Evaluating Adaptation Methods for Cloud Applications: An Empirical Study

Marios Fokaefs, Yar Rouf, Cornel Barna and Marin Litoiu
Department of Electrical Engineering and Computer Science
York University
Toronto, ON, Canada
Email: fokaefs@yorku.ca, yarrouf@my.yorku.ca, cornel@cse.yorku.ca, mlitoiu@yorku.ca

***Abstract*—Web software systems generally reside in highly volatile environments; their incoming traffic may be subject to sharp fluctuations from reasons that cannot always be captured or predicted. Cloud computing provides a solution to this problem by offering flexible resources, like containers, which can be quickly and easily scaled according to the current workload needs. Automating this process is a key aspect for the management of modern web software systems, and there is a plethora of methods to implement autonomic management systems. In this work, we review three of these methods, a threshold-based approach, a control-based approach and a model-based approach. We design and run a number of experiments for all three systems with different workloads to evaluate their ability to manage the software system and how well they do so. Our experiments were conducted on the Amazon EC2 cloud with Docker containers.**

***Keywords*-self-adaptive systems; cloud computing; containers; control theory; performance models;**

I. Introduction

With the advent of cloud computing as the dominant paradigm to support web-based, distributed software systems, and later the emergence of containers and microservices as advancements in virtualization and modular software, the ability to flexibly manage cloud topologies in order to offer high quality services with reduced costs became a reality. This also raised the need for efficient and sophisticated management systems, which will achieve the goals of high quality and low cost automatically, but also consistently. A number of methods have been proposed and used to implement autonomic management systems (AMS), but it is not always evident, which adaptation method is better for what settings, systems or workloads.

Regardless of the particulars of each AMS, it is generally accepted that they should follow the MAPE-K reference architecture [1]. According to this architecture, the AMS starts with a monitoring module, which is responsible of gathering metrics to comprise the state of the managed system. These metrics will be passed on to the analysis module, which will identify any off measurements indicating a potentially problematic case. In case there is a problem, the planning module will assess the situation and will propose the best possible action to address the issue. The planned actions are then forwarded to the execution module, which has access to the software system and the supporting infrastructure and eventually applies the corrective actions. Different AMS implementations and adaptation methods may focus on different modules from the MAPE-K architecture, usually on the analysis and planning as the monitoring and execution modules are third-party tools available by the cloud or the software provider.

In our work, we consider three types of AMS:

1) A **threshold-based** AMS implemented using predefined thresholds on observed metrics and predefined corrective actions. This is one of the most popular methods currently in practice, mainly thanks to its simplicity, its general applicability and its basic efficiency. In practice, the developer defines *a priori* thresholds on performance metrics. If current measurements deviate from these thresholds, it is an indication for a problem. As a response, the AMS takes a predefined action tied to the particular threshold violation (e.g., add/remove container).
2) A **control-based** AMS implemented with a PID controller [2]. This type of controllers is generally domain agnostic, meaning that the controller is not actually aware of the nature of a problem or of the adaptive action. Its sole purpose is to correct errors in the system's observable state by applying specific adaptive actions. This type of AMS traverses both analysis and planning phases of MAPE-K.
3) A **model-based** AMS implemented using a simple mathematical model to linearly represent the system's performance. The model is a set of functions that calculates the utilization of the system's infrastructure by the current traffic and it can estimate the necessary size of the infrastructure to serve this traffic according to certain performance goals. This AMS focuses mostly on the planning phase; when a problem is identified (by any method), the model will determine the exact corrective action to be planned. This means that the action may differ every time according to the problem, making the planning a dynamic process.

Subsequently, we design a set of experiments to evaluate the three AMS and answer the following research questions:

- **RQ1:** *Which method can most effectively maintain the performance goal?* In this case, we use quantitative metrics to determine if each method maintains the performance of the managed system and to what degree. We focus on the error from the given goal and we examine whether there is any significant difference between the three methods.

- **RQ2:** *Which method can most efficiently maintain the performance goal?* Here, we don't want to see if a method simply achieves the goal, but more importantly how well it can do so. We rely on quality metrics to see how *fast* the goal is achieved, how *fast* the system converges to the goal and becomes stable and how *robust* the manager is to sudden and high fluctuations in the workload.

All experiments were executed on Amazon EC2 with Docker containers using a variety of workloads on a 3-tier web application. This paper contributes one of the first, to the best of our knowledge, comprehensive study on AMS for containerized applications. The chosen AMS represent the state-of-the-practice, an AMS offered by all public cloud providers, and the state-of-the-art, two AMS product of research. The study evaluates not only their effectiveness but also their effectiveness by presenting qualitative metrics. Additionally, the execution of our experiments on a real environment with cutting-edge technology gives strength to our conclusions. We kept the workloads and the subject application relatively simple and controlled to allow for a closed experimental setup, which would allow for concrete conclusions. Open-end experiments are the natural extension of this study.

II. Related Work

Adaptive systems have been proposed to automatically manage web applications and react to change. An adaptive system is a system capable to function properly, within parameters defined by the Service Level Objective (SLO), without human intervention [1]. The system is capable to extract data from the environment where the web application resides (using a series of sensors), analyze it (identify problems that might prevent the application to function optimally or within parameters), create an adaptation plan (if necessary) and implement it. The web application and the resources it uses become the *managed resources*, while the rest of the system are part of the *application manager*. Architecturally, autonomic systems follow the Monitor-Analyze-Plan-Execute (MAPE-k) loop sugested by IBM [1].

The multi-tenant nature of the cloud, that allows multiple independent applications to share the same hardware, makes cloud-deployed applications more challenging to manage [3]. The simplest type of adaptation strategy that can be designed is one using policies, which are essentially event-condition-action (ECA) rules: *when **event** happens, if **condition** is true, then execute **action***. These rule-based systems have been investigated to some extent [4], [5]. In practice, events are usually triggered when particular performance metrics cross predefined thresholds.

At the heart of an autonomic system is the decision-making process on when adaptation is required, in other words, how to identify the location of the problem and how to optimally determine the type and quantity of resources that need to be added or removed. To analyze the data and create an action plan, some authors turned to models. Zahorjan et al. [6], Eager and Sevcik [7], Lazowska et al. [8], and Reiser and Lavenberg [9] have presented methods to analyze a system from a performance point of view. Balbo and Serazzi [10], and Litoiu et al. [11] have additionally considered how the potential structure of the workload may influence the performance of the deployed system, how bottlenecks shift when the workload mix changes and when the resources become saturated. A method to uncover the worst workload mix and the minimum population required to saturate a system is presented by Barna et al. [12].

Current state-of-the-art cloud environments implement and promote elasticity by offering software services for the automatic scaling of a cloud topology. More specifically, Amazon [13] offers autoscaling capabilities in conjunction with EC2 and CloudWatch, its monitoring service. The developer can set up CloudWatch alarms to go off when particular metrics go beyond specified thresholds (above or below the threshold) and corresponding adaptive actions are triggered to add or remove servers. Alternatively, if the workload pattern is known, the developer can set up *scaling schedules* to adapt the topology even without CloudWatch. OpenStack [14] offers a similar service as a result of the collaboration between its monitoring service, Ceilometer, and its orchestration service, Heat. In both cases, the scaling policies are threshold-based and result in over-simplified adaptive actions (i.e. add/remove servers). Additionally, the services can react only to threshold violations for metrics that the respective monitoring services can measure. Given that the aforementioned monitoring services were primarily designed for billing purposes, they fall short in capturing important metrics, such as response time, service throughput or application level metrics.

Beyond threshold-based and rule-based techniques, there are various proposed techniques ranging from discrete optimization algorithms to control theoretic approaches. For example, Li et al. [15] propose optimization deployments using bin packing algorithms augmented with integer programming to minimize application response time and infrastructure cost. Other approaches use control theory specific approaches such as model predictive control optimization [16] or other control methods [17], [18], [19]. Depending on the problem and the cost function to be achieved, but also taking into account other criteria, such as maintainability, evolution, cost of development, elasticity designers and implementers can choose an industrial rule based approach or control and optimization based techniques [19].

Proportional-Integral-Derivative (PID) controllers were first introduced as autonomic management components for web applications by Gergin et al. [20], under the assumption that the application was multi-tier and that each tier was implemented in a cluster (i.e. multiple virtual machines performing the same task). Also, it was assumed that the service's demands for resources (e.g. CPU, memory etc.) of each tier was known and constant. In this paper, we deal with more realistic assumptions: not all tiers are clustered and we only have access to the response time of the application and to CPU utilization of the clustered tier, which we can monitor. In practice, we do not assume a model of the application or a model of the cloud as it happens with complex optimization algorithms.

III. Experimental Setup

In our study, we design six experiments to apply three types of AMS on two different workloads. All experiments are performed on a custom web application representing an e-store deployed on the Amazon EC2 cloud with Docker containers. To compare the results of the experiments and evaluate the three AMSs according to the research questions we have put forth, we use such metrics like mean absolute error, overshoot, rise time and settling time among others. In this section, we detail the setup of our experiments, first, with respect to the application and the cloud technical specifications, second, with respect to the design details of the AMSs, third, with respect to the workloads, and, finally, with respect to the evaluation metrics.

A. Subject Application and Cloud Specification

The application used as the subject in our experiments is a minimal three-tier web application [21], as shown in Figure 1. At the front sits an Apache load balancer acting as the front-end interface of the application and distributing incoming requests to the scalable middle layer. There sits a set of Tomcat servers, which host the actual application. At the back-end of the topology, there is a MySQL database. The application is a simple Java application which issues a number of different requests (select, insert, update) of variable intensity (i.e., number of records). We can control the type and intensity of the requests, which allows us to stress test specific resources (CPU, memory, disk, network) and activate particular scaling plans. In our experiments, we focus entirely on CPU saturation and regulation. The simplicity and degree of control over this application allow us the flexibility to focus on testing various AMSs without unexpected external effects from the application.

The application is deployed on Docker containers, which in turn are setup in Amazon EC2 virtual machines. More specifically, we use 4 VMs to set up the Docker Swarm cluster; a micro VM (1 vCPU, 1 GB RAM and 8 GB disk) as the Swarm manager and 3 xlarge VMs (each 3 vCPU, 16 GB RAM and 8 GB disk) as Swarm hosts, where the containers

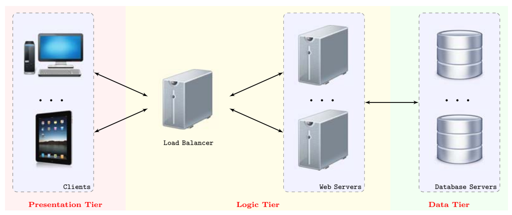

Figure 1. Three-tier architecture for the subject web application.

will actually reside. To enable container scaling, we have to define constraints on the resources that the containers will take from their hosts [22] and, so, we define the notion of 1 computational unit to be 512 MB RAM and 3.75% CPU of the host. This gives us capacity for 32 computational units per host. In order to avoid saturation on the load balancer or the database, we allow the respective containers to be larger allocating them 10 computational units each. Eventually, allocating 1 computation unit for each Tomcat container, we can have capacity for 76 Tomcat web workers in the three hosts.

B. Designing the Autonomic Management Systems

In this work, we consider three types of AMS; a control-based, a model-based and a rudimentary threshold-based. Following the MAPE-K architecture, the monitoring and executor modules, provided in our experiments by Amazon and by Docker, remain common for all three systems. In this section, we detail how each of the AMS we consider implement the analysis and planning modules.

1) Threshold-based Adaptation: Threshold-based techniques are possibly the most prevalent in industrial applications, since they are simple, requiring no special design techniques, and they are applicable in most applications and scenarios. They are used and offered by most cloud providers, including Amazon [13] and OpenStack [14] In practice, these techniques include simple IF-THEN rules. For example, *if average CPU utilization is more than 80%, add 1 resource* (container or VM). Nevertheless, like any other generic method, threshold-based techniques suffer in very specific situations, for example, when we have sudden surges in the workload, or in general when we have highly irregular and volatile workloads. Another shortcoming of this method is that the adaptation strategy, which will respond to the threshold violation is static and decided on design time, e.g., add 1 VM. This may be counterintuitive, since every violation may have to be dealt in a different manner.

The quality of threshold-based techniques lies on what thresholds will be chosen. In this work, we adopt an empirical approach to find when the web cluster (Tomcat servers) are overutilized or underutilized. The thresholds we eventually chose for the average CPU utilization of the web cluster are 80% and 60%. We saw that beyond 80% any additional request starts having a non-linear effect on response time causing great increases and thus significantly deteriorating the quality of the application. On the other hand, we saw that generally below 60% we can remove at least one container and redistribute the load that it would have received to the rest of the cluster without affecting their levels of utilization. In-between these values, response time is well maintained and there are no signs of quality deterioration. Concerning the adaptation strategy, we set up the technique to simply add or remove a single container when the thresholds are crossed.

2) Control-based Adaptation: Control theory has been used to regulate the behaviour of software and computing systems [17], [23]. Among other solutions, Proportional-Integral-Derivative (PID) controllers have been exceptionally popular to certain domains [2] and have been used in software systems as well [20], [24], mainly due to their simplicity, but also their effectiveness. Practically, the PID controller is the function shown in Equation 1. The function receives as input the error, $e(t)$, between the observed value, y, of a metric and the set goal, y^{goal}. The error is processed in three parts; the proportional (with coefficient K_p), corresponding to the current error, the integral (with coefficient K_i), corresponding to the cumulative past error, and the derivative (with coefficient K_d), corresponding to estimation about future error. The three parts are processed and aggregated to produce the *input* to the controlled system, in other words the adaptive action and its intensity necessary to counter the different types of error. In the context of our work, the adaptive action is adding or removing containers on the web (middle) layer of our application, its intensity corresponds to the number of containers we will add or remove and the error is calculated on the average CPU utilization of the web cluster.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{1}$$

The design process of a PID controller includes the calculation of the function coefficients, K_p, K_i, K_d. Although there is a number of methods to tune the coefficients, both automatic and manual [2], we opted to follow an empirical approach. We profiled our application and measured the effect that adding or removing containers have on the CPU utilization of the cluster, as well as the deviation from the set goal for the CPU utilization that can be caused by fluctuations in the incoming traffic. Eventually, we converged to the following values: $K_p = 0.14$, $K_i = 0.0002$ and $K_d = 0$. From these values, it is implied that we put more importance in correcting the latest error, less so on accumulated past errors and none at all on future errors. The last point is because variations in workload may be largely unpredictable and we cannot afford the controller taking actions that will prove invalid shortly after. Finally, the set goal for the average CPU utilization was set to 70%, which is the median of the range used in the threshold-based adaptation.

3) Model-based Adaptation: Performance models are a crucial component in software system design and quality assurance. Their ability to accurately capture the performance of the system under different conditions along with the ability to simulate the actual system and try various scenarios make them a powerful tool in managing their behaviour, maintaining their quality and, eventually, designing adaptation strategies and reaction plans to changes in the environment, including workload fluctuations or infrastructure variations. Many models have been designed and used both in practice and research, including non-linear model [25], favoured for their high accuracy in capturing the system, but also linear ones [26], [18], [23], preferred for their simplicity and analytic efficiency.

In our work, we have opted for a simple linear model. The first step is to find what is the capacity of each container, in other words, how many requests per second a single container can handle before it exceeds the utilization of 70% (around the median of our threshold range). Empirically, for our application and its infrastructure, we measured the average CPU utilization for different workloads and different number of containers, we calculated the nominal workload to increase utilization by 1% and then extrapolated this value for 70% utilization to find the container's capacity. Eventually, the capacity was given by the average of all the measurements for the different workloads. The calculation of capacity is formally given by Equation 2, where λ_i is the arrival rate in measurement i, U_i is the average CPU utilization in measurement i and n is the number of measurements.

$$capacity = \frac{\sum_i^n \frac{\lambda_i}{U_i} \times 0.7}{n} \tag{2}$$

For example, if in one measurement we have 2.9 requests per second generating 30% of utilization, Equation 2 will give us that the capacity of the container is approximately 6.8 requests per second.

Having determined the capacity of a container, we can now continue with defining the adaptation strategy. Scaling, in this case, is triggered exactly as in the previous method, by checking the thresholds. However, if the thresholds are crossed, we no longer add/remove a statically determined number of containers, but rather consult the model with Equation 3 to find how many containers can cover the current arrival rate given the previously calculated capacity.

$$Containers = \left\lceil \frac{\lambda}{capacity} \right\rceil \tag{3}$$

The problem with this method is that the arrival rate, λ, for closed systems, depends on the response time; the higher the response time, the less requests the users will be able to send and vice versa. Therefore, when the cluster is saturated we may not be receiving the highest workload possible and, thus, underestimating the situation with respect to our scaling decisions. To alleviate this situation, we calculate the actual traffic by first estimating the number of users currently using our system using the leftmost part of Equation 4, where N is the number of users, R is the response time and T is the users' average think/process time. Assuming a constant think time, we calculate the number of users for the currently observed response time and arrival rate. Using this number of users and the rightmost part of Equation 4, we calculate the actual rate that these users can send requests in a non-saturated cluster, by replacing the current response time with a more desired value. The new λ is the actual arrival rate and we use this one in Equation 3 to find the number of containers that we need.

$$N = \frac{1000 \times \lambda}{R + T} \Leftrightarrow \lambda = \frac{N \times (R + T)}{1000} \tag{4}$$

For example, let us assume that we receive 24.13 requests per second and our system responds on average in 430ms with the users having a think time of 500ms. This means that we have approximately 22 users. If we want to aim for a response time around 100ms, under these conditions the 22 users can issue about 37.14 requests per second. With these values, Equation 3 informs us that we need 6 containers to cover this demand. It is obvious that if we had used the original arrival rate, we would have used only 4 containers risking further saturation of the cluster.

C. Workloads

We subjected the aforementioned AMSs to two workloads. In the first workload, there are sharp increases or decreases in the number of users and by extension to the arrival rate. The fluctuations are so sharp, so as to create the need to change the infrastructure with more than one container in one adaptive action. Moreover, the sharp changes come at different intensities, so the number of containers may differ from case to case. After each fluctuation there is a short period with no change to the workload, so that the system is allowed to rest and for the adaptive action to take effect and we can clearly observe its impact. We call this workload *sharp* and it has about 400 data points.

The second workload represents a more realistic traffic for a web application. It slowly increases in the beginning and in the same pace decreases close to the end, while in the middle there are shorter and smoother fluctuations. There are no rest periods between the changes in the workload. The workload is designed to resemble the traffic of a web application, i.e., an e-store during a day. We call this workload *smooth* and it has about 800 data points.

D. Evaluation Parameters

In order to compare the three AMSs in all six experiments with respect to how effective they are in adapting the software system and how efficiently they do so, we use certain statistical metrics and methods to evaluate their effectiveness and quality metrics borrowed from control theory [2] to evaluate their efficiency. Due to the dynamic and real-time nature of the software system and the impact from the workload volatility, we had to redefine some of the details of the latter metrics, but the spirit is generally the same. The definition of all the metrics is as follows:

- *Mean Absolute Percentage Error:* We use MAPE to evaluate how accurate and how effective the adaptations each AMS performs as a response to changes in the workload. Given that two of the three AMSs (control and model) have a fixed setpoint, but the threshold-based AMS has a range of setvalues, in order to make the comparison fair, we measure all MAPEs according to a range (between 60% and 80% of CPU utilization). We call the 60-80 range as the *steady-error* zone. Therefore, when the average utilization is within the range, the error is 0, and when it is not, the error is calculated from the closest threshold value. In order to compare if the three methods have different effectiveness and gauge the significance of this difference, we applied the Student t-test to the three MAPEs, pairwise. Given the large size of our samples (more than 400 points for the sharp workload and more than 800 points for the smooth workload), we can assume that our samples follow the normal distribution and thus the use of the Student t-test is possible.
- *Raise time* is defined as the number of iterations from the moment a sharp change in the workload is introduced until the first iteration the utilization enters the 60-80 range. This metric shows how well the AMS judges the situation and issues that adaptive action, which will correct the system's behaviour in the fastest way possible.
- *Overshoot* is defined as the maximum error from the moment utilization crosses the setpoint of 70% until the setpoint is crossed again from the other direction and between all iterations that the utilization stayed above or below 70%. This metric shows how much the AMS may overestimate or underestimate the situation.
- *Settling time* is defined as the number of iterations from the moment a sharp change in the workload is introduced until the moment utilization enters the steady-error zone and stays there for at least 3 iterations. This metric includes the ability of the AMS to react fast but also bring the system back to an eventually stable state.

IV. Validation Experiments

Figures 2 and 3 show the results for our experiments on the sharp workload and on the smooth workload respectively. In each figure, the top plot shows the workload in terms of number of users, the next plot shows the response time of the system, the third plot shows the average CPU utilization of the web cluster and the bottom plot shows the number

Table I
EXPERIMENTAL RESULTS

AMS/workload	Total # containers	Avg CPU	Response time	MAPE	Rise time	Overshoot	Settling time
Threshold (sharp)	8157	67.98	184.25	22.24	8.7	13.61	14
PID (sharp)	7867	70.73	211.17	8.95	3.9	18.89	9.85
Model (sharp)	6860	77.72	222.86	14.71	9.25	17.82	13.7
Threshold (smooth)	18877	70.78	104.24	1.49	–	–	–
PID (smooth)	19122	70.02	107.63	1.97	–	–	–
Model (smooth)	15269	85.33	106.74	9.73	–	–	–

of containers used in each iteration of the experiment. In all plots in both figures, the blue lines correspond to the threshold-based AMS, the red lines to the PID controller and the green lines to the model-based AMS. Table I[1] summarizes the results of all experiments.

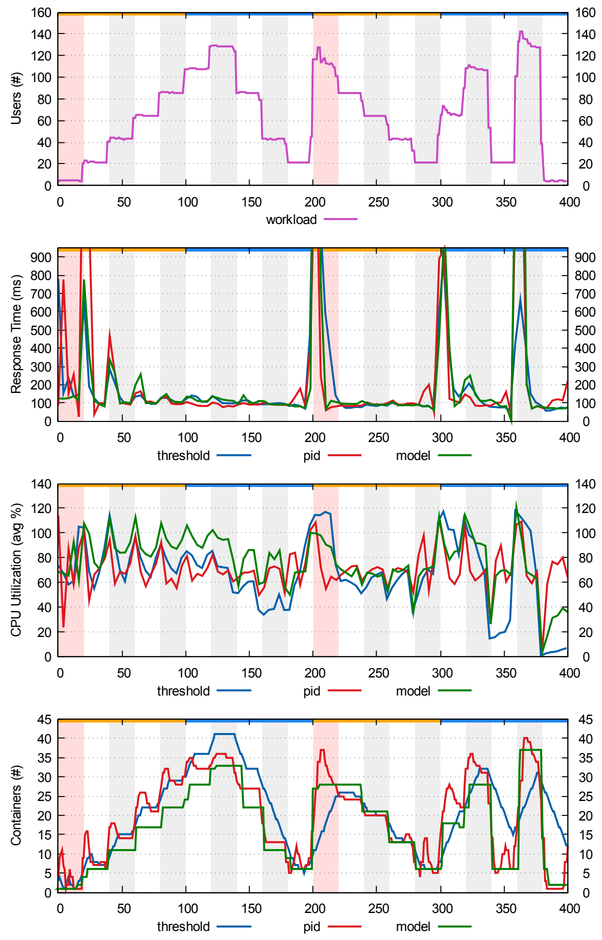

Figure 2. Comparison between the three AMSs for the sharp workload.

[1]The complete dataset of the experiments can be found at https://goo.gl/VuOYe5.

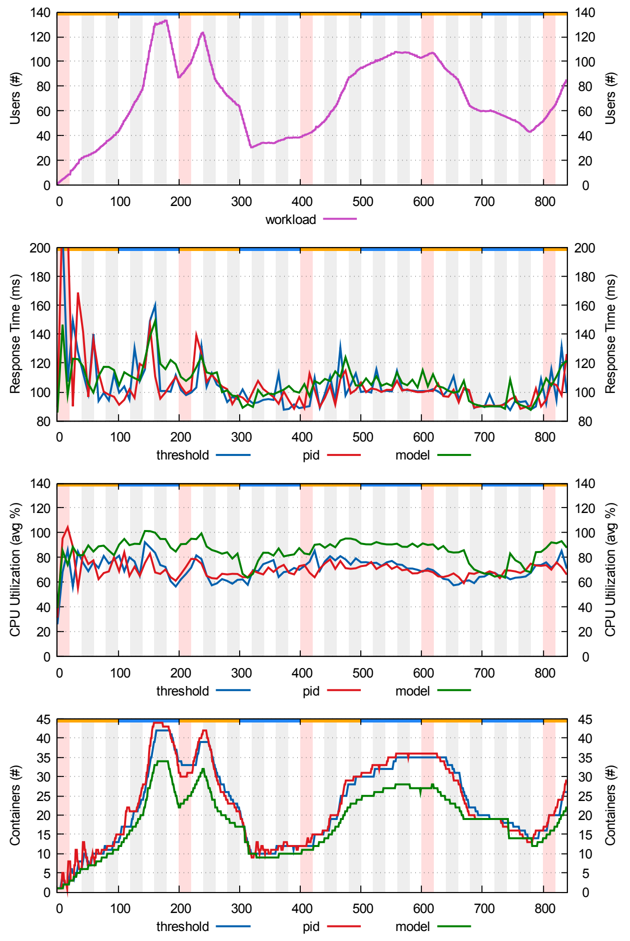

Figure 3. Comparison between the three AMSs for the smooth workload.

A. Effectiveness

In order to answer the first research question (**RQ1**), we have to examine if each AMS achieved the defined goal, how well it did so and what actions it employed in the process. As it can be seen from Table I, with respect to the sharp workload, all three AMSs stayed within the steady-error zone, on average, with the threshold-based AMS closer to the lower bound as it employed more containers in total, thus achieving lower response time, while the model-based AMS stayed closer to the upper bound demonstrating the highest response time, but with the least number of containers, among the three. However, the MAPE metric shows that the threshold AMS deviated the most from the steady-error zone with 22.24%. The pairwise Student t-tests showed that the differences in MAPE between all three AMSs are statistically significant. This means that with respect to this metric the PID AMS was the best among the three. In addition, this AMS stayed closer to the setpoint of 70% utilization using a medium number of containers and achieving a respectable response time.

For the smooth workload, the threshold AMS and the PID AMS demonstrated similar ability with respect to maintaining the CPU utilization as well as the response time, using comparable number of containers. The similarity is also demonstrated in the low MAPE index for both AMSs. In fact, the corresponding Student t-test showed that the difference is not in fact statistically significant. The low MAPE index can be justified by the smoothness of the changes in the workload, compared to the previous case. Conversely, the model AMS did not manage to maintain the CPU utilization within the steady-error zone (the only case out of the six experiments) and demonstrated a relatively higher MAPE index. We also see that this AMS uses a significantly lower number of containers. In retrospect, we found that this was due to the model overestimating the capacity of the containers. On one hand, this is because we used a relatively small number of samples to find the capacity of a container, possibly not capturing the whole spectrum of the resource's capabilities. On the other hand, we reduced the capacity to a single average value to facilitate the efficiency of the calculations, while it is known that cloud system's demonstrate a highly non-linear behaviour in their performance. The impact of the single capacity is evident in both workloads, but it is most prominent in the smooth one, because of its length. It is part of our future plans to extend this study to more types of models and evaluate their capabilities in a similar manner. Overall, there was no clear "winner" between threshold and PID in the smooth workload, however PID demonstrated a robustness to the differences between the two workloads, achieving the goal in both cases.

B. *Efficiency*

To answer the second research question (**RQ2**) we calculated the three control quality indexes, as they were defined before, for the three AMSs. It is important for these quality metrics that a short rest period follows changes in the workload, so that the adaptive action is allowed time to take effect and for the system to reach a stable state. This would not have been possible for the smooth workload, where the change is continuous, and, therefore, we calculated these indexes only for the sharp workload. The results in Table I show that the PID AMS responds to a change in the fastest way possible, requiring less than 4 iterations to reach the setpoint. However, this comes at a cost; in order to be fast, the AMS adds or removes a large number of containers to fix the deviation in CPU, which causes an overshoot of about 18.89% on average. This is also evident at the bottom plot of Figure 2, where as it can be seen, at the beginning of every sharp change, the PID AMS adds/removes a large number of containers, which it immediately tries to correct. Nevertheless, this correction period does not last long, or at least not longer than for the other AMSs, as it can be seen by the settling time index. Conversely, the threshold AMS was found to have exceptionally long settling time. This can be visually confirmed in the response time plot for the sharp workload (Figure 2), where the spikes in the response time observed when there is a sharp change in the workload take longer to be corrected by the threshold AMS. The reason for this is because this type of AMS responds to changes in a stepwise manner, adding or removing a single container at a time, thus it takes longer to reach the setpoint (also evident by the high rise time). Regardless of the overshoot, we can claim that PID is the best overall alternative for the smooth workload, as well. After all, PID quickly accounts for the overshoot through the settling time.

V. Conclusion

Given the volatile nature of web software systems and the flexibility offered by cloud platforms, and even more by containerized infrastructures, autonomic management systems play a crucial role in facilitating administrators and developers. In this work, we presented a comparative study between three popular choices for AMSs, including threshold-based, control-based and model-based solutions. We applied the three AMSs to a simple three-tier web applications deployed on the Amazon EC2 cloud with Docker containers, on two different workloads. We compared the subject management systems with respect to their ability to achieve the performance goal we set, but also by judging the quality of the produced solutions, how fast, accurate and stable they were. Our findings showed that the control-based AMS using PID is a near-optimal solution and generally applicable across different workloads both with respect to the performance goal and the quality of the adaptation.

In the future, our plan is to extend our study with more workloads and more methods to implement AMSs. Additionally, we plan to study optimization techniques and more complex control-based techniques and eventually explore the possibility of hybrid adaptation solutions. Other possible dimensions to extend our work is towards more complex web applications with more scalable tiers, like data analytics applications. '

References

[1] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2005. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf

[2] K. J. Aström and R. M. Murray, *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.

[3] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010. [Online]. Available: http://dx.doi.org/10.1007/s13174-010-0007-6

[4] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking time-varying parameters in software systems with extended kalman filters," in *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2005, pp. 334–345.

[5] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Cluster computing*, vol. 12, no. 1, pp. 1–15, 2009.

[6] J. Zahorjan, K. C. Sevcik, D. L. Eager, and B. I. Galler, "Balanced job bound analysis of queuing networks," in *SIGMETRICS '81: Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1981.

[7] D. L. Eager and K. C. Sevcik, "Performance bound hierarchies for queuing networks," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 99–115, 1983.

[8] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queuing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[9] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queuing networks," *J. ACM*, vol. 27, no. 2, pp. 313–322, 1980.

[10] G. Balbo and G. Serazzi, "Asymptotic analysis of multiclass closed queuing networks: multiple bottlenecks," *Performance Evaluation*, vol. 30, no. 3, pp. 115–152, 1997.

[11] M. Litoiu, J. Rolia, and G. Serazzi, "Designing process replication and activation: A quantitative approach," *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1168–1178, 2000.

[12] C. Barna, M. Litoiu, and H. Ghanbari, "Autonomic load-testing framework," in *Autonomic Computing, 2011. International Conference on*, ser. ICAC '11. New York, NY, USA: ACM, June 2011, pp. 91–100. [Online]. Available: http://dx.doi.org/10.1145/1998582.1998598

[13] Amazon, "Autoscaling," https://aws.amazon.com/autoscaling/.

[14] Openstack, "Heat: Openstack Orchestration," https://wiki.openstack.org/wiki/Heat.

[15] J. Z. Li, M. Woodside, J. Chinneck, and M. Litoiu, "Cloudopt: multi-goal optimization of application deployments across a cloud," in *Proceedings of the 7th International Conference on Network and Services Management*. International Federation for Information Processing, 2011, pp. 162–170.

[16] H. Ghanbari, M. Litoiu, P. Pawluk, and C. Barna, "Replica placement in cloud through simple stochastic model predictive control," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 80–87.

[17] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback control of computing systems*. John Wiley & Sons, 2004.

[18] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 299–310. [Online]. Available: http://dx.doi.org/10.1145/2568225.2568272

[19] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 716–723.

[20] I. Gergin, B. Simmons, and M. Litoiu, "A decentralized autonomic architecture for performance control in the cloud," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 574–579.

[21] M. Fokaefs, C. Barna, and M. Litoiu, "Economics-driven resource scalability on the cloud," in *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2016, pp. 129–139.

[22] M. Fokaefs, C. Barna, R. Veleda, M. Litoiu, J. Wigglesworth, and R. Mateescu, "Enabling DevOps for Containerized Data-Intensive Applications: An Exploratory Study," in *Proceedings of the 2016 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2016.

[23] D. Arcelli, V. Cortellessa, A. Filieri, and A. Leva, "Control theory for model-based performance-driven software adaptation," in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, ser. QoSA '15. New York, NY, USA: ACM, 2015, pp. 11–20. [Online]. Available: http://dx.doi.org/10.1145/2737182.2737187

[24] C. Barna, M. Fokaefs, M. Litoiu, M. Shtern, and J. Wigglesworth, "Cloud adaptation with control theory in industrial clouds," in *Cloud Engineering Workshop (IC2EW), 2016 IEEE International Conference on*. IEEE, 2016, pp. 231–238.

[25] C. M. Woodside, T. Zheng, and M. Litoiu, "Performance model estimation and tracking using optimal filters," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 391–406, 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.30

[26] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Feedback-based optimization of a private cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 104–111, January 2012. [Online]. Available: http://dx.doi.org/10.1016/j.future.2011.05.019