected to each other, the DA application can be started. Since MiCADO works with Docker Swarm, the applications can be started as Docker services. To do so, the operator needs to start DA through the Swarm service locally or remotely. Deploying the application in this way took 140 seconds on average in our experiment.

After the MiCADO infrastructure was built up and the DA application was successfully started, we tested and measured the automated scale-up and scale-down features of MiCADO. First, we put load on the cluster by transferring large volumes of data through the application nodes that run DA. To make the application cluster overloaded, 1 GB files from multiple different sources have been transferred. In the current MiCADO prototype there is an alert defined that in case the load in the Docker cluster exceeds 80% of the available resources, MiCADO automatically creates a new DA node to scale the infrastructure up. In our experiment, it took MiCADO approximately 300 seconds to connect this new node to the cluster and make the application available. On the other hand, destroying existing virtual machines when decreasing the load on the cluster, took only 12 seconds in average. Figure 12 summarizes the above described scale-up and down times and compares these to creating and destroying the complete infrastructure.

Operation	time (sec)
create infrastrurcture	320
destroy infrastructure	15
scale up app node	300
scale down app node	12

Figure 12 Infrastructure and node creation/destroy

The next benchmark was designed to demonstrate MiCADO in actual usage, with scale-up and scale-down events. Figure 13 shows three graphs. These graphs were generated by Grafana [14] based on the input provided by the Prometheus monitoring tool. On the top, the average CPU utilization in the application cluster is illustrated. In the middle graph, the CPU usage of the individual nodes in the application cluster are shown with different colors for each node (which in average gives us the upper graph). Finally, the bottom graph indicates the number of nodes in the Docker cluster.

Overall, Figure 13 illustrates how the number of application nodes changed depending on the actual load on the cluster. The green color on the middle graph shows, that at the beginning (15:34) there was no load on the cluster which could serve user requests perfectly with the help of only one VM. As the file transfers have started, the load soon reached 100% of CPU usage on the only available node. After MiCADO ascertained that the load remains high (the

D6.2 Prototype and documentation of the monitoring service

load needs to be constantly above the alert value of 80% for a certain amount of time, in our case for 180 seconds to avoid unnecessary scaling-up/down), it fired an alert which called Occopus to instruct scaling up the application cluster. The new node (blue marked) was connected to the cluster at 15:43 to decrease the load on the first node that was previously overloaded (15:45). The same up-scaling event can be seen in Figure 13 at 15:49 as two nodes still struggled to serve user requests, and an additional third node was connected to the Docker cluster.

On the “CPU/node” graph we can check that the application nodes share the load between each other, while their load are close to each other. Also worth mentioning, that new nodes are connected to the cluster with instant 100% of load at the beginning. The reason for this is the boot process of the virtual machines. When the nodes are booting they are at full load, and when they finish starting the Docker containers, they remain on the actual load coming from the application.

At the end of this test we successfully transferred 12GB of data using three DA services as maximum in parallel. The transfer took place between 15:36 and 15:56, as we can see in Figure 13. From that point, MiCADO started to scale down the application cluster. The number of nodes decreased by one every time when Prometheus fires up the alert, telling Occopus that the cluster is under-loaded. At 16:06 the number of nodes decreased back to the minimum number of one application node. It is important to note that while the under-loaded alert in Prometheus was still firing, Occopus did not scale below the minimum number of nodes, which was one in our case, to make sure that the application is reachable at any time.

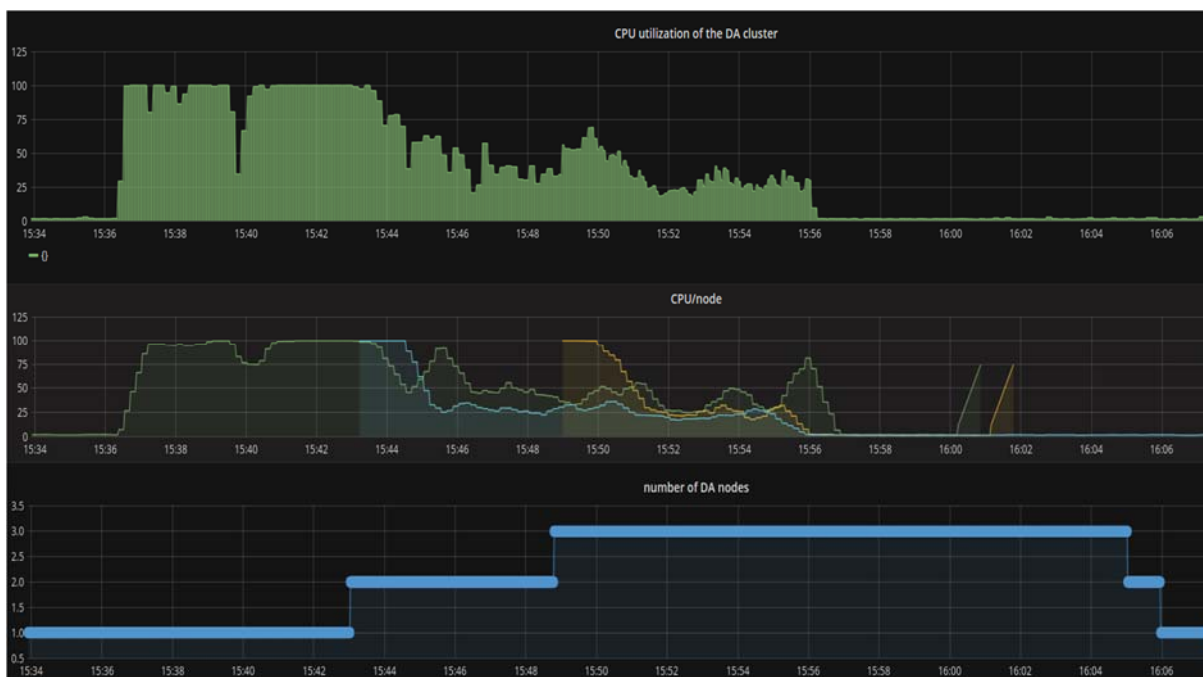


Figure 13 Resource optimization, on both scale up and down events with MiCADO

The above described experiment clearly illustrates that MiCADO works as it is expected. The application was scaled automatically on demand to adopt to actual load. In this experiment, we adjusted the time interval which MiCADO had to wait between two scaling events before scaling again: 300 seconds for scaling up and 60 seconds for scaling down. These time ranges

D6.2 Prototype and documentation of the monitoring service

were selected to fit the DA application, where large file transfers may require more time to finish, while after they finished, the load expected to drop dramatically allowing scaling down more quickly. As these numbers are application specific, they can be set to make sure the scaling events happen in the expected time intervals.

11. Current status and conclusion

In this deliverable, we focused on the design and implementation of the monitoring infrastructure of MiCADO, also explaining how it fits into the overall MiCADO architecture. We started with the big picture in Chapter 5, continuing with the design plan of the orchestration layer in Chapter 6, then focusing on the monitoring alternatives in Chapter 7, and detailing Prometheus as our current candidate as monitoring subsystem in MiCADO in Chapter 8. Chapter 9 continued with the implementation details both for the monitoring system and for the entire MiCADO V3. Finally, performance evaluation has been introduced in Chapter 10.

One of the major achievements of the above described work was the selection and integration of a suitable monitoring tool to MiCADO. The most important reasons for selecting Prometheus as monitoring tool for MiCADO are:

- it has pluggable monitoring architecture, i.e. we can easily add node monitoring and container monitoring
- it supports alerting and notification by defining rules over the attributes
- it supports user-defined notification procedure to integrate with 3rd party tools
- it has good performance and supports push/pull model
- it supports visualization, e.g. through Grafana
- it has good community support

The integration of the Prometheus monitoring tool has been successfully finished and during the integration, we have performed the following main activities:

- created the installation procedure of Prometheus
- configured Prometheus to link with Consul
- developed a new alert notification component for Occopus
- developed auto-deployment for node exporter on the MiCADO worker nodes
- developed scaling rules for MiCADO worker nodes
- developed auto-deployment for Cadvisor on the MiCADO worker nodes
- developed dynamic registration of scaling rules for containers

The current version of MiCADO (V3) has also been introduced in detail to show the progress of the MiCADO framework towards the design plan. The current version of MiCADO implements double controlling loop for resources and containers with the help of the Prometheus monitoring and alerting system.

As a next step, the development of MiCADO will continue with implementing the submitter component that is able to receive container infrastructure specified in TOSCA format. The TOSCA description of the submitted infrastructure must be parsed and realized with the help of Occopus, Docker Swarm and Prometheus. To realize this functionality a strong cooperation with WP5 is needed since WP5 defines the format of the infrastructure description and policies.

The latest release of MiCADO is available from the official COLA webpage [15] under the 'MiCADO' menu. Previous releases are also listed under the 'Archive' menu. Here one can download the package, personalize the settings and deploy MiCADO based on the detailed step-by-step description of the tutorials [16]. The installation procedure of the latest release of Occopus is described in its manual [17] under the 'Setup' section. The source code of MiCADO (including all previous versions) can also be downloaded [13].

12. 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] Tader, Paul. "Server monitoring with Zabbix." *Linux Journal* 2010.195 (2010): 7.
- [4] Massie, Matthew L., Brent N. Chun, and David E. Culler. "The ganglia distributed monitoring system: design, implementation, and experience." *Parallel Computing* 30.7 (2004): 817-840.
- [5] mamagic, Emir, and Dobrisa Dobrenic. "Grid infrastructure monitoring system based on nagios." *Proceedings of the 2007 workshop on Grid monitoring*. ACM, 2007.
- [6] Prometheus website, <https://prometheus.io/docs/introduction/overview/>
- [7] Icinga website, <https://www.icinga.com/docs/>
- [8] GO language, <https://golang.org/>
- [9] Occopus website, <http://occopus.lpds.sztaki.hu>
- [10] Cadvisor website, <https://github.com/google/cadvisor>
- [11] Anshul Gandhi, Anshul, et al. "Adaptive, model-driven autoscaling for cloud applications." 11th International Conference on Autonomic Computing (ICAC 14). 2014.
- [12] Joe Sondow, Auto Scaling Lessons Learned, [14 Augustus 2016]
<https://github.com/Netflix/asgard/wiki/Auto-Scaling-Lessons-Learned>
- [13] MiCADO source repository: <https://github.com/UniversityOfWestminster/MiCADO>
- [14] Grafana, the open platform for analytics and monitoring, [online] Available from: [<https://grafana.com/>](https://grafana.com/)
- [15] COLA website: <http://project-cola.eu/>
- [16] MiCADO tutorials: <http://project-cola.eu/micado-tutorials/>
- [17] Occopus users' guide: <http://occopus.lpds.sztaki.hu/user-guide>
- [18] Hajnal A, Farkas Z, Kacsuk P: Data Avenue, Remote Storage Resource Management in WS-PGRADE, in proceedings of IWSG 2014, 6th International Workshop on Science Gateways, IEEE, 25 August 2014, DOI: [10.1109/IWSG.2014.7](https://doi.org/10.1109/IWSG.2014.7)