Incremental Learning from Multi-Level Monitoring Data and its Application to Component Based Software Engineering

Salman Taherizadeh, Vlado Stankovski
Faculty of Civil and Geodetic Engineering
University of Ljubljana
Ljubljana, Slovenia
vlado.stankovski@fgg.uni-lj.si

***Abstract*—Many new Internet of Things (IoT) applications such a disaster early warning systems, video-streaming, automated driving and similar, are increasingly being built by using advanced component based software engineering approaches. Software components can include various executable images, such as container or Virtual Machine images, scripts and others. Achieving adequate Quality of Service (QoS) for such applications is still a challenging issue due to runtime variations in running conditions intrinsic to the cloud, edge and fog environments. These types of systems should therefore be continuously monitored and hence adapted at various levels including infrastructure, container and application levels. In this work, we present an adaptation method using a new Incremental Learning approach based on Multi-Level Monitoring data. The method dynamically generates a set of rules representing a performance prediction model that allow us to find potential performance bottlenecks and then propose suitable application adaptation actions. Adaptation possibilities in this work include (1) live-migration of application components (such as containers) from the current infrastructure to another one with different characteristics, such as CPU, memory, disk or bandwidth capacity, and (2) dynamic horizontal or vertical scaling of container-based application instances to offer better fitted resource capacities.**

Keywords—software engineering; components; multi-level monitoring; incremental learning; adaptation

I. INTRODUCTION

Internet of Things (IoT) is a paradigm where things/objects/sensors have a pervasive presence in the Internet. In recent years, IoT systems, such as early warning systems, robots, automated cars and similar, are increasingly used by organizations that want to look beyond traditional applications. The cloud computing model is a pay-per-use on-demand offer through which organizations can exploit elastic cloud resources and federated cloud environments. As IoT applications can be virtualized, cloud computing has become a preferable solution for delivery of such applications.

Achieving high Quality of Service (QoS) and high Quality of Experience (QoE) needed by cloud-based IoT applications is still a challenging task. Many times end-user requirements for high QoE translate to requirements for high-performance computing, for example, requirements for the use of high-speed processors, memory and bandwidth between the devices and the processing software components running in the cloud.

Offering favorable application performance as service quality is still a challenging task particularly due to frequent inability to dynamically obtain necessary computing resources required by the application, and highly varying environmental conditions that can affect the application's performance during runtime. Accordingly, the next generation of IoT systems and applications, would need to be highly self-adaptive – they will need to be implemented in a way that does not require human intervention in their operation [1]. In other words they should be able to detect runtime environmental changes and determine their own way of reacting in order to continuously adapt the deployed services for optimal performance.

Nowadays, a popular method for the delivery of cloud applications is via component based software engineering with tools such as Juju and Fabric8 and containers, such as Docker. Due to the lightweight nature of containers and their fast boot time, it is possible to deploy cloud-based IoT applications in various hosting environments faster and more efficiently than using Virtual Machines (VMs) [2].

This research work presents a continuous multi-level monitoring for software components, such as containers running in the cloud. The generated data are fed as input to a Learning Classifier System (LCS) that incrementally learns rules representing a prediction model for the application's QoS. The rules can be used to adapt the application's components during runtime and thus maintain the expected QoS. The goal is to develop a performance model that is fully supportive to determine performance problems that need corrective actions.

II. METHOD

A. Modern component based software engineering practices

In software engineering, there are new tools gaining popularity due to possibilities to quickly develop cloud applications from scratch. For example, Juju is an open source component based modeling tool for service oriented

architectures and application deployments with a set of predefined components connected together. An application developed in Juju can be vertically and horizontally scaled thus addressing some important properties of the cloud. Fabric8 is another open source platform providing the developer with a console for creating, building and deploying micro services, as well as running and managing them in continuous manner. The SWITCH project also provides its own Interactive Development Environment (IDE) for the development of component based cloud applications.

These new software engineering methods and tools are intended to radically shorten the software lifecycle. By using them, it is possible to build IoT applications based on software components and organize the applications into multiple tiers, which can be made to execute across federated cloud environments. Software components can be persistently running in a data center, at the edge of the network, e.g. in micro-severs or in fog computing devices, such as smartphones, robots, cars and actuators. The present work aims to provide learning mechanisms for the adaptation of component-based IoT applications, so that they can continuously maintain high QoS during their operation in multi-cloud environments.

B. Learning Classifier System

A Learning Classifier System (LCS) is aimed at identifying a set of rules that can be stored and used in a Knowledge Base, such as Jena Fuseki. The rules can be applied in order to make performance predictions based on behavior models. The goal is to develop an adaptive system which is able to learn how to perform specific adaptation actions under specific conditions. The LCS module includes three fundamental components: (I) Environment, (II) Learning machine, and (III) Rule compaction.

(I) Environment: The environment is the source of data upon which the LCS learns. As an example, shown in Table I, the environment could be a dataset with some number of training instances. Each instance has some independent attributes that can traditionally have the value of 0 or 1. Moreover, each instance has a single endpoint referred to action. This endpoint is the main variable in the data that we are trying to predict.

To generate this dataset, an implemented monitoring approach able to measure a wide variety of different attributes in the execution environment is proposed. The objective is to autonomously maintain such applications' performance considered as the endpoint in the dataset and hence deliver seamless user experience in different conditions.

(II) Learning machine: The learning machine iterates over the dataset repeatedly until some stop criteria are met, or the maximum number of learning iterations is reached. As the result, the learning machine generates a set of rules together considered as the prediction model. The model will be used to predict the behavior of application performance as single endpoint based on other attributes.

TABLE I. BINARY LEARNING INSTANCES FOR RULES

Endpoint	Attributes		
	Attribute 1	*Attribute 2*	*Attribute N*
1	0	1	1
0	1	1	0
…	…	…	…
1	0	0	1

(III) Rule compaction: Once the last learning iteration is reached, the resulting rules can be applied as the model. However, there is often a post-processing step called "rule compaction" applied to the resulting model after the last learning iteration. Rule compaction strategies typically seek to remove poor, redundant or inexperienced rules from the prediction model. In this way, rule compaction simplifies the model, improves interpretability, and even can enhance predictive performance.

C. Rule population (P)

Rules typically take the form of an {IF:THEN} expression, e.g. {IF 'condition' THEN 'action'}. An individual rule is not itself a prediction model, since the rule is only applicable when its condition is satisfied. The entire population of rules collectively forms the prediction model. Each attribute in a rule can be 0, 1, or '#' as "don't care" symbol (also referred as wild card). For example, the rule (#1###0### ~ 1) as {condition ~ action} can be interpreted in this way: IF the second attribute = 1 AND the sixth attribute = 0 (regardless of other attributes) THEN the prediction class = 1. In the example, the second and sixth attributes have been specified in this rule, while the others were generalized. A rule along with its associated parameters (such as accuracy, fitness and numerosity) is often referred as a classifier. In Michigan-style LCS as the most common type of LCS algorithm, classifiers are contained within a population ([P]) that has a user defined maximum number of classifiers. The [P] starts out empty (i.e. there is no need to randomly initialize a rule population). Classifiers will instead be initially introduced to [P] with a covering mechanism.

As explained before, the learning machine is the core of an LCS and includes several interacting components that operate in a step-wise learning cycle: (Step 1) Training, (Step 2) Matching, (Step 3) Covering, (Step 4) Updating, (Step 5) Subsuming, (Step 6) Genetic algorithm, (Step 7) Deleting.

(Step 1) Training: The beginning step in incremental learning is getting a training instance from the environment. For online learning, LCS will obtain a completely new training instance for each iteration from the environment.

(Step 2) Matching: The next step is finding all rules in the population [P] that have a condition matching the attributes values of the training instance. In other words, every rule in [P] is now compared to the training instance to see which rules match. A rule matches a training instance if all feature values specified in the rule condition are equivalent to the corresponding feature value in the training instance. For example, assuming the training instance is (001001 ~ 0), these

rules would match: (###0## ~ 0), (00###1 ~ 0), (#01001 ~ 1), but these rules would not (1##### ~ 0), (000##1 ~ 0), (#0#1#0 ~ 1). In matching step, the endpoint (action or prediction class) specified by the rule is not taken into consideration. At the end, matching rules are moved to a match set [M]. As a result, the [M] may contain classifiers that propose conflicting actions. Afterwards, since we are performing supervised learning, [M] will be divided into a correct set [C] and an incorrect set [I]. A matching rule goes into the [C] if it proposes the correct action (based on the known action of the training instance), otherwise it goes into [I]. At this point, if no rule has been made into either [M] or [C], then the covering step will be applied.

(Step 3) Covering (as rule discovery): Covering is one of two mechanisms that can introduce new rules to [P] also known as "rule discovery". Covering randomly generates a rule that matches the current training instance. It works by generating a rule condition which randomly specifies a subset of attribute values in the current training instance, and applies wild cards ('#') to the rest. The action or prediction class for the rule is set to the class of the current training instance. Assuming the training instance is (001001 ~ 0), covering might generate any of the following rules: (#0#0## ~ 0), (001001 ~ 0), (#010## ~ 0). Covering step not only ensures that during each learning cycle there is at least one correct, matching rule in [C], but also any rule initialized into the population [P] will match at least one training instance. As mentioned before, [P] typically starts off empty. Because of this, covering step serves as a form of smart population initialization.

(Step 4) Updating: Parameters of any rule in [M] are updated to reflect the new experience gained from the current training instance. For example, we can simply update the accuracy of a rule. Rule accuracy is calculated by dividing the number of times the rule was in the correct set [C] by the number of times it was in the match set [M]. Rule fitness is also updated in this step, and is commonly calculated as a power function based on the inverse of rule accuracy. Numerosity of a classifier means the number of copies of this classifier in the population [P] (if there are multiple copies). Classifiers in the correct set [C] will see an increase in both accuracy as well as fitness. Classifiers in the incorrect set [I] will see a decrease in the accuracy and fitness.

(Step 5) Subsuming: In particular, rules that specify fewer attributes are likely to appear in match set [M] more frequently. Subsuming step is a generalization mechanism that merges classifiers that cover redundant parts of the problem space. In this way, it helps to decrease the size of population set [P] by subsuming a classifier to a more general classifier (and its numerosity has been increased). In other words, the subsuming step examines pairs of rules and looks for a situation in which one of the rules is a subsumer of another one. For example, rule (#####0 ~ 0) is a subsumer of (##1#00 ~ 0). A subsumer rule must cover all of the problem space of another rule, and must be more general and accurate while the more specific rule is eliminated from the population [P].

(Step 6) Genetic algorithm (as rule discovery): This step applies a simple Genetic Algorithm (GA) as the second type of rule discovery mechanisms. While other heuristics could be used to discover rules, the GA is most commonly used. In fact, only two new 'offspring' rules are typically generated by the GA and added to the rule population [P] during each learning cycle.

(Step 7) Deleting: The last step in the LCS learning cycle is to enforce the limited size of the rule population using deletion in order to maintain the maximum population size. The probability of a classifier being selected for deletion is inversely proportional to its fitness. Other factors such as the classifier's numerosity can be applied to increase the probability of deletion (e.g. numerosity divided by fitness). This keeps [P] from being overrun by just a few rules with large numerosities. When a classifier is selected for deletion, its numerosity parameter is reduced by one. When the numerosity of a classifier is reduced to zero, it is removed entirely from the population [P].

D. Prediction

Whether or not rule compaction has been applied, the output of an LCS algorithm is a population of classifiers which can be applied to making predictions on unseen instances as testing data. During the prediction step, the population [P] does not continue to learn from incoming testing data. A test instance is passed to [P] where a match set [M] is formed as usual. Rules in the match set [M] can predict different actions. Therefore a voting scheme will be applied in this situation.

In a simple voting scheme, the action (prediction class) with the strongest supporting "votes" from matching rules wins, and becomes the selected prediction. The strength of the vote for a single rule (a single classifier) is commonly proportional to its numerosity and fitness (Classifier Vote = Numerosity * Fitness). Or another simple voting scheme can be to sum the votes of all rules proposing the same prediction class and chose the class with the highest overall vote.

III. INTERNET OF THINGS APPLICATIONS

A typical example for IoT services is represented in a layered architecture consisting of Sensors (e.g. DHT11), Remote Terminal Units (e.g. Modbus RTU), IP Gateways (e.g. TA900e or Cisco-ASA), Database Server (Apache Cassandra server), Contact Centre Server (Apache Web servers) and perhaps, Call Operators (dedicated and ad-hoc agents).

The Call Operators decide whether or not to send an alert to emergency systems or to the public entities. The Call Centre server checks sensed data stored in Database Server and statistics in real-time and sends notifications (such as e-mail, text messages or voice call via SIP based IP telephony or ordinary PSTN) to Call Operators, if values are outside predetermined thresholds for sensors. The Database Server is a Time Series Database which is used for storing and handling sensed values indexed by time. The IP Gateway is a node that allows communication between networks. It receives data over direct radio link or GSM/GPRS from sensors, aggregates the data and sends the data to the database. Remote terminal units (RTUs) connect to sensors in the process and convert sensor signals to digital data. Sensors can measure temperature, barometric pressure, humidity and other environmental variables.

Sensors, RTU and IP Gateway cannot be virtualized as these components have physical items like attached antennas. In this research work, the CC Server and DB Server are software components that have been implemented by using Docker containers and can thus be deployed in a federated cloud environment.

IV. Multi-Level Monitoring

A multi-level monitoring tool was developed to monitor the application's execution environment at infrastructure-level, which includes VM-level monitoring and end-to-end link quality monitoring, at container-level and at application-level.

JCatascopia [3] has been chosen as baseline technology and extended. In comparison with JCatascopia, the new system concentrates on end-to-end network monitoring, including the collection of metrics such as Packet Loss, Throughput, Average Delay and Jitter. See Table II for more details.

Our monitoring system uses an agent-based client-server approach which is able to support a fully interoperable, highly scalable and light-weight architecture. The distributed nature of this monitoring framework quenches the runtime overhead of system to a number of Monitoring Agents running across different cloud resources.

The system offers a framework to measure, store and report monitoring metrics from different layers of the underlying cloud infrastructure, as well as possible overall performance metrics from the deployed application.

TABLE II. Monitoring Metrics

Level	Metric	Description
Call Centre Server	requestCount	Number of requests served since last collection
	processingTime	Average request processing time
	requestThroughput	Rate at which requests are processed
Database Server	readLatency	Keyspace read latency
	writeLatency	Keyspace write latency
Container-level	rx_bytes	Bytes received
	rx_packets	Packets received
	tx_bytes	Bytes sent
	tx_packets	Packets sent
	cpu_usage	%CPU usage of container
	memory_usage	%Memory usage of container
	blkio_io_bytes_read	Bytes read from hard disk
	blkio_io_bytes_write	Bytes written to hard disk
Infrastructure	cpuUsedPercent	%CPU utilization of VM
	memUsed	Current memory usage of VM
	memUsedPercent	%Memory usage of VM
	diskFree	Amount of available disk capacity
	diskUsed	Amount of used disk capacity
	netPacketsIn	Packets in per second
	netPacketsOut	Packets out per second
	netBytesIn	Bytes in per second
	netBytesOut	Bytes out per second

V. Using the LCS Environment

According to the LCS algorithm, the environment can be considered as dataset upon which the learning machine can make the prediction model. For an IoT application, the preparation of this environment has four different pre-processing steps: (I) Producing monitoring data, (II) Averaging monitoring data, (III) Re-formatting monitoring data, and (IV) Converting monitoring data from numeric to binary format.

(I) Producing monitoring data: To enhance the performance of an application component (e.g. Database Server), one adaptation action can be adding more instances of this component to the pool of servers so that load can be spread across multiple instances for one application component. In a table of collected metrics, rows belong to metrics measured periodically in different times for all Database Servers, all Call Centre Servers and all end-to-end links. The value of metrics can be Long, Double or Float.

(II) Averaging monitoring data: Our solution periodically measures the average value for all metrics at each time, e.g. the average read and write latency of all DB Servers. In this way, for each time period, there exist average values of all metrics for application components, not for individual instances.

(III) Re-formatting monitoring data: Each row in the original format of monitoring data is a measurement record for one metric. The original format needs to be modified since it should be consistent with the environment usable by the LCS algorithm. In this step of preparation procedure, all measurements belonging to the same time interval have been gathered together as one row in the new format. According to the new structure, the overall application performance is the prediction class (endpoint), which we need to enhance as the main goal and hence deliver the result as early as possible for the best real-time user experience.

(IV) Converting monitoring data from numeric to binary format: Machine learning approaches have their own intrinsic limitations mainly as computational complexity, time-consuming process and requirements of large data that impact on their usefulness to function in real-world time-critical use cases. To come up with these challenges, in our method, numeric values of all features stored in the reformatted monitoring data have been converted to binary values. Therefore, in our proposed LCS algorithm, all rows representing states of the environment and consequently rules include attributes that have the binary value. To this end, it is possible to define a threshold for every single monitoring metric in different level as well as for the overall application performance. For instance, the threshold for average CPU usage for each application component in the infrastructure level can be considered 80 percent. Then, if the average CPU utilization is less than 80 percent, it has been converted to 0 and on the other hand, if it is over 80 percent, it has been converted to 1.

VI. ADAPTIVE ARCHITECTURE

The adaptive architecture includes various entities when the application executes. These entities shown in Fig. 1 operate as follows.

(I)The purpose of Monitoring Probes/Agents is to collect data that represents the current state of managed elements (application, container and infrastructure), and then aggregate and transfer the measured values to the Monitoring Server and the Alarm Trigger. The monitored metrics differ from application to application. The Monitoring Probes/Agents should be non-intrusiveness [4], scalable [5], robust [6], interoperable [7] and able to support live-migration [8] as the essential non-functional monitoring requirements.

(II)The Monitoring Server receives the collected data and stores it in a TSDB to build a focused and comprehensive representation of the system state. The TSDB can be implemented by Apache Cassandra technology which is a distributed storage system for managing very large amounts of time-ordered data [9]. The LCS Module also gradually generates and then updates the prediction model which is a set of rules. These rules are usable to define in which conditions the overall application performance meets user requirements. Concurrently, the Alarm Trigger investigates if the measured value of monitored overall application performance overpasses predefined limit. In other words, The Alarm Trigger is a component which processes the incoming monitoring data streams and notifies the Self-Adapter when predefined threshold for the overall application performance is violated. The Monitoring Server, the Alarm Trigger and the LCS Module should be tightly coupled, i.e. running on the same machine in order to save network bandwidth and computational resources needed for data distribution and processing.

(III)When problems are detected, the Self-Adapter is invoked to propose suitable adaptation actions based on the prediction model generated by the LCS Module. This component is able to automatically identify metrics (e.g. CPU or memory) that are the most useful parameters to be changed for the overall application performance. The Self-Adapter specifies a set of adaptation actions for the Control Agent allowing the passage of the whole system from a current state to a desired state. It means that the Self-Adapter reasons which adaptation changes should be done to adapt the system to the desired behavior. Adaptation possibilities can be horizontal or vertical scaling of DB Servers and CC server, or dynamically live-service migration by moving running containers from the current infrastructure to another one either at the same data center or at a different cloud to offer better fitted computational and/or memory capacity.

(IV)The Control Agent, which has the full control of application configurations and infrastructure resources, e.g. VMs/containers, CPU, disk and network bandwidth, finally carries out the adaptation actions defined by the Self-Adapter. In addition to the live-migration, this component is able to increase or decrease the required number of containerized application components (DB Servers and CC servers) providing the service on demand even in different cloud data centers. It is also capable of resizing the CPU resource, memory capacity, disk storage or network bandwidth.

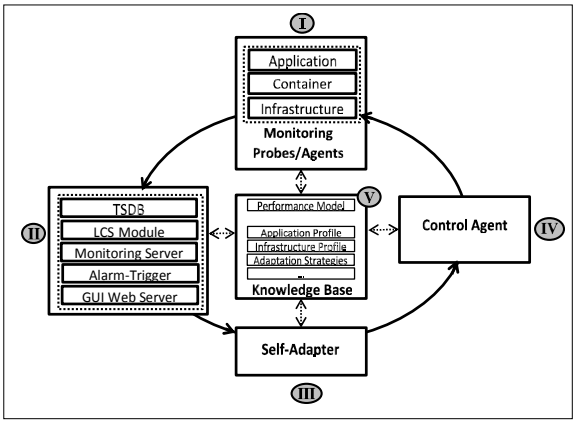

Fig. 1. Adaptive architecture for component based IoT applications

(V)The Knowledge Base will be used to store the prediction model and also all information about the current system metadata, awareness and application configuration for analysis, reuse, reasoning, optimization and refinement of design, topology and execution. The knowledge stored in this component describes profiles of all entities (e.g. application profile, infrastructure profile, performance profile, adaptation strategies, etc.), and it is used to interpret monitoring data [10].

VII. GENERATED RULES

Here, we provide few examples of rules that can be generated by the LCS Module. These rules represent the performance model of the component based cloud-application. Following is an example of a generated rule:

(####################0##########0#####0#### ~ 0) *(rule 1)*

In this rule, the first attribute with the value of 0 is "cpuUsedPercent" at infrastructure level for CC Server (Apache Tomcat), the second and third attributes (defined as 0) are "memUsedPercent" and "diskUsed" also at infrastructure level for DB Server (Cassandra). Other attributes have been set as "don't care" (#). This rule can be interpreted in this way: If the average CPU utilization of the host(s) on which CC Server(s) is(are) running does not exceed its threshold, also if the average memory usage and the amount of used disk capacity of the host(s) on which DB Server(s) is(are) running do not violate their thresholds, the overall application performance of early warning system is acceptable (0) considering users' satisfaction. Therefore, in situations when the overall application performance is not favorable, the Self-Adapter should investigate these three metrics to define if they are violated.

For instance, if "memUsedPercent" and "diskUsed" for DB Server are not presenting a problem, however "cpuUsedPercent" for CC Server is inappropriately very high, regardless of other monitoring attributes, the Self-Adapter suggests increasing the CPU power of the existing virtual machine(s) on which CC Server(s) is(are) running. In this situation, if vertical scaling is not feasible for example since maximum CPU capacity is already reached, the proposed

adaptation plan could be other approaches e.g. live-service migration by moving running CC Server container(s) from the current infrastructure to another one either at the same data center or at a different cloud to offer better fitted computational resources.

As another example, the following rule can be also inferred by the LCS Module as one of the rules in the performance model:

(#1## ~ 1) *(rule 2)*

In this rule, the attribute with the value of 1 is "processingTime" at application level for CC Server (Apache Tomcat). Other attributes have been also defined as "don't care" (#). This rule can be interpreted in this way: If the average response time of CC Server component is unsuitable since it is more than associated threshold, the overall application performance is not acceptable (1). Therefore, in situations when the overall application performance is not appropriate, the Self-Adapter should also consider this metric ("processingTime") to determine, if it is violated. For instance, if it is over the threshold, regardless of other monitoring attributes, the Self-Adapter proposes to dynamically increase the number of running containerized CC Server in order to enhance the overall application performance.

VIII. CONCLUSION

Our presented adaptation method uses a multi-level monitoring system since the adaption of applications should be tuned and handled at various levels of cloud environments—infrastructure, container and application. This work introduced a rule-based adaptation method using the LCS algorithm to automatically adapt the application performance to changing conditions at runtime. In order to achieve this particular aim, the proposed LCS algorithm creates the performance model which is a set of rules. These rules are able to identify any monitored attributes that need corrective adaptation actions. In this way, preventing and predicting potential application performance drops will give more time to take action like horizontal or vertical scaling of application components, or dynamically live-service migration by moving from the current infrastructure to another one either at the same data center or at a different cloud to offer better fitted computational, memory, disk or network capacity.

ACKNOWLEDGMENT

The research and development presented in this article have received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreements No. 643963 (SWITCH project: Software Workbench for Interactive, Time Critical and Highly self- adaptive cloud applications) and No. 644179 (ENTICE project: dEcentralized repositories for traNsparent and efficienT vIrtual maChine opErations).

REFERENCES

[1] M. Koprivica. 2013, "Self-adaptive requirements-aware intelligent things," International Journal of Internet of Things vol. 2 (1), 2013, 4 pages, DOI:10.5923/j.ijit.20130201.01

[2] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder, "Docker containers across multiple clouds and data centers," in: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), 2015, pp. 368–371.

[3] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud," in Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, Chicago, 2014, pp. 226-235.

[4] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, P. Martin, and V. Stankovski, "Runtime network-level monitoring framework in the adaptation of distributed time-critical cloud applications," in Proceedings of the 22nd International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'16), Las Vegas, 2016, 6 pages.

[5] S. Clayman, A. Galis, and L. Mamatas, "Monitoring virtual networks with lattice," in Proceedings of 2011 IEEE/IFIP Network Operations and Management Symposium Workshops (NOMS Wksps), IEEE, 2011, pp. 239-246.

[6] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, "A survey of cloud monitoring tools: Taxonomy, capabilities and objectives," Journal of Parallel and Distributed Computing, 74 (10), 2014, pp. 2918-2933.

[7] K. Alhamazani, R. Ranjan, K. Mitra, F. Rabhi, P. P. Jayaraman, S. U. Khan, A. Guabtni, and V. Bhatnagar, "An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art," Computing, 97(4), 2015, pp. 357-377.

[8] A. Nadjaran-Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments: Challenges, taxonomy, and survey," ACM Computing Surveys (CSUR) 47 (1), 2014, 47 p.

[9] D. Namiot, "Time Series Databases," in Proceedings of the XVII International Conference Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL'2015), Russia, 2015, pp. 132-137.

[10] F. Zablith, G. Antoniou, M. d'Aquin, G. Flouris, H. Kondylakis, E. Motta, D. Plexousakis, and M. Sabou, "Ontology evolution: a process-centric survey," The Knowledge Engineering Review, 30 (1), 2015, pp. 45-75.