D4.6 Validation of the Cloud Data Analytics Service Management and Scalability components

Version 1.0

Document Information

Contract Number	780622
Project Website	https://class-project.eu/
Contractual Deadline	M31, Dec 2020
Dissemination Level	PU
Nature	REP
Author(s)	Roi Sucasas (AtoS) roi.sucasas@atos.net Jorge Montero (AtoS) jorge.montero@atos.net Rut Palmero (AtoS) ruth.palmero@atos.net
Contributor(s)	(ATOS)
Reviewer(s)	(MODENA)
Keywords	Cloud, WP4, Rotterdam, SLA, Machine Learning,

Notices: *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No "780622".*

Change Log

Version	Date	Author	Description of Change
V0.1	25/02/20	Rut Palmero	Initial Schema of the document
V0.1	18/03/21	Jorge Montero Roi Sucasas Rut Palmero	Atos first internal version
V0.2	22/03/2021	Luca Chiantore	CLASS Internal review
V0.3	26/03/2021	Jorge Montero Roi Sucasas Rut Palmero	Terms and Abbreviations updated Glossary removed Citations improved Explanation of experiments results improved
V1.0	31/3/2021		*Release to the EC*
V1.1	19/10/2021		*Addressing EC requirements*

Table of contents

Table of Figures

Table of Tables

Terms and Abbreviations

Acronym	Definition
D	Deliverable
WP	Work Package
M	Month
MS	Milestones
T	Task
QoS	Quality of Service
IoT	Internet of Things
SLA	Service Level Agreement
K8s	Kubernetes[1]
MicroK8s[2]	Micro Kubernetes
COMPSs	Component Superscalar framework (from BSC)
OKD	Openshift Kubernetes distribution
UCs	Use Cases
CaaS	Container as a Service
ML	Machine Learning
EDA	Exploratory Data Analysis
DNN	Deep Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
PCA	Principal Component Analysis

[1] Kubernetes is an open-source container-orchestration system for automating application deployment, scaling, and management: https://kubernetes.io/
[2] Kubernetes version for IoT, Edge devices, workstations etc. https://microk8s.io/

Executive Summary

This deliverable (D4.6) corresponds to the release of the "*Validation of the Cloud Data Analytics Service Management and Scalability components*" for the CLASS project. This includes the results of WP4 task, "T4.4 *Validation of Cloud Computing side*", initially planned to be done between months M30-M36, and finally extended until M39.
The scope of the current deliverable comprehends:

- The validation of the Data Analytics Service and Scalability components integrated into the CLASS architecture from different perspectives.
- The description of the tests performed to assess the Real-time guarantees, flexibility, and elasticity of the solution.
- The description and validation of the mechanisms implemented to improve real-time guarantees.

1 Introduction

1.1 About this deliverable

The focus of this document is the validation of the cloud requirements following the "*Technical Requirements of the CLASS Software Architecture"* described in (D2.1) and the "*Cloud Requirement Specification and Definition"* described in (D4.1). For this we will consider the results presented in MS1 and MS2, and described in (D4.2)"*First release of the Cloud Data Analytics Service Management components*", (D4.4) "*First release of the Cloud Data Analytics Service Scalability components*" and (D4.7)*"Final release of the Cloud Data Analytics Service Management and Scalability components*".

The following image shows the relation to other deliverables and work packages.

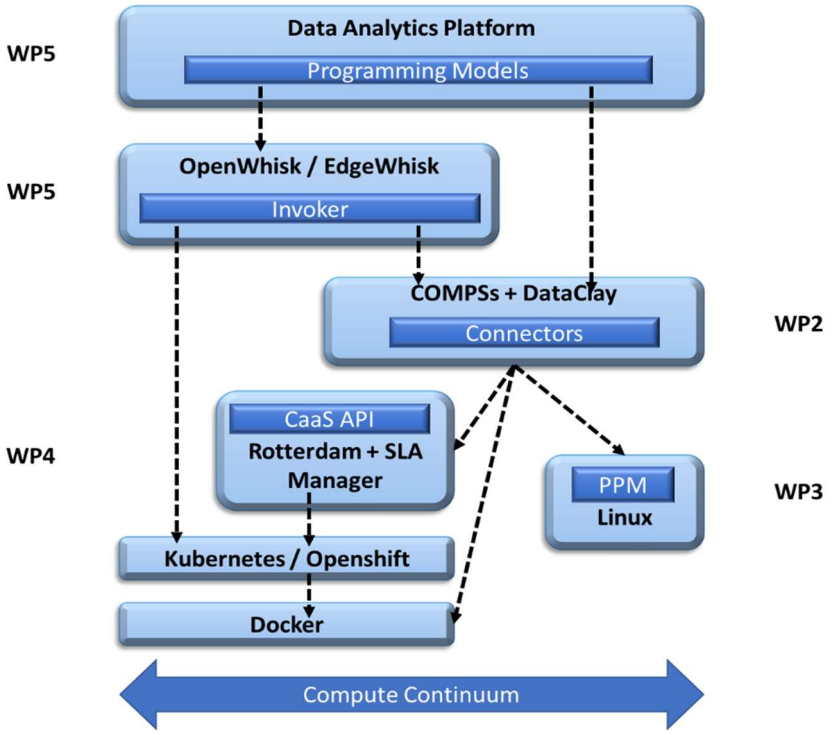

Figure 1 CLASS architecture & WPs

We will also present the experiments performed for the validation of the Cloud Data components and the improvements introduced after these experiments.

1.2 Structure of the document

This document is structured as follows:

- Section 1 contains the introduction, the glossary of terms used in this document, and the description of the document's structure.
- Section 2 describes the validation from the point of view of the initial Requirements.
- Section 3 describes the validation from the point of view of the overall functional requirements and the experiments performed to assess them.

- Section 4 describe the changes in the architecture and new components developed to ensure time guarantee.
- Section 5 presents the result of the validation.
- And finally, section 6 presents the conclusion.

2 Validation of the Cloud Requirement Specification and Definition features

In MS1 after analysing the stakeholders and business goals, the CLASS Cloud Computing Platform system and software requirements were identified and described in (D4.1). The CLASS Cloud Computing Platform comprises two layers: Rotterdam, our CaaS, and the Cloud Infrastructure layer.

Rotterdam	System Requirements	1. Cloud-native deployment 2. Simplified deployment of analytic tasks 3. Microservices style 4. API gateway pattern 5. High throughput / low latency guarantees 6. Analytics performance monitoring 7. Adaptation rules engine 8. Adaptation enactment 9. WS-Agreement-compliant SLA
	Software Requirements	10. API gateway programming language 11. Multi-cloud deployment engine 12. Multi-cloud toolkit 13. Native-cloud toolkit 14. Rules engine 15. SLA engine
Cloud Infrastructure	System Requirements	1. Container orchestration 2. Support to traditional virtualization 3. Public cloud IaaS 4. Private cloud IaaS
	Software Requirements	5. Docker containers 6. Container orchestration solution 7. Kubernetes charts 8. Public cloud infrastructure 9. Private cloud infrastructure 10. Virtual machines on bare metal

Table 1 CLASS Cloud Computing Platform Requirements (extracted from (D4.1))

On next paragraph, extracted from an article in the CLASS project blog (Blog, s.f.), there is a simplified description of the CLASS Cloud Computing Platform. We have linked the different requirements (by means of numbers) with the feature in the description that helps achieving each requirement. Nevertheless, a full description of the Final released version of the Cloud Data Analytics Service Management and Scalability components can be found in (D4.7), where all the features were described in deep.

*"**Rotterdam** is a Container as a Service (CaaS) {1} which facilitates the deployment and lifecycle management and monitoring of multiple containerized applications and cloud data analytics workloads running simultaneously on multiple containerized orchestrators through API calls, abstracting all the cloud infrastructure details away from developers. It includes a lightweight implementation of an SLA system, responsible for enforcing QoS parameters, including real-time. Rotterdam subcomponents provide Data Analytics Service Management and Scalability features:*

- *A **CaaS API gateway** {10} for providing REST APIs {4} {3}to developers for deploying data analytics tasks easily*
- *A **Deployment Engine** {2} for optimal placement of data analytics services on cloud resources*
- *An **SLA Manager** {15} {9} to achieve soft real time guarantees by constantly monitoring and enforcing QoS parameters (such as throughput and latency) {5}.*
- *An **Adaptation Engine** {8} for self-managed and elastic scalability actions based on SLA Manager's inputs."*

Rotterdam was enhanced with new functionalities in MS3, presented in (D4.7), that helped achieve some of the last requirements, like:

- *"Transparent lifecycle management of data analytic workloads in multiple Cloud and Edge clusters {12} {13}*
- *Creation and management of connections to multiple Cloud and Edge containers orchestrators {11}*
- *Real-time QoS guarantees, SLA management and data analytics service scalability {7}*
- *Performance monitoring of data analytics workloads and infrastructures {6}"*

The deployment and management of applications and **serverless** functions in Edge devices was introduced In MS3. That was not listed in the initial requirements, but it was implemented to enhance overall flexibility and elasticity.

***"Cloud infrastructure layer** is required by Rotterdam to deploy and operate containerized {1} analytics tasks. This layer can go from native cloud (i.e., based on Docker containers) to IaaS providing either a private or public cloud {3, 4} {8, 9}, including "traditional" virtualization solutions {2} in data centers. Current implementation relies on Docker for container technology {5, 6}, OpenShift flavor of Kubernetes for Container Management {7} and VMware as Hypervisor {10}. "*

The Cloud infrastructure layer will be fully validated on the use case evaluation.

We can then conclude that all the initial requirements had been met, and some enhancements had been implemented as well, to help achieve all technical requirements of the CLASS Cloud Computing Platform and software architecture.

3 Validation of the Technical Requirements of the CLASS Software Architecture

As stated in (D2.1), after analysing the Business Requirements of the project, the following Technical Goals where identified:

- Increase Software Productivity, in terms of Programmability, Portability and Performance.
- Provide Real-time Guarantees.
- Enable Flexibility and Elasticity.

The Cloud Computing Platform helps achieving the requirements of **software productivity**:

- It provides a way to efficiently manage the underlying computing resources, and to hide the complexity of the compute continuum to the programmer, thus contributing to **programmability**.
- It oversees dealing with the internals of each specific technology (including edge and cloud), maximizing the performance capabilities, thus contributing to **portability**.
- It is responsible of distributing and orchestrating the data analytics execution across the compute continuum, making it possible to exploit the capabilities of the architectures where the final functionalities will ride on, thus contributing to **performance**.

To assess up to which point the Cloud Computing Platform contributes to **Provide Real-time Guarantees** and **Enable Flexibility and Elasticity** to the CLASS Software Architecture, we have conducted some tests. We have some parameters to measure:

- Is the missed deadline[3] a significant QoS metric?
- After a missed deadline, what if a workflow takes less than the time required to bring up more workers? Is the scale in/out (horizontal elasticity) always the best choice?
- How is elasticity affecting the execution time (Real-time guarantees)?

In the next section we describe our tests and present our conclusions.

3.1 Contribution to Real-time guarantees, flexibility, and elasticity

To validate the improvements brought by the use of the Cloud Data Analytics Service Management and Scalability components of the CLASS platform, a set of tests have been done in the two clusters deployed in Modena Data Center. These two cluster infrastructures have the following configuration:

[3] A missed deadline indicates that a computation task is taking more time to complete than expected by QoS. A missed deadline starts a scale in/out in the number of tasks

- **Openshift** cluster (Figure 2): Cluster composed by 4 VMs with the following configuration:
  - 1 master node: CentOS 7.5, 4 vCPUs, 16GB RAM, 80GB disk
  - 3 worker nodes: CentOS 7.5, 1 vCPU, 8GB RAM, 80GB disk

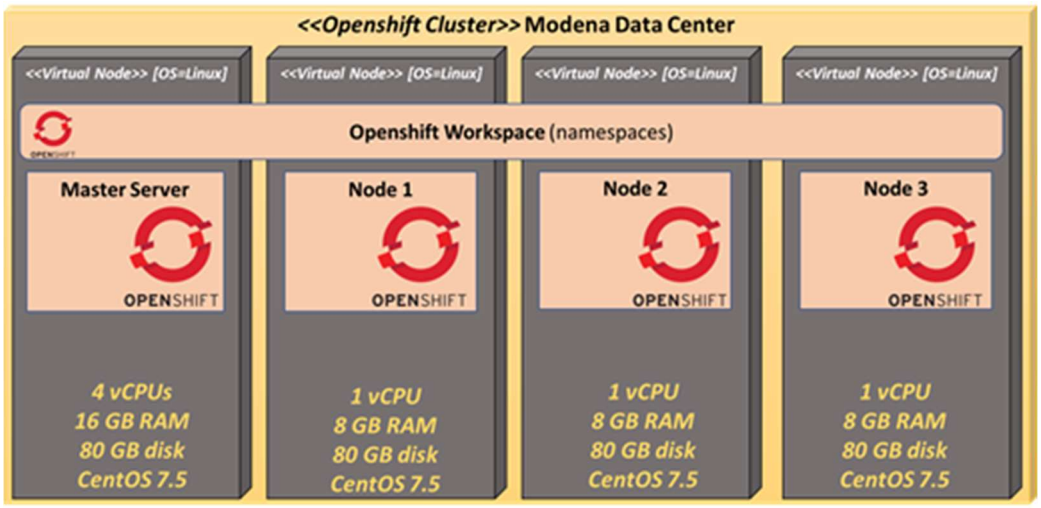

Figure 2 Openshift cluster in Modena Data Center

- **Kubernetes** cluster (Figure 3): Cluster composed by 5 VMs with the following configuration:
  - 1 master node: Ubuntu 18.04, 4 vCPUs, 8GB RAM, 60GB disk
  - 3 worker nodes: Ubuntu 18.04, 4 vCPUs, 8GB RAM, 60GB disk
  - 1 additional worker node: Ubuntu 18.04, 16 vCPUs, 8GB RAM, 60GB disk

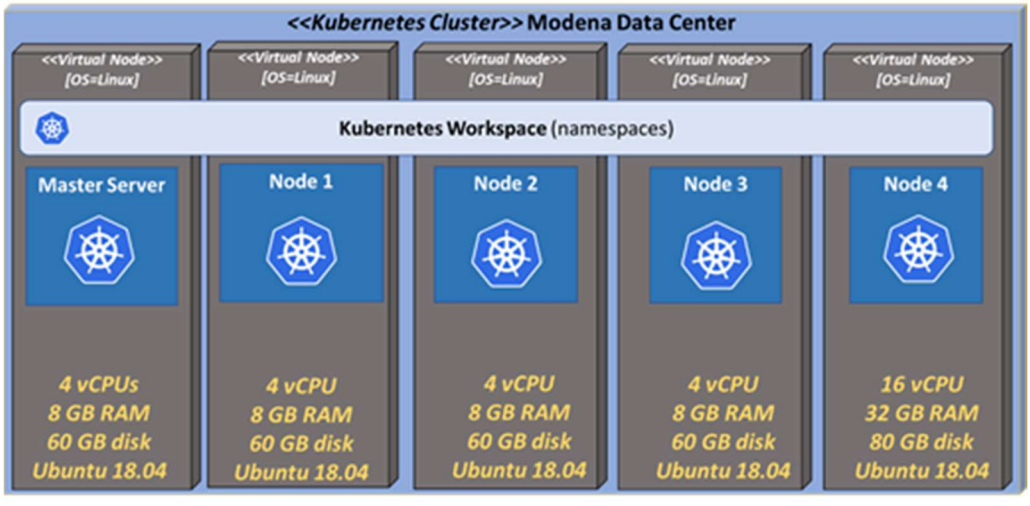

Figure 3 Kubernetes cluster in Modena Data Center

The application tested in this environment is a COMPSs workflow application[4], responsible for executing a java matrix multiplication (simulating a big-data analytics workload) in a distributed environment. This workflow uses one or more workers (i.e.,

[4] https://hub.docker.com/r/bscppc/class-pycompss-rotterdam

slaves, instances, pods) to distribute its computation tasks and execute them in parallel. The number of workers is defined at launch time, but it can be scaled in or out during execution time depending on the assessment done by the SLA Manager component.

To run these tests in both clusters we prepared two different scenarios:

- **COMPSs workflow without auto-scaling** (Figure 4): In this scenario the COMPSs master application connects to the REST API provided by Rotterdam to launch a workflow in the Cloud environment (Openshift / Kubernetes cluster). First, the master defines the number of workers used to execute the parallel tasks (**1**), and then Rotterdam creates these workers (pods) in the cluster at deployment / launch time (**2**). Finally, the COMPSs master application connects to these workers to execute the computation tasks in parallel (**3**). Also, in this scenario the Rotterdam's adaptation engine will be disabled; thus, it will not scale out or in the number of workers during run time.

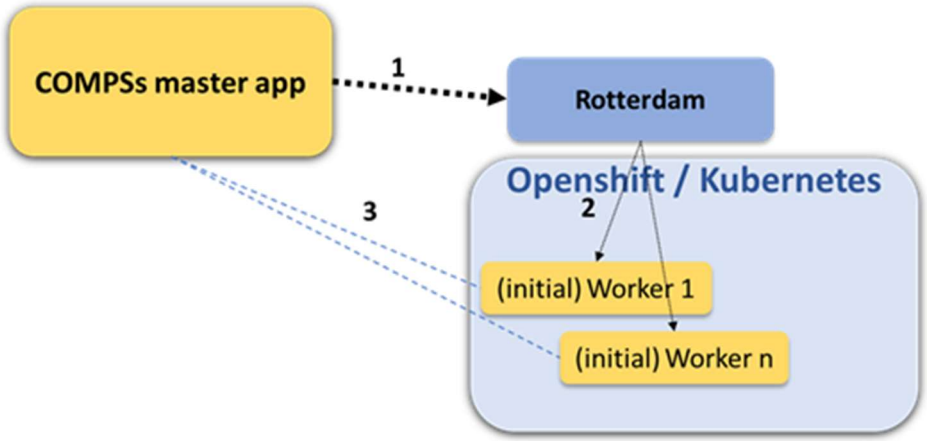

Figure 4 Scenario I

- **COMPSs workflow with auto-scaling** (Figure 5): In the second scenario, we will use the same COMPSs master application and Rotterdam with its adaptation engine enabled. An SLA will be created for each workflow at deployment / launch time. These SLAs will be used to check the missed deadlines of the workflow computation tasks (**5**). These missed deadline times are metrics generated by the COMPSs master application and they are stored in Prometheus (**4**). These values are related to the expected total execution time of the workflow. If there is a missed deadline (a computation task takes more time to complete than expected), it means that the workflow will probably last more than expected. In this case the SLA will generate a violation to notify Rotterdam that the workflow will last more than desired. After receiving this violation, Rotterdam will take the required actions: in this case, it will scale out the number of workers (**6**). Finally, the COMPSs master application will redistribute the computation tasks in the new worker nodes (**7**) before continuing the execution of the workflow.

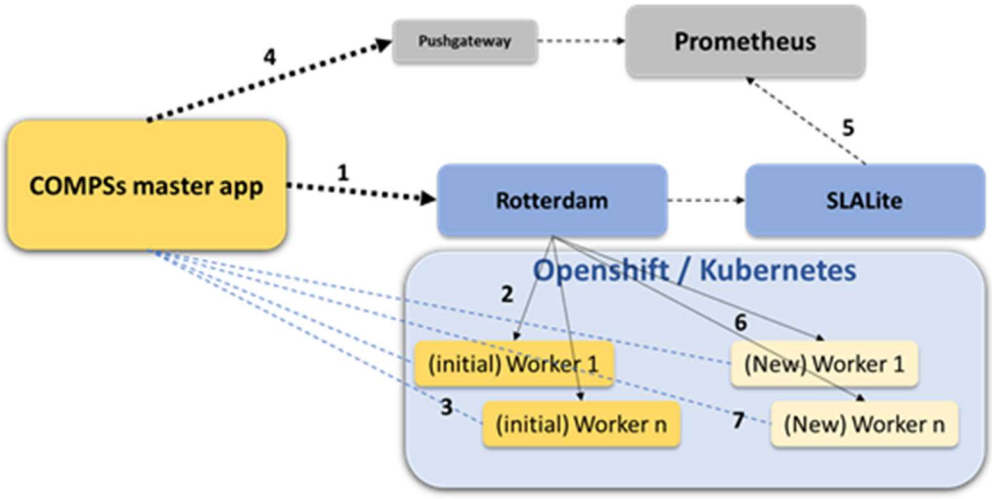

Figure 5 Scenario II

3.2 COMPSs workflow without auto-scaling

The following table shows the results of running the same COMPSs workflow in a "static" environment (fixed number of workers for each execution and same conditions for each cluster), without the use of the features provided by Rotterdam. First, we tried with only one worker per workflow, then with three and so on. The results (**execution time in ms**) of these executions in OpenShift are in red, and the executions in the Kubernetes cluster are marked in black:

Table 2 Workflows execution times (in ms)

#	1 worker	3 workers	6 workers	9 workers	12 workers
1	621786	617820	522618	407144	399776
2	600364	571378	454466	415762	374831
3	594230	566439	464412	416799	378867
4	615012	564199	464266	424616	384708
5	604306	578790	455849	413798	395198
6	622013	567978	466922	416003	395060
7	617566	565619	468000	411904	401901
8	588759	565052	463951	405911	376899
9	598650	572151	455543	410987	380001
10	615332	582628	472430	410940	372098
11	599151	636311	459940	411981	392121
12	611090	593063	457041	405350	399140
13	609145	595968	464953	410500	382955
14	603099	599054	455312	404989	381880

15	631566	590142	471328	429132	398812
	608804,6	*584439,5*	*466468,7*	*413054,4*	*387616,5*

In principle we didn't notice any significant difference in the results obtained by executing the workflow in both the Kubernetes cluster and the OpenShift cluster.

We can also see that if we assign only one worker to the workflow to execute all the internal computation tasks, the average duration of the whole workflow is around 608 seconds. This duration decreases a bit, to 584 seconds, if we assign 3 worker nodes to execute these tasks in parallel. If we continue assigning more workers, then we can observe a great decrease in the execution time. 466 seconds with 6 workers, and 388 with 12 workers. Of course, there is a limit in the number of workers we can assign to the workflow, before it stops decreasing the total duration. In principle, it seems that with 12-15 workers for this workflow we found this limit.

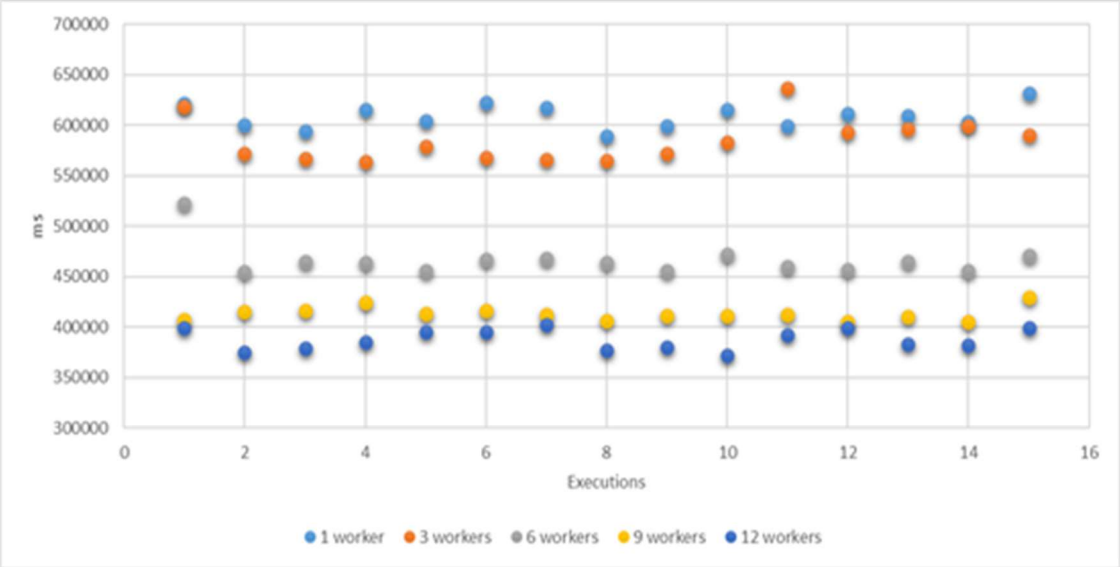

Figure 6 Workflow execution times of scenario I

3.3 COMPSs workflow with auto-scaling

The following tables show the results of running the COMPSs workflow using the scalability features provided by Rotterdam. First, we present the results of executing the workflow with only one initial worker before starting to apply the autoscaling features.

Table 3 Workflows execution times using scalability features with1 initial worker (in ms)

#	1 -> 3 workers	1 -> 6 workers	1 -> 9 workers	1 -> 12 workers
1	688950	601899	551234	531098
2	691934	600123	567912	562983
3	650124	595904	601832	544982
4	668102	599400	575894	545722

5	710723	601543	569001	587153
6	659100	602473	569012	555913
7	667921	660888	578147	532450
8	651098	599919	599157	530001
9	688090	603751	580800	549550
10	681012	615821	561613	578776
	675705,4	608172,1	575460,2	551862,8

As we can see in the previous table, if we start the workflow execution with only one initial worker, and then we scale out to three workers during the execution time, the final execution time is higher (about 70 seconds more) than executing the same workflow (only one worker) without scaling it out. This happens because of the following reasons:

1. The missed deadline metric can appear several minutes after the workflow execution starts.
2. The SLA Manager takes some time to detect that the COMPSs application needs more workers. As the SLA Manager evaluates the associated metrics every 20-30 seconds, it can take some time before detecting these violations.
3. Rotterdam also needs some time to create the new workers and services required by the COMPSs workflow.
4. The COMPSs master application also takes some time to detect that there are new free workers available. Once it detects these new workers, it also needs some time to resynchronize and redistribute the computation tasks in the new workers.

We estimate that all of these "scalability" tasks can take between 60 and 90 seconds. And they can be run at the beginning or in the middle of the workflow execution.

Only after scaling out from one worker to nine or more, we start to see an improvement in the execution time.

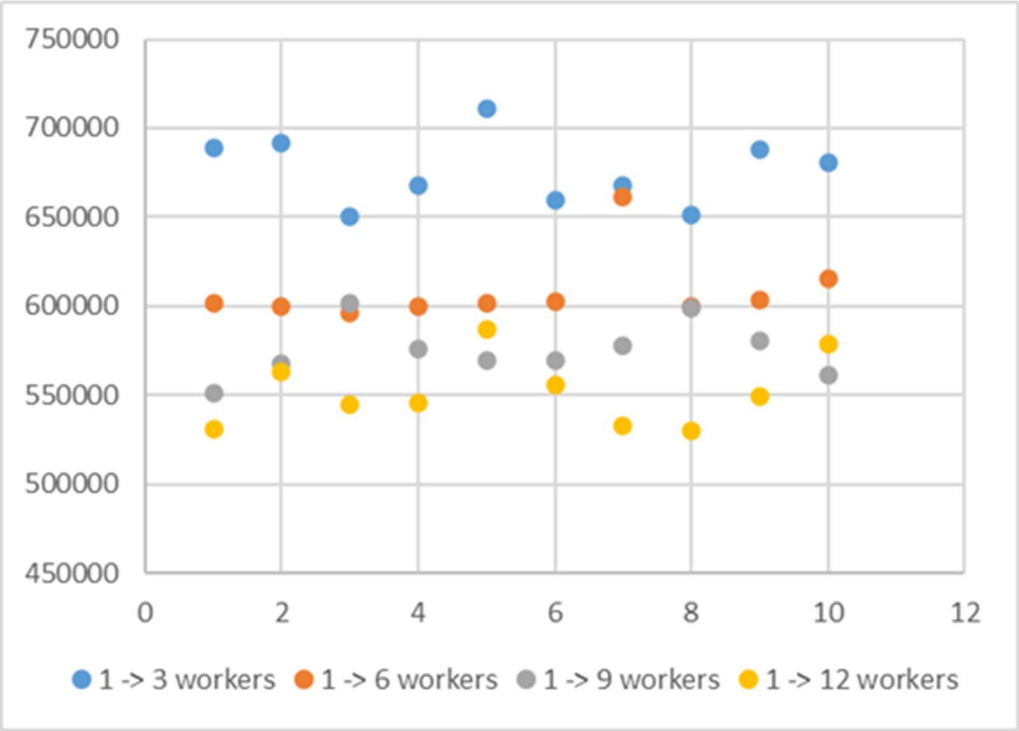

Figure 7 Workflow execution times of scenario II – 1 initial worker

The following table presents the results of executing the workflow with three initial workers:

Table 4 Workflows execution times using scalability features with 3 initial workers (in ms)

#	3 -> 6 workers	3 -> 9 workers	3 -> 12 workers
1	595918	540754	519871
2	601912	541675	498912
3	600112	553989	521812
4	590001	555585	559093
5	595567	571359	548762
6	598591	572998	530925
7	600159	569673	531973
8	585051	573845	538479
9	579141	544647	578245
10	592625	555359	560823
	593907,7	557988,4	538889,5

Scaling up the workflow from three initial workers to six during the execution time brings no difference. In fact, the results show that the execution time is also a bit higher than without scaling up the application. This is the same situation we presented

in the previous tests. But when we scale to nine or 12 workers, we can see a small improvement in the final execution time.

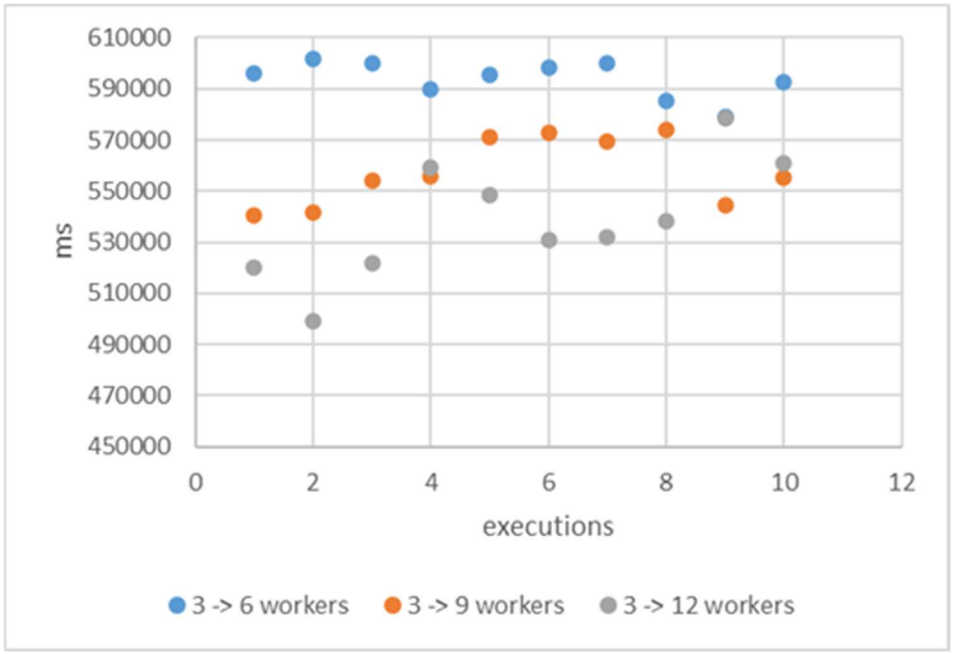

Figure 8: Workflow execution times of scenario II – 3 initial workers

Long-time processes or applications can benefit from the Cloud Data Analytics Scalability. In the case of medium-time applications, like the one used in these tests, we get an improvement in the final execution time, but this improvement is not as optimal as desired.

4 Architecture with time guarantee

As a result of our tests, we have identified that the overhead of the scalation process can affect elasticity, especially in short-time applications. Adding more workers doesn't always lead to better execution times and missed deadline might not be the optimal metric to drive QoS. As we focus on the development of cloud services towards real-time response, we should address this issue.

What if we could foresee a situation that will lead to a longer execution time after a horizontal scaling out? What if we could predict the optimal number of workers for a desired execution time? In this section we will describe how we have improved our SLA Manager with an intelligent module, called **SLA Predictor** that helps us anticipate to missed deadlines by obtaining the optimal number of workers for a desired execution time, on a certain state of the system.

4.1 Rotterdam and SLA Predictor integration

To implement a predictive SLA management for COMPSs workflows, we have added a new service in the SLA Manager. This new service is responsible for calling the Machine Learning subsystem, called **SLA Predictor** (depicted in Figure 8), to get the

recommended number of workers for a COMPSs workflow based on a desired total execution time.

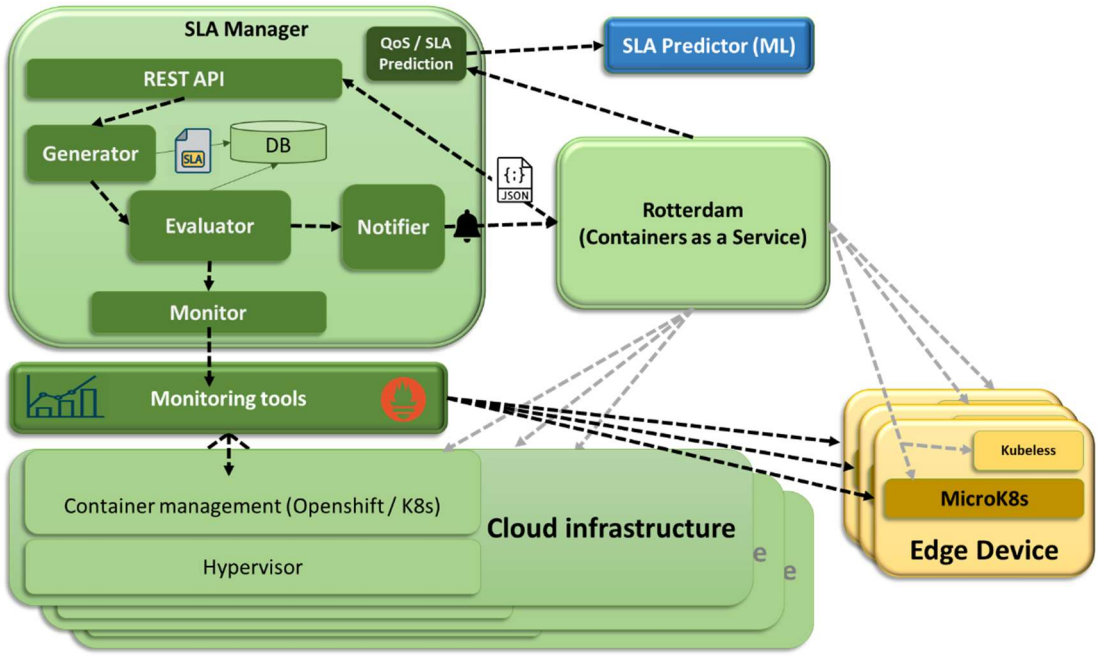

Figure 8 Rotterdam and SLA architecture

Instead of using the missed deadline metrics as described in 3.1 we use a new metric called "*execution_time*", which express the desired total execution time of the workflow. This metric is defined in the JSON task definition[5] (in the QoS properties field) used to deploy applications with Rotterdam. The following diagram (Figure 9) shows the new sequence for COMPSs workflows deployment using the new ML capabilities provided by the SLA Predictor component:

[5] The format of JSON files used to deploy applications / tasks in the Cloud Data Analytics Service Management and Scalability framework was described in (D4.4) and (D4.7)

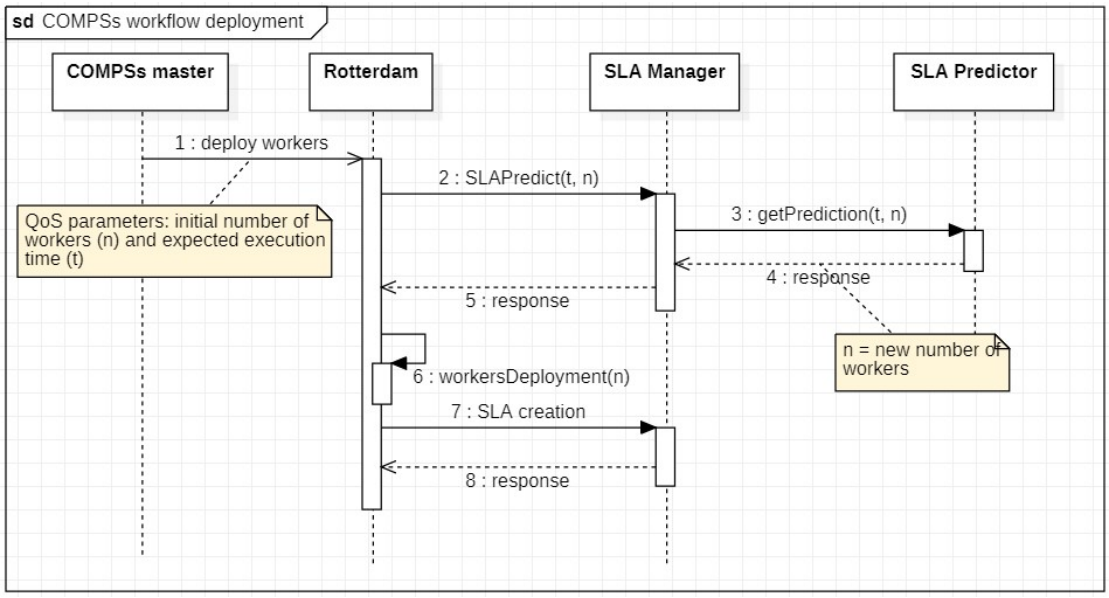

Figure 9 New sequence diagram of COMPSs workflows

1. The COMPSs master application defines (JSON) a new workflow, specifying the number of initial workers and a QoS for a desired total execution time.
2. Rotterdam calls the SLA Manager to check that the QoS can be fulfilled with the given number of workers and desired execution time.
3. The SLA Manager calls the ML subsystem to get the number of workers needed to complete the workflow in the time specified in the QoS.
4. The ML subsystem (SLA Predictor) gets the information based on the status of the cluster, and the results of previous executions. It checks if the execution time will be fulfilled with the given number of workers.
5. This information is sent back to Rotterdam.
6. Rotterdam deploys the workers based on the information obtained by the SLA Manager. If more workers are needed to fulfil the defined execution time, then Rotterdam deploys more workers according to the result generated by the SLA Predictor.
7. Rotterdam creates the SLA for this execution time.

The new version of Rotterdam and the SLA Manager that integrates with the SLA Predictor is available at https://github.com/class-euproject/Rotterdam

In section 4.3 you can see a description of the improvement in the execution time obtained with this new architecture, but let's first describe the SLA Predictor component in deep.

4.2 SLA Predictor

This section describes the steps followed to implement a Machine Learning (ML) model to estimate a predicted execution time for an application based on the status of the environment by analyzing the metrics that Prometheus provides, and identifying the best set of them. It is split in different parts, from a data exploratory

analysis (EDA), through different approaches to build the model, to end exposing the model to be used by the SLA Manager.

The different Python Notebooks describing the EDA phase and the different building model approaches, along with the microservice deployment, are available at https://github.com/class-euproject/SLA-predictor.

4.2.1 Exploratory Data Analysis

The first step is to analyse what we have available in our premises to predict those execution times.

At this phase, we look into different metrics that Prometheus exposes over different integrated collectors:

- **Go collector[6]:** collects the information from Go's runtime such as details about go routines, system threads or garbage collector.
- **Node_exporter collector[7]:** collects different metrics related to the system hardware and kernel.
- **Process collector[8]**: collects basic information from the system proc file (CPU, memory, file limits, and more)

All those metrics have been extracted from the Prometheus API, with the help of a Python client[9], and transformed to a table to be later filtered and get those we want to analyse. In our case, we have filtered them to get only the metrics with a gauge type, where the values range over the time and fits in a timeseries object. Counter, summary, and histogram metric types have been discarded as do not comply our objective of having timeseries data.

The total amount of metrics from Prometheus goes up to 259 metrics, but after the gauge filter, our input metrics went down to 157.

With this set of metrics filtered, we can start our exploratory data analysis to extract relevant information of each metric, as details about the data distribution, stationarity, or trends. The following functions have been used:

- **get_timeseries_from_metric**(metric_name,start_time,end_time,chunk_size):
  - this function executes a query in Prometheus to retrieve a timeseries of each metric for a given period of time.
- **draw_hist_norm**(data,metric_name):
  - plot the timeseries data and compare it to a normal distribution.
- **get_skew**(data):
  - to get if the distribution is symmetrical or skewed.
- **get_kurt**(data):

[6] https://github.com/prometheus/client_golang
[7] https://github.com/prometheus/node_exporter
[8] https://github.com/deadtrickster/prometheus_process_collector
[9] https://github.com/AICoE/prometheus-api-client-python

- to know if the values are spread out, near the mean or the extremes, or close to a normal distribution.

- **get_adfuller**(data):
  - perform a stationarity test to a metric data
- **get_hurst**(data):
  - to get if the metric data contains trends, random walks, or mean reversion characteristics.

An example of those functions for one metric is described in the following images:

```
                            go_memstats_alloc_bytes
timestamp
2021-03-10 14:25:02.020999936                2436064.0
2021-03-10 14:25:32.020999936                2794576.0
2021-03-10 14:26:02.020999936                4090680.0
2021-03-10 14:26:32.020999936                3284552.0
2021-03-10 14:27:02.020999936                3387208.0
...                                                ...
2021-03-10 15:23:39.080000000                8899120.0
2021-03-10 15:23:54.080000000                5018160.0
2021-03-10 15:24:09.080000000                5577432.0
2021-03-10 15:24:24.080000000                6051408.0
2021-03-10 15:24:39.080000000                6609368.0

[840 rows x 1 columns]


       go_memstats_alloc_bytes
count             8.400000e+02
mean              4.359654e+06
std               1.953795e+06
min               1.747592e+06
25%               2.782858e+06
50%               3.523200e+06
75%               5.576548e+06
max               9.285232e+06
```

Figure 10. Timeseries data and summary

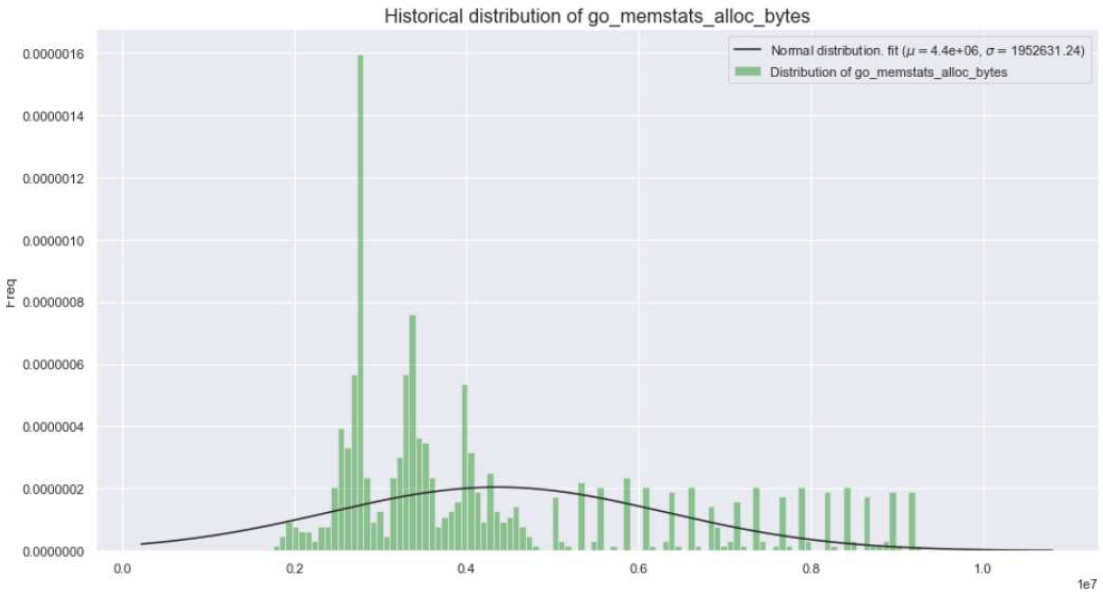

Figure 11. Timeseries distribution and comparation to a normal distribution

```
Skewness: 1.055132
  highly skewed
Kurtosis: -0.101013
  mesokurtic distribution. Values are moderately spread out.


T-test: -7.046238324334879
P-value: 5.677255131792454e-10
Critical values: {'1%': -3.4381774989729816, '5%': -2.8649951426291, '10%': -2.568609799556849}
  Null hypothesis discarded -> stationarity with 5%


Hurst exponent = 0.2360
  anti-persistent process (mean revert)
```

Figure 12. Skewness, kurtosis, stationarity test and hurst exponent

The last part of this EDA is focused on getting the correlation matrix for our metrics extracted. To do that, we first merge all the metrics data into one table and aggregate them in 1-minute frequency to have a common time distribution for all of them.

Figure 13 shows the correlation matrix for all the gauge metrics:

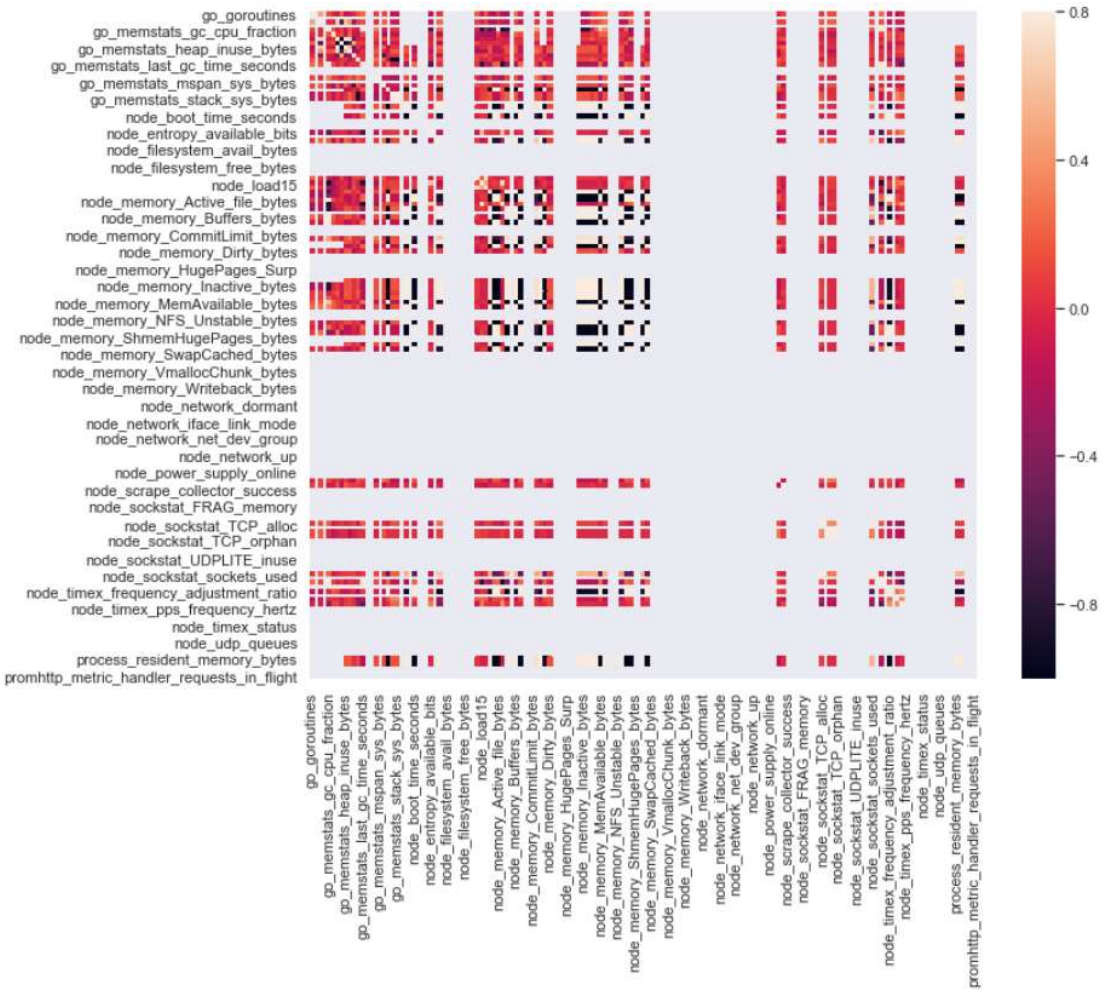

Figure 13. Correlation matrix with all the metrics

This correlation matrix contains a lot of unnecessary information (grey parts) due to different reasons, such as the metric data does not contain information, or all the values are the same. After filtering out the irrelevant metrics, in Figure 14 we get the following correlation matrix:

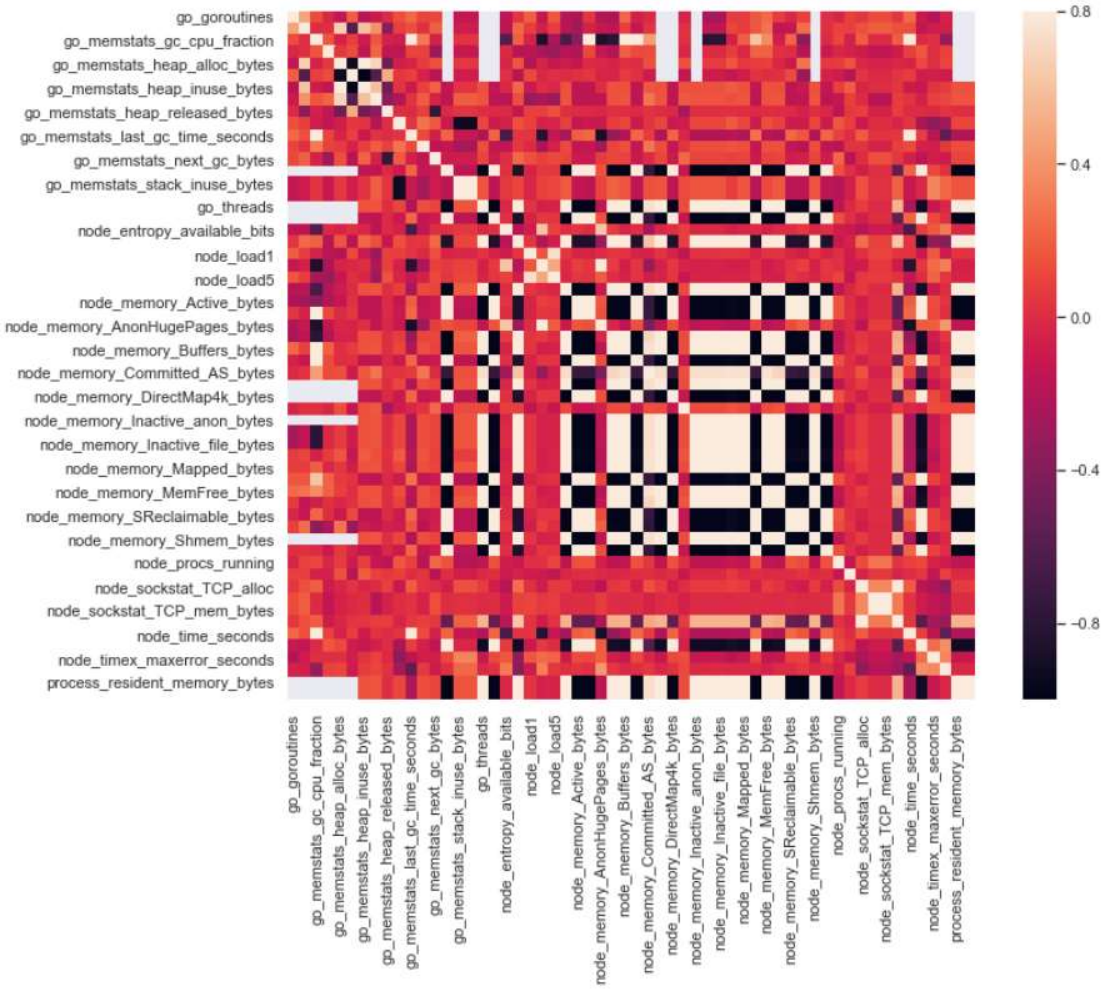

Figure 14. Correlation matrix with selected metrics

From this EDA, we have extracted and listed in Table 5 the following 57 metrics that contains useful information for our objective, and will be used to build our ML model:

	name	**type**	**Description**
1	go_goroutines	gauge	Number of goroutines that currently exist.
2	go_memstats_alloc_bytes	gauge	Number of bytes allocated and still in use.
3	go_memstats_gc_cpu_fraction	gauge	The fraction of this program's available CPU time used by the GC since the program started.
4	go_memstats_gc_sys_bytes	gauge	Number of bytes used for garbage collection system metadata.

5	go_memstats_heap_alloc_bytes	gauge	Number of heap bytes allocated and still in use.
6	go_memstats_heap_idle_bytes	gauge	Number of heap bytes waiting to be used.
7	go_memstats_heap_inuse_bytes	gauge	Number of heap bytes that are in use.
8	go_memstats_heap_objects	gauge	Number of allocated objects.
9	go_memstats_heap_released_bytes	gauge	Number of heap bytes released to OS.
10	go_memstats_heap_sys_bytes	gauge	Number of heap bytes obtained from system.
11	go_memstats_last_gc_time_seconds	gauge	Number of seconds since 1970 of last garbage collection.
12	go_memstats_mspan_inuse_bytes	gauge	Number of bytes in use by mspan structures.
13	go_memstats_next_gc_bytes	gauge	Number of heap bytes when next garbage collection will take place.
14	go_memstats_other_sys_bytes	gauge	Number of bytes used for other system allocations.
15	go_memstats_stack_inuse_bytes	gauge	Number of bytes in use by the stack allocator.
16	go_memstats_stack_sys_bytes	gauge	Number of bytes obtained from system for stack allocator.
17	go_threads	gauge	Number of OS threads created.
18	node_boot_time_seconds	gauge	Node boot time, in unixtime.
19	node_entropy_available_bits	gauge	Bits of available entropy.
20	node_filefd_allocated	gauge	File descriptor statistics: allocated.
21	node_load1	gauge	1m load average.
22	node_load15	gauge	15m load average.
23	node_load5	gauge	5m load average.
24	node_memory_Active_anon_bytes	gauge	Memory information field Active_anon_bytes.
25	node_memory_Active_bytes	gauge	Memory information field Active_bytes.

26	node_memory_Active_file_bytes	gauge	Memory information field Active_file_bytes.
27	node_memory_AnonHugePages_bytes	gauge	Memory information field AnonHugePages_bytes.
28	node_memory_AnonPages_bytes	gauge	Memory information field AnonPages_bytes.
29	node_memory_Buffers_bytes	gauge	Memory information field Buffers_bytes.
30	node_memory_Cached_bytes	gauge	Memory information field Cached_bytes.
31	node_memory_Committed_AS_bytes	gauge	Memory information field Committed_AS_bytes.
32	node_memory_DirectMap2M_bytes	gauge	Memory information field DirectMap2M_bytes.
33	node_memory_DirectMap4k_bytes	gauge	Memory information field DirectMap4k_bytes.
34	node_memory_Dirty_bytes	gauge	Memory information field Dirty_bytes.
35	node_memory_Inactive_anon_bytes	gauge	Memory information field Inactive_anon_bytes.
36	node_memory_Inactive_bytes	gauge	Memory information field Inactive_bytes.
37	node_memory_Inactive_file_bytes	gauge	Memory information field Inactive_file_bytes.
38	node_memory_KernelStack_bytes	gauge	Memory information field KernelStack_bytes.
39	node_memory_Mapped_bytes	gauge	Memory information field Mapped_bytes.
40	node_memory_MemAvailable_bytes	gauge	Memory information field MemAvailable_bytes.
41	node_memory_MemFree_bytes	gauge	Memory information field MemFree_bytes.
42	node_memory_PageTables_bytes	gauge	Memory information field PageTables_bytes.

43	node_memory_SReclaimable_bytes	gauge	Memory information field SReclaimable_bytes.
44	node_memory_SUnreclaim_bytes	gauge	Memory information field SUnreclaim_bytes.
45	node_memory_Shmem_bytes	gauge	Memory information field Shmem_bytes.
46	node_memory_Slab_bytes	gauge	Memory information field Slab_bytes.
47	node_procs_running	gauge	Number of processes in runnable state.
48	node_sockstat_TCP_alloc	gauge	Number of TCP sockets in state alloc.
49	node_sockstat_TCP_mem	gauge	Number of TCP sockets in state mem.
50	node_sockstat_TCP_mem_bytes	gauge	Number of TCP sockets in state mem_bytes.
51	node_sockstat_sockets_used	gauge	Number of IPv4 sockets in use.
52	node_time_seconds	gauge	System time in seconds since epoch (1970).
53	node_timex_frequency_adjustment_ratio	gauge	Local clock frequency adjustment.
54	node_timex_maxerror_seconds	gauge	Maximum error in seconds.
55	node_timex_offset_seconds	gauge	Time offset in between local system and reference clock.
56	process_resident_memory_bytes	gauge	Resident memory size in bytes.
57	process_start_time_seconds	gauge	Start time of the process since unix epoch in seconds.

Table 5. Selected metrics description

4.2.2 Data acquisition and training data generation

The second step aims to create our training set that later will be used to train and validate the ML model.

We started from 40 executions in the premises to extract the metric data linked to those executions. Different distributions of workers and hour of the day have been considered to catch how the workers affect in the final execution time, and also be able of catching different levels of stress of the instances.

Table 6 contains a summary of the executions:

	day	Hour	execution	exectime	workers
1	2021/02/22	15:10	T1	617820	3
2	2021/02/22	15:29	T2	571378	3
3	2021/02/22	16:11	T3	566439	3
4	2021/02/22	16:24	T4	564199	3
5	2021/02/23	09:12	T5	578790	3
6	2021/02/23	09:25	T6	567978	3
7	2021/02/23	09:50	T7	565619	3
8	2021/02/23	10:23	T8	565052	3
9	2021/02/23	10:40	T9	572151	3
10	2021/02/23	10:54	T10	582628	3
11	2021/02/23	11:09	T11	636311	3
12	2021/02/23	11:34	T12	593063	3
13	2021/02/23	11:53	T13	522618	6
14	2021/02/24	08:00	T14	454466	6
15	2021/02/24	08:13	T15	464412	6
16	2021/02/24	08:24	T16	464266	6
17	2021/02/24	08:35	T17	455849	6
18	2021/02/24	08:47	T18	466922	6
19	2021/02/24	09:02	T19	468000	6
20	2021/02/24	09:15	T20	709955	6
21	2021/02/24	09:35	T21	693223	6
22	2021/02/24	09:52	T22	595968	3
23	2021/02/25	10:26	T23	463951	6
24	2021/02/25	10:38	T24	455543	6
25	2021/02/25	10:49	T25	472430	6

26	2021/02/25	11:00	T26	459940	6
27	2021/02/25	11:15	T27	588759	1
28	2021/02/25	11:29	T28	600364	1
29	2021/02/25	11:48	T29	594230	1
30	2021/02/25	12:06	T30	599054	3
31	2021/02/25	12:32	T31	457041	6
32	2021/02/25	12:45	T32	464953	6
33	2021/03/01	08:01	T33	407144	9
34	2021/03/01	08:12	T34	415762	9
35	2021/03/01	08:24	T35	416799	9
36	2021/03/01	08:38	T36	424616	9
37	2021/03/01	08:50	T37	413798	9
38	2021/03/01	09:02	T38	416003	9
39	2021/03/01	09:13	T39	411904	9
40	2021/03/01	09:25	T40	429661	9

Table 6. Executions summary

To create the training data, a function has been developed (based on the "get_timeseries_from_metric" from EDA) to retrieve the last hour of metrics until the execution starts, for the list of metrics extracted in the EDA section. At the end, each metric contains 60 rows and 57 columns for each execution.

4.2.3 Model development

The critical point is the model development. Different approaches have been followed to find the best model for our problem.

4.2.3.1 Neural Networks

Our starting point was based on creating a regression model to predict the execution time based on the metrics data extracted and the number of workers set by the SLA Manager.

We split our training data in input variables (metrics + workers) and target (execution time).

To get this objective, we first research on neural network models for timeseries forecast using Keras[10]. This research can be summarized in Deep Neural Network models (Figure 15) and Recurrent Neural Network models, such as LSTM, (Figure 16) to capture how the timeseries evolve in relation to the execution time.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_56 (Dense)             (None, 58)                3422
_________________________________________________________________
dense_57 (Dense)             (None, 30)                1770
_________________________________________________________________
dense_58 (Dense)             (None, 15)                465
_________________________________________________________________
dense_59 (Dense)             (None, 8)                 128
_________________________________________________________________
dense_60 (Dense)             (None, 1)                 9
=================================================================
Total params: 5,794
Trainable params: 5,794
Non-trainable params: 0
```

Figure 15. Deep Neural Network definition

```
Model: "model_26"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_32 (InputLayer)        [(None, 60, 58)]          0
_________________________________________________________________
lstm_40 (LSTM)               (None, 60, 10)            2760
_________________________________________________________________
lstm_41 (LSTM)               (None, 60, 5)             320
_________________________________________________________________
lstm_42 (LSTM)               (None, 1)                 28
_________________________________________________________________
dense_27 (Dense)             (None, 1)                 2
=================================================================
Total params: 3,110
Trainable params: 3,110
Non-trainable params: 0
```

Figure 16. LSTM definition

We have discarded this approach because in our problem, the workers variable is very important to predict the execution time, not only the metrics analyzed, and this type of models set different variable importance based on internal implementations. In this scenario, the workers variable is not important to those models, and the same execution time is predicted based on a set of metrics values, whatever how many workers are set.

[10] https://keras.io/

4.2.3.2 Regression models

After discarding the neural network models, a benchmark has been created using different regression models provided by Scikit-learn[11], and a randomized grid of values for the parameters. The list of regression models used is the following:

- LinearRegression()[12]
- Ridge()[13]
- Lasso()[14]
- ElasticNet()[15]
- Lars()[16]
- LassoLars()[17]
- BayesianRidge()[18]
- SGDRegressor()[19]
- KernelRidge()[20]
- KNeighborsRegressor()[21]
- RadiusNeighborsRegressor()[22]

[11] https://scikit-learn.org/stable/index.html

[12] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression

[13] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html#sklearn.linear_model.Ridge

[14] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html#sklearn.linear_model.Lasso

[15] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html#sklearn.linear_model.ElasticNet

[16]

[17] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lars.html?highlight=lars#sklearn.linear_model.Lars https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LassoLars.html#sklearn.linear_model.LassoLars

[18] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.BayesianRidge.html?highlight=bayesianr#sklearn.linear_model.BayesianRidge

[19] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html#sklearn.linear_model.SGDRegressor

[20] https://scikit-learn.org/stable/modules/generated/sklearn.kernel_ridge.KernelRidge.html#sklearn.kernel_ridge.KernelRidge

[21] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html#sklearn.neighbors.KNeighborsRegressor

[22] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.RadiusNeighborsRegressor.html#sklearn.neighbors.RadiusNeighborsRegressor

- GaussianProcessRegressor()[23]
- DecisionTreeRegressor()[24]
- RandomForestRegressor()[25]
- AdaBoostRegressor()[26]
- GradientBoostingRegressor()[27]

The training set defined for neural networks models does not fit with this kind of models. To adapt it, we have transformed the 60 rows for each execution to 1 by getting the mean, max, min and std values for each metric timeseries. Our new input variables are now this aggregated information and the number of workers. The target, that has not change, is execution time.

Figure 17 details the score of each regression model with the best combination of parameters for one benchmark run:

[23] https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessRegressor.html?highlight=gaussian#sklearn.gaussian_process.GaussianProcessRegressor

[24] https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html#sklearn.tree.DecisionTreeRegressor

[25] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html#sklearn.ensemble.RandomForestRegressor

[26] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html#sklearn.ensemble.AdaBoostRegressor

[27] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html#sklearn.ensemble.GradientBoostingRegressor

```
Best model linear
    best_params: {}
    best_score: 1.0
    score en test: 1.0
-----------------------------------------
-----------------------------------------
Best model ridge
    best_params: {'alpha': 0.9325417032901051, 'solver': 'svd'}
    best_score: 1.0
    score en test: 1.0
-----------------------------------------
-----------------------------------------
Best model lasso
    best_params: {'alpha': 0.25731122014169316, 'selection': 'random'}
    best_score: 0.9999997130353292
    score en test: 0.9999982057662904
-----------------------------------------
-----------------------------------------
Best model elastic
    best_params: {'alpha': 0.46036614848292556, 'l1_ratio': 0.1958024901452704, 'selection': 'cyclic'}
    best_score: 0.9999995228292181
    score en test: 0.9999994502815811
-----------------------------------------
-----------------------------------------
Best model lars
    best_params: {}
    best_score: -3.885573675110852e+114
    score en test: -4.588420755087938e+115
-----------------------------------------
-----------------------------------------
Best model lassolars
    best_params: {'alpha': 0.9136463059020467}
    best_score: 0.8672415075164664
    score en test: 0.9043635061047156
-----------------------------------------
-----------------------------------------
Best model bayesian
    best_params: {'alpha_1': 0.5850421661010451, 'alpha_2': 0.17544010803256882, 'lambda_1': 0.42352436033841323, 'lambda_2': 0.24385665791920264}
    best_score: 0.9422981870853206
    score en test: 0.7352534238451848
-----------------------------------------
-----------------------------------------
Best model sgd
    best_params: {'alpha': 0.39244818614694665, 'epsilon': 0.5913739161256906, 'l1_ratio': 0.8155115052091472, 'learning_rate': 'optimal', 'loss': 'epsilon_insensi
tive', 'penalty': 'l1'}
    best_score: -3.359760145271304e+24
    score en test: -8.652893799229716e+24
-----------------------------------------
-----------------------------------------
Best model kernel
    best_params: {'kernel': ExpSineSquared(length_scale=1.67, periodicity=7.74), 'alpha': 0.01}
    best_score: 0.9999915296553354
    score en test: 0.9999935153456607
-----------------------------------------
-----------------------------------------
Best model knn
    best_params: {'weights': 'distance', 'n_neighbors': 4, 'algorithm': 'auto'}
    best_score: 1.0
    score en test: 1.0
-----------------------------------------
-----------------------------------------
Best model rnn
    best_params: {'algorithm': 'ball_tree', 'p': 1, 'radius': 0.587983375520108, 'weights': 'distance'}
    best_score: 1.0
    score en test: 1.0
-----------------------------------------
-----------------------------------------
Best model gaussian
    best_params: {}
    best_score: 1.0
    score en test: 1.0
-----------------------------------------
-----------------------------------------
Best model tree
    best_params: {'splitter': 'best', 'max_features': 'auto', 'max_depth': None, 'criterion': 'mae'}
    best_score: 1.0
    score en test: 1.0
-----------------------------------------
-----------------------------------------
Best model randomforest
    best_params: {'max_features': 'sqrt', 'max_depth': None, 'criterion': 'mse', 'bootstrap': True}
    best_score: 1.0
    score en test: 1.0
-----------------------------------------
-----------------------------------------
Best model adaboost
    best_params: {'learning_rate': 0.07652902902086312, 'loss': 'square', 'n_estimators': 50}
    best_score: 0.9959408877715477
    score en test: 0.9960071372098764
-----------------------------------------
-----------------------------------------
```

Figure 17. Benchmark run of regression models

We have discovered that the scores are sometimes overfitted and other times too bad. But the main issue is that we have the same behavior we have with neural networks for regression, the workers variable importance is too low that the same execution time is predicted all the time for different workers values.

4.2.3.3 Classification models

At the end, we need to solve the workers variable importance, and to do that we have change our perspective to get our objective.

This change is based on:

- First, classify how stressed is our system by analysing the metrics data extracted in the EDA. It can be classified in 3 classes: low, normal, or high.
- And second, return an estimated execution time for that level of stress and the number of workers. Those rules about "stress → workers → execution time", have been extracted from the 40 experiments to create our training set, and is described in Table 7. Un upper value for high stress is set to 1000000 to indicate that the execution time could be too long.

Workers	Stress	Execution time
1	Low	(550000,580000)
1	Normal	(580001,610000)
1	High	(610001,1000000)
3	Low	(550000,570000)
3	Normal	(570001,600000)
3	High	(600001,1000000)
6	Low	(430000,450000)
6	Normal	(450001,470000)
6	High	(470001,1000000)
9	Low	(390000,410000)
9	Normal	(410001,420000)
9	High	(420001,1000000)

Table 7. Rules definition for linking workers, stress, and execution time

The training set has been transformed in the same way as with the regression models to have the aggregated information for each execution.

In this case, a preprocessing of the training set has been taken into consideration to balance it and getting a similar amount of data for each stress class we want to classify. Others preprocessing methods have been researched like a transformation of the variables by applying standardization and scaling the data and applying PCA techniques to reduce the amount of variable to those that contains most of the variance.

The same benchmarking process has been followed but with a list of classifiers models:

- LogisticRegression()[28]

[28] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic#sklearn.linear_model.LogisticRegression

- SGDClassifier()[29]
- KNeighborsClassifier()[30]
- DecisionTreeClassifier()[31]
- RandomForestClassifier()[32]
- AdaBoostClassifier()[33]
- GradientBoostingClassifier()[34]
- MLPClassifier()[35]

The summary of the benchmark run for the classifier models using the balanced dataset is described in Figure 18:

[29] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html?highlight=sgdcla#sklearn.linear_model.SGDClassifier
[30] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier
[31] https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier
[32] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier
[33] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier
[34] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier
[35] https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier

```
Best model logistic
   best_params: {'C': 0.043543533944407464, 'penalty': 'l1'}
   best_score: 0.8305084745762712
   score en test: 0.8
----------------------------------------
----------------------------------------
Best model sgd
   best_params: {'alpha': 0.22324436907502176, 'loss': 'modified_huber', 'penalty': 'elasticnet'}
   best_score: 0.3728813559322034
   score en test: 0.26666666666666666
----------------------------------------
----------------------------------------
Best model kneighbors
   best_params: {'weights': 'distance', 'n_neighbors': 6, 'algorithm': 'kd_tree'}
   best_score: 0.8813559322033898
   score en test: 0.8666666666666667
----------------------------------------
----------------------------------------
Best model tree
   best_params: {'criterion': 'gini', 'max_depth': None, 'max_features': 10, 'min_samples_split': 2}
   best_score: 0.8983050847457628
   score en test: 0.8
----------------------------------------
----------------------------------------
Best model random_forest
   best_params: {'bootstrap': False, 'criterion': 'gini', 'max_depth': None, 'max_features': 3, 'min_samples_split': 7}
   best_score: 0.8305084745762712
   score en test: 0.8666666666666667
----------------------------------------
----------------------------------------
Best model adaboost
   best_params: {'algorithm': 'SAMME.R', 'learning_rate': 0.6497617283318484, 'n_estimators': 50}
   best_score: 0.9152542372881356
   score en test: 0.8
----------------------------------------
----------------------------------------
Best model gradientboosting
   best_params: {'learning_rate': 0.738310950168935, 'n_estimators': 50}
   best_score: 0.8813559322033898
   score en test: 0.8666666666666667
----------------------------------------
----------------------------------------
Best model mlp
   best_params: {'activation': 'identity', 'alpha': 0.9623671707014817, 'hidden_layer_sizes': (10,), 'learning_rate_init
   best_score: 0.3389830508474576
   score en test: 0.26666666666666666
----------------------------------------
----------------------------------------
```

Figure 18. Benchmark run of classification models

4.2.4 Model selected

Based on the result from the classifier models benchmark, the KNeighborsClassifier model has been selected to be the best model for our scenario, with an accuracy of 0.8666666666666667.

The best combination of parameters for this model are:

- **Weights**: “distance”
- **N_neighbors**: 6
- **Algorithm**: “kd_tree”

After evaluating the model, the confusion matrix and classification reports are provided in Table 8 and Table 9, where we can discover that the model classifies the stress level of the system with a high accuracy and precision. This model sometimes predicts that a normal stress level of the system is high, which is an error that we could manage.

		Predicted label		
		High	Low	Normal
True label	High	4	0	0
	Low	0	5	0
	Normal	2	0	4

Table 8. Confusion matrix

	precision	recall	F1-score	Support
High	0.67	1.00	0.80	4
Low	1.00	1.00	1.00	5
Normal	1.00	0.67	0.80	6
Micro avg	0.87	0.87	0.87	15
Macro avg	0.89	0.89	0.87	15
Weighted avg	0.91	0.87	0.87	15

Table 9. Classification report

4.2.5 Model exposition

SLA Predictor has been exposed as a microservice to be accessible using REST calls. To achieve this objective, the microservice is encapsulated in a Docker container using Python and Flask[36].

The "predictSLA" method could be called by passing the number of workers and the desired execution time from the SLA Manager. It performs a query to Prometheus to retrieve the last hour of our selected metrics, classify it to know how much the system is stressed, and return the most similar number of workers that fits the input data based on the rules defined for "stress→workers→execution time". If there is not a valid number of workers to the SLA requirements, it returns -1.

Some examples of this endpoint:

- **/predictSLA?workers=3&exectime=620000**
  - Internal result: [(1, (580001, 610000)), (3, (570001, 600000)), (6, (450001, 470000)), (9, (410001, 420000))]
  - Output: 3
- **/predictSLA?workers=3&exectime=520000**
  - Internal result: [(6, (450001, 470000)), (9, (410001, 420000))]
  - Output: 6

[36] https://flask.palletsprojects.com/en/1.1.x/

4.2.6 Future work

To continue developing more features and capabilities, some options can be taken into consideration. One possible update is adding more experiments to the training data with a wide variety of different combinations of workers and stress system.

Another update is oriented of improving the classification model by implementing recurrent neural networks to identify how the values evolve instead of having a static image of the system. This path is hard to follow due to the problems described on this section with a regression perspective, so is needed to take a particular care when researching this line.

Finally, this solution is easily adaptable to other applications by defining new rules for that executions, and also it is possible to retrain the model to classify the system stress level along the time with other behaviors that could appear.

4.3 Validation of the architecture with time guarantee

We have done a set of tests using this new component (SLA Predictor). In these tests we have used the same application described in section 3.1. On one side, we have defined an initial number of workers for the COMPSs workflow. And on the other side, we have defined a total execution time in the QoS parameters. This way, Rotterdam and the SLA Manager create an SLA where the constraint is to execute the workflow in the desired time or less (time guarantee).

The objective of these experiments is to show the impact of using the SLA Predictor before deploying the application in the Cloud and compare it with the previous experiments. Instead of scaling out the application during the execution time, now we use ML technics to predict the behaviour of the workflow in the Cloud, to adjust the initial number of workers before launching the application.

Here follow the two sets of experiments, and the result of 6 executions each. The table shows REAL final times obtained with the number of workers returned by the SLA Predictor:

1. Experiment 1 in COMPSs master:
    - Initial number of workers: 3
    - Desired total execution time: 540000 ms
    - Result of the call to SLA Predictor: 6 workers

	6 workers
1	503947
2	525037
3	521731
4	528276
5	542583
6	539082
Mean	**526776**

Table 10 – Real execution times obtained after calling SLA Predictor with Target Execution time < 540000 and 3 workers (in ms)

2. Experiment 2 in COMPSs master:
    - Initial number of workers: 3
    - Desired total execution time: 480000 ms
    - Result of the call to SLA Predictor: 9 workers

	9 workers
1	487218
2	477542
3	461500
4	479326
5	481609
6	472977
Mean	**476695**

Table 11- Real execution time obtained after calling SLA Predictor with Target Execution time < 480000 and 3 workers (in ms)

Although the results of executing the first set of experiments (initial workers: 3, execution time: *540000 ms* - Table 10) show that the desired execution time (SLA) is fulfilled (*t=526776 ms*), we can observe that the average execution time is higher than the results we obtained by executing the same workflow with 6 initial workers without using the scalability features (*t=466469 ms*) shown in Table 2. The same applies to the second set of experiments shown in Table 11, compared with the results in Table 2 with 9 workers.

From the logs provided by the SLA Predictor, we have seen that some of these last experiments were executed on the most stressed level of the system (when running the first experiments we didn't have this information, provided by SLA Predictor).

```
[(6, (470001, 1000000)), (9, (420001, 1000000))]
[pid: 48|app: 0|req: 23/39] 192.168.7.28 () {34 vars in 488 bytes} [Tue Mar 16
08:20:37 2021] GET /predictSLA?workers=3&exectime=540000 => generated 1 bytes in
12138 msecs (HTTP/1.1 200) 2
```

Also, we have seen that the execution times predicted by the SLA predictor, tend to be more optimistic than real final execution times obtained on the experiments:

```
[(6, (450001, 470000)), (9, (410001, 420000))]

[pid: 49|app: 0|req: 17/40] 192.168.7.28 () {34 vars in 488 bytes} [Tue Mar 16
08:36:04 2021] GET /predictSLA?workers=3&exectime=480000 => generated 1 bytes in
14645 msecs (HTTP/1.1 200) 2 headers in 78 bytes (1 switches on core 0)
```

The results obtaining are constrained by:

- The level of stress of the cluster and network infrastructures.

- The initial data used for training the model.

From all 12 sample experiments, **the average execution time always fulfils the SLA**, and in 9 cases it's **lower than the target value**. Thus, there are cases (3) where the SLA manager generates a violation. This is because the SLA Predictor selects the minimum

number of workers, to avoid scalation and its overhead and thus sometimes having a slightly higher execution time than desired.

Now, to really assess the benefits of the SLA Predictor, we must **compare these new results with the results of scaling out the number of workers during execution time.** When avoiding to scale from 3 to 6 workers , we can see an improvement: *526776 ms* vs *593907 ms* (Table 10 (mean) vs Table 4 ('3->6' mean)). The same applies when avoiding to scale the number of workers during execution time from 3 to 9 workers: *476695 ms* vs *557988 ms* (Table 11 (mean) vs Table 4: '3->9' mean).

We can conclude that with the SLA Predictor and the new QoS metric (execution time vs deadline misses), we avoid the generation of violations in most cases, and we guarantee the execution time even in stressed states of the system. We have also identified some elements susceptible to be improved in the future.

4.4 Demonstration

To show the improvement in execution time from the initial architecture to the architecture with SLA Predictor, we have recorded a demonstration in a video that is available at the CLASS intranet:

https://class-project.eu/user/login

A dedicated user has been created for demonstration purposes, with limited access to deliverables and related videos. The credentials to access this service are the following:

Username: ***EC_user***

Password: ***@Hz.52qXXF#K23***

After logging in, click on "Intranet", the demonstration videos and files of this deliverable are located in "PU_D4-6Report" directory.

5 Result of the Validation

In Section 2 we verified that the Cloud initial requirements identified in MS1 had been met in MS3 as presented in Final release of the Cloud Data Analytics Service Management and Scalability components. Some enhancements, not identified as initial requirements, had been implemented as well. We can conclude that all initial requirements have been achieved.

In Section 3 we verified that the Technical Requirements of the CLASS Software Architecture have been met, but that there was a margin to improve in providing real-time guarantees, as described in section 3.1.

In Section 4 we presented a new architecture to secure time guarantee, and a new component called SLA Predictor. The results of the experiments in section 4.3 allow us to assert that all the Technical Requirements of the CLASS Software are met by the Cloud Data Analytics Service Management and Scalability components.

6 Conclusion

This deliverable reported on the work done in WP4 from M30 to M39. The target at milestone MS4 of task T4.4 Validation of Cloud Computing side has been successfully achieved and documented in this deliverable.

This deliverable also presents the code and a demonstration video of the improvements implemented into the Final release of the Cloud Data Analytics Service Management and Scalability components.

The progress done in the last milestones has helped us to achieve all WP4 objectives successfully.

This validation will be complemented with the Use Case Evaluation that will be presented in D1.6 in M42.

References

Blog, C. (n.d.). *CLASS Cloud Computing Platform - first integration in the City of Modena Data Center*. Retrieved from https://class-project.eu/news/class-cloud-computing-platform-first-integration-city-modena-data-center

D2.1, C. (n.d.). *D2.1 CLASS Software Architecture Requirements and Integration Plan.*

D4.1, C. (n.d.). *D4.1 Cloud Requirement Specification and Definition, Deliverable D4.1 of CLASS project.*

D4.2, C. (n.d.). *D4.2 First release of the Cloud Data Analytics Service Management components, Deliverable D4.2 of CLASS project.*

D4.4, C. (n.d.). *D4.4 First release of the Cloud Data Analytics Service Scalability components, Deliverable D4.4 of CLASS project.*

D4.7, C. (n.d.). *D4.7 Final release of the Cloud Data Analytics Service Management and Scalability components.*