A Platform to Enable Self-Adaptive Cloud Applications Using Trustworthiness Properties

José D'Abruzzo Pereira
University of Coimbra, CISUC, DEI
josep@dei.uc.pt

Rui Silva
University of Coimbra, CISUC, DEI
rfsilva@student.dei.uc.pt

Nuno Antunes
University of Coimbra, CISUC, DEI
nmsa@dei.uc.pt

Jorge L. M. Silva
University of Campinas, IC
jorgem@unicamp.br

Breno de França
University of Campinas, IC
breno@ic.unicamp.br

Regina Moraes
University of Campinas, FT
regina@ft.unicamp.br

Marco Vieira
University of Coimbra, CISUC, DEI
mvieira@dei.uc.pt

ABSTRACT

Self-Adaptive Systems (SASs) reflect on both their state and on the environment and change their behavior to satisfy the expected objectives. Cloud systems are self-adaptive by nature, especially considering the resources used in a pay-as-you-go manner. Satisfying trustworthiness (worthiness of a service based on evidences of its trust) properties also demands self-adaptation capabilities. Unfortunately, developers lack an easy-to-use platform to support the assessment of such properties and to execute the required adaptions. This paper presents TMA, a platform that implements a MAPE-K control loop for cloud systems, supported by a distributed monitoring system based on probes. Quality Models are used to express trustworthiness properties, resulting in scores, which are used to plan adaptations through evaluation rules. These plans are executed by actuators. A demo shows the scaling up/down of the number of containers in a cloud application of a set of web services from TPC Benchmarks, as a result of changes observed in the environment.

KEYWORDS

self-adaptive systems, trustworthiness, cloud applications, quality model

ACM Reference Format:
José D'Abruzzo Pereira, Rui Silva, Nuno Antunes, Jorge L. M. Silva, Breno de França, Regina Moraes, and Marco Vieira. 2020. A Platform to Enable Self-Adaptive Cloud Applications Using Trustworthiness Properties. In *IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3387939.3391608

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SEAMS '20, October 7–8, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7962-5/20/05.
https://doi.org/10.1145/3387939.3391608

1 INTRODUCTION

Cloud applications are deployed through models such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS), and the clients pay per the use of the resources [16]. During the life cycle of an application, several versions with new features are released to production. Clients usually have applications offered to their end-users in a SaaS model, but they may also sell the use of such applications to their own customers. In this context, it is important to monitor cloud applications and services both at design-time (in the developed source code), and at run-time to detect potential inconsistencies as early as possible, thus avoiding losses.

In Cloud computing, *trustworthiness* can be defined as the worthiness of a service and its provider for being trusted [15], thus including a multitude of properties (e.g., reliability, availability, security, privacy, dependability, etc. [17]). Continuously monitoring and assessing the trustworthiness of cloud systems is not trivial due to many factors, such as the number of properties involved in trustworthiness. Also, trust is a subjective concept that is built based on guarantees, experiences, transparency, and accountability. A trustworthiness life cycle can be defined to assess cloud applications both before (design-time) and after (run-time) deployment: at design-time, the source code can be analysed; and at run-time, the cloud applications need to be monitored, and adaptations may be performed. However, current self-adaptive capabilities are rather limited and based only on *CPU usage* and *memory consumption* [10].

This demo paper intends to contribute to the **assessment and improvement of the trustworthiness of cloud applications**, considering the relevant properties and a trustworthiness life cycle inspired on the MAPE-K cycle [11]. The TMA platform supports these activities by providing a solution for **Cloud applications that require self-adaptation for maintaining/achieving trustworthiness without the need for creating a managing element from scratch**. Compared to existing solutions (e.g., HPA [10]), our proposal adds flexibility that allows the user to prepare their systems to adapt according to a wide range of properties. The interface between the cloud application and TMA is supported by *probes* to monitor the status of the application, and by *actuators* to perform the adaptations [11]. Quality Models (QM) aggregate data collected

from the cloud application, and generate trustworthiness scores. A basic trustworthiness model is explained in [15], and a privacy QM can be seen in [5].

TMA supports the task of engineering more reliable software systems, as it provides developers and owners of the applications with an easy way to monitor and maintain the key properties of the system. Each MAPE-K component is designed as a microservice [13] that can be easily deployed in a container-based system (e.g., Kubernetes, Docker Swarm).

A usage scenario shows the applicability and flexibility of the platform. In practice, we monitor resource consumption (CPU usage and memory consumption) and performance (response time and throughput) of a Cloud application. TMA calculates its trustworthiness level, and adaptations are made when needed. The scale up/down actions of the application server are dispatched according to the workload. Results allow observing how the scores can guide the system adaptation.

The remaining of this paper is structured as follows. Section 2 presents background and related work. Architecture and implementation details are presented in Section 3. The usage scenario is presented in Section 4, and the results and discussion in Section 5. Finally, Section 6 concludes the paper and presents future steps. Further information about TMA can be found online:

– http://tma.dei.uc.pt/

2 BACKGROUND AND RELATED WORK

The most used adaptation control loop was introduced by IBM [11] and is named MAPE-K. Its name is an acronym for five components: Monitor, Analyze, Plan, Execute, and Knowledge. The *Monitor* is responsible for collecting details about the managed resource, while *Analyze* reasons over the data collected in the previous phase. Decisions are made in the *Plan* to achieve the goal and objectives. *Execute* is responsible for interacting with the *managed element* (for example, a Cloud Application) to assure that the adaptations are made. Finally, *Knowledge* is a repository that supports other phases. The component that implements the adaptation control loop is called *managing element* [2].

All the interaction between the managing element and the managed element happens through the manageability endpoints. *Sensors* or *Probes* are used to send all the data from the managed element to the managing element. *Effectors* or *Actuators* perform the adaptations on the managed element.

There are different approaches to promote self-adaptation, such as the architecture-based solution used by Rainbow [8]. It adds the adaptation control loop in the architectural layer, while probes and effectors stay in the system layer along with the managed element. The architecture-based self-adaptation allows a global perspective of the system, and system-level properties and integrity constraints are exposed.

In a cloud environment, some self-adaptation features are found. For instance, the infrastructure provider Amazon Web Services has the AWS Auto Scaling service [1], which provides a rule-based autoscaling service for the resources deployed on the Amazon Cloud. For container-based systems (e.g., Kubernetes [9], Docker Swarm [7], Apache Mesos [3]), orchestrator frameworks are usually responsible for autoscaling features. For instance, Horizontal Pod Autoscaler (HPA) [10] for systems deployed using Kubernetes. Both Amazon AWS and Kubernetes HPA usually use metrics of memory consumption and CPU usage to support the scaling.

Our solution differentiates from existing ones in terms of flexibility (of monitoring and adaptation and on supporting diverse systems). Also, we introduce Quality Models (QMs) to aggregate diverse metrics according to user-defined preferences, and that simplifies the task of decision-making based on several properties. In practice, it is useful in scenarios that require: *1)* more diverse or more complex adaptations; *2)* the analysis of diverse trustworthiness properties and scores (e.g., security, privacy, dependability, and coherence); and *3)* the computation of scores that require inputs from multiple levels of the managed element (e.g., at the client level, service level).

3 TMA-PLATFORM ARCHITECTURE

The TMA-Platform supports the trustworthiness assessment that involves different attributes, properties, and characteristics, which vary depending on the objectives of the system that is being assessed [15]. Our trustworthiness framework for cloud applications is composed of:

- A definition of the relevant properties and metrics used to characterize trustworthiness;
- A trustworthiness lifecycle (Fig. 1), inspired in the MAPE-K cycle and that covers two main phases: design-time (application in development), and run-time (application in use);
- A monitoring platform that receives measurements and events from the managed application;
- Measurement instruments that allow which gathering the information to be used (measurements and events);
- Quality Models (QMs) that define how the measurements will be used to compute the scores;
- Actuators that implement the adaptation logic are included in the managed system and allow adaptations that aim at improving the system trustworthiness.

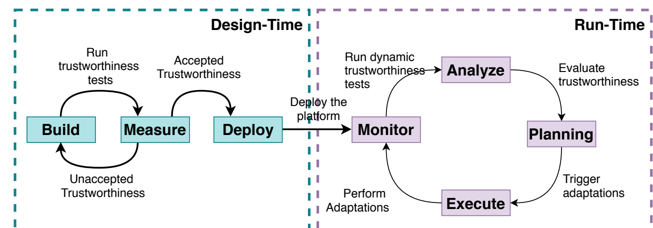

Figure 1: Lifecycle of Trustworthiness Assessment

A platform to support the trustworthiness monitoring and assessment of Cloud applications was developed. Figure 2 presents a high-level architecture of the TMA platform, which follows a microservice architecture [18], where each component represents one MAPE-K function [11]. Each one of the components is deployed into a container inside a Kubernetes pod. Kubernetes is an open-source system that allows automatizing the deployment and management of container-based applications [9]. Kubernetes starts the containers and deploys them according to the specification of the pod (wrapper defined by Kubernetes and its base unit of management).

The implementation details of each component are described in the following sections. To assure that the communication among

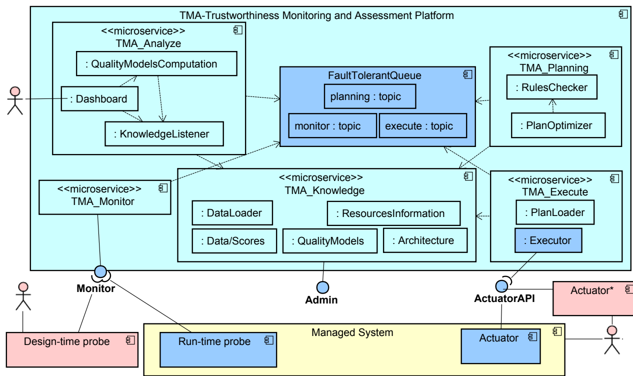

Figure 2: Architecture and Interfaces of TMA.

the components is handled in a reliable way, a fault-tolerant mechanism is used. The FaultTolerantQueue is implemented using Apache Kafka, which allows creating topics that the components can subscribe to consume the messages [4]. Details on how to create a probe and an actuator are described in Sections 3.1 and 3.4, respectively. A Docker image and a YAML (YAML Ain't Markup Language) file were created for each component of the TMA, to support the deployment into a Kubernetes cluster. The YAML file contains the specification of a Kubernetes object, and Kubernetes uses it to decide what to deploy (e.g., the image container that will be used and the ports that will be open to be invoked). We automate the platform deployment through bash scripts and recipes (https://github.com/eubr-atmosphere/tma-yaml).

3.1 Monitor Component

The **Monitor** component provides a RESTful API interface for the *probes* to post JSON messages (observations) of the managed element (i.e., the Cloud Application) to TMA. The content of the message contains its data type, which can be: *i) measurement* (e.g., execution time, memory allocated, CPU in use, response time) or *ii) event*: (e.g., the scale up process has started). Measurements are numeric values that will be used to calculate the scores based on a Quality Model (QM), while events are evidence of the occurrence of a fact. The service is deployed using the web microframework Flask [21]. The data pushed by probes through this interface is enqueued in the *monitor topic* to assure that it will be stored in the *Knowledge* component in a reliable way. TMA is also prepared to receive the results of design-time assessment tasks, in which performance and scalability requirements are not so stringent.

This component validates all data collected by probes according to a JSON schema. SSL/TLS encryption is used to secure communication between the probes and the Monitor component. Hence, all probes must have the Monitor digital certificate, acquired during the initial registration of the probe in the platform. Later, when a probe sends data to the Monitor, its certificate is used to authenticate.

In addition to the data type (measurement or event), the JSON message contains value, and time of the observation. If the data received are valid, the Monitor sends them to the FaultTolerantQueue through an Apache Kafka topic named monitor. Otherwise, the data are discarded, and an error message is returned to the probe.

The QueueListener subcomponent is implemented through the DataLoader component. It pulls data from the monitor topic and executes the data normalization process, so the data are correctly inserted in the Knowledge database.

A few probes were developed to demonstrate and validate the TMA platform, and libraries to develop new probes were created for Java, Python, and C#, as well as a Docker image. Using the supporting tools described above, we make available 12 concrete probes for different measurements and information about them can be found online (tma.dei.uc.pt/probes). The ones listed below are used in the usage scenario (see Section 4):

- probe-k8s-metrics-server – gathers information about pods and nodes in a Kubernetes cluster. It collects measurements of *CPU usage* and *memory consumption* using the monitoring open-source component *metrics-server* (github.com/kubernetes-incubator/metrics-server), which replaces the deprecated Heapster [9];
- probe-client-java – collects performance metrics of a client that performs REST requests to a server. It is able to monitor *response time*, *throughput*, and *rate of served requests under a predefined threshold*.

3.2 Analyze Component

The **Analyze** component is responsible for reasoning over the data gathered by the Monitor component, and aggregating the measurements into a trustworthiness score. The data are read from the Knowledge component, and the scores are calculated and stored. Several aspects may be considered to support the decision-making (adaptation) process. Hence, a combination of different sources of information is necessary.

A way of doing this is through a Quality Model (QM), by aggregating the measurements and come up with a score. The focus of the Analyze component is on trustworthiness properties. Hence, the integration of QMs of various trustworthiness properties (security, privacy, coherence, isolation, stability, fairness, transparency, and dependability) allows forming complex trustworthiness QMs [17]. They can be used to compute trustworthiness scores. A Dashboard that allows users to analyze information at runtime and adjust the parameters of the QMs and thresholds (objectives) is also available.

The Performance QM is used as an example, and the data are obtained through the probe-client-java probe (described in Section 3.1). This probe is deployed on the client-side, and it collects measurements from the user perspective. The leaf attributes contain either values obtained from the probe (e.g., *throughput* - A1, *response time* - A2, or *rate request under contracted* - A3), or obtained from the calculation of values by the probes (e.g., *rate served requests* - A4, which is the division of throughput by demand).

Among the leaf attributes, some of them are benefit (*throughput* - A1 and *rate served requests* - A4), and others are cost attributes (*response time* - A2 and *rate requests under contracted* - A3). They are represented in Figure 3 with different colors: benefit attributes in green, and cost attributes in orange.

An adjustment needs to be done in the cost attribute to be interpreted as a benefit attribute since the performance score should be interpreted as a benefit attribute. As all the attributes are normalized and are in a 0-to-1 range, the transformation is done through the following formula: $benefit_attribute = 1 - cost_attribute$. After the transformations, the attributes can be combined.

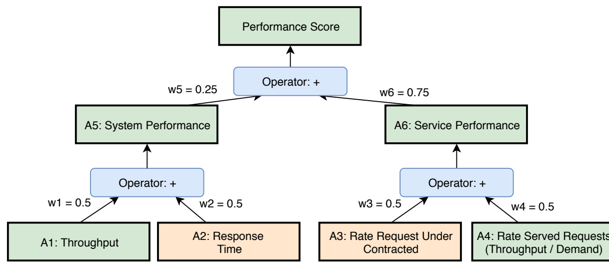

Figure 3: Performance Quality Model used for this usage scenario

The first partial score is *System Performance* (A5). It is composed of the *throughput* (A1), and *response time* (A2). The second partial score is the *Service Performance* (A6), with attributes *rate request under contracted* - A3, and *rate served requests* - A4. The final score is obtained with the formula:

$score = 0.25 * (0.5 * A1 + 0.5 * A2) + 0.75(0.5 * A3 + 0.5 * A4)$

As the *Service Performance* partial score is more important, it received weight $w6 = 0.75$, while the *System Performance* partial score has weight of $w5 = 0.25$. This score is calculated every second.

Additionally, the Analyze component needs to know the following configurations in advance: *i) periodicity of calculation*, and *ii) observation window*.

3.3 Planning Component

The **Planning** component is responsible for checking the scores and coming up with a plan in case an adaptation is needed. An adaptation plan is a set of actions to achieve the required goals or to recover the desired trustworthiness levels. There are different adaptation decision approaches, e.g., models, rules/policies, goals, or utility [12]. A rule-based approach [14] is used by the TMA.

TMA uses the business rules management system *Drools* [19], which is a Java-based tool. It provides an easy-to-read and easy-to-understand language to specify rules, with conditions and actions in case the conditions are met. The decision to adapt is made based on the score calculated by the *Analyze* component. If the score is either above or below a threshold, an adaptation is dispatched. In case the user needs to extend it, a Java class can be created and invoked by the rules.

A Drools rule used by TMA is shown on Listing 1. The *performance* score (calculated using the QM detailed in Section 3.2) is used to specify the condition through the when directive. When the score value exceeds the threshold of 0.08, and the number of pods is smaller than 2, an adaptation is dispatched to increase the number of pods to 2. The adaptation plan is specified through the then directive. Details about the execution of the plan are presented in Section 3.4.

Listing 1: Scale up Drools rule example

```
rule "Score validation - Wildfly Scale up"
  when
    $score: TrustworthinessScore ( performanceScore.score >
          0.08, podCount < 2 )
  then
    Action action = new Action(1, "scale", 9, 5);
    action.addConfiguration(
        new Configuration(
        2, "metadata.name", "wildfly"));
    action.addConfiguration(
        new Configuration(
        3, "spec.replicas", "2"));
    AdaptationManager.performAdaptation( action,
          AdaptationManager.obtainMetricData($score) );
end
```

A similar rule is needed to scale down the number of pods. In that case, a different, lower bound threshold should be used.

3.4 Execute Component

The **Execute** component is responsible for invoking the actions of the adaptation plan defined by the *Planning* component and notified through the *execute topic*. The TMA interacts with the management element (target of the adaptations) through the *Actuators*. Each actuator provides a RESTful API service invoked by the *Execute* component. All communication is secure, and all the messages are encrypted using the keys of both the Executor component and the actuator to be used.

Currently, two libraries are provided to ease the communication with actuators developed in Java and Python. Three actuators are available: *i)* kubernetes-actuator: used to scale up and scale down Kubernetes pods; *ii)* email-actuator: used to send e-mail notifications when the scores are not in the expected thresholds; and *iii)* api-actuator: used to interact with a third-party API.

This actuator kubernetes-actuator is used in the scenario presented in Section 4, and it was developed using the Actuator Java Library. Complete usage instructions and demos are available online in http://tma.dei.uc.pt/.

3.5 Knowledge

The **Knowledge** component is responsible for storing all the data, such as measurements and events, QM definitions, trustworthiness scores, information about the application architecture, resources and assets available, and adaptation plans. Its implementation contains a MySQL DBMS (*knowledge* database) and a block-storage solution Ceph.

Data collected by the probes are inserted in the corresponding table. The TMA database follows a star schema as the one shown on Figure 4. The fact table Data contains the basic numerical facts provided by the probes, and the dimensions include all the different perspectives needed to characterize them. The remaining information about the architecture is represented in the Resource and Probe tables. The Description table contains the specification about either the measurements or events provided by each probe. Finally, the Time dimension is represented in the conceptual diagram, but the information about the time is stored directly in the Data table (for performance reasons).

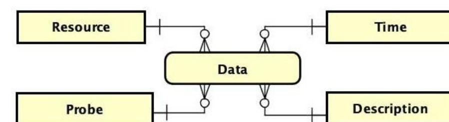

Figure 4: Data model used by TMA for the Monitor Component.

The model used by the *Execute* component reflects the data to dispatch the adaptations, and it can be seen online[1] (not included

[1]http://tma.dei.uc.pt/planning.html

here due to space constraints). Information about the actuator as well as the actions that they can perform is stored in the tables represented in the adaptation model.

An application called Admin was developed to ease the configuration of the platform via the knowledge database. It allows registering probes and actuators to the database, besides information about the architecture of the managed element. It is split between two different components: the REST API and the GUI. Both components are deployed as a Kubernetes pod.

The REST API is implemented in Java, using the Spring framework to expose the REST services. The API takes care of every request and proceeds to update the Knowledge database. The GUI provides user-friendly interface that allows the TMA administrator to invoke the REST API service. Features such as adding a probe, adding an actuator and configuring their actions, and adding the resources of the managed system are available in this application.

4 USAGE SCENARIO: SCALING CONTAINERS

The usage scenario presented in this section shows one application that consists of a set of web services from TPC Benchmarks [22]. To store the data, both the web application server Wildfly [20] and MySQL database are used. We focus on performance-related experiments as these are the ones that have alternatives even if limited (e.g., HPA), and they are easy to understand. However, TMA can be used with other metrics such as dependability, privacy, and security.

The experiment consists of varying the request workload to the application and observing the adaptations and the scores. Four different scenarios are used: **A:** No adaptation; **B:** Using Kubernetes HPA [10]; **C:** Using TMA with Performance QM; **D:** Using TMA with Resource Consumption QM.

The application is deployed on Kubernetes. It is configured using the controller StatefulSet from Kubernetes, which guarantees the order in which the pods are created and scaled. As it is using the StatefulSet controller, every time the number of pods changes through a scale action, a pod is either created or deleted when the remaining ones are running properly to avoid disruptions in the service. This solution allows scaling the number of pods, as the requests are sent to the same endpoint. All the load is balanced among the pods by Kubernetes.

For this usage scenario, only one Wildfly pod is initially created. When the workload increases, more pods are created to balance the load. The MySQL database does not scale during the experiment.

All the experiments were performed in a machine with the following configuration: **CPU**: Intel(R) Xeon(R) Gold 5118 CPU 2.30GHz x 24; **Memory**: 96GB DDR4 RAM; **Disk**: Dell SSD PERC H330 Adp 1TB. This server, used to create Virtual Machines (VMs), uses the tool Infrastructure Manager (IM) [6]. IM automates the deployment and configuration of VMs. For the usage scenario, 4 VMs were created for the Kubernetes cluster (each VM with 4 CPU cores, 16GB of Memory, and 200GB of disk storage):

- **Master Node**: instance that contains the master node of the Kubernetes cluster. It allows creating Kubernetes objects (e.g., working nodes, pods, volume storage);
- **Working Node (2 instances)**: instances where the pods with Docker containers are deployed and run. After joining the cluster through the master node, pods can be deployed in the working nodes by the master node;
- **Storage Node**: instance where the block storage is configured (Ceph [23]). As data stored in the pods are ephemeral, there is a need for a persistent solution.

When both the managed element and the TMA are properly set up, the experiment is started. The scenario consists of varying the demand of requests to the application (requests per second - rps) for 30 minutes. Each slot runs for three minutes, and the demands of each slot vary.

During the phases of the experiment, two scores were calculated: *i)* the *Performance* Score, and *ii)* the *Resource Consumption per Pod* Score. They are used to decide when scale up or scale down actions are needed. The Performance QM was detailed in the Section 3.2. The Resource Consumption per Pod QM, which uses the probe probe-k8s-metrics-server, is not fully detailed due to space constraints. The formulas to calculate the score are listed below:

$A1 = cpu_pod/cpu_node$

$A2 = memory_pod/memory_node$

$score = (0.65 * A1 + 0.35 * A2)/pod_count$

The values *cpu_node* and *memory_node* are constants known in advance. All the others are obtained through the probe called probe-k8s-metrics-server.

5 RESULTS AND DISCUSSION

Figure 5 (a) shows the run chart of the scores when no adaptation mechanism is enabled. The *performance score* is presented in red, as well as their partial sub-scores (dashed lines). Between instant 105 and 160, the performance score drops, as one replica cannot handle a load of 2,500 rps. The blue line presents the *resource consumption per pod* score, which varies based on both the CPU usage and memory. If the data are analyzed, the score is linearly related to the CPU usage, which is influenced by the demand (rps).

Different from the previous experiment, Kubernetes HPA [10] is used as an adaptation approach, and one chart is shown on Figure 5 (b). As it can be seen on the *performance score*, it does not vary so much, although there are some peaks. Throughout this experiment, a new pod is created. However, even when the demand decreases, and when no demand is present, the number of pods remains equals to two. Regarding the *resource consumption per pod score* line, there is a peak around instant 61. This is related to the scaling promoted by HPA, which happened some instants earlier. When a new pod is created, both CPU usage and memory are high. Consequently, the score increases. When the pod set up stabilizes, the values also go back to normal.

The remaining run charts show the results when the adaptation is dispatched by the platform. Every time a score calculated through a QM is either above an upper threshold or below a lower threshold, an adaptation plan is created. The adaptation plan is executed by the Execute component, which invokes the kubernetes-actuator. It is first invoked when the score exceeds the upper threshold.

Figure 5 (c) shows the scores during an experiment execution using the performance QM, and the peaks on the *resource consumption per pod score* chart represent the creation of the new pods. As the *performance score* is being used to decide about the adaptation actions, it varies more during the experiment.

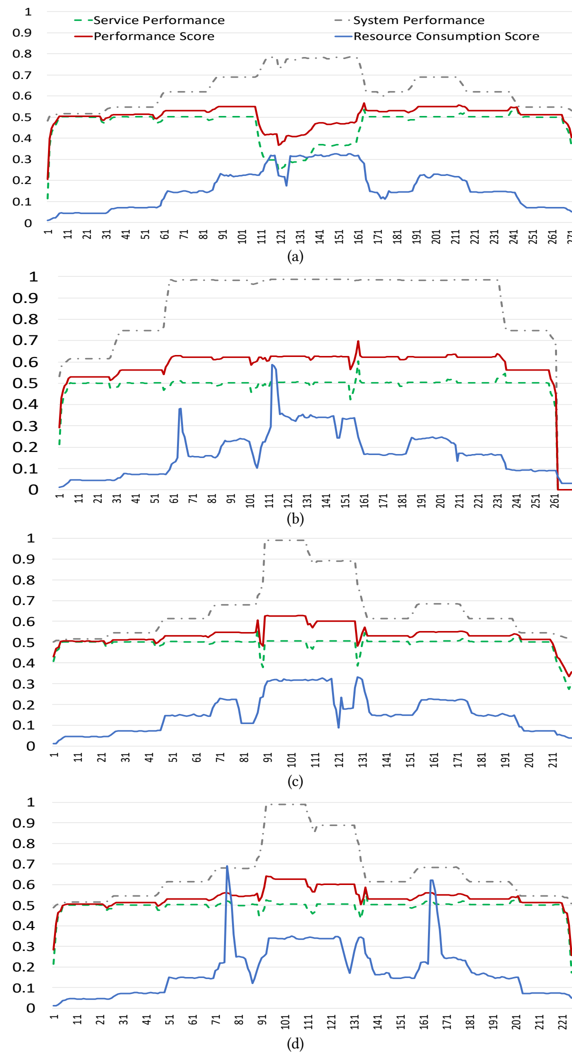

Figure 5: Charts of the Resource Consumption per pod and Performance Scores during one experiment of the TPC Benchmark usage scenario. **(a) without adaptations; (b) with adaptations dispatched by HPA; (c) adaptations using the Performance QM; (d) adaptations using the Resource Consumption per pod QM.**

The adaptations triggered by the resource consumption per pod QM have similar behavior as the performance QM. When the adaptations are performed using the resource consumption QM, four adaptations are performed (two scale up, two scale down). Figure 5 (d) shows the scores during one execution. The two peaks on the *res. consump. per pod score* chart represent the creation of the new pods. Different from the adaptation with HPA, the scale down happens. Consequently, the scores fall when no demand is being performed and the resources are not allocated when they are idle. Also, the *performance score* does not vary so much, but there are some peaks during the higher load of the experiment.

The Table 1 shows the average response time (top of the cell) and the average throughput (bottom of the cell) per slot configuration. The stars (*) represent the slots when an adaptation happens (either scale up or scale down). It can be noticed that only one adaptation happens in the HPA configuration, while the adaptations using the platform scale up and scale down during the experiment.

The response time values show that the mean is higher when no adaption is performed. In configurations B, C, and D, the response time is lower, because a new replica is created, which means that the requests are distributed by the two replicas. This decrease can be observed with more impact in slots V and VI when the response time decreases about 2.5 milliseconds.

The throughput values show that the mean is higher in configurations that automatically creates a new replica of the service. With two replicas, the service can handle loads of slots V and VI (2,500 and 2,000rps respectively), which increases the mean in the configurations B, C, and D in comparison with configuration A.

Table 1: Average response time and throughput of each configuration (A - No Adaptation, B - HPA, C - Performance, D - Resource Consumption)

		Response time (ms) / Throughput (rps)			
Slot	**Demand**	**A**	**B**	**C**	**D**
I	150	4.64 149.94	4.77 149.97	4.59 149.97	4.58 149.97
II	300	4.37 299.82	4.42 299.89	4.50 299.53	4.54 299.62
III	650	4.68 649.62	5.22 649.92 *	4.85 649.02	4.85 649.93
IV	1000	5.40 999.59	6.37 998.30	6.18 998.19 *	5.89 996.10 *
V	2500	6.40 1403.63	3.61 2499.71	2.88 2499.96	3.21 2499.51
VI	2000	6.38 1451.96	3.81 1996.14	2.69 1999.2	3.37 1999.13
VII	650	5.95 649.03	4.75 649.94	5.04 649.94 *	5.04 649.93 *
VIII	1000	5.39 998.81	5.32 999.62	4.87 999.82 *	4.82 999.09 *
IX	650	4.69 649.94	4.67 649.96	4.93 649.92 *	4.86 649.95 *
X	300	4.37 299.66	4.42 299.87	4.58 298.93	4.45 299.82
Average	920 (100.0%)	5.23 755.20 (82.1%)	4.73 919.33 (99.9%)	4.51 919.45 (99.9%)	4.56 919.31 (99.9%)

6 CONCLUSION AND FUTURE WORK

A usage scenario of self-adaptation in a cloud application was presented. It is supported by a trustworthiness TMA-platform, which consists of an assessment lifecycle, a monitoring platform, and measurement instruments (probes) and adaptation services, and it allows defining a trustworthiness level for the cloud applications. Quality models (QMs), which are defined using attributes, weights, operators, and thresholds, are used to portray the trustworthiness level of a cloud application. The attributes reflect the measurement of trustworthiness sub-properties. As future work, we plan to develop other trustworthiness QM and allow the user to add new models according to the need. We also plan to create design-time scores to be integrated into the Continuous Integration (CI) workflow.

ACKNOWLEDGMENTS

This work was partially supported by the project **ATMOSPHERE**, funded by the Brazilian Ministry of Science, Technology and Innovation (Project 51119 - MCTI/RNP 4th Coordinated Call) and by the European Commission under the Cooperation Programme, H2020 grant agreement no. 777154, and project **METRICS**, funded by the FCT – agreement no `POCI-01-0145-FEDER-032504`.

REFERENCES

[1] Amazon. 2009. AWS Auto Scaling. (2009). https://aws.amazon.com/autoscaling/ Accessed: 2018-12-01.

[2] Jesper Andersson, Luciano Baresi, Nelly Bencomo, Rogério Rogerio De Lemos, Alessandra Gorla, Paola Inverardi, and Thomas Vogel. 2013. Software Engineering Processes for Self-Adaptive Systems. *Software Engineering for Self-Adaptive Systems II SE - 3* 7475 (2013), 51–75. https://doi.org/10.1007/978-3-642-35813-5_3

[3] Apache. 2009. Apache Mesos. (2009). http://mesos.apache.org Accessed: 2018-11-29.

[4] Apache. 2011. Apache Kafka. (2011). https://kafka.apache.org Accessed: 2019-02-08.

[5] Tania Basso, Hebert Silva, Leonardo Montecchi, Breno de França, and Regina Moraes. 2019. Towards trustworthy cloud service selection: monitoring and assessing data privacy. In *Anais do XX Workshop de Testes e Tolerância a Falhas*. SBC, Porto Alegre, RS, Brasil, 6–19. https://doi.org/10.5753/wtf.2019.7711

[6] Miguel Caballer, Ignacio Blanquer, Germán Moltó, and Carlos de Alfonso. 2015. Dynamic Management of Virtual Infrastructures. *Journal of Grid Computing* 13, 1 (01 Mar 2015), 53–70. https://doi.org/10.1007/s10723-014-9296-5

[7] Docker. 2017. Swarm Mode Overview. (2017). https://docs.docker.com/engine/swarm/ Accessed: 2018-11-29.

[8] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure Rainbow: Architecture- Based Self-Adaptation with Reusable Infrastructure. *Computer* 37, 10 (2004), 46–54. https://doi.org/10.1109/MC.2004.175

[9] Google. 2014. Kubernetes - Production-Grade Container Orchestration. (2014). https://kubernetes.io Accessed: 2018-11-29.

[10] Google. 2014. Kubernetes Horizontal Pod Autoscaler. (2014). https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ Accessed: 2018-12-01.

[11] IBM. 2006. An architectural blueprint for autonomic computing. *IBM White Paper* 31 (2006), 1–6.

[12] Christian Krupitzer, Felix Maximilian Roth, Sebastian Vansyckel, Gregor Schiele, and Christian Becker. 2015. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing* 17, PB (Feb. 2015), 184–206. https://doi.org/10.1016/j.pmcj.2014.09.009

[13] James Lewis and Martin Fowler. 2014. Microservices. (2014). https://martinfowler.com/articles/microservices.html Accessed: 2018-11-09.

[14] Frank D. Macías-Escrivá, Rodolfo Haber, Raul del Toro, and Vicente Hernandez. 2013. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications* 40, 18 (2013), 7267 – 7279. https://doi.org/10.1016/j.eswa.2013.07.033

[15] Nádia Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. 2017. Towards an Approach for Trustworthiness Assessment of Software as a Service. In *2017 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 220–223. https://doi.org/10.1109/IEEE.EDGE.2017.39

[16] Peter Mell and Timothy Grance. 2011. The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology. *National Institute of Standards and Technology, Information Technology Laboratory* 145 (2011), 7. https://doi.org/10.1136/emj.2010.096966 arXiv:2305-0543

[17] Nazila Mohammadi, Sachar Paulus, Mohamed Bishr, Andreas Metzger, Holger Könnecke, Sandro Hartenstein, and Klaus Pohl. 2013. An Analysis of Software Quality Attributes and their Contribution to Trustworthiness. In *CLOSER 2013 - Proceedings of the 3rd International Conference on Cloud Computing and Services Science, Aachen, Germany, 8-10 May, 2013*. SciTePress, 542–552.

[18] C. Pahl and B. Lee. 2015. Containers and Clusters for Edge Cloud Architectures – A Technology Review. In *2015 3rd International Conference on Future Internet of Things and Cloud*. 379–386. https://doi.org/10.1109/FiCloud.2015.35

[19] RedHat. 2011. Drools - Business Rules Management System. (2011). www.drools.org Accessed: 2020-01-20.

[20] RedHat. 2014. Wildfly. (2014). http://www.wildfly.org/ Accessed: 2019-02-14.

[21] Armin Ronacher. 2010. Flask - Web development, one drop at a time. (2010). http://flask.pocoo.org/ Accessed: 2019-02-08.

[22] TPC. 2014. TPC-App (Application Server) Standard Specification, Version 1.3. (2014). http://www.tpc.org/tpc_app/ Accessed: 2019-01-23.

[23] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, USA, 307–320.