2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

Delivering Elastic Containerized Cloud Applications to Enable DevOps

Cornel Barna, Hamzeh Khazaei, Marios Fokaefs and Marin Litoiu
Department of Electrical Engineering and Computer Science
York University
Toronton, ON, Canada
{cornel,mlitoiu}@cse.yorku.ca, {hkh,fokaefs}@yorku.ca

***Abstract*—Following recent technological advancements in software systems, like microservices, containers and cloud systems, DevOps has risen as a new development paradigm. Its aim is to bridge the gap between development and management of software systems and enable continuous development, deployment and integration. Towards this end, automated tools and management systems play a crucial role. In this work, we propose a method to develop an autonomic management system for multi-tier, multi-layer data-intensive containerized applications based on a performance model of such systems. The model is shown to be robust and accurate in estimating and predicting the system's performance for various workloads and topologies, while the AMS is capable of regulating the application's behaviour by taking independent actions on its various parts.**

***Keywords*-devops; cloud computing; autonomic management systems; performance models; continuous delivery; containers; scaling; multi-tier big data applications;**

I. Introduction

In recent years, there have been parallel advancements in software engineering and the infrastructure that supports the software applications. The associated technologies, including cloud computing, containers, microservices and web services, have provided additional flexibility in the development, delivery and maintenance of software. More importantly, lightweight infrastructure technologies, like containers, and small self-contained software functionality, like microservices, have given rise to the *DevOps* paradigm. DevOps [1] is a development model, whose mission is to bridge the gap between development and operations management, i.e., that part which guarantees the proper function of the software system and its quality after it has been developed or even after it has been deployed. Bringing Ops closer to development enables changes to be incorporated in production code seamlessly and faster with less impact to users and clients.

Autonomic management systems (AMS) [2] and the ability of the software system to self-adapt to external stimuli and regulate its own behavior are integral parts of the DevOps process. In practice, AMS constitutes the automated tooling of the operation management side. As a consequence, it is vital for the process to have good adaptive capabilities [3] to guarantee what became known as *continuous delivery*, i.e., the ability to absorb changes of any kind in a transparent manner to the clients. The wealth of available technologies in cloud computing for autonomic management and the abundance of available solutions to solve recurring performance problems may actually create additional challenges for the developers. One important decision is about the functionality and the architecture of the AMS. For complex software systems that consist of many microservices and utilize a number of virtualization technologies, one has to wonder whether there is enough control and knowledge over the module to design a single, overarching AMS or multiple independent ones for each module. Another question is if there are multiple adaptive actions that can achieve the same goal, which one we pick and according to what criteria. In this situation, *models* can be of particular help as they give the opportunity to consider a number of different actions and solutions, evaluate them and, eventually, through systematic decision-making processes pick the optimal one. The models are part of the analysis and planning modules of an AMS.

In this work, we focus on multi-tier and multi-layer data intensive applications and their autonomic scaling. This class of applications has web, analytics and data tier clusters and need to scale all tiers to accommodate the web traffic. We provide responses to two relevant research questions:

Research Question (RQ1): *How can we model the performance of a multi-layer, multi-tier, data intensive web system?* For a performance model, used by the analyzer and planner of the proposed AMS, we need to consider a minimal but comprehensive structure which is easy to create by the development team; its parameters on the other hand have to be collected or estimated at runtime, during operations management. The high dimensionality of the problem would impose a multi-tier and multi-layer performance model and respective analysis and planning phases for the AMS.

Research Question (RQ2): *Can we design and implement a single autonomic management system for multi-layer, multi-tier web architectures?*
As far as the AMS is concerned, we are interested in exploring primarily its feasibility. Considering the multi-dimensional nature of the problem, the design and development of such a system is not trivial and a number of parameters need to be taken into account; monitoring agents need to be coordinated, interdependent actions need to be taken, externalities to be considered, and, eventually, manage to have a robust manager in place to guarantee the system's proper behaviour.

Through practical work and software development to answer

978-1-5386-1550-8/17 $31.00 © 2017 IEEE
DOI 10.1109/SEAMS.2017.12

the previous questions, our work results in two contributions:

1) A *self-tuning performance model* for multi-tier and multi-layer containerized applications deployed on cloud, based on layered queuing networks [4]. For a four-tier, three-layer architecture, including clients, a web tier, an analytics tier and a data tier, the model is able to receive the traffic for the system (in number of active users/clients) and its underlying topology (VMs and containers), and output its performance in CPU utilization for each tier, throughput and response time. In the presence of an anomaly, i.e., irregularly high or low metrics, the AMS can input different topologies to the model, varying the number or size of containers, in an attempt to find actions that rectify the problem.
2) An architecture and a working implementation for an *autonomic management system* for multi-tier, multi-layer data-intensive web applications. The proposed AMS has monitoring agents, which gather performance measurements from all layers (VMs, containers and software), uses the performance model to analyze problematic situations and plan for potential adaptive actions to fix them, and, finally, it can connect to any execution engine exposing an API, and perform the actions on the system to regulate its behaviour.

The rest of the paper is organized as follows. In Section II, we outline relevant literature on the topics of DevOps, container scaling and performance modeling. Section III details a novel performance model for multi-layer, multi-tier data-intensive web applications, while Section IV presents the architecture and implementation of the respective autonomic management system. In Section V, we present a set of experiments to validate the performance model and the AMS and prove their applicability on actual settings. Finally, Section VI concludes our work.

II. Related Work

DevOps [1] can be considered as a novel development process or model or paradigm or even philosophy. Its primary goal is to bridge the gap between development and operation management. In this capacity, it recommends the use of tools and processes, as well as knowledge and skill sets, which would span across the entire lifecycle of the software. The concept existed, even before it was named DevOps [5]; developers would tinker with system administration tools and concepts to better understand how the software is to be deployed, while IT operators would occasionally merge with the development team to better understand the system's functionality and ensure higher quality. Thanks to virtualization technologies and the self-adaptation concepts, DevOps is the beginning of a new software development culture and a new breed of hybrid developers and IT specialists.

Another mission of the DevOps model is to shorten the release cycle of the software [6]. The goal is for every change to be incorporated seamlessly to the system in production, while high quality is ensured and maintained [7]. Netflix is already employing *chaos engineering* techniques [8], which, in the context of DevOps, enable concepts such as *continuous deployment* and *continuous delivery*.

Balalaie et al. [9] present their experience in migrating a monolithic mobile-back-end-as-a-service (MBAAS) to a microservice architecture. Apart from the common pains and suffering of migrating a legacy system to a service-oriented architecture, they had some interesting experience with respect to the DevOps side of the process and the final system. First, they had to change the team organization from a horizontal structure (development, QA, operations) to more vertical teams with all layers responsible for the smaller services. Second, they report on the importance of monitoring the system and, finally, on the use of containers to bridge the gap between the development and the production phases.

Even though containers is a technology whose popularity has only recently been increasing, scaling, and even autoscaling, is an issue, which has already received attention by practitioners and researchers. In practice, scaling may refer to either the VM hosts or the actual containers. Focusing on the latter, there are several tools that support scaling of containers, including popular choices, such as Kubernetes [10], [11] from Google, Docker Swarm in conjunction with Docker Compose [12], [13], the native orchestration service of Docker, and Mesos [14] with Marathon [15], [16]. Other solutions are offered by Microscaling [17], which offers autoscaling capabilities, and the Amazon EC2 containers service based on alarms from the CloudWatch monitoring service [18]. These tools provide mostly reactive, threshold-based solutions, when they allow for autoscaling capabilities, where the scaling plans are usually static and defined in design time. Conversely, the proposed solution, thanks to the performance model, can be dynamic with respect to the scaling plan and, potentially, it can also be proactive in the sense that it can alleviate a problematic situation immediately and avoid incremental actions, with slow reaction, that can compromise the system's performance. Practically, the model can be used as a complement to any AMS. On the other hand, the proposed AMS is capable of handling complex architectures with multiple tiers, as well as perform actions on multiple layers. This capability is not definitively proven in existing tools.

Layered performance models have been proposed before for client-server systems [4], but not for containerized applications and for cloud. In cloud, as shown in the next sections, we have to account for unknown delays. As far as parameter estimation is concerned, Woodside et al. [19] and Epifani et al. [20] presented frameworks to tune non-functional properties including performance. Epifani et al. provide a Bayesian technique to re-estimate probabilities, which can be applied to different formal models such as Markov chains or queuing networks. The methodology presented in this work is based on Kalman filters [21] and focuses on tuning a multi-layer and multi-tier performance model and its applicability to modifying the application topology on demand in response to changes in a volatile environment such as cloud. Huber et al. [22] follow a similar model-based approach, based this time on the Descartes Modeling Language (DML) to propose an adaptation framework for cloud resource management. The framework is quite comprehensive providing end-to-end adaptation from the virtualization layer to the application layer, complemented with predictive capabilities.

Performance analysis of cloud services considering containers as a virtualization option is in its infancy. Most of the works compare the implementation of various applications deployed either on VMs or containers. Joy [23] conducted a performance comparison including a three-tier cloud application for storing and retrieving data for the Joomla framework. It was shown that containers outperform VMs in terms of performance and scalability. Container deployment processes 5x more requests compared to VM deployment and also containers outperformed VMs by 22x in terms of scalability. This work shows promising performance indexes when using containers instead of VMs for service delivery to end users.

Ali et al. [24] performed a comprehensive study on performance evaluation of containers under different deployments. They used various benchmarks to study the performance of native deployment, VM deployment, native Docker and VM Docker. In native deployment, the application is installed in a native OS; in VM deployment, the application is installed in a vSphere VM; in native Docker, the application is installed in a container that runs on a native OS; and finally, a Docker container including the application is deployed on a vSphere VM that itself is deployed on a native OS. Redis[1] has been used as the back-end datastore in this experiment. All in all, they showed that in addition to the well-known security, isolation, and manageability advantages of virtualization, running an application in a Docker container in a vSphere VM adds very little performance overhead compared to running the application in a Docker container on a native OS. Furthermore, they found that a container in a VM delivers near native performance for Redis and most of the micro-benchmark tests.

III. Modeling the Performance of Multi-Tier Multi-Layer Data-Intensive Applications

In this section, we first outline the general architecture and characteristics of the subject application, which our model and AMS are intended to be applied on and then we go into details as to how we model the performance of such applications.

A. Subject Application

In the context of our work, we focus on four-tier, data-intensive web applications, employing a web service layer, a big-data analytics tier and a NoSQL database (see Figure 1). The application is intended to have high throughput and be able to handle a large and increasing number of users. The three functional, and scalable, tiers of the application are deployed on containers, which in turn are hosted on virtual machines. This deployment constitutes the multi-layer nature of the subject architecture. Figure 1 depicts the architecture and the deployment scheme of the subject application as used in our experiments. Without loss of generality, this

[1]Redis is a NoSQL key-value datastore.

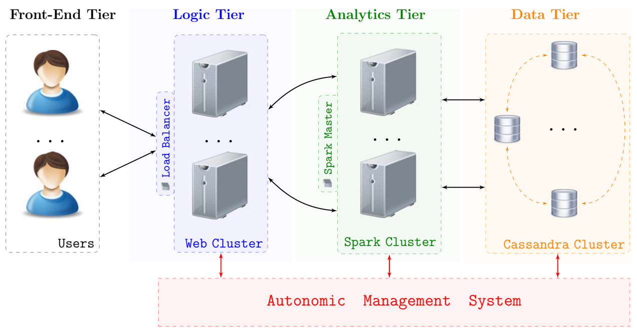

Fig. 1: *A typical four-tier, tree-layer application.*

implementation assumes Apache Tomcat[2] as the application server, Apache Spark[3] as the analytics engine and Apache Cassandra[4] as the NoSQL database. The container technology of choice is Docker[5].

The functionality and architecture of the subject application are largely based on the LEGIS application [25], as it was used in our previous experiments on Docker elasticity [26]. LEGIS is a distributed navigation service, which receives directions requests and finds optimal paths based on current traffic data. A set of original paths are forwarded from Tomcat to Spark, which then finds the closest deployed traffic sensors and uses these as keys to query Cassandra for local and current traffic data. Once this data is acquired, it is analyzed and Spark annotates the original paths with a score indicating the navigability of each segment. The annotated paths are returned to the users.

LEGIS architecture is common; it can be found in any web application that includes real-time analytics such as recommender systems. Given the focus on container technology and big data analytics, our work covers a wide range of applications, where, up to a degree, the same challenges exist, especially with respect to monitoring and the execution of the adaptive actions. Even with respect to the performance model, the use of established theoretical background, such is that of layered queuing networks, implies that our model remains topical and can be applied to applications with any number and type of layers and queues (tiers), with minimal modification and extension.

B. Performance Model

Since one of the goals of the DevOps method is dynamic scaling (elasticity), to find the required amount of resources (containers and VMs) that brings the system back to a desired state, we propose a multi-tier, multi-layer performance model to support containerized and analytics-based applications in cloud. Additionally, the model acts as the integrator between the development and operations teams. The former is responsible for profiling the application and identifying its demands

[2]http://tomcat.apache.org/
[3]http://spark.apache.org/
[4]http://cassandra.apache.org/
[5]https://www.docker.com/

for resources, such as CPU, memory, disk network and so on. In design time, these demands are used to build the model. In runtime, the operations team uses the model to regulate the performance of the application and guarantee its service level objectives. Moreover, the operations team is responsible for updating the model at runtime to maintain its accuracy as the environmental conditions of the application change. Our model-based automated management system automates the last two steps, which are the responsibility of the operations team, effectively creating a DevOps tool for the development team.

The model creates a representation of the system using three queuing networks (layers) based on the layered queuing network (LQN) theory [4]. It captures the details of the software resources in the first layer, the containers in the second layer and the virtual machines (VM) in the third layer. Each resource has an associated demand, which is the processing time necessary to handle a single request at that resource. Ideally, for hardware resources (CPU, disk), the demands are measured, possibly as the result of profiling. If measurement is not possible (either because there are no tools available or the overhead is too big), then a method to estimate them is required. For the software resources, the demands are in fact response times from the layer below (see Figure 2). Therefore, the service time of requests at an application resource is given by the response time of the container queuing network and the service times from the container layer are determined by the hardware queuing network.

Figure 2 shows the three-layered performance model of a four-tier application for which the three back-end tiers have been modeled, excluding the presentation/front-end tier. The three back-end tiers include: logic tier represented by the `web` software resource, analytics tier represented by the `Spark` resource and the data tier represented by the `Cassandra` resource. The software resources (`web`, `spark` and `Cassandra`) run in containers (`container1`, `container2` and `container3` respectively), and each container runs on one or more CPUs.

To solve the model, we leverage Schweitzer's approximation algorithm [27] for mean value analysis (MVA) to find the target performance metrics such as utilization, response time and throughput for each layer. This algorithm is iterative, where at each iteration it checks if the configuration is stable (the configuration is considered to be stable if, in two consecutive iterations, the queue lengths at the resources change less than a given value). If the configuration is not stable, the users will be redistributed among the resources and a new iteration starts. If the configuration is stable, the algorithm stops and outputs the calculated metrics. To iterate between layers, a fix point algorithm can be used.

Overall, the performance model can be seen as a function $\mathcal{M}$ defined in Equation 1.

$$[U_e, RT_e, X_e] = \mathcal{M}(N, Z, container_{\#}, VM_{\#}, D) \quad (1)$$

where U_e is estimated CPU utilization for all resources, RT_e is the estimated response time and X_e is the estimated

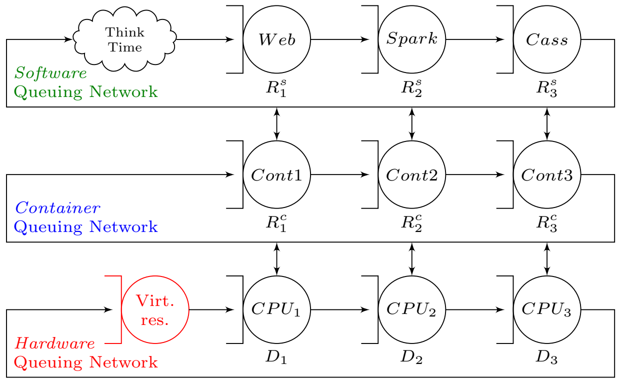

Fig. 2: *A three-layered queuing network of resources for modeling containerized cloud applications. Each resource can have a multiplicity factor associated—i.e., there are multiple copies of that resource, and the requests are evenly distributed among the copies.*

throughput. N indicates the number of users in the system (workload), Z denotes the think time of the users and D is the service demand for all resources. U and D are vectors.

Note that in Figure 2, we have multiplicity for service units at all three layers. More specifically, in the hardware layer, we have multi-core CPUs for VMs, in the container layer, we may have multiple containers for Web, Spark or Cassandra clusters and finally in the software queuing network we have multi-threading. As a result, all queues are multi-server.

C. Dealing with structural uncertainties in the model

When an application is deployed on a containerized topology in cloud, there are extra delays introduced by the cloud infrastructure or cloud management. The interactions among application components and services reach layers of the cloud and network that are not known or accessible to the application developer. Although these extra delays are reflected in the measured application performance metrics, their source cannot always be identified [28], [29] and, thus, their source or intensity should be considered as *uncertainty*. In other words, even if we know everything about our application, it is not enough from a performance point of view, since we cannot capture all infrastructure performance related factors. We call these unmodeled dynamics *"structural uncertainties of the model"*, i.e., missing components, delays and queues that we have not knowledge of. To account for unmodeled dynamics, we propose to add a serial virtual resource, in the cloud layer, representing a delay center. This serial virtual resource will account for the delays in the application requests processing, due to additional delays in the cloud (see Figure 2). The parameters of this delay center, such as service times and visit probabilities are unknown at design time and they will be identified at runtime together with other parameters of the model. The reason the virtual resource is added on the cloud layer is due to the assumptions that from a user/manager perspective the software and container layers are controllable and generally known, while, especially for public clouds, the bottom layer is out of scope. Therefore, this is where the uncertainty lies.

D. Estimating parameters and tuning the model

In order to solve the layered performance model presented in Figure 2 we need to specify the workload (arrival rates or number of users), the topology of the application (the resources and how a request moves from one resource to the next) and the demand for each resource. These have to be measured or estimated. Among these, the hardest to measure are the demands, since they require instrumentation of each request at the kernel level and thus yield large overheads. Since utilization (U) and throughput (X) for each resource are easier to measure, we can use the formula $U = XD$ to estimate the demands D. However, the measured value might contain noise and measurement errors, and the formula would produce wrong values. A better way to estimate D is to take into account the measurement and modeling noise and use estimators. This has been used in the past with very good results for non-cloud deployments [30], [31].

In this paper we use Kalman filter [21] to estimate the model parameters, which we cannot measure. Figure 3 shows the interactions among Kalman filter and performance model that are being orchestrated by autonomic manager. As can be seen in Figure 4 the autonomic manager connects the "Management Loop" to "Model Loop" by directing the measured data toward performance model. The measured metrics are: $Mes = [RT, X, U_w, U_a, U_c]$, where RT is the response time of the application, X is the throughput, U is the mean utilization and w, a, c denote the web, the analytics and the Cassandra cluster. The estimated metrics are $D = [D_w, D_a, D_c, D_v]$ where D stands for Demand and w, a, c for web, analytics and Cassandra clusters and v for the virtual resource. The model $\mathcal{M}$ (defined by Equation 1) will predict the vector $P = [RT^m, X^m, U_w^m, U_a^m, U_c^m]$. The estimation of the vector D is done periodically at run-time to maintain the accuracy of the performance model (Figure 3). When a new set of measured values Mes is available, if the model would produce significantly different predictions P (that is, the square root of $P - Mes$ greater than some acceptable value) then Kalman filter is used to estimate new demands D. These demands are then fed to the model. The number of measured metrics is greater than the number of estimated ones and therefore the necessary condition of convergence is ensured [19].

Calibration is a relatively light operation so that it can be done for every single system measurement without affecting reaction time of the autonomic manager.

IV. Autonomic Management System

The proposed AMS (see Figure 4) follows the MAPE-K architecture [32] for adaptive software systems. According to this architecture the AMS, first, has a **M**onitoring module, which manages a set of sensors and monitoring agents responsible for gathering measurements from the components, layers and tiers of the managed applications. Second, it has an **A**nalysis module responsible for analyzing the gathered

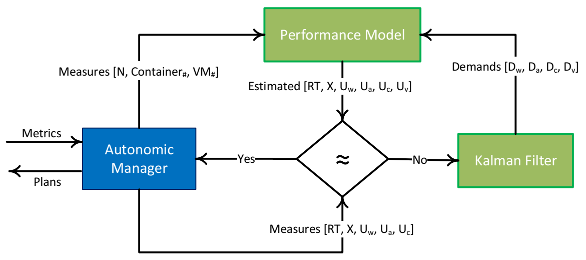
Performance Model
Measures [N, Container#, VM#]
Estimated [RT, X, Uw, Ua, Uc, Uv]
Demands [Dw, Da, Dc, Dv]
Metrics
Autonomic Manager
Plans
Yes
≈
No
Kalman Filter
Measures [RT, X, Uw, Ua, Uc]

Fig. 3: *Interactions among performance model, Kalman filter and autonomic manager; this figure elaborates the "Model Loop" of Figure 4.*

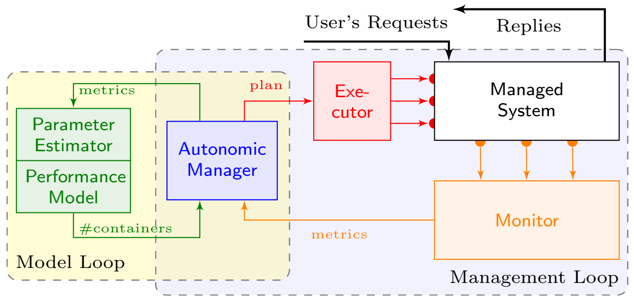

Fig. 4: *The architecture of the proposed Autonomic Management System. The AMS continuously monitors the managed system, evaluates it's health and makes changes if necessary.*

measurements and identifying if there are any problematic situations now or in the near future, for example, resource saturation or high response time. Third, it has a **P**lanning module, whose responsibility is to find an adaptive action to rectify the problematic situation identified before. Finally, the AMS has an **E**xecution module responsible for applying the adaptive actions as identified during the planning phase. This last module can either act directly on the system or connect to other execution engines provided by the cloud or the software infrastructure.

In this section, we describe in general the characteristics of the MAPE architecture for the proposed AMS, in addition to the specific details of our implementation, which is also used in the consequent experiments.

A. Monitoring

The monitoring module is responsible for keeping track of those metrics that may indicate a problem with the application's performance and help developers or the AMS, if the process is automatic, guarantee the application's health. These metrics may span across all layers and tiers, thus different tools and agents may be necessary. The proposed monitoring module gathers CPU utilization for all the back-end tiers of the applications (Web, Spark and Cassandra) on the container level, CPU utilization of the host VMs, as well as response time and throughput from the entire system. Measurements are gathered through third party tools including Ceilometer[6] from the OpenStack cloud and Docker Stats API[7].

B. Analysis and Planning

The proposed manager uses a simple analysis module; specific thresholds are placed on the monitored performance metrics, for example, "average CPU utilization on web or Spark cluster cannot exceed 70%". The particular values for the thresholds greatly depend on the application, the demand it generates and the service level objectives. For this reason, the process of defining the thresholds requires rigorous profiling of the application and testing in production, after deployment, also a DevOps requirement [8]. Upon violation of one of the thresholds, the planning module is activated. Normally, a different plan is called for every threshold violated; if memory is saturated, the plan is to increase the memory resource in the topology, if CPU is saturated the plan is to increase the number of cores in the infrastructure and so on.

Our planning phase uses the performance model to decide what actions to take. In current state-of-practice, like OpenStack Heat[8] or Amazon EC2 AutoScaling[9], the adaptation plans are defined statically at design time. For example, when the developers set up the auto-scaling capabilities, they say that every time an upper threshold is crossed one virtual machine or one container will be added in the topology. We argue that the nature of the adaptive actions should be more dynamic as a threshold can be crossed in a variety of scenarios. Therefore, while the action that is to be taken is important, its *intensity* is also crucial.

The proposed performance model allows us to respond to each problematic situation differently. With the ability of the model to try different topologies for given workloads on the fly, it is possible that each time it is invoked it may return a different combination of resources to address the problem. In addition to the model that provides us with dynamic scaling actions depending on the given situation, we use an additional step to determine whether the scaling action is actually necessary at that point. The fact, and the challenge, with respect to web traffic is that it is rather irregular, meaning that there can be sudden and short spikes, upwards or downwards, which can cause the system to temporarily look out of order. These spikes can throw an AMS off and cause unnecessary scaling actions, which will have to be undone almost immediately. Considering that a monitoring service gathers new measurements approximately once every minute and a decision is made at that point, if every check resulted in an action, such frequent and violent oscillations in the system's infrastructure could have a detrimental effect on the end-users as well as on the system's budget. Current AMSs used by practitioners provide the option to define a *recurrence* factor for threshold violations. In other words, the AMS takes action only when a threshold is violated n times, where n is defined in design time.

In our work, we take this guard one step further and we propose the "Scaling Heat Algorithm" (Algorithm 1) for containers. The algorithm is used in conjunction with the performance model right after the analysis phase. Central concept is that of *"heat"*, which determines the buildup for adding or removing containers. A violation of upper thresholds, indicating a saturation of a container cluster, will result in increasing the heat factor (lines 1–4), while a violation of lower thresholds, i.e., underutilized clusters, will result in decreasing heat (lines 5–8). As the heat factor grows, it increases the chance of adding containers and vice versa. After the result of the analysis is determined, the heat factor is changed accordingly. If there is no violation, heat returns gracefully to zero, one step at a time (lines 9–13). When it hits a given number n (we use 5 in our experiments), either positive for additions or negative for removals, the algorithm informs the AMS to execute the scaling action as directed by the model (lines 14–19). After an action is performed, the heat factor returns to zero immediately. The algorithm is executed at every decision point when new measurements are acquired from the monitoring service.

The main difference between the scaling heat and the traditional recurrence factor is that the latter requires for the violations to reoccur consequently. If a spike of the opposite direction occurs while we are within the recurrence range, then the factor resets. This way the system can get stuck in violation for a long time. By simply adjusting the heat factor when opposite or no violations occur and not resetting it, we avoid these perpetual situations. Figure 5 shows a slice of data from our experiments that shows the progress of the heat factor for the web cluster (top plot) and for the Spark cluster (middle plot) and the CPU utilizations (bottom plot) that cause the fluctuations in heat. Taking iteration 210 as an example, we can see that Spark has generated a heat of 4, but in the next iteration we see a reduction. This would have reset a traditional recurrence factor, thus negating and delaying a scaling action, which becomes necessary only two iterations later.

In general, the available actions in our AMS are *"add/remove Tomcat container(s)"* and *"add/remove Spark container(s)"*. The latter is rather straightforward as the addition of a Spark worker requires only the configuration of Spark master so that it knows about the new member of the cluster. On the other hand, adding a Tomcat worker implies also the addition of Spark workers. The reason lies in how Spark operates[10]. A Tomcat server contains a *driver*, a process that manages all the tasks that are to be executed by the analytics service. In order to actually run a job, a driver needs to be tied with an *executor*, a process on the Spark side. When a new Tomcat worker starts, but there are no available Spark workers to connect the driver with the executor, then the new web worker would be blocked and result in high response

[6]http://docs.openstack.org/developer/ceilometer/

[7]https://docs.docker.com/engine/reference/api/docker_remote_api_v1.23/#/get-container-stats-based-on-resource-usage

[8]http://docs.openstack.org/developer/heat/

[9]https://aws.amazon.com/autoscaling/

[10]Spark Cluster Mode Overview: http://spark.apache.org/docs/latest/cluster-overview.html

Algorithm 1: Scaling Heat Algorithm: The decision making algorithm for adding and removing containers to a cluster. This algorithm is executed every time a new set of measurements is available and a decision has to be made regarding cluster elasticity.

```
input  : $utilization$ — the average CPU utilization of containers in a
         cluster
input  : $lower\_threshold$ and $upper\_threshold$ — the limits of the
         desired range for the CPU utilization of containers in a
         cluster
input  : $heat$ — a value indicating the buildup for adding/removing
         containers to/from a cluster
output : $heat$ — the new value to be used in the next iteration

if $utilization \geq upper\_threshold$ then
    // cluster overload
    if $heat < 0$ then
        // reset any buildup for container removal
        $heat \leftarrow 0$;
    $heat \leftarrow heat + 1$;
else if $utilization \leq lower\_threshold$ then
    // cluster underload
    if $heat > 0$ then
        // reset any buildup for adding containers
        $heat \leftarrow 0$;
    $heat \leftarrow heat - 1$;
else
    // the utilization is within range
    // move heat one unit towards 0
    if $heat > 0$ then
        $heat \leftarrow heat - 1$;
    else if $heat < 0$ then
        $heat \leftarrow heat + 1$;
if $heat = n$ then
    Invoke model to find number of extra containers to be added;
    $heat \leftarrow 0$;
else if $heat = -n$ then
    Invoke model to find number of excess containers to be removed;
    $heat \leftarrow 0$;
return heat
```

time. By adding new Spark workers along with the Tomcat worker, we guarantee that there will be at least one free executor slot for the new web worker. In our experiments, we add two Spark workers for every new Tomcat to increase the possibility for a free executor. If the web worker connects to an already available Spark worker, rendering the new Spark workers unnecessary, this situation could be resolved in the next iteration, where the extra Spark worker will be removed, since it is not being used.

As far as the management of virtual machines is concerned, this is achieved indirectly. As it was explained in our previous work [26], in order to enable autoscaling of containers in a meaningful manner, one has to constrain the resources that become available from the host VM to the containers, effectively building what we called *computation units*. As a consequence, this creates a notion of *capacity* for the VMs, i.e., how many computation units they can hold depending on their size. In the process of scaling containers, if the capacity of the available VMs is exhausted, or about to, the AMS automatically adds another virtual machine. The new VM is added in the Docker Swarm cluster and becomes available as a container host.

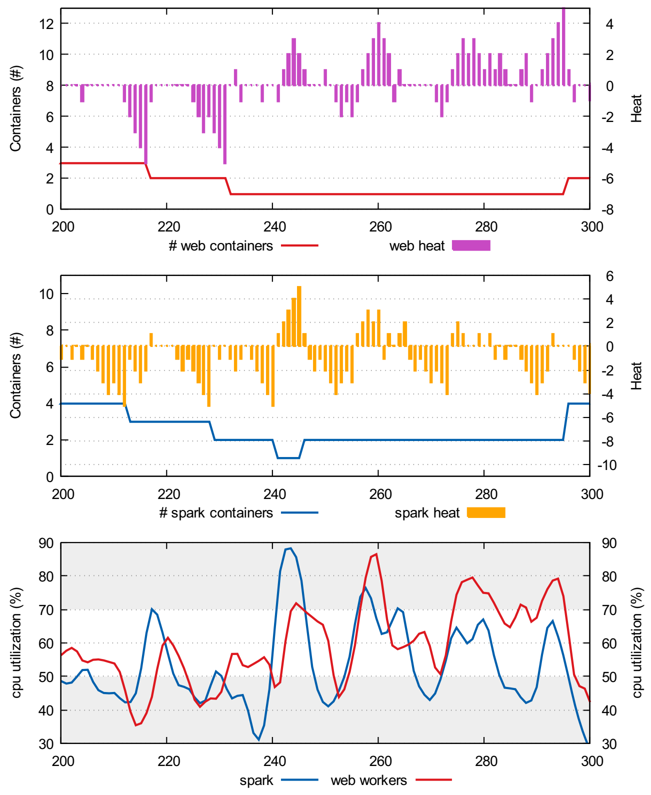

Fig. 5: *The evolution of heat in Experiment 3.*

C. Execution

To apply the planned adaptive actions, the proposed AMS relies on third-party services. For example, to add or remove virtual machines, the manager uses a REST API, which is usually available by the cloud provider. Similarly, it uses the Docker API for corresponding actions with containers. A noteworthy detail on container scaling is that Docker Swarm[11], responsible for container placement in host VMs, does not allow the developer to control to what host VMs the containers will land. Nevertheless, the developer can configure Swarm's placement strategies[12], choosing between *spread*, *random*, and *binpack*. In our experiments, we used the binpack strategy to minimize the number of required VMs.

V. Experiments

In order to evaluate our work in a complete manner, we designed and performed two sets of experiments. The first focuses on the autonomic management system. The experiments in this case are on an actual cloud environment with a real application and the goal is to demonstrate the ability of

[11]The version of Docker Swarm we use is 1.2.4.
[12]https://docs.docker.com/swarm/scheduler/strategy/

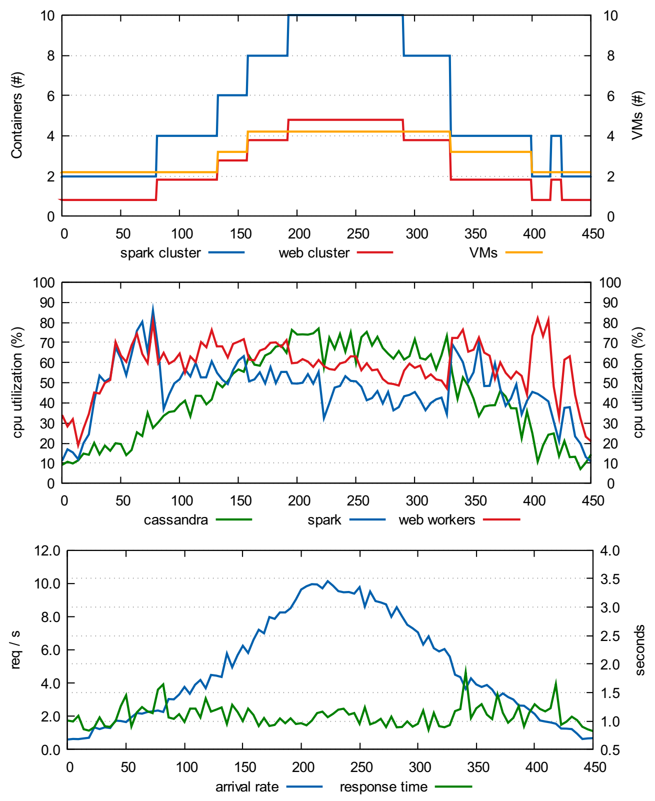

Fig. 6: ***Experiment 1****: Coupled clusters with linearly increasing/decreasing workload. The Web and Spark containers are added/removed together to guarantee that there are enough Spark containers available for the Web containers.*

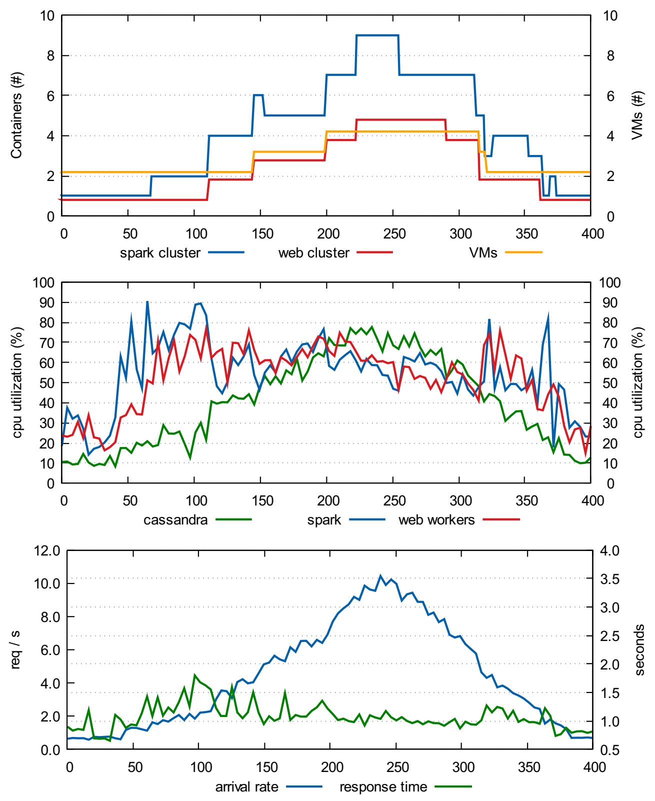

Fig. 7: ***Experiment 2****: Decoupled clusters with linearly increasing/decreasing workload. Addition and removal of containers happens independently for the two tiers. Web containers compete with each other to grab Spark containers.*

the AMS with the model to properly manage the behavior of the application. The experiments also show the ability of the AMS to handle either consistent or variable workload with the same efficiency and, additionally, they illustrate the correlation between the tiers in terms of performance metrics and how the AMS can regulate the performance of the whole system with independent adaptive decisions on parts of it. The second set of experiments focuses on the model, where we empirically validate its ability to accurately capture the performance of the multi-tier, multi-layer application and its ability to quickly converge after recalibration.

A. Evaluation of Autonomic Management System

The evaluation of the autonomic management system focuses on its applicability on real applications and real cloud settings, as well as its efficiency in regulating the application's performance with the assistance of the model given dynamic workloads. We designed three experiments; one with regular workload (increasing, then decreasing) and coupled clusters (Spark containers are added/removed only along with Tomcat containers), one with regular workload but decoupled clusters (Spark and Tomcat containers are scaled independently), and one with more realistic workload and decoupled clusters.

The experiments are performed on the LEGIS application

TABLE I: *Container Settings*

	Size	RAM	Quota of Host's CPU
Load Balancer	large	4 GB	25.0%
Web Node	large	4 GB	25.0%
Spark Master	large	4 GB	25.0%
Spark Node	medium	2 GB	12.5%
Cassandra Node	large	4 GB	25.0%

deployed on SAVI [33], an OpenStack research cloud, with Docker containers. The Docker Swarm cluster consists of Ubuntu 16.04 VM hosts, with 16GB RAM, 8 vCPU and 160GB disk. In the experiments, we use Docker version 1.11, Apache2 load balancer version 2.4.7, Tomcat version 7, Spark version 1.6.1, and Cassandra version 2.2.7. As far as the containers are concerned, we define two sizes, medium and large, based on constrained resources. The sizes and how they are used in the topology are outlined in Table I.

In all three experiments we captured the progress of the performance metrics (average CPU utilization for each cluster, response time) while the system experiences workload (arrival rate). The metrics are measured directly from the system using the monitoring module. During the experiments, the number of containers and VMs change and are the direct product of the manager's activity as a response to variations in workload. The goal for the AMS is to maintain the CPU utilization between 50% and 70%.

Figures 6 and 7 show the results for the first two experiments when the system experienced regular workload, the difference being that in the first one the scaling of Spark is tied with that of Tomcat. This can be seen in the upper plots of the two figures; in Figure 6 whenever we have an addition/removal of a Tomcat container, we have a symmetric addition/removal of two Spark containers, while in Figure 7 we can see a Spark container being removed (around iteration 150) or being added (around iteration 320) without a removal/addition of a Tomcat container. The independent decision making results in lower number of containers in the latter case, as well as better management of resources for Spark; in the first experiment, we overprovision Spark containers, resulting in relatively lower CPU utilization, indicating underutilized clusters. In addition, it resulted in slightly better VM management; the fourth VM is added slightly later around iteration 200, while the third one is removed a little earlier in the second experiment, around iteration 320. Finally, we can also see that thanks to the model, the AMS can make dynamic decisions, evident around iteration 320 in either figure, where two Tomcat containers are removed, when usually the AMS changes one container at a time.

Figure 8 shows the result of the third experiment, where scaling in the two cluster is also decoupled, but the workload is irregular. The experiment shows that the manager maintains the CPU utilization and response time in this scenario as well. This indicates the robustness of the AMS in handling any kind of workload. In this figure, the independent decisions are more evident, with respect to Spark scaling. Finally, in all three experiments, the response time is kept almost constant, which is the ultimate goal of scaling.

B. Model Validation

To validate the accuracy of the proposed model, we used traces from the actual LEGIS application. First, we used the OPERA tool [34] to implement the proposed layered multi-tier performance model and be able to conduct the experiments in a streamlined manner. To acquire the trace, we deployed the application on the SAVI OpenStack cloud with Docker containers. We then fed it with a linearly increasing workload, one extra user at a time, and measured the CPU utilization for the Web and Spark clusters, response time and throughput of the system. The next step was to get the workload and the topology (number of containers in each layer) for each iteration, provide them as input to the model and estimate the new measurements. We performed a few trial runs to identify how often we need to recalibrate the model using Kalman filters. We found that to achieve error rate less than 10%, we will have to calibrate every 10 iterations approximately, for less than 5% every 2 iterations and for less than 1% error rate every 1 iteration.

Given these calibration rates as parameters for our experiments, we executed three runs of the model, one for each rate, and compared the measured values, from the application trace,

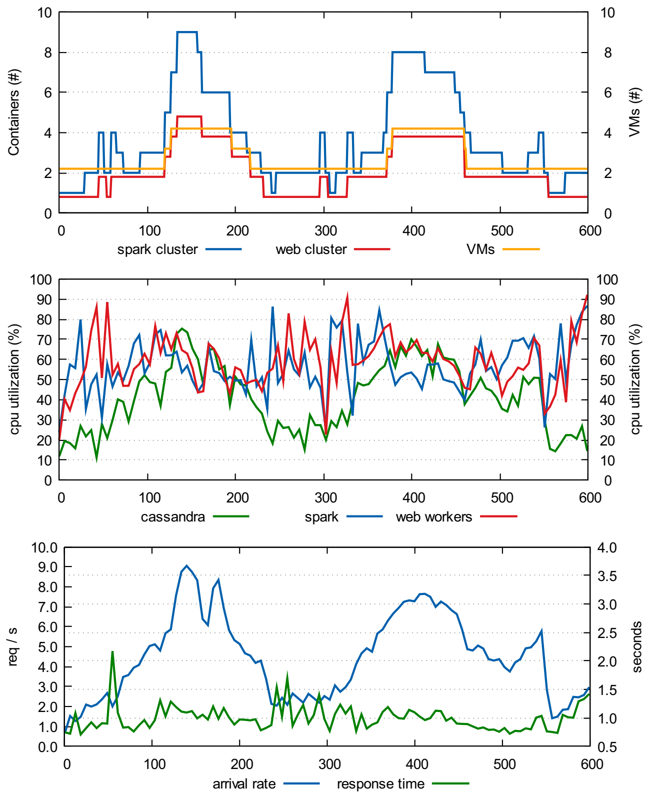

Fig. 8: ***Experiment 3**: Decoupled clusters with irregular workload.*

with the estimated ones, from the model. Table II presents the results of the three runs in terms of the average error rate. As it was expected, the more frequently Kalman is invoked, the more accurate the model becomes. For calibration at every iteration, the model achieves error rate between less than 0.1% to 0.2% for all four metrics. As the frequency of the calibration decreases, the error rate increases, but disproportionally; between 4% and 4.8% for every 2 iterations and between 5.9% and 8.9% for every 10 iterations. While an error rate of less than 10% is acceptable, the Kalman calibration is actually a cheap process. Therefore, we can invoke it as frequently as we want to further decrease our error rate.

TABLE II: *Evaluation results for model accuracy and calibration rate*

Calibration rate (#iter)	1	2	10
CPU Web Error (%)	0.089	4.838	7.868
CPU Spark Error (%)	0.156	4.496	7.789
Response Time Error (%)	0.219	4.043	8.993
Throughput Error (%)	0.164	4.537	5.926
Precision (CPU Web)	0.99	0.86	0.65
Recall (CPU Web)	0.99	0.75	0.67
F-measure (CPU Web)	0.99	0.80	0.66
Precision (CPU Spark)	1.00	0.83	0.65
Recall (CPU Spark)	0.98	0.84	0.80
F-measure (CPU Spark)	0.99	0.84	0.72

Besides the absolute error rate of the model, it is interesting to investigate its relative error with respect to scaling, in other words the potential of the model to make accurate scaling decisions. To validate this ability, we measured the number of times where the model agreed with the trace that a scaling action is required, i.e., CPU utilization greater than 70% or lower than 50% in either cluster (*true positive*). We also measured the times where the model did not agree with the trace, when a scaling action was in fact required (*false negative*) or when it was not required (*false positive*). Based on these counts, we are able to calculate precision, recall and F-measure, also presented in Table II. Calibration is also crucial in this case, as it increase precision and recall to 99% if performed in every iteration. This implies that the model can be trusted as a decision mechanism for scaling actions. As the frequency of the calibration decreases, precision and recall greatly deteriorate, meaning that the model becomes more prone to bad scaling decisions.

VI. Conclusion

Our work revolves around the concept of DevOps and the goal is to integrate automated operation tools in the development process. This is to allow developers to better manage their application either during development, deployment or production and IT experts to get a more complete picture about the deployed application, thus effectively bridging the gap between the two teams. This particular paper focuses on the autonomic management of a complex deployed application, with multiple application tiers and infrastructure layers, with the use of a model to predict and estimate the performance of the system under various workloads and topologies and be used to determine a topology that would regulate its behavior in the event of high or low workload.

Our experiments demonstrate the accuracy and robustness of the proposed model to estimate the application's performance and to be used by an autonomic management system for scaling. The model is as simple as possible, at the same time accurate for its purpose: finding a deployment architecture that meets performance requirements. The model is designed to handle complex multi-tier and multi-layer data-intensive applications, a popular and topical domain for web software systems. The proposed autonomic management system based on the aforementioned performance model is also shown empirically to be applicable and robust against a variety of workloads and topologies. The AMS is also capable of making decisions for the various tiers independently and result in more efficient scaling actions in terms of resource management.

Acknowledgment

This research was supported by IBM Centres for Advanced Studies (CAS), the Natural Sciences and Engineering Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network and the Ontario Research Fund for Research Excellence under the Connected Vehicles and Smart Transportation(CVST) project.

References

[1] M. Hüttermann, *DevOps for developers*. Apress, 2012.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[4] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2009.

[5] D. Spinellis, "Being a devops developer," *IEEE Software*, vol. 33, no. 3, pp. 4–5, 2016.

[6] L. Zhu, L. Bass, and G. Champlin-Scharff, "Devops and its practices," *IEEE Software*, vol. 33, no. 3, pp. 32–34, 2016.

[7] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[8] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[9] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[10] Kubernetes, "Production-Grade Container Orchestration," 2016. [Online]. Available: http://kubernetes.io/

[11] C. Sanchez, "Scaling Docker with Kubernetes V1," 2015. [Online]. Available: http://www.infoq.com/articles/scaling-docker-kubernetes-v1

[12] D. Inc., "Docker Compose," 2016. [Online]. Available: https://docs.docker.com/compose/

[13] B. Christner, "How to scale Docker Containers with Docker-Compose," 2015. [Online]. Available: https://www.brianchristner.io/how-to-scale-a-docker-container-with-docker-compose/

[14] A. Mesos, "Program against your datacenter like its a single pool of resources," 2015. [Online]. Available: http://mesos.apache.org/

[15] I. Mesosphere, "A container orchestration platform for Mesos and DCOS," 2015. [Online]. Available: https://mesosphere.github.io/marathon/

[16] ——, "Autoscaling Marathon services using CPU and memory," 2015. [Online]. Available: https://docs.mesosphere.com/1.7/usage/tutorials/autoscaling/cpu-memory/

[17] F. Ltd, "Microscaling - real-time auto-scaling using constant feedback from your running system," 2016. [Online]. Available: https://microscaling.com/

[18] A. EC2, "Amazon EC2 Container Service - Developer Guide (API Version 2014-11-13)," 2016. [Online]. Available: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/cloudwatch_alarm_autoscaling.html

[19] M. Woodside, T. Zheng, and M. Litoiu, "The use of optimal filters to track parameters of performance models," in *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*. Washington, DC, USA: IEEE Computer Society, 2005, p. 74. [Online]. Available: http://dx.doi.org/10.1109/QEST.2005.40

[20] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–121.

[21] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[22] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bahr, "Model-based self-aware performance and resource management using the descartes modeling language," *IEEE Transactions on Software Engineering*, 2016.

[23] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*. IEEE, 2015, pp. 342–346.

[24] Qasim Ali, Banit Agrawal, and Davide Bergamasco. (2016, 6) Docker containers performance in vmware vsphere. [Online]. Available: http://blogs.vmware.com/performance/2014/10/docker-containers-performance-vmware-vsphere.html

[25] M. Fokaefs, D. Serrano, R. Veleda, and M. Litoiu, "Locality-Enhanced Geographic Information System," in *4th International IBM Cloud Academy Conference*, 2016.

[26] M. Fokaefs, C. Barna, R. Veleda, M. Litoiu, J. Wigglesworth, and R. Mateescu, "Development and management of containerized data-intensive applications: An exploratory study," in *CASCON 2016: Proceedings of the 2016 conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 2016.

[27] P. J. Schweitzer, G. Serazzi, and M. Broglia, *A survey of bottleneck analysis in closed networks of queues*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 491–508.

[28] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[29] M. Smit, B. Simmons, and M. Litoiu, "Distributed, application-level monitoring for heterogeneous clouds using stream processing," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2103–2114, 2013.

[30] A. Filieri, L. Grunske, and A. Leva, "Lightweight adaptive filtering for efficient learning and updating of probabilistic models," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 200–211.

[31] T. Zheng, C. M. Woodside, and M. Litoiu, "Performance model estimation and tracking using optimal filters," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 391–406, May 2008.

[32] "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2005.

[33] SAVI. (2015, June) Cloud platform. http://www.savinetwork.ca.

[34] OPERA, "Optimization, Performance Evaluation and Resource Allocator (OPERA)," 2013. [Online]. Available: http://www.ceraslabs.com/technologies/opera