Cloud Orchestration at the Level of Application

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:

D6.1 Prototype and documentation of the cloud deployment orchestrator service

Due date of deliverable: 30/06/2017 Actual submission date: 30/06/2017

WPL: Peter Kacsuk

Dissemination Level: PU

Version: WIP

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	22/06/17	Jozsef Kovacs
Reviewed:	Alex Worrad-Andrews	28/06/17	Alex Worrad-Andrews
Approved:	Tamas Kiss	30/06/17	Tamas Kiss

Table 1 Status Change History

Version	Date	Pages	Author(s)	Modification
V0.1	13/06	ALL	Jozsef Kovacs	Empty Skeleton
V0.2	14/06	Section 5	Tamas Kiss, Jozsef Kovacs	MiCADO generic architecture
V0.3	15/06	Section 7,9	Eniko Nagy, Jozsef Kovacs	Cloud Orchestration and Occopus
V0.4	16/06	Section 8	Attila Farkas, Jozsef Kovacs	Container Orchestration
V0.5	19/06	Section 6	Jozsef Kovacs	MiCADO Orchestration Layer
V0.6	20/06	Section 10	Botond Rakoczi, Jozsef Kovacs	MiCADO implementations
V0.7	21/06	Section 11	Jozsef Kovacs	Conclusion
V0.8	22/06	Section 10,11	Jozsef Kovacs	Small corrections
V0.9	23/06	Section 6, 7, 8, 10, 11	Tamas Kiss	Small corrections
V1.0	28/06	Section 5, 6, 7, 8, 10	Gabor Terstyanszky	Small corrections
V1.1	29/06	Section 8	Attila Farkas, Jozsef Kovacs	Add more explanation for the selection
V1.2	29/06	Section 4	Jozsef Kovacs	Introduction of WP6 deliverables
V1.3	29/06	Section 7	Jozsef Kovacs	Adding Terraform description and additional explanation for the selection

V1.4	29/06	Section 6	Jozsef Kovacs	Add discussion on requirements defined by D8.1 for COLA use cases

Table 2 Deliverable Change History

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

Table 3 Glossary

4. Introduction

DoW specifies D6.1 “Prototype and documentation of the cloud deployment orchestrator service” deliverable as follows:

> “This document will contain the technical and user documentation of the MiCADO cloud deployment orchestrator service.”

This deliverable aims at describing the design and implementation of MiCADO focusing on the orchestration of both cloud resources and container services. MiCADO is a compound service providing automatic scaling and orchestration of user submitted 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 policies specified by the user together with the infrastructure description. This overall functionality is realized by the MiCADO framework which will be introduced in the following chapters.

The core part of D6.1 is structured as follows:

- **Chapter 5** – MiCADO generic architecture
  This chapter summarises the layers of the entire system to be realized in the COLA project to clearify the big picture and location of the MiCADO orchestration layer.
- **Chapter 6** – Design plan of MiCADO Orchestration Layer
  This chapter details the architecture of the MiCADO orchestration layer together with an overview of its proposed internal operation and interactions among the components.
- **Chapter 7** – Cloud orchestration
  This chapter focuses on the Cloud Orchestration component of the MiCADO architecture. It introduces several Cloud Orchestrator tools and proposes one candidate to be used for implementing the MiCADO orchestration layer.
- **Chapter 8** – Container Orchestration
  This chapter focuses on the Container Orchestration component of the MiCADO architecture. It introduces several Container Orchestrator tools and proposes one candidate to be used for implementing the MiCADO orchestration layer.
- **Chapter 9** – Occopus
  This chapter gives a short introduction of the Occopus cloud orchestrator tool.
- **Chapter 10 –** Implementation of MiCADO Orchestration Layer
  This chapter introduces the MiCADO Orchestration Layer implementations released until the deadline of this deliverable and also proposes the next step in the implementation.
- **Chapter 11** – Conclusion
  This chapter concludes the overall deliverable, the results introduced and implementations released.

The rest of the deliverable contains the obligatory parts, Table of Contents in Chapter 1, List of Figures and Tables in Chapter 2, Status, Change History and Glossary in Chapter 3 and References in Chapter 12.

In order to understand the synergy of the deliverables in WP6, here is a small explanation:

- **D6.1 – Prototype and documentation of the cloud deployment orchestrator service**: this deliverable focuses on Cloud deployment and Orchestration functionality of the MiCADO Orchestration layer. Deployment in this context means deployment of the virtual machines in the cloud and deployment of the containers in the docker cluster. This deliverable will focus on these two topics.

- **D6.2 – Prototype and documentation of the monitoring service**: this deliverable will focus on the monitoring subsystem operating inside MiCADO to collect information for scaling related decisions.
- **D6.3 – Prototype and documentation of the scalability decision service**: this deliverable will focus on how the scalable decisions will be implemented, what kind of policies are implemented.
- **D6.4 – Prototype and documentation of the price/performance optimization service**: the deliverable will detail what kind of optimization calculations can be executed to affect the scalable decisions and will detail the internal architecture for it.

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 large enough to be sufficient for worst-case requirements scenario. However, most of the time, the worst case scenario is not present and allocated resources of the monolithic infrastructures are wasted.

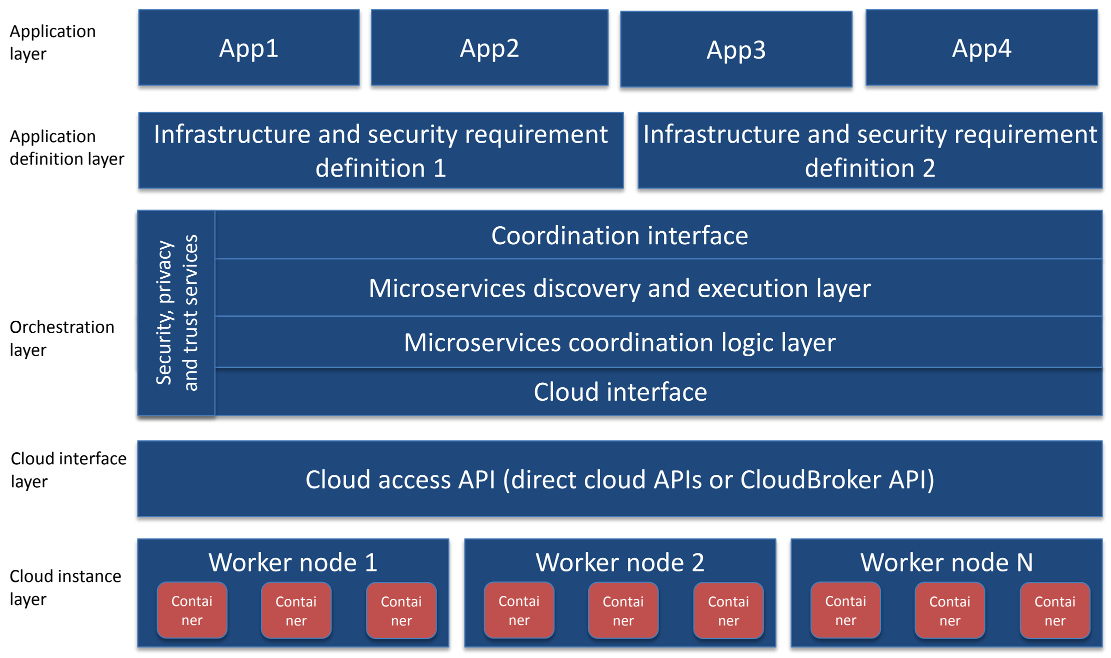
Application layer
App1
App2
App3
App4
Application definition layer
Infrastructure and security requirement definition 1
Infrastructure and security requirement definition 2
Orchestration layer
Security, privacy and trust services
Coordination interface
Microservices discovery and execution layer
Microservices 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 above described microservices-based concept are as follows:

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

it, 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 orchestration control and decouples the orchestration layer from the application definition layer. This set of APIs enables application developers to utilize the dynamic orchestration capabilities of the underlying layer and supports the convenient development of dynamically and automatically scalable cloud-based applications by embedding these API calls into application code.
   b. **Microservices discovery and execution**. This sub-layer manages the execution of microservices and keeps track of services running. Execution management combines both start-up and shut down of microservices. Service management gathers information about currently running services, such as service name, IP address and port where the service is reachable and optional service tags to help in 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 to abstract 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 3b and 3c) 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 on 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 then be 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 also be used at this layer to support accessing multiple, heterogeneous and distributed clouds via a uniform access layer.
5. **Cloud instance layer**. This layer contains cloud instances and provided by IaaS cloud providers. These instances can run various containers that execute actual microservices. This layer typically represents state-of-the-art of cloud technology, as provided by various public or private cloud providers.

6. Designing the MiCADO Orchestration Layer

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 since 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. The design below has taken the deliverable D8.1 entitled “Business and technical requirements of COLA use cases” as an input which specifies the requirements of the 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 basically forms a cluster which is able to dynamically allocate and 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 node. Master node is the head of the cluster performing the collection of information on microservices, the calculation of optimized resource usage, the decision making and the realization of decisions related to handling resources and to scheduling microservices. Worker nodes are volatile components representing execution environments for the microservices, i.e. they are executing the 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.

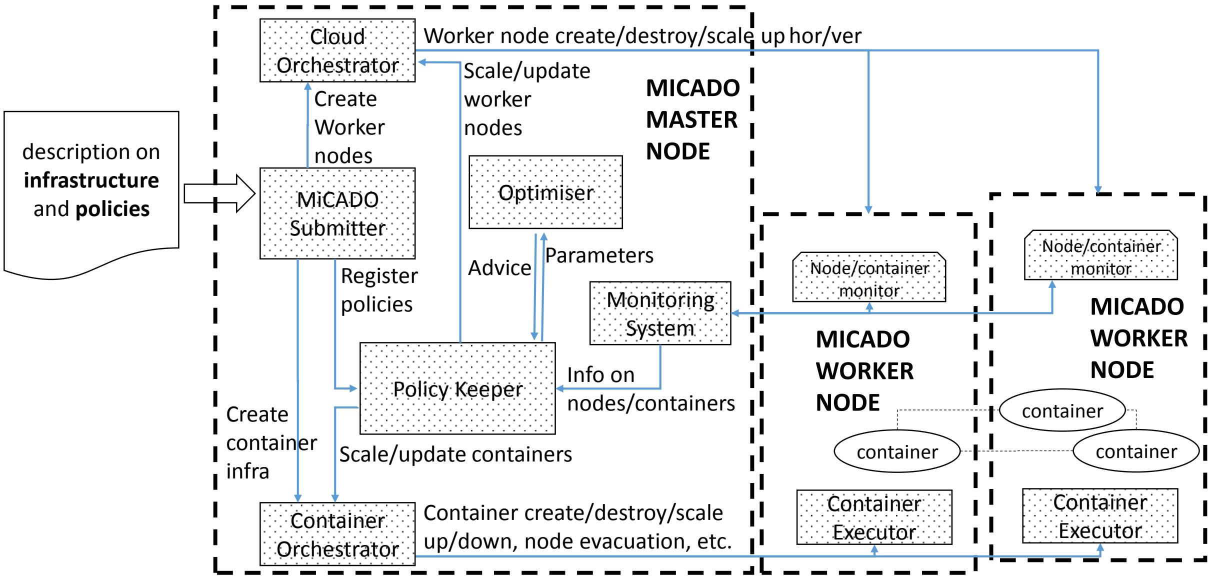

Figure 2 Architecture of the MiCADO Orchestration Layer

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

- **MiCADO Submitter** is the primary service endpoint for creating an infrastructure to run an application and managing this infrastructure and the application itself.

Submitted infrastructures are received by this component. The incoming description (e.g. TOSCA format) are interpreted and related parts are forwarded to the other key components.

- **Cloud orchestrator** is responsible for communication with the Cloud API for allocating and releaseing resources, for building up/shutting down new MiCADO worker nodes when necessary.
- **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 the 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 requires 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 makes sure 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 and the result 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 the information on the nodes and containers and the Policy Keeper starts updating the deployment (including both the worker nodes and containers) when necessary. The Optimizer performs calculation in the background and provides advice for the Policy Keeper time to time.

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 start realizing the controlling loop. Once the initial deployment has been done, any update can only be 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 is planned to be technology neutral that will not be depending on one particular component implementation.

There are five categories of requirements defined in D8.1 by the COLA use cases: system requirements, data requirements, performance requirements, security requirements. other requirements.

- System requirements relates to the underlying operating system which is Ubuntu in most of the use cases except for Saker use case, where windows is the base operating system. There are alternatives in executing windows applications in MiCADO which are being investigated. Current promising alternative is using windows emulator software on linux, however native windows solution is also a possibility with or without containers.
- Data requirements for the use cases are low however using external database is a good alternative for data intensive applications if needed.
- Performance requirements is 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 MiCADO is planned to be able to setup VPN, encrypted channels and to apply firewall settings. Private and public clouds will be supported.
- Other requirements in D8.1 mentions data protection, robustness, quality of service, etc. These does not strongly related to the MiCADO architecture, but will be addressed later during the project together with WP7.

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 option to replace them with an alternate solution if necessary.

In chapter 7, cloud orchestrator tools are investigated and a candidate for the current implementation is selected. The same happens in chapter 8 for the container orchestration functionality. Finally, Chapter 10 summarizes the current status of the MiCADO developments.

7. Cloud orchestration

In the MiCADO framework, the cloud orchestration functionality is required for allocating virtual machines in order to join additional resources to MiCADO. This functionality is implemented by a cloud deployment orchestrator tool. In this section, we list the investigated cloud orchestration tools, select a candidate and provide justification for our selection.

7.1 Investigated tools

In this section we give a short overview of relevant cloud orchestrator tools. We have investigated several academic prototypes (Occopus, Live Cloud, Roboconf, GRyCAP) and commercial products (Cloudify, Heat, CloudFormation) in this field. We shortly describe each of them to clarify their advantages and/or disadvantages.

Occopus [3] is a powerful, easy-to-use, configurable, hybrid, multi-cloud orchestration tool developed by MTA SZTAKI. It is an open source software providing features for configuring and orchestrating distributed virtual infrastructures both on single and multi-cloud environments. Occopus has been designed to be cloud-agnostic and is developed in a way to handle dynamically replaceable plugins to realize the cloud-dependent interactions through the various cloud interfaces. Such plugins are designed not only for handling different cloud interfaces simultaneously (multi-cloud), but also for interacting with various configuration management tools at the same time (multi-config), for utilizing different contextualization methods and for implementing various service health-check facilities. As a result, Occopus can be utilized in a broad range of environments by applying any combination of its plugins. Moreover, such plugins can be added by the user without recompilation. Building and maintaining an infrastructure can be performed through different interfaces. Occopus has CLI and REST API. Both, provides the main functionalities, like building, maintaining, scaling or destroying. Moreover, the CLI and the REST interfaces can be used in an alternate way, which means after building an infrastructure by the CLI one may continue the maintenance of the infrastructure with the help of the REST API. During maintenance the opposite direction is also a supported use case. There is a third interface namely the library API that enables developers to integrate the functionalities of Occopus as a library and use it by invoking the API methods. It is able to handle CloudBroker virtual machines and easily deployable.

The Roboconf [4] cloud orchestrator written in JAVA aimed at supporting service deployment, maintenance and migration among various types of clouds including multi-cloud systems. In order to achieve this goal they have developed a hierarchical language to allow fine-grained administration of cloud services. The main drawback of Roboconf that it is not able to utilize external configuration manager tools and currently does not support CloudBroker either.

LiveCloud [5] is a management framework for resources in cloud data centers that integrates low level network oriented resources into data center orchestration and service provision. It also includes resource allocation algorithms but in a static way. LiveCloud is not able to support distributed systems like multi-cloud including CloudBroker.

GRyCAP [6] realized a good concept. Their descriptors are based on RADL (Resource Application Description Language) for defining the infrastructure and the nodes. They put significant effort into combining and integrating their cloud orchestrator tool with a repository of (1) virtual machine images, (2) RADL descriptors and (3) application recipes. GRyCAP is a good candidate, however CloudBroker support is not implemented yet.

Openstack's Heat [7] is template-based, provides auto-scaling features through integration with Telemetry, and has Chef/Puppet integration. Heat can be used with a CLI, API, or the Horizon Dashboard. Heat has its own template format, HOT (Heat Orchestration Template), but can process CloudFormation templates. Heat is open source, however, it only supports OpenStack clouds.

CloudFormation [8] is one of the most mature and heavyweight contenders between orchestrators developed by Amazon. Infrastructures are defined as JSON templates which can be submitted through CLI, API or the AWS management console to the AWS EC2 Cloud. Unfortunately, it does not support any other cloud providers or backends. The vendor lock-in makes it hard to use for academic purposes and it is not an open source software.

Marpaung et al. [9] discuss Altocumulus, AppScale, Cloudify and mOSAIC. Altocumulus focuses on deploying web applications to a variety of public clouds, which limits its usability in private or hybrid clouds. It does not provide monitoring or dynamic changes to services. AppScale is an open-source product that supports execution of Google Application Engine applications and therefore restricted to this particular technology. mOSAIC provides a set of APIs to application developers to tackle cloud deployment issues. The limitation of mOSAIC is the implementation of these APIs as the application developer needs to integrate these to application components.

Cloudify [10] is one of the latest orchestrators, with its first release being in 2014. It is an open source cloud orchestration framework, allowing the user to model applications and services and automate their entire life cycle, including deployment on any cloud or data center environment, monitoring all aspects of the deployed application, detecting issues and failure, manually or automatically remediating them and handle ongoing maintenance tasks. However, some advanced features including the Web UI are only available in the commercial (premium) edition. It has a TOSCA editor with deployment and orchestrator. It provides access to multiple clouds and a complete framework to describe microservices and execute them either in Docker containers or on cloud metal. Cloudify also provides dynamic service upscaling and downscaling based on microservice dependent parameters, for example number of transactions, number of threads, etc. Cloudify does not provide a container portability framework, nor do its metrics span dockerised microservices and the cloud metal executing them. CloudBroker support is also missing.

OpenTosca [11] provides an open source ecosystem for the OASIS Topology and Orchestration Specification for Cloud Applications developed by Stuttgart University. OpenTosca is divided into three parts: a TOSCA runtime environment (OpenTosca container), a graphical modelling TOSCA tool (Winery) and a self-service portal for the application available in the container (Vinothek). Although OpenTosca is a generic framework, it does not support run-time orchestration.

Terraform [12] is developed by HashiCorp for building and changing infrastructures. Terraform can manage existing and popular service providers as well as custom in-house solutions. Terraform is integrated with HashiCorp's other tools, however, on its own it lacks advanced features, and is only capable of deployment of an infrastructure without lifecyclemanagement, scaling, error-handling. It can integrate with Chef to provide configuration management and is open-source. Terraform has a large developer community behind and it provides similar functionality to CloudFormation, but without the vendor lock-in.

7.2 Selected Service: Occopus

In the current stage of the project, the candidate for performing cloud orchestration in MiCADO is the Occopus tool developed by MTA SZTAKI. We must emphasize again that the cloud orchestration in the MiCADO system is designed to be an interchangeable component. Our main argument for Occopus are as follows:

- it is a mature orchestrator tool for virtual machine allocation/release
- it is cloud-independent
- it supports CloudSigma and CloudBroker
- it is easily updatable (modifications, new features, etc.) according to the requirements of the COLA project
- it is well-documented
- it supports easy-to-create descriptors
- it is easily deployable as a container
- it supports scaling
- it has command-line and REST API interface

The weakest point of Occopus is that it currently does not support TOSCA descriptor format. This disadvantage will be addressed by introducing the MiCADO submitter component (Figure 2) which will interpret TOSCA description and pass the necessary information to the services running on the MiCADO server node. During this step, the submitter can extract the necessary parameters for Occopus. Descriptors for Occopus can be generated based on the combination of templates and attributes coming from the TOSCA description. Moreover, later it is still an alternative to introduce TOSCA descriptor support in Occopus.

It is important to note that based on the proposed implementation design of MiCADO the orchestrator tool is planned to be completely hidden from the users of MiCADO. The tool will only be interfaced by the components running on the MiCADO server (e.g. policy keeper). The only interface of MiCADO is the submitter through which TOSCA descriptors will be processed. This way MiCADO users should not even recognise if the cloud orchestrator tool is replaced by another. Therefore the selection of the Orchestrator tool do not need to be affected by the popularity of the tool, only by the available functionalities.

8. Container Orchestration

In the MICADO framework the applications and services are executed as containers. In this chapter the three most popular and mature Docker Clustering tools will be investigated and finally a candidate will be selected to be integrated into the MiCADO framework.

8.1 Investigated tools

Rancher

Rancher is an open source software developed by Rancher Labs. Rancher provides entire software stack that is needed to manage containers in production. The Rancher platform consists of four major components: infrastructure orchestration, container orchestration and scheduling, application catalog and enterprise-grade control. The entire software stack is running in one Docker container. Rancher provides a web and a command line interface to manage the Rancher server.

Rancher takes virtual machines from cloud provider, data center or PCs to create an infrastructure for the container cluster. Rancher has an own container orchestrator and scheduler framework called Cattle, but it also supports other popular frameworks like Docker Swarm, Kubernetes and Mesos.

Rancher has a built-in, public application catalog to deploy applications. In this catalog users can create their own private catalog to manage their private applications. This catalog works as an application repository for the created infrastructure.

Rancher supports enterprise grade user management with integrated LDAP and Active Directory authentication. The Rancher architecture and the provided tools are showed in Figure 3.

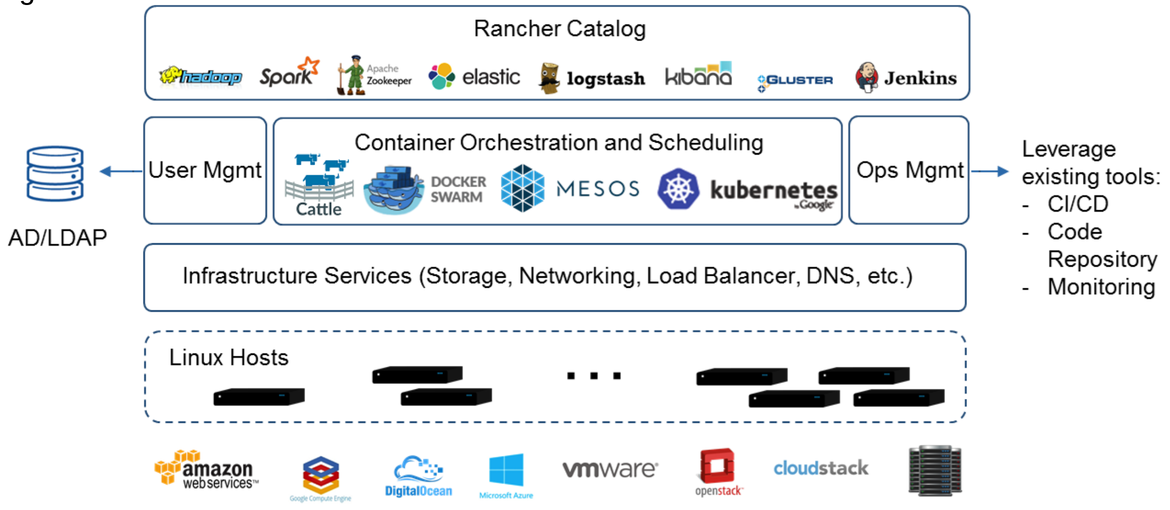

Figure 3 The Rancher architecture

Rancher creates environment to separate different clusters from each other. Every host, container and Rancher resources are created and belong to an environment.

The Rancher server will manage every resource and cluster inside the environment. Rancher takes Linux hosts to build a cluster in an environment. There are some prerequisites against the hosts. They have to support Docker 1.10.3 or higher version, they should have minimum 1GB ram, ability to connect the Rancher server and ability to be routed to any other host in the same environment on the Rancher network. After a node is added to the Rancher environment, a rancher agent container is launched on the node. After that, the node can be used to create a cluster inside the environment.

Rancher provides a container-to-container overlay network, using IPsec tunneling. On this network the containers will be assigned to the Docker bridge network and the Rancher managed network. Containers on different nodes can communicate with each other on the Rancher managed network. Rancher implements a distributed DNS service by using its own DNS server. Each healthy container is added to the DNS service and reachable with their service name.

In Rancher the default environment and cluster management platform is Cattle. Cattle uses the Rancher Compose tools which is the multi-host version of the Docker Compose. Docker Compose is used for service definition meanwhile Rancher Compose will deploy and schedule the defined service inside the environment.

Rancher uses HAProxy as the default load balancer. Every environment has their own load balancer. The load balancer uses the Round Robin algorithm from the HAProxy to select the target node. Rancher load balancer also supports Layer 4 and Layer 7 load balancing. With Layer 4 load balancing Rancher can link services and direct traffic to a defined port. The Layer 7 load balancing uses 1 defined port and uses HTTP headers to separate the services from each other and directs the traffic to the services.

Rancher in the Cattle environment uses distributed health monitoring system by running network agents on the hosts for health checking the containers. The network agent uses HAProxy to internally validate the health status of the running services in the environment. Users can define health check policy with Rancher Compose files.

The user can create scheduling policies inside a Cattle environment. Users can set labels on the hosts and on the containers, with these labels users can define scheduling rules. With these rules users can define which container to be launched on a host with a specific host label or name.

Kubernetes

“Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.” [13]

Kubernetes can schedule and run application containers on clusters of physical or virtual machines which located in a public or private cloud or in a datacenter. However, Kubernetes also allows developers to move from a host-centric infrastructure to a container-centric infrastructure. Kubernetes provides the infrastructure to build a truly container-centric environment. [13]

Kubernetes defines a new component, the pod. The pod is a group of one or more Docker containers, the shared storage for those containers, and options about how to run the containers. Pods are always co-scheduled, run in a shared context and one pod is assigned to one node. A pod models an application-specific logical group, it contains one or more application container which is relatively tightly connected.

“Containers within a pod share an IP address and port space, and can find each other on localhost address. Applications within a pod also have access to shared volumes, which are defined as part of a pod and are made available to be mounted into each application’s filesystem.”

Pods will not survive scheduling failures or node failures. Due to the fact, that pods represent running processes on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed. Each container in a pod has its own image. Kubernetes only supports Docker images. Kubernetes supports public registry like Docker Hub, and also supports private registry, like Docker Registry, to pull the defined images.

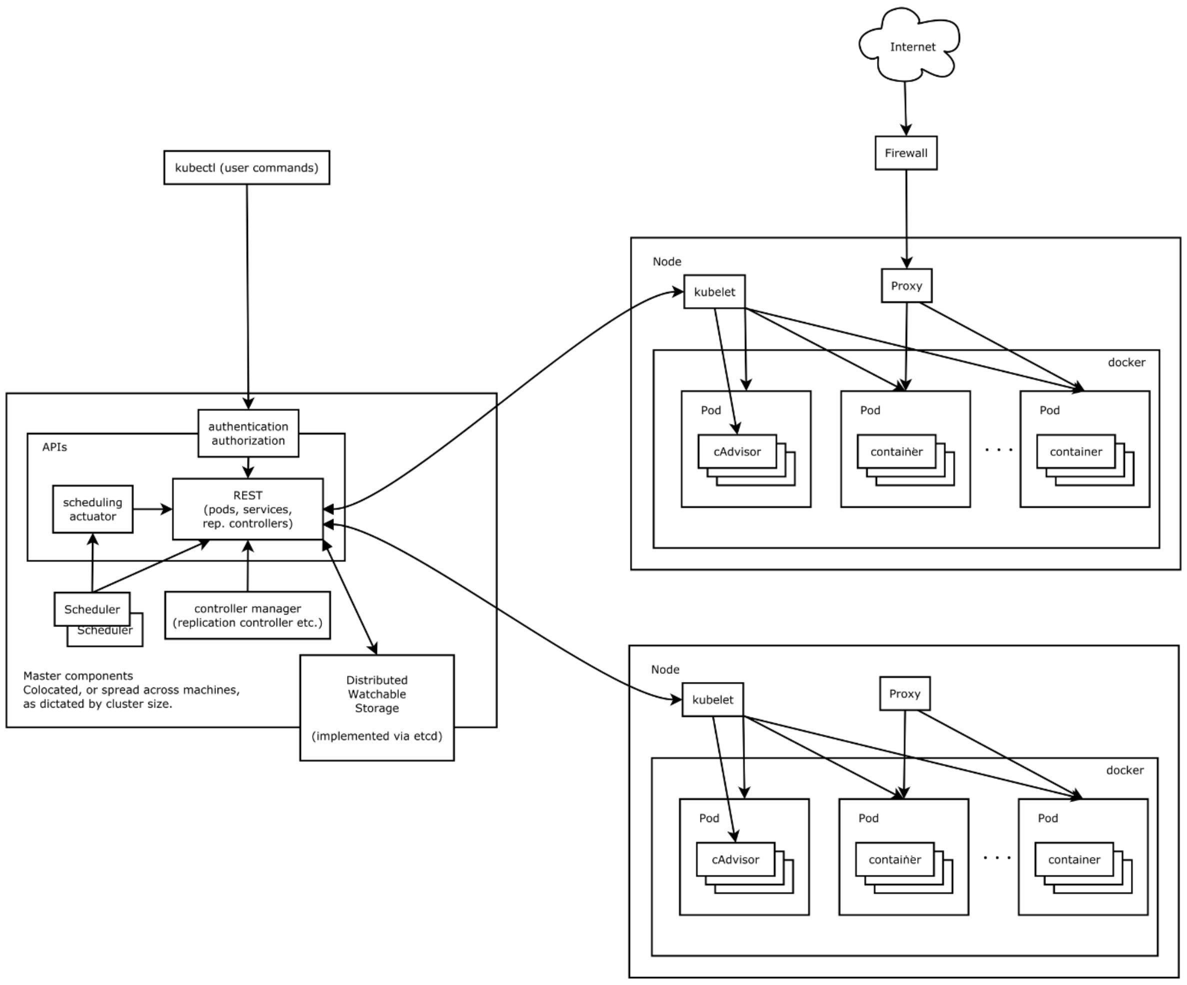
Internet
Firewall
kubectl (user commands)
Node
kubelet
Proxy
docker
Pod
cAdvisor
container
authentication authorization
APIs
scheduling actuator
REST (pods, services, rep. controllers)
Scheduler
controller manager (replication controller etc.)
Master components Colocated, or spread across machines, as dictated by cluster size.
Distributed Watchable Storage (implemented via etcd)

Figure 4 The Kubernetes architecture

The pods are mortal, therefore Kubernetes uses Kubernetes Services for long term services. A Kubernetes Service is an abstraction which defines a logical set of pods and a policy by which to access them.

A Kubernetes cluster can contain many components and they can run many services on each node. Kubernetes node has to run necessary services to run application containers and be managed from the master system.

The master components are those who provide the cluster’s control plane. The master components are responsible for making global decisions about the cluster, like scheduling and detecting or responding to cluster events, like starting a pod. Master components can be run on any node in the cluster. The kubelet is the primary node agent which runs the pod’s containers via Docker, mounts the pod’s required volumes, or periodically executes any requested container liveness probes. The front-end tool of the Kubernetes cluster is the kube-apiserver which expose the Kubernetes API. “The kube-controller-manager is a binary that runs controllers, which are the background threads that handle routine tasks in the cluster. Logically, each controller is a separate process, but to reduce the number of moving pieces in the system, they are all compiled into a single binary and run in a single process.

The kube-scheduler watches newly created pods that haven’t got a node assigned, and selects a node for them to run on it. The scheduler is pluggable, and the Kubernetes will support multiple cluster schedulers and even user-provided schedulers in the future.”

The etcd is used as Kubernetes’ backing store and all the cluster data about the cluster is stored here. Each node runs Docker, which takes care of the details of downloading images and running containers. The architecture of the Kubernetes platform is shown in Figure 4.

The main tool to manage Kubernetes cluster is the kubectl. The kubectl is a command line interface to run commands against the cluster. The kubectl will communicate with the Docker client on every node through the native Docker CLI. Many Docker command has a kubectl equivalents like docker run or docker ps.

Kubernetes also provides a web user interface called Dashboard. Dashboard is enabled since version 1.2 of Kubernetes by default. With the Dashboard the user can inspect and manage the Kubernetes resources and also able to deploy containerized applications.
Kubernetes supports many network implementations which satisfy the Kubernetes fundamental requirements. Kubernetes imposes that all containers can communicate with each other and with all nodes without NAT, a container and others see the same IP address which is allocated to the container. Kubernetes uses the “IP-per-pod” model. Every pod has different IP address and the containers inside the pod can communicate with each other on localhost. One of the most popular network implementation is Flannel. Flannel is a very simple overlay network that satisfies the Kubernetes requirements. With Flannel, every pod can communicate with each other inside the cluster.

All communications from the cluster to the master is terminated at the apiserver. By default, the apiserver is listening on a HTTPS port. For secure communication nodes are provisioned with the public root certificate of the cluster. Master can communicate with nodes on two primary paths. The first is between the master apiserver and the node kubelet process. This connection is terminated on the kubelet HTTPs port. In this path the master can fetch logs for pods or attach to running pods. The second path is from the master apiserver to any node, pod or services through the apiserver’s proxy function. This is a plain HTTP connection.

Kubernetes provides liveness probes to detect the health status of the pods. The diagnostic is performed periodically on a container. There are three possible diagnostic options. The kubelet can execute a command inside the container or perform a tcp check on the specified port or perform a HTTP Get against the container IP address to get the health status. The probe can return with Success, Failure or Unknown status. The user can create a probe description which will be executed by the liveness probes.

Kubernetes also provides a Node Problem Detector daemon which monitors the node health. It collects node problems from various daemons and reports them to the apiserver. It provides a kernel issue detection which supports only the file based kernel logs in version 1.2 of Kubernetes.

Kubernetes has a built in auto scaler component. The Kubernetes autoscaler was implemented as a control loop. Every pod has a defined target CPU utilization. The autoscaler periodically queries CPU utilization of the pods. Then, it compares the value of the pods’ CPU utilization with the target and adjust the number of replicas if needed.

Kubernetes supports external load balancer which can be provisioned from the cloud provider or users can create their own external load balancer.

The Kubernetes cluster has two type of users, the service accounts managed by Kubernetes and the normal users. Normal users are managed by an external, independent service. An admin has to add the users to a Kubernetes cluster. Kubernetes does not have objects which represent normal user accounts, and users cannot be added to a cluster through an API call. This type of account is for humans who will use the cluster. The service accounts are for processes which run in pods.

The service accounts are managed by the Kubernetes API. These accounts are tied to a set of credentials which are mounted into pods allowing in cluster processes to talk to the Kubernetes API.

Docker Swarm

Docker Swarm [14] is a native clustering tool for Docker. It turns a pool of Docker hosts into a single, virtual Docker host. Docker Swarm serves the standard Docker API therefore every tool which uses the standard Docker API can be used with Docker Swarm.

Since the 1.12 version of the Docker Engine, the Docker Swarm is included into the Docker Engine like swarm mode. With swarm mode, the Docker Engine can natively manage a cluster of Docker Engines. To deploy application services to the swarm and to manage swarm behavior is also possible through the Docker CLI.

In a swarm cluster there are two type of nodes: the manager and the worker nodes. The manager node gets the service definition. It divides the service into tasks and dispatches them to the worker nodes. It also performs the cluster management. The worker node receives and executes the task from the manager nodes. The service is a definition of tasks and the tasks are the atomic scheduling units in the swarm. The task is carried in a Docker container and assigned to one node. It cannot be moved to another node, therefore the task can only run or fail on the original node. Figure 5 shows the architecture of the swarm cluster.

There are two types of service deployment: replicated and global service. The global service runs on every node in the cluster and when the new node is added to the cluster the global service will automatically start on it. The replicated service runs in a specified number in the cluster.

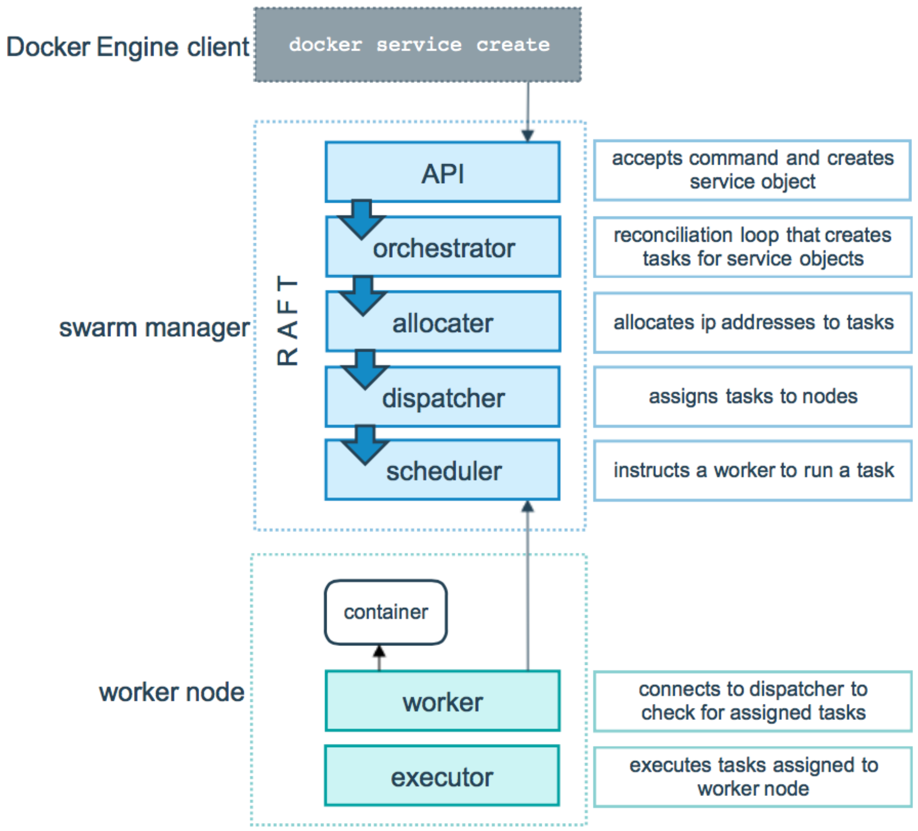
Docker Engine client
docker service create
swarm manager
RAFT
API
orchestrator
allocater
dispatcher
scheduler
accepts command and creates service object
reconciliation loop that creates tasks for service objects
allocates ip addresses to tasks
assigns tasks to nodes
instructs a worker to run a task
worker node
container
worker
executor
connects to dispatcher to check for assigned tasks
executes tasks assigned to worker node

Figure 5 The Docker Swarm architecture

The swarm mode's public key infrastructure is built into the Docker Engine. The nodes in the swarm use Transport Layer Security (TLS) to authenticate, authorize and encrypt the communication between themselves. In the initialization state of the swarm cluster, the first manager node creates a root Certification Authority (CA) to secure the communication, or it uses the specified CA which is set in the initialization command parameter. The swarm should use minimum version 1.2 of TLS to secure the communication.

Creating a swarm cluster consists of many steps. First, the user has to initialize nodes with Docker Engine in swarm mode. After that, the user has to choose the swarm manager to create the cluster. Finally, the user has to add the other nodes to the swarm cluster as swarm workers. The prerequisites of the nodes are that they can run the 1.12 or higher version of Docker Engine in swarm mode and they can reach each other on the network. Optionally, the user can add more swarm manager to the cluster. All the swarm managers will be members of the Raft [1]consensus group. On a built swarm cluster the defined services are deployable and scalable on demand. The scheduling of the created task is made by the swarm manager.

Swarm mode natively supports the overlay network to create a container-to-container network. Multiple services can be attached to one overlay network and the service discovery

[1] https://raft.github.io/

assigns a virtual IP address and DNS entry to each service. Therefore, all the services on an overlay network are reachable with their service names.

Swarm mode makes it possible to publish ports for services, making them available from the outside of the cluster. All nodes are a member of an ingress routing mesh. The routing mesh enables each node in the cluster to accept connections on a published port, and routes all incoming requests to an active container in the cluster. Swarm uses an easy to use external load balancer like HAProxy to route request to a swarm service.

Docker swarm cluster's health can be monitored through the node API in JSON format. All the swarm cluster nodes' health can be queried from any manager node. Since Docker Engine 1.12, it can perform a descripted health check on the application inside the container and the status can be queried through the Docker CLI. The health check method has to be described in the Dockerfile.

"Docker Stacks and Distributed Application Bundles are experimental features introduced in Docker 1.12 and Docker Compose 1.8, alongside the concept of swarm mode, and Nodes and Services in the Engine API. Similarly, a docker-compose.yml can be built into a distributed application bundle, and stacks can be created from that bundle. In that sense, the bundle is a multi-service distributable image format. As of Docker 1.12 and Compose 1.8, the features are experimental. Neither Docker Engine nor the Docker Registry support distribution of bundles." [15]

8.2 Selected service: Swarm

The three tools outlined above have many similar features. The features are compared in the following table:

Table 4 Comparison of Docker Clustering tools

Function	Docker swarm mode	Rancher	Kubernetes
Docker container support	X	X	X
Native command line interface	Docker CLI	Rancher CLI	Kubectl
Graphical User Interface	-	X	X
Private repository support	X	X	X
User authentication	-	X	X
Container level network	X	X	X
Secure communication within the cluster	X	X	X
Application monitoring	X	X	X

Station monitoring	X	-	X
Application scaling	X	X	X
Automatic application scaling	-	-	X
Load balancing	X	X	X
Scheduling	X	X	X
Live update	X	X	X
External storage support	X	X	X
Modular design	-	-	X

Table 4 shows the evaluation of all the features the three different clustering tools have. However there are several functionalities in the table we do not rely on when integrating one of the tools as a Container Orchestrator in MiCADO.

- Graphical User Interface is not needed for Container Orchestration in MiCADO since MiCADO will offer its services through its submitter, and will hide all other internal services to avoid by-passing the submitter. Moreover, graphical user interface will be implemented on the client side of the MiCADO submitter service.
- Since Cloud Orchestrator is considered as an internal service, it is not exposed for external usage, there is no need for User authentication either.
- Decision on application scaling is going to be implemented by MiCADO therefore Automatic application scaling is not needed in our case.
- Live update and Modular design aspects are not relevant in case of MiCADO since for stability reasons we stick to a specified version and do not plan to integrate to separate modules.

Having a look at the table again, the tools are more or less equivalent regarding the features MiCADO will rely on. The selection therefore is not driven by features but by observations and experiences. SZTAKI has deployed all the three systems and used them for several weeks. Here is the summary of the experiences.

Docker Swarm uses default Docker CLI therefore every tool which is compatible with the default CLI can use Swarm. Kubernetes uses a kubectl command line interface which is different from the Docker CLI and the two CLIs are not compatible with each other.

Kubernetes defines a new container format, the pod. Pods contain one or more Docker containers and this is the smallest schedulable component in Kubernetes. This component provides some new features but introduce a new layer and a new service format in the system. The user or developer must create more descriptors to create pods and services comparing to Swarm.

Kubernetes has a modular architecture containing many components. This is a good feature because the architecture is more flexible and swappable but more complex, too. All components must be configured during creation of MiCADO worker node which is more difficult and generates overhead. Based on WP6 experiences these Kubernetes components generate more network overhead than Swarm. Swarm cluster scales faster than Kubernetes.

Docker Swarm is a built-in solution in the Docker engine therefore the cluster building process is easier and does not require any other component. However, Kubernetes has a more complex building process because of the modular architecture and requires more third party components like etcd key-value store. Swarm has a built-in container network however the Kubernetes uses third party container network plugins.

Docker Swarm is an “out of the box” container cluster solution which developed by the Docker developers while the Kubernetes is a more complex and robust system providing extra features but claim a stable and permanent base infrastructure and generate more overhead on the cluster than Swarm.

Because of the new features and components, users and developers who know the Docker technology have a bigger learning curve with Kubernetes than with Swarm.

Comparing the developer communities working with the tools, Kubernetes has a wider community behind than Swarm.

However, it is important to mention that during integration of Swarm in MiCADO, interoperability and interchangeability aspects are taken into account to enable switching between clustering tools later if necessary with reasonable work.

9. Occopus

9.1 Main characteristics

This section gives an overview of the most significant and valuable characteristics of the Occopus solution including multi-cloud support, multiple configuration management support, health monitoring, descriptors, multiple node definition support, scaling, on-the-fly, dynamic reconfiguration of an infrastructure, interfaces and error reporting support.

Occopus supports orchestration activities on various cloud types, i.e. on public, private, multi and hybrid clouds. Occopus does not depend on any cloud type specific feature, therefore it is operational in any circumstances provided that the Cloud API is accessible. The orchestration in Occopus involves the startup of the virtual machines with contextualization and optionally health monitoring remotely. In the current version Occopus is mainly utilizing cloud-init for contextualization, however it is not a requirement and can be skipped if not needed for setting up services. The health monitoring of the virtual machines can be optionally disabled in case they are behind in a private cloud and no access to the virtual machines are enabled.

The current cloud APIs Occopus is able to interact with are EC2, Nova, Occi, Docker, CloudBroker and CloudSigma. These interfaces covers the most important ones where MiCADO is designed to be used.

In some situation it may happen that a predefined node which is reusable in different infrastructures (e.g. mysql) is configured in a way that it requires a certain configuration manager (e.g. chef) while other nodes in the same infrastructure require another type of configuration manager (e.g. puppet). This requires the orchestration to coordinate the infrastructure deployment with several different kind of Config Management services. Occopus is able to handle these situations and, even more, nodes with the same type of Config Manager but on different location does not cause any problem either.

When building an infrastructure running services on a node is the basic goal in every situation. The infrastructure is operational when all the services running on the node are properly working. There are solutions where config management tools (e.g. chef) can check if a service is running correctly. However, in certain cases, for example, when the node does not utilize any configuration management tool Occopus itself should provide some simple health monitoring primitives. These include testing the network access of the node (e.g. ping), testing the access of a port or an url of a node and testing the mysql database connectivity. These have been selected as the most important primitives however, new primitives can be added if required.

Occopus operates based on descriptors that describes the infrastructure layout, the individual nodes, the resources to be used, the configuration management details, the contextualization of the nodes and the way the services on the nodes can be monitored. First of all, an infrastructure description is needed to list the nodes, their names, their numbers (scaling) and their dependencies based on which the order of deployment is calculated by Occopus. The infrastructure description tells Occopus what needs to be instantiated, while the next level descriptors called node definition tells how the nodes should be created. A node may have multiple implementations (e.g. one for Amazon, one in the user's local OpenStack cloud, etc.) and the selection is done by Occopus, however the available implementations can be filtered for example to exclude the unwanted implementations.

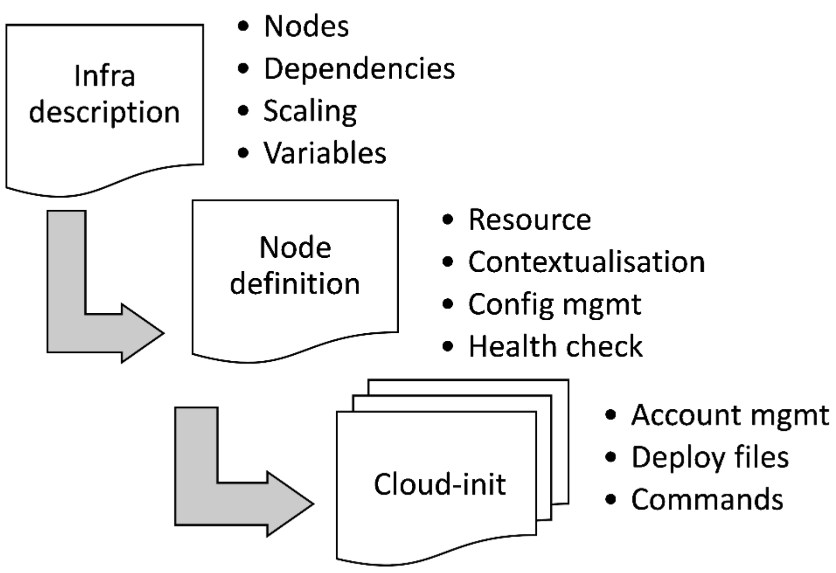

Figure 6 Occopus descriptors

The node definition contains four main sections. The first and obligatory section called "Resource" defines the target resource (cloud interface endpoint, image id, etc.) where the virtual machine (or container in case of Docker) must be instantiated. The next three sections are optional. The "Config management" section defines the configuration manager tool to utilize (e.g. chef) and the main parameters associated with it (e.g. chef server endpoint, role, validator key, etc.). The "Health-check" section details how the service - running on the node - can be monitored externally. Currently supported primitives are ping, port checking, database access, URL checking. Finally, the section called "Contextualization" contains the description how the instantiated virtual machine is contextualized. Optionally, it may point to an additional file containing the details of contextualization, for example in case of cloud-init, it is the user-data file. With the user-data file for cloud-init one may perform account management i.e. create users/groups, may deploy credentials i.e. public keys, may deploy files/scripts, may install software packages by OS package management or configuration manager (e.g. chef/puppet), but may execute any command or script to perform any manipulation and preparation at start-up. However, Occopus is not stuck to cloud-init, contextualization handlers are also implemented as plugins, since different clouds may require own contextualization facility.

Each node has a node definition which defines what to instantiate. In case a node has multiple implementations, one may perform selection by filtering the appropriate one through the infrastructure description. For example, a node has a definition where the service deployment is implemented by a config manger (e.g. chef) while another version implements service deployment with another config manager or without so the infrastructure developer can decide which version to use. It is also possible that one node definition describes the details of the instantiation for one cloud while another describes for another target cloud. Selecting among these definitions means selecting the appropriate target cloud.

Horizontal scaling functionality is a kind of natural requirement towards the orchestration tools where the number of instances of a certain node can be increased and decreased. When increasing the number of instances, it is the task of the infrastructure to recognize the new instance and include it in the flow of operations. Occopus only performs the creation and contextualization of the new instance. It is the task of the deployed infrastructure to recognize the new instance. For example, Consul can be used for this purpose as it is detailed in some use cases at [3]. Hard limits for the number of instances can be defined in the infrastructure description as minimum and maximum. At start up Occopus creates the minimum number of instances for a node. Occopus will not let the number of instances go beyond these limits. Vertical scaling - where the capacity of the virtual machine is updated - can be implemented based on the multi node definition feature of Occopus in case of stateless services.

There are rare situations when an already running infrastructure must be updated not only from scaling aspect, but also its layout i.e. a new node must be added or one of the running ones must be excluded from the entire infrastructure. A possible use case for that is when a database backend is replaced with a different one. For this functionality, Occopus gives the possibility for the building phase to specify an already existing infrastructure, which is then rebuilt. Occopus first takes the existing infrastructure then compares it to the target one and performs realizing the difference i.e. creating new nodes and destroying unwanted ones. Similarly to the scaling functionality, the infrastructure must be prepared for such situation since Occopus itself does not alter the configuration of an already existing node instance.

Building and maintaining an infrastructure can be performed through different interfaces. Occopus has CLI and REST API. Both, provides the main functionalities, like building, maintaining, scaling or destroying. Moreover, the CLI and the REST interfaces can be used in an alternate way, which means after building an infrastructure by the CLI one may continue the maintenance of the infrastructure with the help of the REST API. During maintenance the opposite direction is also a supported use case. There is a third interface namely the library API that enables developers to integrate the functionalities of Occopus as a library and use it by invoking the API methods.

Both during development and maintenance Occopus provides error reporting mechanism and logging to ease the development and maintenance of the infrastructure. When developing an infrastructure the focus is on creating the appropriate descriptors for the node and for the infrastructure. Occopus performs syntax and in some cases semantic checking on the descriptors. Once a descriptor, e.g. cloud-init configuration has some syntax error, error report details the exact location and the nature of the error in a user friendly form. During the maintenance of the infrastructure, errors occurring during the communication of the Cloud APIs are reported in details and the logging information enables the analysis of the history of activities Occopus performed.

Occopus is able to build, maintain, scale and destroy the infrastructure. As a summary, Occopus is a light-weight, easily deployable and usable orchestration tool with high level of flexibility and cloud-independence.

9.2 Latest developments

Occopus is developed by MTA SZTAKI. The aim of the developments during the COLA project is two-folded. One main direction in developments is to increase the user experience by fixing handling issues, by improving interactions and by introducing user-friendly features.

Another important direction in developments is to improve functionalities required by the COLA project by adding new plugins and by updating behavior to fit to the MiCADO architecture. In the last 6 months, there have been three releases with the following release notes:

Release v1.5

- Reimplemented cloudbroker plugin: handle instances, not jobs
- Remove cloud-broker node resolver (replaced by cloud-init)
- Add multiuser support in handling redis server
- Improve error handling and logging in cloudsigma, ec2 and occi plugins
- Improve nova plugin to handle interruption
- Add infra and node name syntax checking
- Add new Occopus installer script
- Improve parallel node creation

Release v1.4

- Improve node handling in cloudsigma plugin
- Improve floating ip handling in nova plugin
- Precise syntax error reporting for descriptors
- Unique VM name for nodes as default
- Introduce user defined VM name templates
- Improve error/exception handling and reporting
- Fix logging and evaluation in schema checker
- Fix calculating default scaling min, max
- Restructure health-check reporting
- Deprecate 'network_mode' attribute in docker plugin
- Introduce attach and detach functions in rest
- Compatible REST and cmd-line functions

Release v1.3

- New Puppet config-manager plugin: server-free, called "puppet_solo"
- Remove external redis config for occopus-import command
- Remove attribute dependency from plugins
- Reimplement floating_ip handling in nova plugin
- Fix bug in filtering
- New tutorial for puppet_solo plugin
- New tutorial to introduce autoscaling with prometheus

Here are the most important developments, serving the requirements of the MiCADO integrations in the COLA project:

- Occopus configuration has been updated in order to be docker compliant. Internal configuration was modified to separate the Occopus service from its database service, called redis. After the update, Occopus and its redis database can be executed in two separate containers under docker.

- Occopus command-line functions has been modified in order to let the user handle the same infrastructure through REST-API and with command-line functions in an alternate way. The infrastructure deployed through the REST API can be manipulated by command-line functions and vica versa.
- Occopus docker plugin has been tested and updated to be compatible with the latest docker version.
- Health-check reporting has been updated in order to be simpler and more understandable for the user.
- Improving garbage collection in cloudsigma plugin in order to handle deployment interruption in a way to deallocate all partially allocated resources.
- Introduce new Occopus installer script to provide a one command way of installation and default configuration. Currently, Occopus can be installed as docker container, too.
- New cloudbroker plugin which allocates cloudbroker instances instead of cloudbroker jobs/applications. With this new plugin it is possible to deploy the MiCADO architecture and to scale up and down the number of MiCADO worker nodes.

Beyond the current state, Occopus is continuously developing towards the MiCADO requirements raising during building the MiCADO prototype.

10. Implementations of MiCADO Orchestration Layer

In Chapter 6 the development plan of MiCADO Orchestration Layer has been introduced detailing the architecture, components, functionalities and behavior at high-level. This chapter details the current status of MiCADO at the implementation level.

There are already several versions of MiCADO (1/a, 1/b, 2/a 2/b) developed and released by the time of this deliverable. We give a short overview of these releases. The very next step in the development is also described.

Based on chapter 7, WP6 will use Occopus as the cloud orchestrator in the MiCADO platform. The initial versions of MiCADO has been implemented by using Occopus as cloud orchestrator. Similarly, according to chapter 8 Swarm will be used as container clustering tool in the first versions of MiCADO. In order to realize the controlling loop in MiCADO, a monitoring system was also necessary. The current selection for the monitoring system is Prometheus about which the next deliverable D6.2 will contain a detailed description.

10.1 MiCADO v0

This version has two sub-versions: v0/A and v0/B. MiCADO v0/A (see Figure 7) uses only Linux services to develop and create configuration files in a Linux based environment. We soon discovered that supporting the latest virtualization technologies, such as Docker containers makes MiCADO better. So we implemented the version v0/B using Docker. Both versions are highly scalable.

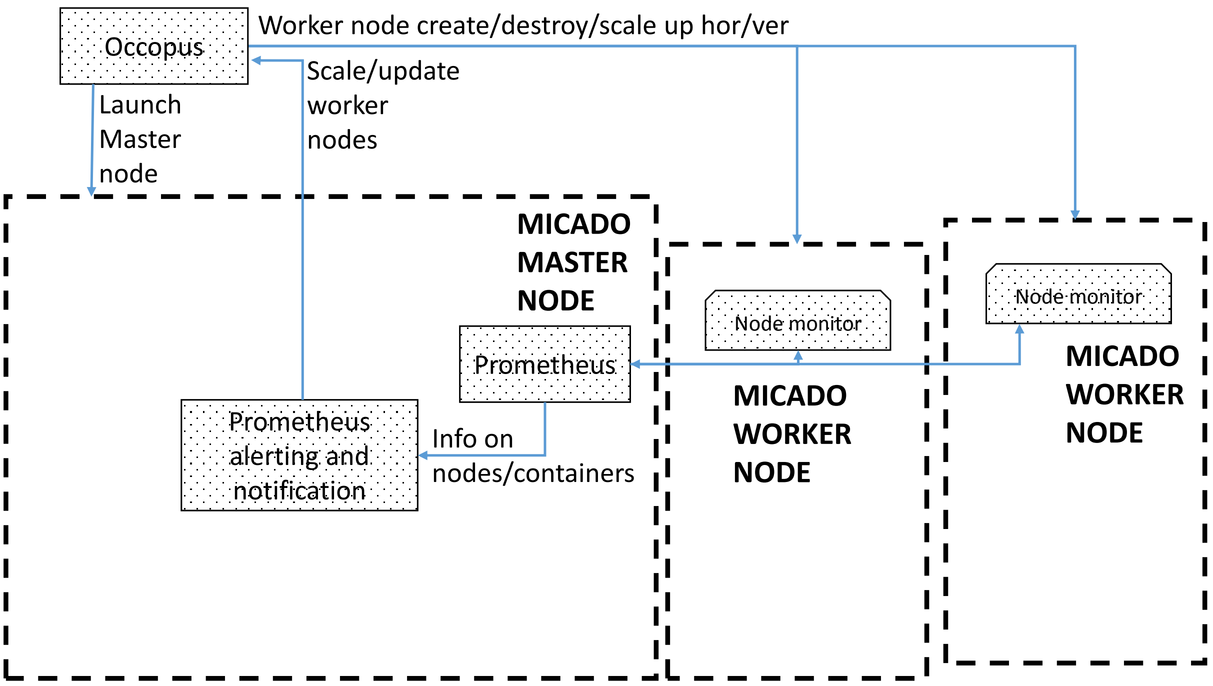

Figure 7 Architecture of MiCADO v0/A

Scaling is executed in two layers: first user requests have to be shared in a load balancer layer, and then the requests have to be executed in the worker node where the computational tasks are getting done. MiCADO automatically scales both the load balancers (not part of Figure 7) and the worker nodes and makes sure that the implemented application works as it is expected with the optimum number of resources.

Version v0/A utilizes Linux based virtual machines to run all MiCADO services as Linux services. This is the base infrastructure of all further versions and all developments are aimed at making it better while using the same concept. In this version user applications are hard coded into the worker node start-up configuration files because knowledge of the cloud-init files is required.

Version v0/B extends v0/A by implementing Docker, a well-known virtualization technology. In this version every service is dockerized to create shorter configuration files which are more understandable by the users. Implementing user applications doesn't require intensive knowledge of the cloud-init files but the application still have to be specified in these files. Instead of creating all the configuration files and setting up the runtime environment, users can simply paste their application with a "docker run" command into the end of the application node descriptor file.

Comparing to the development plan, presented in Chapter 6, most of the components are missing, but automatic scaling of worker nodes is already operational. In this version the entire MiCADO infrastructure including the Master and Worker nodes were deployed by Occopus as an external service instead of being part of the Master node. This modification of the original design was made to simplify the development and testing, but later it will be eliminated.

10.2 MiCADO v1

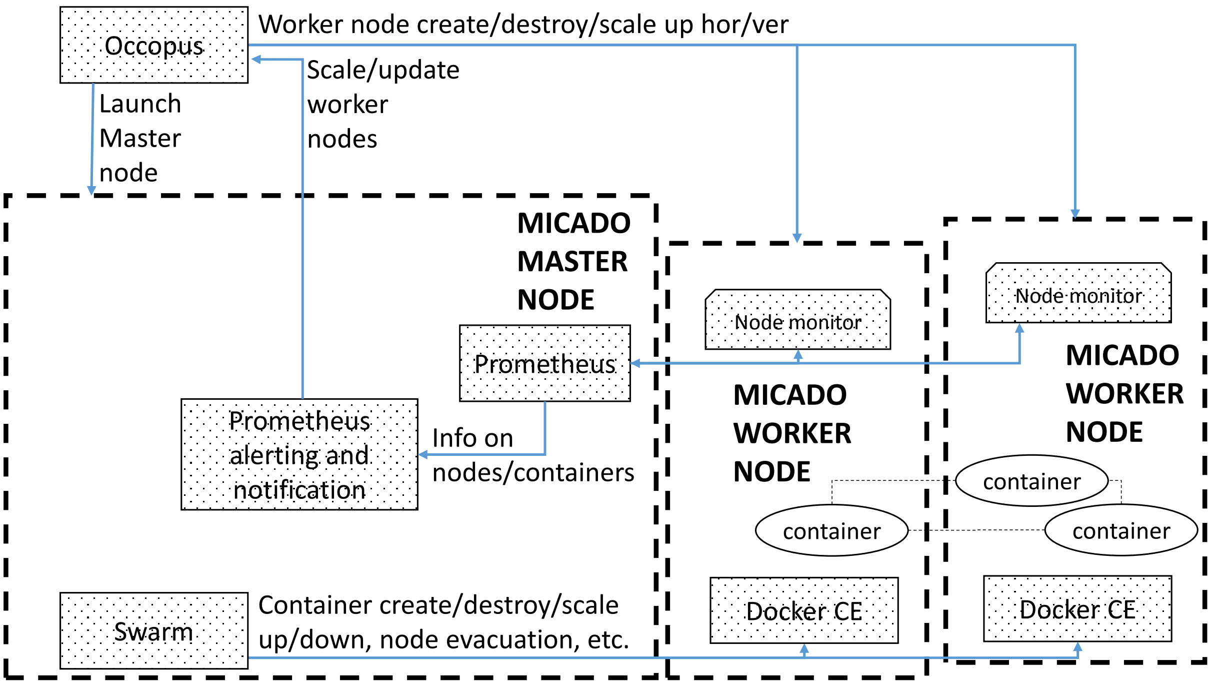

Figure 8 Architecture of MiCADO v1

In MiCADO v1 (see Figure 8) WP6 improved two major features. The first one was that not every application makes advantage of a fully scalable load balancing layer so there were unnecessary virtual machines that was eliminated. The second and probably the bigger one was to make it easier to change user applications. The two problems were solved by implementing Docker Swarm. With the help of Swarm users do not need to write their applications into the configuration files before they build up the infrastructure and do not need to know cloud init configuration files in advance. Instead, with the help of Swarm they can start applications with a simple command. Swarm also has a built-in load balancer that can be used in case the application is more a computation heavy and not used by a huge number of users and to deal with user requests (V1/A). In other situations the previously implemented load balancing layer is necessary (V1/B).

MiCADO v1/A version extends v0/B by adding Docker Swarm to the orchestration layer. In this version there is no separate load balancing layer because Swarm's built-in load balancer is used. It can only scale the worker nodes. As a result users do not need to modify cloud-init files at all. Users can start the application as a Docker service on the Master node through the Swarm Docker API. This requires less knowledge of the cloud-init files making easier replacing the user application later since the application is not hard-coded anymore into the cloud-init files.

Version v1/B extends v1/A with a scalable load balancing layer to support user heavy applications where Swarm's load balancer is a bottleneck and a fully scalable load balancing layer is required. WP6 implemented the same layer used in v0/B which means that both the load balancing and the application layer can be scaled up/down independently. User applications can be started as Swarm services as in the v1/A version.

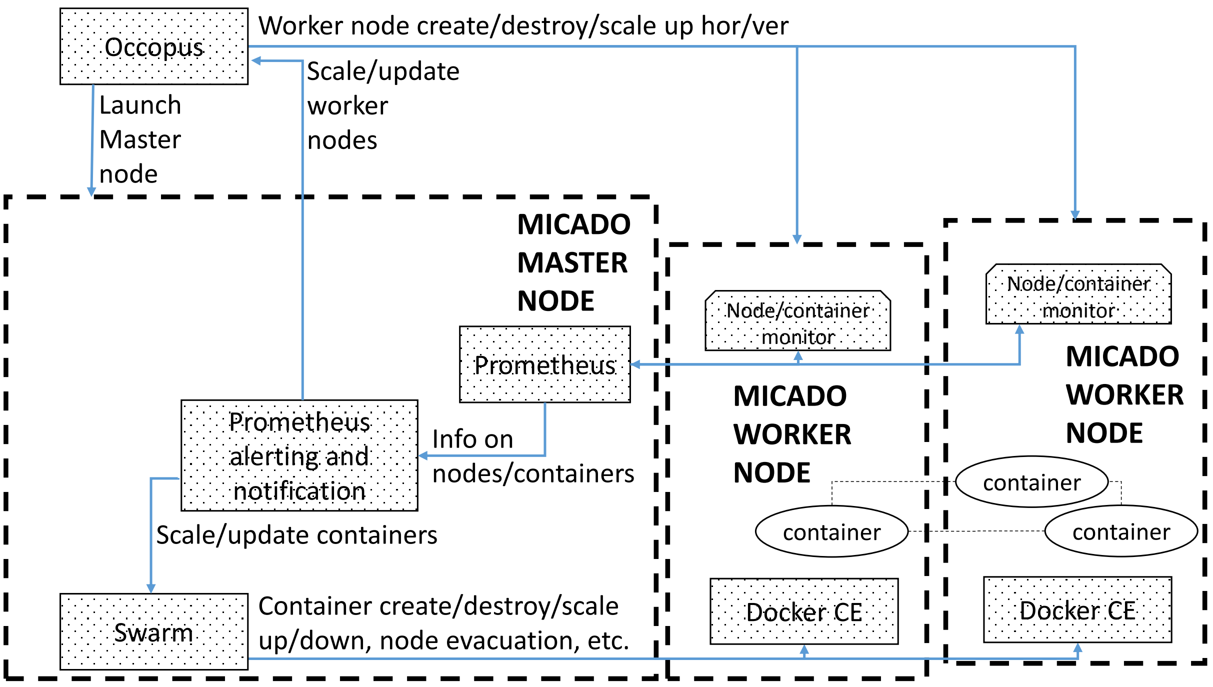

Figure 9 Architecture of MiCADO v2

This version (see Figure 8) implemented the orchestration layer in the MiCADO platform combining with Occopus, Swarm and Prometheus to realize the controlling loop for the resource allocation and release. Based on the information Prometheus collected on the load of the worker nodes, the properly configured alerting system (which is a partial realization of policy keeper) of Prometheus is able to instruct Occopus (through its REST interface) to scale up/down the MiCADO worker nodes. Occopus is still implemented as an external service in this version. Swarm has been integrated to act as a container orchestration component.

10.3 MiCADO v2

In MiCADO v2 (see Figure 9) we implemented container monitoring, and with the help of the collected information, now the deployed application can be also scaled up/down in the container level. It means that changing the number of containers of the application and the worker nodes is only required if no resource left on any of the worker nodes. This gives faster feedback in the control loop and with container level scaling this solution can meet real time demands better. Deploying multiple applications on the same infrastructure has improved as well and limiting the resource usage of the applications works. Application specific alerts in Prometheus are generated and removed automatically when container application starts or finishes.

In MiCADO v2 (see Figure 9) development targeted the implementation of the control loop of scaling up and down the number of container instances for a given container service. Comparing to the previous version, the link between the box denoted by ‘Prometheus alerting and notification’ and the box denoted by ‘Swarm’ has been implemented. This new version is now able to continuously monitor the resource consumption of the container services (‘Node monitor’ on worker node in Figure 8 became ‘Node/container monitor in Figure 9) and scales up if resource consumption is above a certain threshold. Occopus is still implemented as an external service in this version.

10.4 MiCADO v3 (under development)

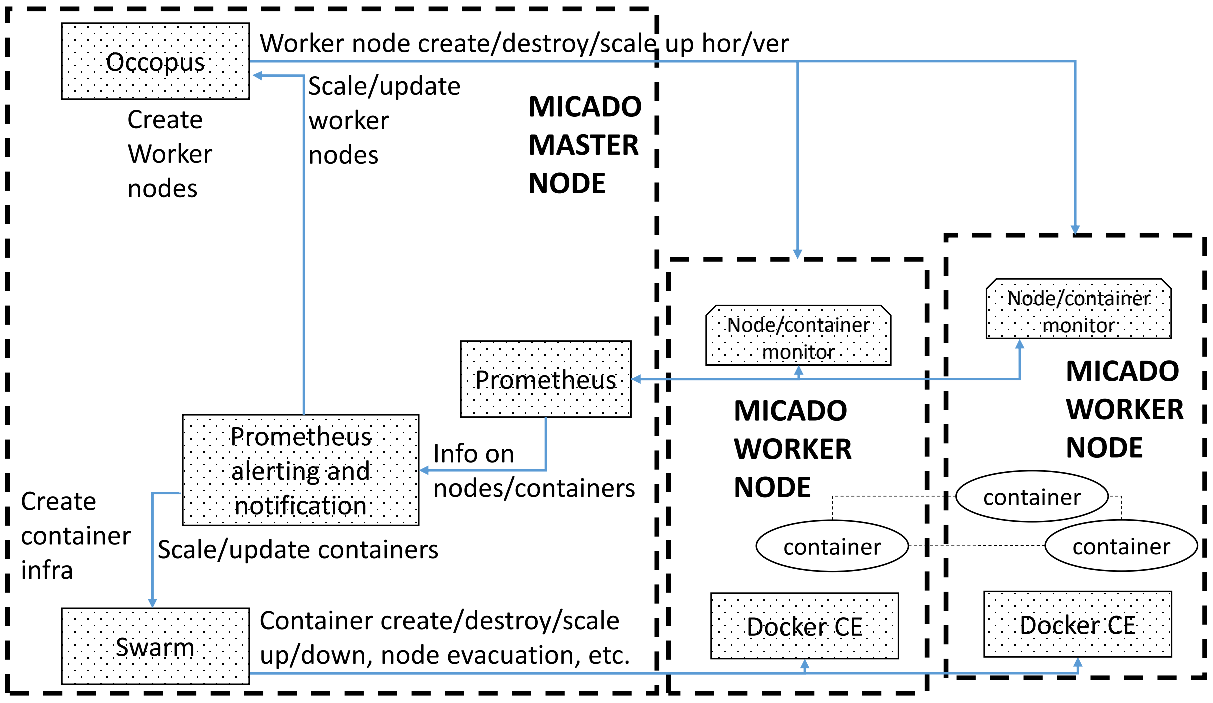

Figure 10 Architecture of MiCADO v3

In MiCADO v3 (see Figure 10) the next proposed step, which is currently being implemented is the integration of Occopus in the MiCADO master node.

Occopus will be deployed as a microservice inside the MiCADO Master node to support the creation and shutdown of the worker nodes. Once Occopus is integrated, the deployment of the Master node must be automatized. For this purpose, there are several alternatives: using Occopus with a one node description, using docker compose with a description of containers services on the Master node or using any other orchestrator tool.

11. Conclusions

In this deliverable we introduced the design and implementation of the MiCADO framework starting with the big picture in Chapter 5, continuing with the design plan of the orchestration layer in Chapter 6, detailing the investigation and selection of the tools for the implementation in Chapter 7, 8, 9 and finally showing the phases of MiCADO development in Chapter 10.

The design plan of MiCADO covers the deployment, orchestration for cloud resources and containers, the monitoring of cloud resources and containers, the decision making functionality of the controlling loops and finally the submitter functionality.

For cloud deployment and orchestration Occopus has been chosen, for container orchestration Swarm with Docker has been chosen. For monitoring the Prometheus tool has been selected. Based on these tools several implementations of MiCADO have already been done.

Implementation versions of MiCADO have shown the progress of the MiCADO framework towards the design plan step by step. Deployment and orchestration in MiCADO have been successfully implemented by the time of writing this deliverable, since the key components namely Occopus and Swarm are fully integrated together with Prometheus to form the double controlling loops for resources and containers.

The next step for the implementation of the MiCADO framework is MiCADO version 3 which has been introduced in Chapter 10.4. Beyond MiCADO v3 the implementation must continue with the MiCADO submission service and later with the Optimizer component. Implementation of the MiCADO submitter and Optimizer requires a strong cooperation with WP5 which defines the format of the infrastructure description and policies.

Releases of the Occopus tool are listed on the official Occopus webpage [3] under the 'Releases' menu. The installation procedure of the latest release is described in the Occopus manual located at http://occopus.lpds.sztaki.hu/user-guide under the 'Setup' section. The source code of Occopus (including all previous versions) can be downloaded from https://github.com/occopus .

Releases of the MiCADO framework are listed on the official COLA website [16] under the 'Tutorials' menu. Here one can download the package, personalize the settings and deploy MiCADO based on the detailed step-by-step description of the tutorials located at http://project-cola.eu/micado-tutorials/ .

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

[4] Linh Manh Pham, Alain Tchana, Didier Donsez, Noel De Palma, Vincent Zurczak, et al. Roboconf: a Hybrid Cloud Orchestrator to Deploy Complex Applications. 2015 IEEE 8th International Conference on Cloud Computing, Jun 2015, New York, United States. <10.1109/CLOUD.2015.56>. <hal-01228353>

[5] X. Wang, Z. Liu, Y. Qi and J. Li, "LiveCloud: A lucid orchestrator for cloud datacenters,"4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, Taipei, 2012, pp. 341-348. doi: 10.1109/CloudCom.2012.6427544

[6] Caballer, M., Segrelles, D., Moltó, G., and Blanquer, I. (2015) A platform to deploy customized scientific virtual infrastructures on the cloud. Concurrency Computat.: Pract. Exper., 27: 4318–4329. doi: 10.1002/cpe.3518.

[7] Heat, https://wiki.openstack.org/wiki/Heat

[8] Cloudformation, https://aws.amazon.com/cloudformation/

[9] Marpaung, Sain, & Hoon-Jae Lee. (2013). Survey on middleware systems in cloud computing integration. Advanced Communication Technology (ICACT), 2013 15th International Conference on Advanced Communications Technology, 709-712.

[10] Cloudify, http://getcloudify.org/

[11] Binz, Breiter, Leyman, & Spatzier. (2012). Portable Cloud Services Using TOSCA. Internet Computing, IEEE, 16(3), 80-85.

[12] Terraform, https://www.terraform.io/

[13] Kubernetes overview, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

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

[15] Docker stacks and bundles, https://docs.docker.com/compose/bundles/

[16] COLA website, http://project-cola.eu/