



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:

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

Due date of deliverable: 31/04/2018

Actual submission date: 31/04/2018

WPL: Peter Gray

Dissemination Level: PU

Version: 1.0

1 Table of Contents

1 Table of Contents	2
2 List of Figures and Tables	4
3 Status, Change History and Glossary	5
4 Glossary	5
5 Introduction	7
6 Requirements gathering	7
7 Performance benchmarking: Methodology and Tools	9
7.1 Code repository - GitHub	9
7.2 Continuous integration/automated testing tool - Travis & Jenkins	11
8 Performance benchmarking of MiCADO services	11
8.1 Occopus (SZTAKI)	12
8.1.1 Benchmarking Occopus	12
8.1.2 Swarm deployment by Occopus	14
8.1.3 MiCADO deployment by Occopus	16
8.2 Prometheus	18
8.2.1 Automated stress testing	18
8.3 Docker Swarm	19
8.3.1 Automated setup of containers using MiCADO	19
9 Test plan for the security components	20
9.1 Crypto Engine	21
9.1.1 Test Items	21
9.1.2 Test Features	21
9.1.3 Features not to be tested	21
9.1.4 Approach	22
9.2 Credential Manager	22
9.2.1 Test Items	23
9.2.2 Test features	23
9.2.3 Features not to be tested	23
9.2.4 Approach	24
9.3 Credential Store	24
9.3.1 Test Items	24
9.3.2 Test features	25
9.3.3 Features not to be tested	25
9.3.4 Approach	25

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

9.4 Security Policy Manager	25
9.4.1 Test Items	25
9.4.2 Test features	25
9.4.3 Features not to be tested	26
9.4.4 Approach	26
10 Conclusion	27
11 References	28

2 List of Figures and Tables

Figures

Figure 1 Occopus benchmark test with 10 nodes infrastructure	13
Figure 2 Occopus benchmrk test with 100 nodes infrastructure	13
Figure 3 Swarm deployment page on Jenkins	15
Figure 4 Occopus swarm deployment in different clouds	16
Figure 5 MiCADO deployment time in different clouds	17
Figure 6 Grafana displaying real-time Prometheus metrics (average and individual CPU load on nodes and containers) during automated load testing	18
Figure 7 Docker Swarm Visualiser showing test_stress containers (smaller green boxes) running inside worker virtual machine nodes (larger boxes) as they are scaled-down.....	19

Tables

Table 1 Status Change History	5
Table 2 Deliverable Change History	5
Table 3 Glossary	6
Table 4 Occopus Swarm deployment average.....	16
Table 5 MiCADO deployment average.....	17

3 Status, Change History and Glossary

Status:	Name:	Date:	Signature:
Draft:	Peter Gray	20/04/2018	Peter Gray
Reviewed:	Gabor Terstyanszky	26/04/2018	Gabor Terstyanszky
Approved:	Tamas Kiss	30/04/2018	Tamas Kiss

Table 1 Status Change History

Version	Date	Author	Modification
v0.1	8/03	Peter Gray	Structure document
v0.2	26/03	James Deslauriers	Add content sections 8.2 / 8.3
v0.3	04/04	Mark Emodi	Add content 7.2, 7.3 (travis), 8.1
v0.4	10/04	Mark Emodi	Add content under section 8.1.x
v0.5	12/04	Nicolae Paladi	Add security section 9
v0.6	16/04	Peter Gray / Bogdan Despotov	Add requirements gathering section 6 and performance benchmarking methodology and tools section 7
v1.0	18/04	Peter Gray / Bogdan Despotov	Consolidate content, add Introduction and conclusion

Table 2 Deliverable Change History

4 Glossary

CLI	Command Line Interface
CM	Credential Manager
CO	Container Orchestrator
CS	Credential Store
GUI	Graphical User Interface
HTTPS	Hyper Text Transfer Protocol (Secure)
ON	OpenNebula
SPM	Security Policy Manager
SQL	Structured Query Language
SSD	Solid State Drive
TOSCA	Topology and Orchestration Specification for Cloud Applications
REST	Representational State Transfer
SSH	Secure Shell
YAML	Yet Another Markup Language / YAML Ain't Markup Language
QoS	Quality of Service

Table 3 Glossary

5 Introduction

The purpose of the COLA project is to develop a generic framework that will enable cloud applications to utilise the dynamic and elastic capabilities of underlying IaaS cloud infrastructure. The framework should allow for the performance of these applications to be optimised taking both execution/response time and also economic cost and viability into consideration. However, to achieve this we must first ensure adequate performance of the MiCADO services layer and underlying cloud infrastructure.

This deliverable relates to objective **4.3: To investigate optimal container size and infrastructure requirements of microservices (MiCADO services)** and **4.4: To assess infrastructure performance for the optimisation of cloud applications**. According to DoW this deliverable reports on the initial functional and non-functional cloud infrastructure and access layer level requirements of typical MiCADO microservices, and analyses the first performance benchmarks of these microservices.

This deliverable will inform WP5 regarding QoS policies including deployment and scaling of services and their security policy, and WP6 to advise on price/performance optimisation.

In Section 6, we outline our approach to understanding performance requirements at application-level by studying the requirements of the COLA use-cases described in D8.1 and D8.2, the use-case templates documented in D5.4, and more specifically at the service-level (MiCADO Services) requirements documented in D6.2. In Section 7, we describe our approach to performance benchmarking the MiCADO services (methodology and tools). In Section 8, we document the results from the performance benchmarking of the core MiCADO Services (Occopus, Prometheus, Docker SWARM). In Section 9, we outline a test plan for core security components comprised on the Crypto Engine, Credential Manager, Credential Store and Security Policy Manager.

6 Requirements gathering

WP4 is tasked with providing the cloud access layer and testbed/production cloud infrastructure optimised for the MiCADO microservices. In this respect, most of the requirements gathering is undertaken by the study and correlation of the outputs from project deliverables.

We examined **D8.1: Business and Technical Requirements of COLA Use-Cases**, and **8.2: Customisation and Further Development of Software Applications** to extrapolate the high-level requirements common to the use-cases and assess whether the performance of the current MiCADO implementation in the cloud is likely to meet the demands of the applications. Details of these requirements can be found in the above referenced documents and a short summary is provided below.

Outlandish / The Audience Agency Finder Application

Querying needs to be in the range of hours and not days. Some degree of elastic scalability to respond in a timely fashion to soak up load. Spin up time to be less than AWS' auto-scaling groups for a standard AMI based deployment. We would prefer the scaling out to be similar to the speed at which a Kubernetes cluster can be deployed.

Saker Solutions / SakerGrid simulation platform

The system should have a target of a linear increase in performance from the additional cloud resources deployed. The time to start up a machine instance should be similar to that to power up a desktop PC with an SSD – i.e. no longer than one minute.

Inycom / Eccobuzz platform

The performance of the system depends strongly on how many crawlers are configured, how often they are launched and how much information they gather to be processed later. The objective in this use case for the regional government is that the crawlers are run every 2 hours and in the meantime all the information collected has been processed.

CloudSME / Data Avenue

The whole cluster should respond (deliver pages) fast (20 up to 100 milliseconds). This varies by complexity of the delivered pages and type of page (e.g. to be cached before or not).

To further understand the application-level performance requirements we also studied the application description templates (ADTs) and services relating to each use-case including the additional Data Avenue use-case, as documented in **D5.4: First Set of Template and Services of Use Cases**. Finally, technology selection and design decisions related to the MiCADO framework and its implementation have been influenced by D6.2 Prototype and documentation of the monitoring service.

7 Performance benchmarking: methodology and tools

In order to validate the automated scalability features provided by MiCADO and evaluate the performance of the MiCADO prototype implementation we refer to Section 10 of **D6.2: Prototype and Documentation of the Monitoring Service**. From the experiments conducted, we learn the time it takes to create and destroy the MiCADO infrastructure and scale up/down the application nodes.

Operation	Time (Sec)
Create infrastructure	320
Destroy infrastructure	15
Scale up app node	300
Scale down app node	12

The results provide evidence that MiCADO performs as required in that the times recorded for each scaling event are well within the expected performance ranges. However, to assess in more detail the potential impact on application performance, and efficient resource utilisation we must first understand the service-level performance requirements. Therefore, in Section 8 of this deliverable we drill-down further to test the performance of the core MiCADO services, comprised of (1) Occopus for the orchestration of the MiCADO infrastructure, (2) Docker Swarm for the automated setup of containers, and (3) Prometheus as the monitoring tool.

It is important to perform baseline testing for core microservices periodically. It also makes more sense to run microservice tests at unit level. For this reason, we have decided to use common tools such as GitHub, for source code management, Travis for unit testing and Jenkins for performance testing. The results from the tests that follow in Section 8 will provide a performance baseline allowing for further performance comparisons to be made per release. This will make it easier to measure any performance degradation at later stages of development and allow the developers to pinpoint possible cause much more quickly and accurately.

7.1 Code repository – GitHub

GitHub is chosen as the source code management platform that will contain all the code related to the COLA project. It is a web based hosted solution, allows for collaboration and is integrated with Travis, a tool used for unit testing. Currently the development repository for MiCADO is at: <https://github.com/micado-scale/>

MiCADO releases are posted on the project website, while this repository is used for collaboration between the teams at UoW and SZTAKI.

The guidelines for the development of MiCADO are the following.

Code style

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

We use the following coding style under the development process. For Python we use PEP-8 [1] as a standard coding style and for YAML we use the 4-space wide indent.

Versioning

For versioning purposes, we adapted semantic versioning 2.0.0 [2]. Consider a version format of X.Y.Z (Major.Minor.Patch). Given a version number major, minor, patch increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

This infers the following renaming of previous releases:

V3 -> 0.3.0 (further modifications will result in 0.3.1, 0.3.2, etc.)

V4 -> 0.4.0

Previous releases (V1 and V2) should remain untouched.

Repositories

We should use separate GIT repositories for each separate logical unit within the micado-scale github.com organization as follows:

1. Each component should be placed in its own repository with the prefix component-, e.g., component-alert_manager.
2. The MiCADO repository is the main repository, containing "glue" files (e.g., cloud-init and/or docker-compose related ones).
3. Documentation should go into the docs repository in either markdown or restructured text format.

Each repository should contain a README.md file explaining the purpose of the repository, basic functionality, and pointers for further documentation (in the docs repository). The master branch of each repository should contain an ISSUE_TEMPLATE.md for issue reporting. This should be copied over from the master branch of the MiCADO repository.

Branching

We adapt the successful GIT branching method [3] with the following modifications:

1. The master branch should always represent the latest stable release.
2. The develop branch is for development.
3. In each repository, from its develop branch, for each major release, a release branch should be created.
4. Release branches should be named based on their major and minor versions: v_MAJOR_MINOR.x
 - E.g., for the 0.4 releases, the branch should be called 0.4.x (x is literal, represents that all 0.4 releases, e.g., 0.4.0, 0.4.1, etc. are based on this branch).
 - Releases

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

- For 0.4.0 and 0.3.0 we should use the following procedures, as 0.4.0 is considered a new implementation, but 0.3.x requires fixes and refactoring.

For the current (0.4.x) release branch:

1. Merge the release branch to develop in each affected repository.
2. Merge the release branch to master.
3. Create tag with the release number (e.g., 0.4.0).

For the 0.3.x release branch:

Create a tag on the release branch (no merging with develop branch, but selected fixes can be added).

Commit Guidelines

We adopt the angular.js commit guidelines [4] with some modifications. We do not pre-define scopes. The scope part should describe the affected part in the commit message.

Dev Docker registry

We use a private docker registry to develop MiCADO components. This registry is running on CloudSigma and protected with basic authentication. The address of the registry: cola-registry.lpds.sztaki.hu. To get access, contact with DevNull group (devnull@lists.lpds.sztaki.hu) at SZTAKI. The naming convention in the registry: username/imagename:version

To use the registry during the development:

- Replace the default image name with dev image name
- Insert this into the cloud-init file

7.2 Continuous integration/automated testing tool: Travis & Jenkins

Under the development process we are planning to use Travis as a **unit testing tool**. Travis is a hosted solution, so we do not have to maintain the Travis infrastructure. It is distributed, easy to use and free, if open source is the project.

Jenkins is an industry-standard open-source **Continuous Integration server**. It downloads code from a repository, resolves dependencies, builds the code, tests it and then deploys it. While Jenkins is typically used for building and deploying software, it can be easily repurposed for more interesting tasks. It is an effective way to monitor the execution of externally-run jobs, such as cron jobs, even those that are run on a remote machine. Jenkins keeps those outputs and makes it easy to see when something goes wrong. It can be used to boost productivity and automate repetitive tasks using a consistent and easy to use GUI, providing an audit trail of each run, as well as access to the output of the run.

8 Performance benchmarking of MiCADO services

MiCADO is an implementation of a generic and pluggable framework that supports the optimal and secure deployment and run-time orchestration of cloud applications. It is based on the concept of microservices and designed to work in a cloud environment. Cloud execution offers the possibility to optimize resource allocation and thus manage resource cost dynamically. MiCADO implements an autoscaling functionality which will provide end users with a convenient way of optimizing costs. MiCADO contains the following major services: Occopus to deploy virtual machines in the cloud, Docker Swarm, to install and manage containers, and Prometheus, to monitor execution in the cloud.

8.1 Occopus (SZTAKI)

Occopus [5], [6] is an open-source cloud orchestration and management framework for heterogeneous multi-cloud platforms. Occopus provides a language to specify infrastructure descriptions and node definitions based on which Occopus can automatically deploy and maintain the specified virtual infrastructures in the target clouds.

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 includes the startup of the virtual machines with contextualization and optionally health monitoring remotely. Health monitoring includes 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.

Building and maintaining an infrastructure can be performed through different interfaces. Occopus has CLI and REST API. Both provide 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 and vice versa.

During development and maintenance Occopus provides error reporting mechanism and logging to ease the development and maintenance of the infrastructure.

8.1.1 Benchmarking Occopus

Occopus is able to run in simulation mode, which means that Occopus skips the cloud API calls and emulates their successful outcome. This option is useful to simulate Occopus behaviour and performance without spending time and money for instantiating the virtual machines. The test creates two infrastructures that consist of 10 and 100 nodes. The test measures the time of simulated deployment for both infrastructures. After that, the test removes the infrastructure, and archives the logs and results. The test is implemented on Jenkins and utilises the same container version of Occopus as built into MiCADO. Figure 1 and Figure 2 show a quick summary of 30 tests.

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

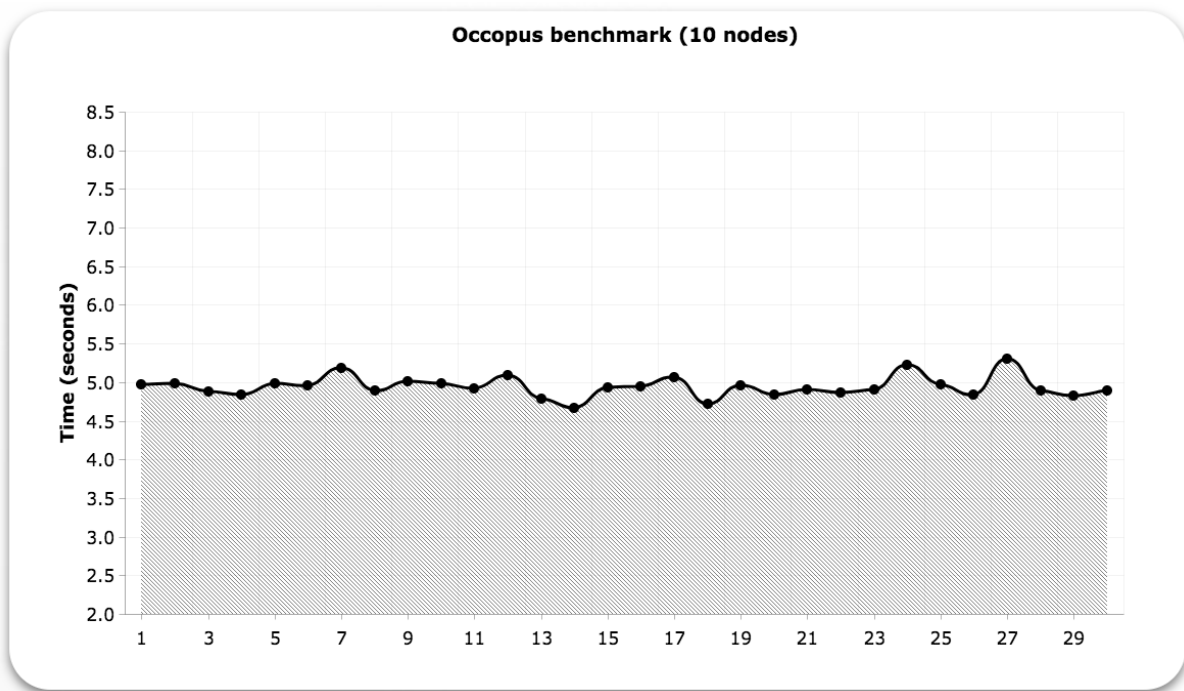


Figure 1 Occopus benchmark test with 10 nodes infrastructure

As we can see in Figure 1, Occopus generates approximately 5 seconds overhead when creating 10 nodes (i.e. 0.5 seconds per node) and the fluctuation is very small. We consider this 0.5 seconds overhead acceptable since the overall time for creating a virtual machine on cloud infrastructure is in the range between 30 and 90 seconds.

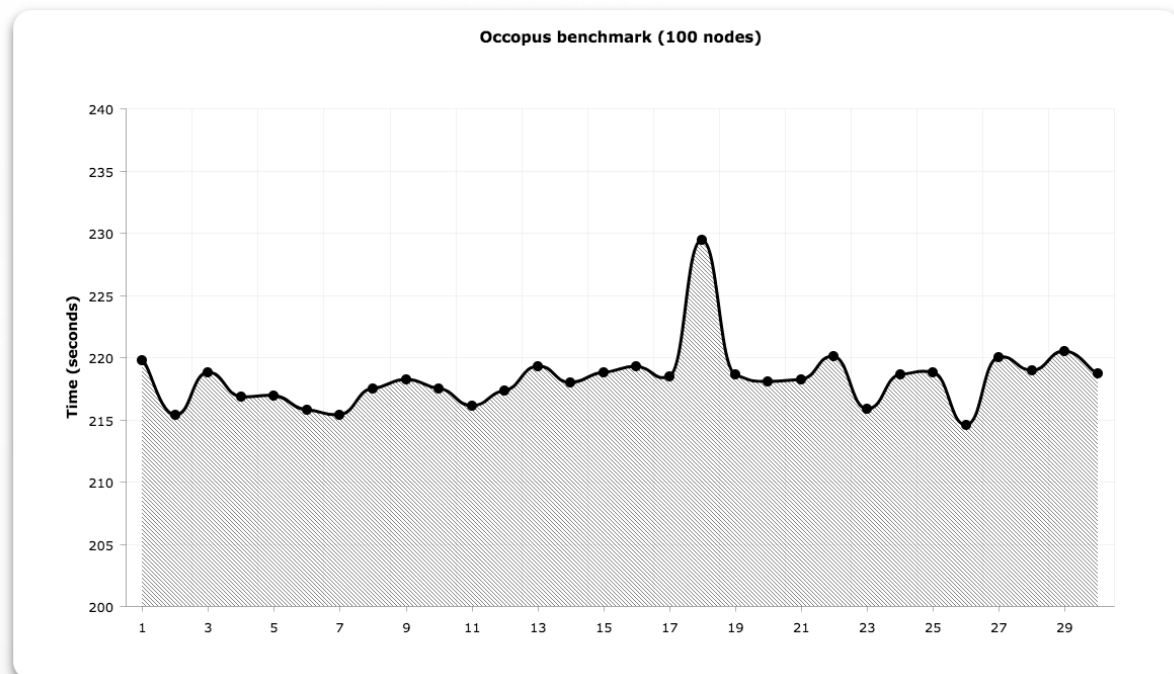


Figure 2 Occopus benchmark test with 100 nodes infrastructure

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

As it can be seen in Figure 2, Occopus generates approximately 220 seconds overhead when creating 100 nodes (i.e. 2.2 seconds per node) and the fluctuation is within 5 percent. We consider this 2.2 second period still acceptable since the overall time for creating a virtual machine on cloud infrastructure is in the range between 30 and 90 seconds.

However, the overall performance of building 100 virtual machines in parallel in a Cloud significantly depends on the capacity of the cloud and its background cloud services performance.

8.1.2 Docker Swarm cluster deployment by Occopus

In this test a Docker Swarm cluster with 3 nodes (1 master and 2 workers) have been created on six different combination of 4 cloud providers. The six scenarios are listed below with their reference in brackets:

- Amazon (AWS)
- CloudSigma (CS)
- SZTAKI OpenNebula (ON)
- CloudBroker-AWS (CB-AWS)
- CloudBroker-CloudSigma (CB-CS)
- CloudBroker-SZTAKI OpenNebula (CB-ON)

First, Occopus creates the master node and installs the Docker CE on it. The readiness of the master is detected by monitoring the port of Swarm API (tcp 2375 port) when it becomes open. Afterwards Occopus creates the worker nodes in parallel. Detecting the readiness of the worker nodes is not a straightforward task since no service comes to life on the worker nodes, thus there is no possibility to detect the existence of a service on the worker nodes.

To detect the end of the configuration stage of the worker nodes with Occopus the cloud-init file has been extended with an Nginx web server installation. Nginx opens port (TCP 80) which can be detected by Occopus. The open port represents the successful finish of the worker node creation. This part of the test could be replaced later with SSH or other lightweight daemon. Our measurements prove that the web server installation takes a few seconds, and a few percentage of the overall worker node installation and configuration, so it does not affect negatively the outcome of the test. After a worker connects to the Swarm cluster, it continues with the installation of a web server. Occopus maintains the infrastructure and is triggered when the http port (80) becomes open. This is a sign for Occopus, which indicated that the worker node deployment is completed. The test measures the required time and saves the logs and results.

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

	SZTAKI Opennebula Test	CloudBroker Platform SZTAKI OpenNebula Test	CloudSigma Test	CloudBroker Platform CloudSigma Test	AWS Test	CloudBroker Platform AWS Test
	6min 8s	6min 8s	7min 29s	9min 26s	2min 35s	15min 50s
#27 Apr 05 07:45 No Changes	10min 8s	5min 37s	5min 30s	5min 24s	2min 7s	8min 8s
#26 Apr 04 17:24 No Changes	7min 38s	5min 54s	6min 45s	8min 8s	2min 38s	8min 23s
#23 Apr 04 12:05 No Changes	4min 7s	4min 53s	5min 24s	3min 7s	2min 22s	7min 39s
#22 Apr 04 12:05 No Changes	3min 37s	4min 53s	5min 19s	12min 8s	2min 37s	8min 8s
#21 Apr 04 11:18 No Changes	5min 22s	4min 52s	11min 55s	5min 38s	2min 22s	7min 38s
#20 Apr 04 10:33 No Changes	4min 8s	5min 22s	5min 49s	11min 24s	2min 37s	7min 38s
#19 Apr 04 08:43 No Changes	5min 22s	4min 53s	5min 14s	13min 8s	2min 37s	9min 53s
#18 Apr 04 08:43 No Changes	6min 8s	6min 38s	18min 55s	9min 23s	2min 22s	8min 8s

Figure 3 Swarm deployment page on Jenkins

The Swarm cluster architecture is similar to what we use in MiCADO. The mechanism is implemented in Jenkins and is made for the container version of Occopus. You can see the Jenkins page on Figure 4. This test is appropriate for validating the Occopus resource handlers and for measuring the provisioning time between different cloud providers. Although, the test depends on the average load of the cloud providers we can still get a general idea of the average speed of different clouds.

Measurements were repeated 10 times in each scenario and each measurement result of each scenario are shown in Figure 3. This way we can get a more comprehensive picture about the measurements.

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

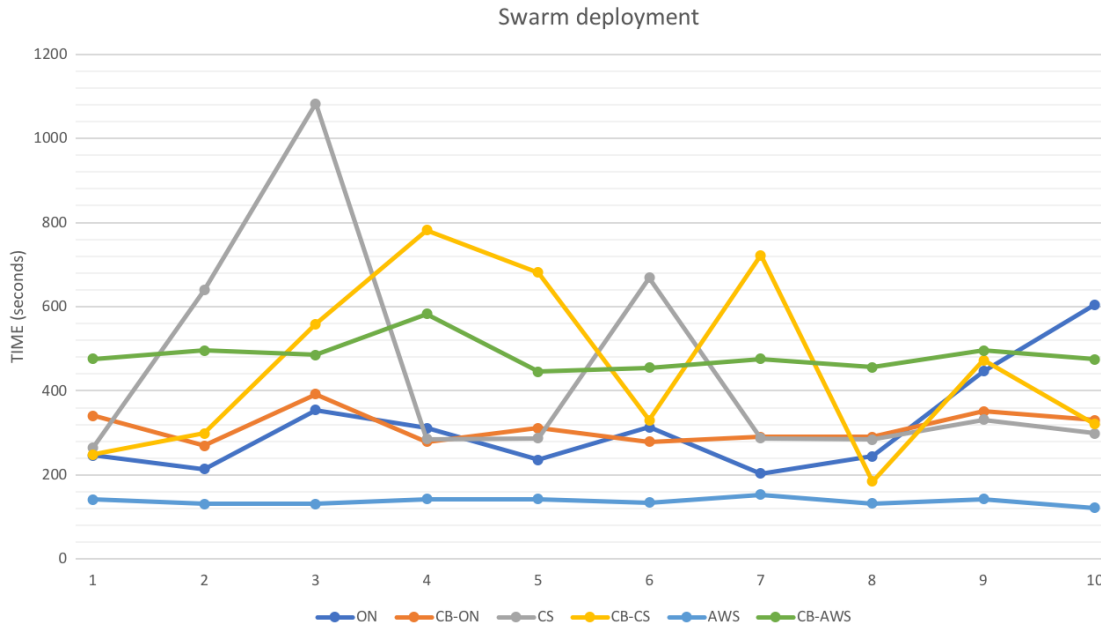


Figure 4 Occopus swarm deployment in different clouds

	ON	CB-ON	CS	CB-CS	AWS	CB-AWS
AVG	317,384 s	313,355 s	443,097 s	459,967 s	137,396 s	484,258 s
MEDIAN	278,96 s	300,78 s	293,3 s	401,605 s	137,995 s	475,64 s
AVG PER NODE	105,795 s	104,452 s	147,699 s	153,3223 s	45,7987 s	38,4816 s
DEVIATION	125,203 s	39,364 s	271,176 s	214,176 s	8,867 s	38,482 s

Table 4 Occopus Swarm deployment average

The average deployment time, median, average time per node and deviation can be found in the Table 4. Median is better off filtering out the measurement error, and the other noises which can distort the result.

As we can see in Table 4, AWS cloud has the lowest deployment time in our measurements and it is about 3 time faster than any other provider. OpenNebula and CloudSigma deployment time are close to each other. CloudBroker generates some extra overhead on top of the target cloud. One way to get the fluctuation, is to calculate deviation. The fluctuation was small on the AWS cloud and CloudBroker's OpenNebula, however the others providers generate peaks on the measurements.

8.1.3 MiCADO infrastructure deployment by Occopus

The test builds up a MiCADO infrastructure on various cloud providers. First, it creates the MiCADO master node, where the required packages are installed, configurations created,

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

and the main components pulled from Docker Hub. Once, the appropriate configurations are set, Occopus creates the worker node. The worker cloud-init file which is located in the official GitHub repository is changed, in order to detect the MiCADO deployment. The change is a command order modification, it pulls the worker components first, then joins the Swarm. The test periodically fetches the number of nodes through the Swarm REST API, and when it reaches a predefined number, the infrastructure creation is considered to be finished successfully. This way we can measure and investigate the time of MiCADO deployment and the operability of Occopus as well. Moreover, the deployment of the MiCADO infrastructure itself is also realized by Occopus. The test has been implemented in Jenkins, which is running on Cloudsigma.

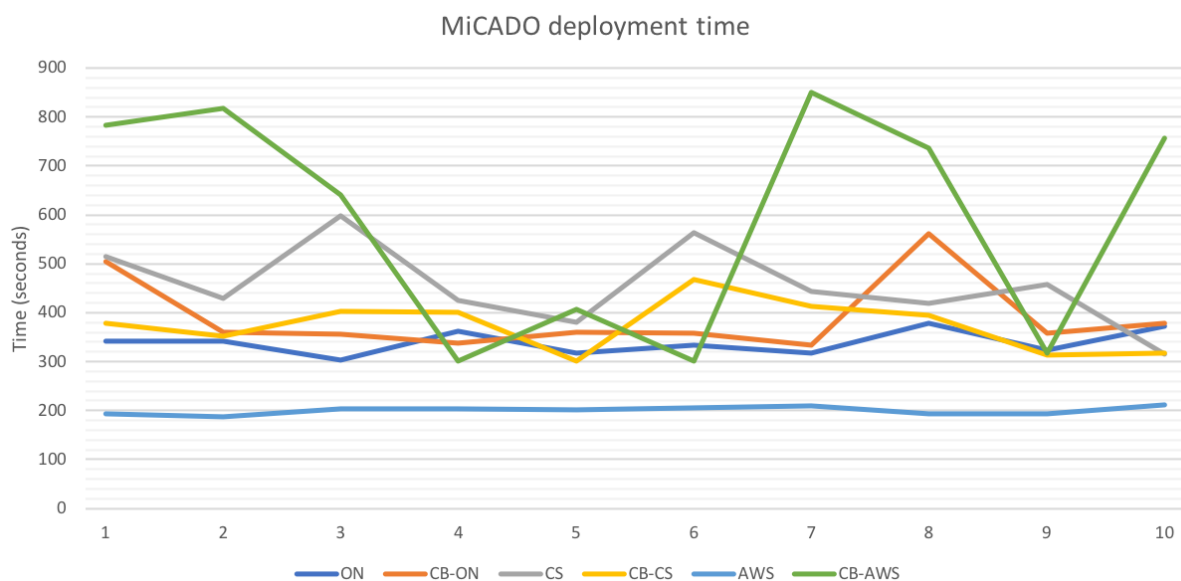


Figure 5 MiCADO deployment time in different clouds

	ON	CB-ON	CS	CB-CS	AWS	CB-AWS
AVG	339 s	391 s	455 s	374 s	200 s	591 s
MEDIAN	338 s	360 s	437 s	387 s	203 s	689 s
AVG PER NODE	169,6 s	195,55 s	227,4 s	187,1 s	100,2 s	295,7 s
DEVIATION	25,12 s	76,98 s	83,87 s	52,41 s	7,73 s	231,63 s

Table 5 MiCADO deployment average

The average deployment time is between 200 and 600 seconds, presented in Table 5, and is similar to the Swarm cluster deployment time. The reason for this is that the installation process is quite similar, although the image is being downloading from Docker hub in parallel on the worker and master nodes.

8.2 Prometheus

Prometheus [7] is an open-source monitoring and alerting tool which collects a variety of metrics from an extensible list of sources. Prometheus in the current MiCADO implementation monitors hardware resources and generates statistics from the virtual machines and containers which make up the deployed application. Prometheus can be extended to collect specific measures from databases, web applications and more. The metrics reported by Prometheus are used in the decision making processes which drive Occopus and Docker Swarm to scale nodes and containers respectively.

8.2.1 Automated stress testing

A complete infrastructure test has been automated inside Jenkins to monitor the various components of MiCADO and the interactions between them. The test relies on a Docker container running *stress-ng* [8], an extension of the *NIX *stress* tool, to deliberately increase the load on the host CPU (to be reflected in the metrics scraped from both virtual machine node, and container) to a set percentage, 85%.

The test then observes the MiCADO response to the increased load by either following the Docker events log, making calls to component APIs, or by attaching to component logs. After MiCADO successfully completes a scale-up to a set target of four nodes and four containers, the CPU load in *stress-ng* is reduced significantly, to 5%. Again, the test observes the MiCADO response during the scale-down phase, ensuring that both virtual machine nodes and containers are scaled back to one each. The test periodically queries for alerts through the Prometheus API, and reports changes in alert state back to the Jenkins console.



Figure 6 Grafana displaying real-time Prometheus metrics (average and individual CPU load on nodes and containers) during automated load testing

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

The success of the test proves that Prometheus is correctly reporting the CPU consumption of both virtual machine nodes and containers, and that the information is made available to the correct components within MiCADO. Based on the console output, the time between a Prometheus alert and a corresponding scale response by Occopus or Docker Swarm can be inferred.

To go further and follow the Prometheus metrics directly, we take advantage of the open-source data graphing tool, Grafana [9], which offers built-in support for the graphical display of Prometheus metrics. Figure 6 shows a Grafana dashboard during the automated stress test as it monitors the increase in node and container CPU load to force a scale-up response, and then as it decreases to force the scale-down response.

8.3 Docker Swarm

Docker [10] is an OS-level virtualisation platform based on Linux containers which allows for the deployment of applications in isolated environments. Swarm [11] extends this functionality by managing the orchestration of containerised applications across multiple hosts which have been clustered into a single virtual instance of Docker. In the current implementation of MiCADO, Docker Swarm is the container orchestrator of choice, used to manage application deployment and application-level scalability.

8.3.1 Automated setup of containers using MiCADO

The automated stress test described in 8.2.1 is also used to test Docker Swarm and its handling of containers within the MiCADO infrastructure. The *stress-ng* container is created and destroyed twice during testing, and Docker receives instructions to scale up or down based on the current CPU load.

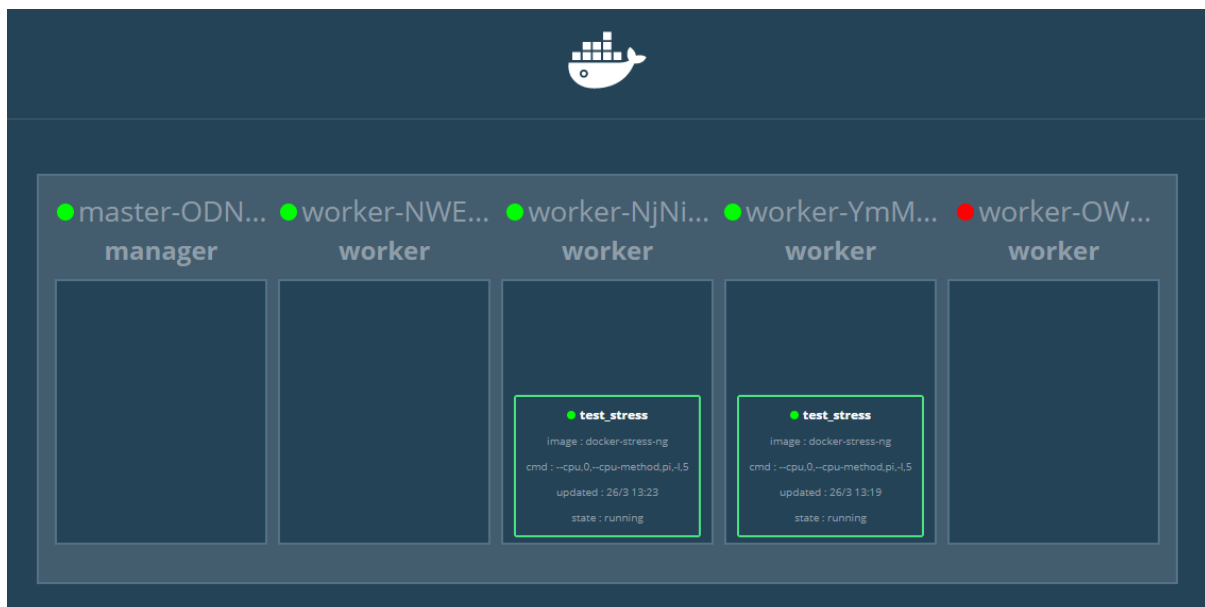


Figure 7 Docker Swarm Visualiser showing *test_stress* containers (smaller green boxes) running inside worker virtual machine nodes (larger boxes) as they are scaled-down.

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

The success of the test shows that Docker Swarm is able to handle setup and teardown of containers, both in response to the automated request from the test script and from the MiCADO components during autoscaling phases. The test attaches to the Docker events stream and follows containers as they are replicated up and down. This information is output to the console log of the test. The testing is visualised in Figure 6, where the *docker-swarm-visualiser* tool [12] displays virtual machine nodes and containers in real-time as they are added or removed from the infrastructure in scale-up or scale-down events.

9 Test plan for the security components

The following subsections outline the test plan for the COLA security enablers to be developed within the COLA project. The test plan does NOT include test plans for existing security enablers that have been developed earlier by the industrial partners in the project. A detailed description of each security architecture component including the Crypto Engine, Credential Manager, Credential Store and the Security Policy Manager can be found in Section 4 of Deliverable 7.2.

Please note that as the implementation of these security components only started in April 2018 (according to the original COLA DoA), results of these tests cannot be presented in this document. The tests will be completed following the implementation of security components and will be reported by WP7.

9.1 Crypto Engine

The Crypto Engine generates cryptographic keys to enable secure interaction between different entities within the MiCADO framework.

9.1.1 Test Items

#	Items to Test	Test Description
1	Key generator	Test whether the component can generate keys according to specifications.
2	Nonce generator	Test whether the component can generate random numbers according to specifications.
3	Encryption library	Test whether the component can encrypt and decrypt messages according to specifications.

9.1.2 Test Features

#	Function of Test	Test Description
1	Generate 256-bit symmetric key	Test whether the function correctly generates a 256-bit symmetric key with sufficient entropy.
2	Generate 2048-bit asymmetric key	Test whether the function correctly generates a 2048-bit public-private keypair with sufficient entropy.
3	Generate random nonce	Test whether the function generates random numbers with sufficient entropy.
4	Encrypt and decrypt data	Test whether the function works properly and correctly encrypts and decrypts sample inputs using a given key, encryption algorithm and encryption mode.
5	Generate X509 certificate	Test whether the function correctly generates a well-formed X509 certificate.

9.1.3 Features not to be tested

Some features are not tested at this phase because they will be delayed for developing later or they belong to another test phase.

#	Feature not to be tested	Test Description
1	Symmetric Searchable encryption	Test whether the component correctly implements symmetric searchable encryption.

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

2	Asymmetric Searchable encryption	Test whether the component correctly implements asymmetric searchable encryption.
3	Probabilistic encryption	Test whether the component correctly implements probabilistic encryption.

9.1.4 Approach

#	Function to Test	Test data Description	Metrics to be collected	Pass/Fail criteria
1	Generate 256-bit symmetric key	Data involves: input command, key type.	Correct/ Incorrect “Correct” means function produces a uniformly distributed 256-bit sequence; Incorrect otherwise	Precision = # of incorrect/ # of test runs Pass if precision = 1 Fail if precision < 1
2	Generate 256-bit asymmetric key	Data involves: input command, key type.	Correct/ Incorrect “Correct” means function produces a uniformly distributed sequence of a given size; Incorrect otherwise	As above
3	Generate random nonce	Data involves: input command, nonce size	Correct/ Incorrect “Correct” means function produces a pseudorandom sequence (for encryption) that equals the input plaintext when decrypted (for decryption); Incorrect otherwise	As above
4	Encrypt and decrypt data	Data involves: input command, input data, encryption/decryption on key, encryption/decryption on cipher and mode	Correct/ Incorrect “Correct” means function produces a valid X509 certificate with correct input data. Incorrect otherwise	As above
5	Generate X509 certificate	Data involves: input command, certificate input data, encryption/decryption on cipher and mode	Correct/ Incorrect “Correct” means function produces a uniformly distributed sequence of a given size; Incorrect otherwise	As above

9.2 Credential Manager

The Credential Manager securely stores the credentials of entities with access to the MiCADO service.

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

9.2.1 Test Items

#	Item to Test	Test Description
1	Security Policy Manager	Test whether the component can communicate with CM, and works properly or not
2	Credential Manager	Test whether the component can communicate with SPM, and works properly or not

9.2.2 Test features

#	Function to Test	Test Description
1	Verify authenticator	Test whether the function works properly and returns correct response
2	Add new identity	Test whether the function works properly and returns correct response
3	Change authenticator	Test whether the function works properly and returns correct response
4	Reset authenticator	Test whether the function works properly and returns correct response
5	Delete identity	Test whether the function works properly and returns correct response
6	Integration of #1 and #2	Test whether the two functions cooperate smoothly to deliver the function of adding a new identity or not
7	Integration of #1 and #3	Test whether the two functions cooperate smoothly to deliver the function of changing authenticator or not
8	Integration of #1 and #4	Test whether the two functions cooperate smoothly to deliver the function of resetting authenticator or not

9.2.3 Features not to be tested

Some features are not tested at this phase because they will be delayed for developing later or belong to another test phase.

#	Features not to be tested	Test Description
1	Lock-out mechanism	Test whether the function works properly and returns correct response
2	Verifying password strength	Test whether the function works properly and returns correct response
3	Reset authenticator by user himself	Test whether the function works properly and returns correct response
4	Collision of random authenticator	Test whether new random generated authenticator matches with any of other generated ones in the past
5	Forcing user to change the default authenticator	Test whether users changed their default authenticator from the first log-in or not
6	Testing for credentials transported over protected channel	Test whether credentials are transported with POST method through HTTPS protocol or not. This test should involve all sensitive requests, such as log in request, TOSCA file submission.
7	Testing for bypassing authentication	Test whether user can bypass authentication by means such as directing to another page which is not under access control, parameter modification, session Id prediction, SQL injection.
8	Test for non-specific announcement for	Test whether user knows if username or password fails or not.

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

	failed login	
9	Test for default credentials	Test whether user is using common default credentials or not. For e.g., common usernames are admin, qa, test, root. Common passwords are blank password, pass123, 123, nopass, password.

9.2.4 Approach

#	Function to Test	Test data description	Metrics to be collected	Pass/Fail criteria
1	Verify authenticator	Data involves not existed identity, existed identity with wrong authenticator, existed identity with matched authenticator	Correct/Incorrect	Precision = # of incorrect/ # of test runs Pass if precision = 1 Fail if precision < 1
2	Add new identity	Data involves not existed identity, existed identity	Correct/Incorrect	As above
3	Change authenticator	Data involves not existed identity, existed identity with wrong authenticator, existed identity with matched authenticator but empty new authenticator, existed identity with matched authenticator and non-empty new authenticator	Correct/Incorrect	As above
4	Reset authenticator	Data involves not existed identity, existed identity	Correct/Incorrect	As above
5	Delete identity	Data involves not existed identity, existed identity	Correct/Incorrect	As above
6	Integration of #1 and #2	Combination data from #1 and #2	Correct/Incorrect	As above
7	Integration of #1 and #3	Combination data from #1 and #3	Correct/Incorrect	As above
8	Integration of #1 and #4	Combination data from #1 and #4	Correct/Incorrect	As above
9	Integration of #1 and #5	Combination data from #1 and #5	Correct/Incorrect	As above

9.3 Credential Store

The Credential Store securely stores the Authentication Credentials used to manage passwords, keys, tokens, and other secrets in the system.

9.3.1 Test Items

#	Items to Test	Test Description
1	Security Policy Manager (SPM)	Test whether the component can communicate with CO and CS, and works properly or not
2	Credential Store (CS)	Test whether the component can communicate with SPM, and works properly or not
3	Container Orchestrator (CO)	Test whether the component can communicate with SPM, and works properly or not

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

9.3.2 Test features

#	Function to Test	Test Description
1	Initialize Credential Store	Test whether the function works properly and returns correct response
2	Write secrets to Credential Store	Test whether the function works properly and returns correct response
3	Read secrets from Credential Store	Test whether the function works properly and returns correct response

9.3.3 Features not to be tested

This is not applicable to the Credential Store.

9.3.4 Approach

#	Function to Test	Test data description	Metrics to be collected	Pass/Fail criteria
1	Initialize Credential Store	Data involves two cases: correct URL of Credential Store, incorrect URL of Credential Store	Correct/ Incorrect ("correct" means that Credential Store is initialized successful if providing URL is correct, and vice versa)	Precision = # of incorrect/ # of test runs Pass if precision = 1 Fail if precision < 1
2	Write secrets to Credential Store	Data involves: empty secret, 1 secret, multiple secrets	Correct/ Incorrect	As above
3	Read secrets from Credential Store	Data involves: not existed secret name, existed secret name, and combination.	Correct/ Incorrect	As above

9.4 Security Policy Manager

The Security Policy Manager is required for the enforcement of user-defined security policies.

9.4.1 Test Items

#	Items to Test	Test Description
1	Security Policy Manager (SPM)	Test whether the component can communicate with CO and CS, and works properly or not
2	Credential Store (CS)	Test whether the component can communicate with SPM, and works properly or not
3	Container Orchestrator (CO)	Test whether the component can communicate with SPM, and works properly or not

9.4.2 Test features

#	Function to Test	Test Description
1	Initialize Credential Store	Test whether the function works properly and returns correct response
2	Write secrets to Credential Store	Test whether the function works properly and returns correct response

D4.2 Requirements Gathering and Performance Benchmarking of Microservices

3	Read secrets from Credential Store	Test whether the function works properly and returns correct response
---	------------------------------------	---

9.4.3 Features not to be tested

Some features are not tested at this phase because they will be delayed for developing later or they belong to another test phase.

#	Features not to be tested	Test Description
1	Lock-out mechanism	Test whether the function works properly and returns correct response
2	Verifying password strength	Test whether the function works properly and returns correct response
3	Reset authenticator by user himself	Test whether the function works properly and returns correct response
4	Collision of random authenticator	Test whether new random generated authenticator matches with any of other generated ones in the past
5	Forcing user to change the default authenticator	Test whether users changed their default authenticator from the first log-in or not
6	Testing for credentials transported over protected channel	Test whether credentials are transported with POST method through HTTPS protocol or not. This test should involve all sensitive requests, such as log in request, TOSCA file submission.
7	Testing for bypassing authentication	Test whether user can bypass authentication by means such as directing to another page which is not under access control, parameter modification, session Id prediction, SQL injection.
8	Test for non-specific announcement for failed login	Test whether user knows if username or password fails or not.
9	Test for default credentials	Test whether user is using common default credentials or not. For e.g., common usernames are admin, qa, test, root. Common passwords are blank password, pass123, 123, nopass, password.

9.4.4 Approach

#	Function to Test	Test data description	Metrics to be collected	Pass/Fail criteria
1	Initialize Credential Store	Data involves two cases: correct URL of Credential Store, incorrect URL of Credential Store	Correct/ Incorrect ("correct" means that Credential Store is initialized successful if providing URL is correct, and vice versa)	Precision = # of incorrect / # of test runs Pass if precision = 1 Fail if precision < 1
2	Write secrets to Credential Store	Data involves: empty secret, 1 secret, multiple secrets	Correct/ Incorrect	As above
3	Read secrets from Credential Store	Data involves: not existed secret name, existed secret name, and combination.	Correct/ Incorrect	As above

10 Conclusion

In this deliverable we have presented our approach to the performance testing of the core MiCADO components in accordance with the performance requirements and specifications of the MiCADO orchestration layer, matched against the high-level performance requirements of the four project use-case applications. We have also provided a plan for testing the key security enablers to be developed. To reiterate, the deliverable will inform WP5 regarding QoS and scaling services, and WP6 to advice on price performance optimisation, while providing the developers of the core MiCADO components a baseline for further testing. As a result, application developers and end-users will be able to set QoS, security, performance and economic requirements, and make modification to the requirements on the fly. Furthermore, an optimized price/performance ratio will make the cloudification of applications more feasible for SMEs.

11 References

- [1] PEP 8, <https://www.python.org/dev/peps/pep-0008/>
- [2] Semantic Versioning 2.0.0, <https://semver.org/>
- [3] A successful Git branching model, <http://nvie.com/posts/a-successful-git-branching-model/>
- [4] AngularJS developers.md, <https://github.com/angular/angular.js/blob/master/DEVELOPERS.md#-git-commit-guidelines>
- [5] Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures, J. Kovács and P. Kacsuk, “Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures,” Journal of Grid Computing, Nov 2017. [Online]. Available: <https://doi.org/10.1007/s10723-017-9421-3>
- [6] Occopus website, <http://occopus.lpd.sztaki.hu/>
- [7] Prometheus website, <https://prometheus.io/docs/introduction/overview/>
- [8] Stress-ng documentation, <http://kernel.ubuntu.com/~cking/stress-ng/>
- [9] Grafana website, <https://grafana.com/grafana>
- [10] Docker website, <https://www.docker.com/what-docker>
- [11] Swarm documentation, <https://docs.docker.com/engine/swarm/>
- [12] docker-swarm-visualizer on Github, <https://github.com/dockersamples/docker-swarm-visualizer>