Environment. Technology. Resources, Rezekne, Latvia
Proceedings of the 11th International Scientific and Practical Conference. Volume II, 52-57

Auto-scaling and Adjustment Platform for Cloud-based Systems

Jānis Kampars, Krišjānis Pinka
Riga Technical University, Institute of Information Technology. Address: Sētas iela 1, Riga, LV-1048, Latvia

Abstract. For customers of cloud-computing platforms it is important to minimize the infrastructure footprint and associated costs while providing required levels of Quality of Service (QoS) and Quality of Experience (QoE) dictated by the Service Level Agreement (SLA). To assist with that cloud service providers are offering: (1) horizontal resource scaling through provisioning and destruction of virtual machines and containers, (2) vertical scaling through changing the capacity of individual cloud nodes. Existing scaling solutions mostly concentrate on low-level metrics like CPU load and memory consumption which doesn't always correlate with the level of SLA conformity. Such technical measures should be preprocessed and viewed from a higher level of abstraction. Application level metrics should also be considered when deciding upon scaling the cloud-based solution. Existing scaling platforms are mostly proprietary technologies owned by cloud service providers themselves or by third parties and offered as Software as a Service. Enterprise applications could span infrastructures of multiple public and private clouds, dictating that the auto-scaling solution should not be isolated inside a single cloud infrastructure.

The goal of this paper is to address the challenges above by presenting the architecture of Auto-scaling and Adjustment Platform for Cloud-based Systems (ASAPCS). It is based on open-source technologies and supports integration of various low and high level performance metrics, providing higher levels of abstraction for design of scaling algorithms. ASAPCS can be used with any cloud service provider and guarantees that move from one cloud platform to another will not result in complete redesign of the scaling algorithm. ASAPCS itself is horizontally scalable and can process large amounts of real-time data which is particularly important for applications developed following the microservices architectural style. ASAPCS approaches the scaling problem in a nonstandard way by considering real-time adjustments of the application logic to be part of the scalability strategy if it can result in performance improvements.

Keywords: cloud computing, auto-scaling, microservices, big data.

I. INTRODUCTION

Cloud computing platforms provide a virtually unlimited pool of computing power and storage resources for hosting various enterprise applications. Customers of such platforms are charged by the amount of resources they have used on a pay-per-use basis. Therefore, it is important to minimize the infrastructure footprint and balance it with the required level of Quality of Service (QoS) and Quality of Experience (QoE). Usually this is achieved by scaling the application up during peak times and removing the surplus of resources later. To assist with that cloud service providers are offering Application Programmable Interfaces (APIs) that support:

1. horizontal resource scaling by changing the number of currently running virtual machines or containers (e.g. add two more web-server nodes),
2. vertical resource scaling by changing the capacity of individual cloud hosted nodes (e.g. increase the RAM by 2GB and add 2 CPU cores).

Existing scaling solutions mostly concentrate on low-level metrics like CPU load and memory consumption which doesn't always reflect the actual QoE. Such metrics should be preprocessed and joined with high level application metrics when deciding whether the cloud based solution should be scaled up or down. Most of currently available scaling platforms are proprietary technologies owned by cloud service providers or third parties. Even if the enterprise application is designed in a platform-independent way, changing the cloud service provider might result in redesign of the applied scaling algorithm. Complex enterprise systems could also be hosted in multiple clouds, both private and public, dictating that the auto-scaling solution should not be isolated inside a single cloud platform. It could span multiple public clouds and some parts of it could be deployed on premises. The chosen deployment model should not be dictated by the limitations of the auto-scaling platform; it should be the choice of the software architect. Although there are existing specialized scaling platforms that are cloud-independent, they are proprietary technologies that come with the risk of vendor lock-in. Their extendibility is limited due to the closed source.

Design of such platforms is challenging because of the scalability requirements and the amount of real-time data that needs to be integrated and

ISSN 1691-5402

http://dx.doi.org/10.17770/etr2017vol2.2591

processed for decision-making purposes. This can be even more complex for applications designed in microservices architectural style, since each service acts as a data source with one or more measurable properties.

The goal of this research is development of the Auto-scaling and Adjustment Platform for Cloud-based Systems (ASAPCS) that addresses the challenges described above.

The structure of this work is as follows. Section II gives a brief look at the related work in scalability, microservices and concepts of Capability Driven Development (CDD). Section III defines the requirements for the auto-scaling platform. Section IV describes the overall architecture of the platform in a technology independent manner while Section V looks at the technological stack that was used to build the first prototype of the platform. Section VI concludes with final remarks and future work directions. Section VII contains acknowledgments.

II. RELATED WORK

Most of the academic work in the area of scalability is concentrated on scaling algorithms and strategies aiming at maximizing the performance metrics and minimizing the related costs or on architectures that should be applied to ensure that the application would effectively scale. Auto-scaling strategies are categorized as [1]:

- reactive – a scaling operation is performed immediately as soon as performance values have fallen out of a previously defined interval during the last time window,
- conservative – a scaling operation is performed if during the last few time windows performance values have fallen out of a previously defined interval,
- predictive – performance values for the next time window are predicted and acted upon similarly as with the reactive strategy.

The step of the scaling operation can be fixed (e.g. one node at a time) or adaptive (based on the difference between current demand and resource capacity).

There has been little research done on executing scaling algorithms in production environment for large scale applications that require real-time analysis of big amounts of data. When scaling cloud-based systems, the auto-scaling platform must also be made of components that can scale horizontally, ensuring its resilience and scalability [2]. The auto-scaling platform must scale together with the applications or it will become the single point of failure and the bottleneck of the whole solution.

There have been developments in the area of platforms providing auto-scaling, however, most of them lead to vendor lock-in. Notable examples are platforms by the major cloud service providers - AWS Auto-scaling, Microsoft Azure auto scale and Rackspace Auto Scale. These auto-scaling solutions are tied to the underlying cloud computing platforms and services. Therefore, moving from one cloud service provider to another might result in redesign of the scaling solution. There are other options like RightScale and New Relic which are not limited by the use of a single cloud computing platform, however both are proprietary technologies being offered in Software as a Service (SaaS) model and their users have to consider threats arising from the vendor lock-in. Since these are closed-source programs they can't be easily extended. Kubernetes, an open-source platform for orchestrating containers, provides basic auto-scaling capabilities. Similarly, simple auto-scaling scenarios can be implemented on Mesos using Marathon, however this is not the main concern of the platform. In both cases the auto-scaling functionality can only be used for container-based solutions hosted on the specific platforms, which is not suitable for systems spanning multiple environments.

Developments in the area of auto-scaling platforms can be especially beneficial for applications designed following the increasingly popular microservices architectural style. Microservices have become a dominant architectural style choice for service oriented applications [3]. Traditionally cloud based systems are divided into service groups like database, web and application. Those groups consist of homogenous servers that can be scaled independently of other groups. This approach gives a certain level of control and allows to scale up just the web server if there is lack of capacity in this specific tier. In reality this might lead to inefficient use of resources and waste of money. For example - if extra resources are needed just for the video transcoding process, the only option to achieve this might be to scale up the whole application server cluster. In reality this might turn out to be a significant overhead since application servers might be running many other processes which were having no lack of capacity. Microservices, being a cloud-native architecture [4], address this challenge by dividing the system into small and lightweight services that are purposely built to perform a very cohesive business function, and it is considered to be an evolution of the traditional service oriented architecture (SOA) [5, 6, 7]. Scalability is often mentioned as one of the advantages of the microservice architecture and it is quite obvious since in terms of architecture it gives a fine-grained control over how application scales during varying load. Unfortunately, still very little research is done in the area of microservices [3, 8] and even less so on their scalability.

Kukade et al. [9] and Toffetti [2] are among few investigations done in the area of microservice auto-scaling. The work by Kukade et al. [9] is the only literature source covering auto-scaling aspects from ones reviewed in the systematic mapping study done

by Pahl [8]. Yet Kukade [9] only briefly covers the auto-scaling aspects and puts more emphasis on containerization as a suitable technology for implementing microservice architecture. The main parts of their solution are:

- Service Container Monitor – providing container health checks and detecting faulty containers,
- Request Monitor – counting the number of requests each container receives,
- Memory Load Monitor – measuring container's memory consumption,
- Scale – adding extra containers when existing ones have reached the top threshold of used memory and received requests, removing containers when the bottom margin is reached.

Little information is provided about the implementation of auto-scaling solution. Also the number of requests and memory consumption will not always provide a good correlation with SLA and might result in inefficient use of resources, therefore it can be concluded that proposed solution will be useful only in very specific scenarios.

Toffetti et. al [2] propose an architecture for auto-scaling microservices. To specify the relation between various service groups and instances a service type graph and instance graph is used (see Fig. 1).

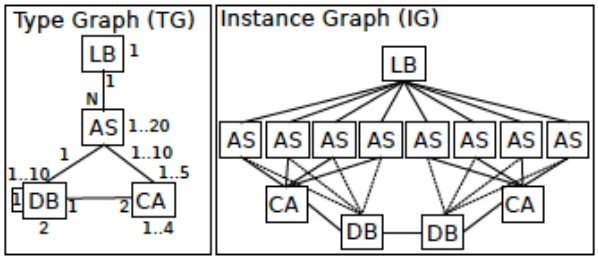

Fig. 1. Type and instance graphs [6]

The type graph defines service groups and their corresponding scaling limits (e.g. each database server, DB, is linked to exactly two caching instances, CA, there are 1 to 5 database server instances, DB) while instance graph keeps track of active instances (e.g. currently 4 database server instances are online). The graph data is stored in etcd, a distributed, consistent key value store. All cloud based components are aware of their type and can discover other nodes by using the etcd directory where they are registered upon deployment. The architecture makes sure that the required number of cloud nodes are online through provisioning of new ones and monitoring their health. Although the proposed solution might work well in some scenarios, it would be hard to implement complicated scaling algorithms and debug them.

The goal of the majority of auto-scaling solutions is achieving the desired QoS levels. Hoßfeld et. al [10] have concluded that QoE as perceived by users has the potential to become the guiding paradigm for managing quality in the cloud. We strongly support this opinion and propose to use some of the concepts of CDD (Capability Driven Development) [11] in scaling microservices, since they are organized around capabilities [5]. CDD aims at capturing the relation between context and capabilities enabled by information systems (IS) and adjusting the IS according to the contextual situation. Capability can be defined as the ability and capacity that enable an enterprise to achieve a business goal in a certain context. The fluctuations in application load could be considered as examples of varying context of the application while scaling of an application is an example of real-time adjustment. CDD provides ways for altering the business logic of the IS based on the contextual situation during run-time. This approach could also be applied in the auto-scaling problem area. Since QoE could be even more important than QoS, run-time adjustments in the business logic of application itself could become part of the auto-scaling strategy allowing to achieve results that wouldn't be possible by using infrastructure level adjustments all alone. Changing internal logic of the application in response to increased load (e.g. data consistency level in the database tier or limiting the free service when requests from paid customers are not served properly) could result in higher number of served requests while ensuring smaller infrastructure footprint.

III. REQUIREMENTS FOR THE AUTO-SCALING PLATFORM

This section lists the requirements for the auto-scaling platform that are largely derived from the previous section and review of related work in the area of scalability, microservices and CDD:

- QoE above QoS – the platform should facilitate constant monitoring of the QoE and QoS,
- Real-time data integration and processing – the platform should be capable of processing and analyzing large amounts of real-time data originating from application and infrastructure nodes,
- Windowing support – the platform should be able to provide basic windowing functionality (e.g. sliding window),
- Scalability and resilience – the platform should be resilient and scale together with the system that is scaled by it,
- Machine learning capabilities – in order to allow predictive scaling of applications platform should provide machine learning functionality,
- Graph processing capabilities – since distributed application is often described by type and instance graphs the platform should be capable of processing them,

- Abstraction – in order to develop the algorithms independently of cloud computing platforms and data sources a level of abstraction should be established,
- Run-time improvements of algorithms – users should be able to alter the behavior of scaling algorithms during run-time,

Open-source – platform should be based on open source technologies thus avoiding risks of vendor lock-in and facilitating extendibility of the platform.

IV. ARCHITECTURE

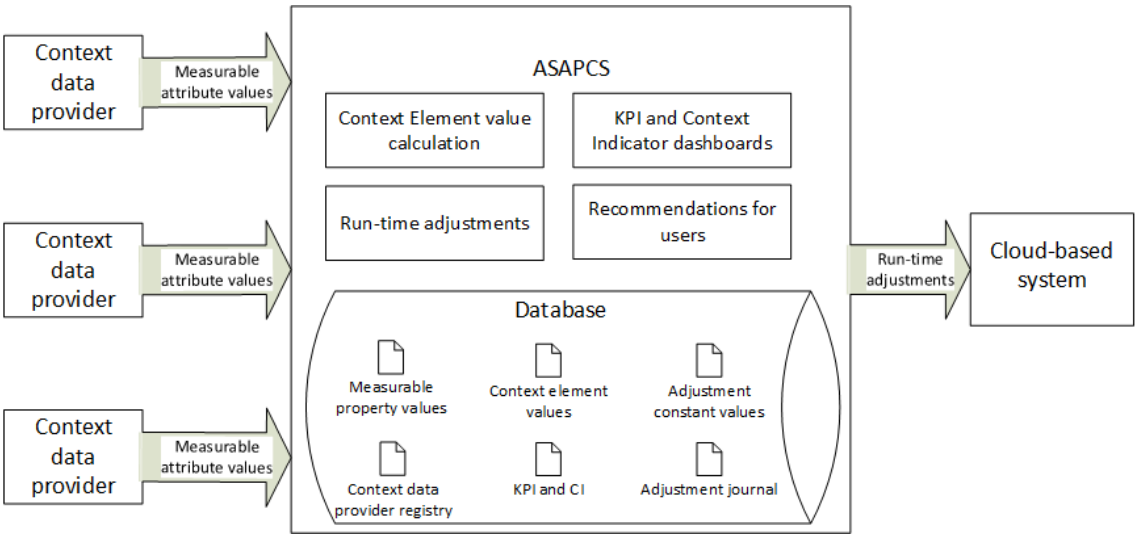

Fig. 2. Overview of the ASAPCS components

The architecture of the ASAPCS is largely inspired by the CDD approach [11] therefore it uses term "context" to address all information that can be used for scaling the application while actual scaling operations are referred to as run-time adjustments. Main components of the ASAPCS are illustrated in Fig. 2 and related concepts are discussed below.

Context data providers (CDP) are entities providing availability of the context data. Examples of CDP are monitoring tools like Zabbix and customized agents providing information about various aspects of the application performance and infrastructure. CDP is responsible for providing information about the current contextual situation. There is no need for the CDP to store historical data since ASAPCS takes care of that, which means that a relatively small effort is needed to create a CDP.

The contextual data provided by CDP is further divided into measurable properties (MP). An example of MP is the current queue length or memory consumption. Generally, MPs are data of low granularity that needs to be preprocessed and aggregated to be used in scaling algorithms.

While changing the granularity of MP it is transformed into a Context element (CE) – entities positioned at higher levels of abstraction that are not directly linked to a single CDP anymore. A typical example of CE could be an average number of visits during last minute with a sliding interval of 5 seconds (CE value is recalculated once in every 5 seconds). CEs could also be created as compound structures consisting of multiple MPs – for example a server load CE measured as high, medium and low could be expressed as a function from memory consumption and CPU usage. In a similar way as MPs, historical values of a CEs are stored in a temporal database of ASAPCS. New CEs can be defined even after the data collection has started. Values of the newly created CEs can be recalculated from the historical values of MPs available in the temporal database.

Another concept that has been derived from the CDD is Adjustment Constant (AC). The value of the AC can be changed during run-time by the user and it can be used to alter the scaling algorithm (for example change the interpretation of what is considered high, medium and low server load).

Run-time adjustments (RTA) are used for ASAPCS initiated scaling operations. Technically it is an in-code defined scaling algorithm that uses values of the AC and CE as input parameters. RTAs are used for transforming manually scalable solutions into ones that scale automatically based on the contextual situation. Thanks to the CE abstraction RTAs are not coupled to the CDPs therefore moving the system from one cloud to another would not result in the complete rewrite of the scaling algorithm. Only actual calls to the cloud platform scaling API might need to be rewritten. In theory these can also be abstracted using interfaces and their platform-specific realizations. Each scaling operation performed by the

run-time adjustment algorithm is logged to the adjustment journal for later reviewing. RTA are triggered by a combination of CE values falling below or exceeding a previously defined threshold. These margins can also be defined using AC allowing to easily change them during run-time.

If a certain contextual situation can't be handled with RTA, a notification can be shown in the user interface of the ASAPCS providing recommendations for further actions. These notifications are also triggered by certain values of CEs.

Key Performance Indicators (KPI) are indicators that have direct effect on the cloud-based system. KPI show how the strategic goals are met and in case of scaling applications it could be linked to the SLA, QoS and QoE. The current value of a KPI is calculated by using one or multiple CE values. The target value is specified using historical CE values or manually entered number. KPIs are visualized as dashboards thus giving an instant view at the value interpretation. Generally, KPIs show how well the scaling algorithms allow to provide the needed level of service and based on them the user can decide whether the values of AC should be altered or another implementation of the RTA should be deployed to ASAPCS.

If the real-time value of the CE is important for the user of the ASAPCS it can be transformed into a Context Indicator (CI). For example, although the current number of web-serer nodes is not a KPI it can still be valuable information for the ASAPCS user. The main difference between CI and KPIs are that the platform is not concerned with interpreting the value of CI and this is left to the end-user. If there would be a target value of the web-server nodes it would be a KPI rather than a CI.

V. ASAPCS TECHNICAL SOLUTION

The technical stack that was chosen for implementing the ASAPCS is shown in Fig. 3.

MP data originating from the CDP is sent to a DNS balanced Haproxy (a reliable, high performance TCP/HTTP load balancer) cluster. The Haproxy node receiving the data forwards it to one of the Kafka proxy nodes residing at the Kafka proxy cluster.

A proxy cluster is used for ensuring extra level of security and flexibility in defining ASAPCS MP API. Kafka proxies are implemented using Node.js and they forward the MP data further to one of the Kafka cluster nodes.

Kafka is chosen since it is horizontally scalable, fault-tolerant, ensures the right order of the messages and exactly-once processing. It is also known to perform well with streaming apps and other real-time data.

MP values from Kafka are processed by Spark Streaming consumer and transformed into CE. Data is persisted into Cassandra database which is known to perform well with temporal data and has good integration with Apache Spark platform.

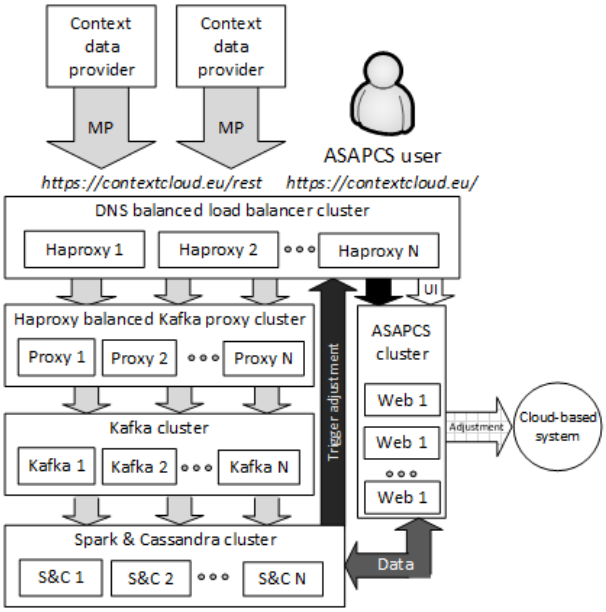

Fig. 3. ASAPCS prototype

Apache Spark was chosen because besides streaming component it also provides machine learning (MLlib) and graph processing (GraphX) libraries, which are important for meeting the previously defined requirements.

If it is determined that the current contextual situation identified by values of one or more CE requires execution of a RTA, Spark Streaming application makes a request to the DNS balanced Haproxy cluster which in turn calls the RTA. RTA is made available as a REST web service and is implemented together with the ASAPCS user interface as a NodeJS application. While calling the RTA Spark Streaming also passes the input parameters such as AC and CE values.

Based on the input data RTA scales the corresponding cloud application. ASAPCS ships together with a set of RTAs that are used for ensuring availability of ASAPCS itself. Currently we are investigating whether Docker containers would be a good fit for containerizing adjustments to provide wide support of programming languages and various libraries that could be used for RTA implementation.

Users of the ASAPCS platform access it through the DNS load balanced Haproxy cluster that forwards the requests to web-based ASAPCS UI, which is also implemented as a cluster of Node.js servers.

The prototype of the ASAPCS is in its early stages and is hosted on the CloudStack based RTU's open-source cloud computing platform. Currently ASAPCS is being validated with a use-case of video transcoding application requiring auto-scaling and altering data replication logic during run-time. This is done in collaboration with Komerccentrs DATI Grupa, a Latvia based IT company.

VI. CONCLUSION

Developments in the area of auto-scaling platforms are specifically important for microservice applications however traditional cloud-based systems would also benefit from existence of open-source auto-scaling platforms.

This paper presents architecture of ASAPCS and gives a brief overview of the current technological stack. Main advantages of ASAPCS are:

- ability to scale horizontally together with cloud applications,
- capability to process vast amounts of real-time data,
- definition of auto-scaling algorithms in a platform independent way thanks to extra level of abstraction,
- unlimited extendibility that comes from ASAPCS being truly open-source based,
- real-time monitoring of CIs and KPIs,
- run-time alteration of auto-scaling algorithms via graphical interface by changing values of ACs,
- machine learning and graph processing capabilities inherited from the Apache Spark platform,
- journaling of performed auto-scaling actions.

ASAPCS is still in active development and it must be extensively tested before the technological stack can be finalized. We are also looking into making ASAPCS more platform independent through providing support for other stream and batch processing platforms like Apache Flink. Upon reaching a sufficient level of maturity the source code of ASAPCS will be publicly released.

VII. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the research project "Competence Centre of Information and Communication Technologies" of EU Structural funds, contract No. 1.2.1.1/16/A/007 signed between IT Competence Centre and Central Finance and Contracting Agency, Research No. 1.12 “Configurable parameter set based adaptive cloud computing platform scaling method".

REFERENCES

[1] M.A.S. Netto, C. Cardonha, R.L.F. Cunha and M.D. Assuncao, “Evaluating auto-scaling strategies for cloud computing environments”, in Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS, Sept. 2014, pp. 187 – 196. https://doi.org/10.1109/MASCOTS.2014.32

[2] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet and A. Edmonds, “An architecture for self-managing microservices”, in Proceedings: Automated Incident Management in Cloud (AIMC), 1st International Workshop, in conjunction with EuroSYS 2015, pp. 19-24. https://doi.org/10.1145/2747470.2747474

[3] N. Alshuqayran, N. Ali and R. Evans, "A Systematic Mapping Study in Microservice Architecture," in 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), Macau, 2016, pp. 44-51. https://doi.org/10.1109/SOCA.2016.15

[4] A. Balalaie, A. Heydarnooria and P. Jamshidi, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture”, in IEEE Software, Vol., 33, Issue 3, art. no. 7436659, May - June 2016, pp. 42-52. https://doi.org/10.1109/MS.2016.64

[5] J. Lewis and M. Fowler. Microservices, March 2014 [Online]. Available: http://martinfowler.com/articles/microservices.html, [Accessed: Feb. 10,2017]

[6] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. https://doi.org/10.1109/SAINTW.2003.1210138

[7] A. Sill,”The Design and Architecture of Microservices” in IEEE Cloud Computing, Vol. 3, Issue 5, Sept. – Oct. 2016, pp. 76-80. https://doi.org/10.1109/MCC.2016.111

[8] C. Pahl and P. Jamshidi, “Microservices: A systematic mapping study”, in Proceedings of the 6th International Conference on Cloud Computing and Services Science Vol. 1, 2016 , pp. 137-146. https://doi.org/10.5220/0005785501370146

[9] P.P. Kukade and G Kale, “Auto-Scaling of Micro-Services Using Containerization” International Journal of Science and Research (IJSR), Vol. 4, Issue 9, Sept. 2015, pp. 1960-1963

[10] T. Hoßfeld, R. Schatz, M. Varela and C. Timmerer, “Challenges of QoE management for cloud applications” in IEEE Communications Magazine, Vol. 50, Issue 4, April 2012, pp. 28-36. https://doi.org/10.1109/MCOM.2012.6178831

[11] S. Berziša, G. Bravos, T.C. Gonzalez, U. Czubayko, S. España, J. Grabis, M. Henkel, L. Jokste, J. Kampars, H. Koç, J. –C. Kuhr, C. Llorca, P. Loucopoulos, R.J. Pascual, O. Pastor, K. Sandkuhl, H. Simic, J. Stirna, F.G. Valverde and J. Zdravkovic, “Capability Driven Development: An Approach to Designing Digital Enterprises”, in Business and Information Systems Engineering, Vol. 57, Issue 1, March 2015, pp. 15-25. https://doi.org/10.1007/s12599-014-0362-0