Improving Rule-Based Elasticity Control by Adapting the Sensitivity of the Auto-Scaling Decision Timeframe

Demetris Trihinas, Zacharias Georgiou, George Pallis, Marios D. Dikaiakos

Department of Computer Science,
University of Cyprus
{trihinas, zgeorg03, gpallis, mdd}@cs.ucy.ac.cy

Abstract. Cloud computing offers the opportunity to improve efficiency with cloud providers offering consumers the ability to automatically scale their applications to meet exact demands. However, "auto-scaling" is usually provided to consumers in the form of metric threshold rules which are not capable of determining whether a scaling alert is issued due to an actual change in the demand of the application or due to short-lived bursts evident in monitoring data. The latter, can lead to unjustified scaling actions and thus, significant costs. In this paper, we introduce AdaFrame, a novel library which supports the decision-making of rule-based elasticity controllers to timely detect actual runtime changes in the monitorable load of cloud services. Results on real-life testbeds deployed on AWS, show that AdaFrame is able to correctly identify scaling actions and in contrast to the AWS auto-scaler, is able to lower detection delay by at least 63%.

1 Introduction

Cloud computing is dominating the interests of multiple business domains revolutionizing the IT industry to the point where any person, with even basic technical skills, can access via the internet, vast and scalable computing resources by shifting IT spending to a pay-as-you-use model [1]. For small businesses and startups, this well-established argument is sound. Cloud computing eliminates capital expense of buying hardware and diminishes costs for configuring and running on-site computing infrastructures of any size [2]. Nevertheless, driving cloud adoption is *elasticity*, that is the ability of cloud services to acquire and release dedicated resources to meet current demand [3].

Albeit, while elasticity is one of cloud computing most-hyped features, the reality does not necessarily measure up to cloud providers' promises. For instance, application traffic from sudden user demand can explode rapidly, and the need for immediate scalability to address demands comes with many impediments. Cloud providers, such as AWS, offer "auto-scaling" by automatically provisioning VMs when certain user-defined metric thresholds are violated. Metric thresholds are usually reactive and rule-based in the IF-THEN-ACTION format (e.g., IF cpuUsage $>$ 75% THEN addVM) [4]. However, auto-scaling is challenging, especially when determining whether a scaling alert is issued due to actual change in the demand of an application, or due to sudden and short-lived (e.g., few seconds) spikes on highly sensitive monitoring data (e.g., cpu usage). The latter may resort to "ping-pong" effects where resources are provisioned and de-provisioned rapidly, but most importantly are billed although real demand does

not exist [5]. Thus, *rapid scaling could, in fact, end up being detrimental resulting in unwanted charges*. On the other hand, delaying to determine an actual change in the application monitorable load by extending the ruling to include a time window that the scaling alert must persist (e.g., AWS default timeframe is 5min), inhibits the possibility of a severe performance penalty affecting the overall application quality-of-service. In contrast to rule-based auto-scaling, a number of interesting and more advanced approaches have been proposed to offer better elasticity control based on machine learning and control theory [6] [7] [8] [9]. However, cloud providers, for the time being, refrain from embracing such approaches as they suffer from practical limitations that derive from the complexity of the algorithmic process in a fully automated environment or the assumption that users have a priori knowledge of optimal parameter configuration.

In this paper, we introduce AdaFrame, a library for cloud provider rule-based and reactive auto-scalers to improve elasticity control, by supporting the decision-making process to timely detect actual runtime changes in the statistical properties of the monitorable load of cloud services. To achieve this, our library employs an online, low-cost and probabilistic algorithmic process based on runtime change detection which allows for elasticity controllers to reduce the possibility of falling victims to ping-pong effects without the need to resort to large decision timeframes which inquire significant performance penalties to cloud applications and their owners. In our evaluation with two real-world testbeds deployed on AWS, we show that AdaFrame is able to correctly notify of when scaling actions should be executed, and in comparison to AWS auto-scaling, AdaFrame is able to lower detection delay by at least 63%.

The rest of this paper is structured as follows: Section 2 elaborates the motivation. Section 3 introduces our library and the algorithmic process of the approach. Section 4 presents a comprehensive evaluation in real-life settings, while Section 5 presents the related work. Finally, Section 6 conclude this paper.

2 Motivation

Fostered by autonomic computing concepts, "auto-scaling" is now a fundamental process for market leading cloud service providers. This is commonly implemented as a decision-making problem, where resource allocation for an application consists of periodically monitoring the application load, the current allocated resources (e.g., number of VMs) and based on some scaling policy, decide to (de-)allocate resources in order to maintain the performance as close as possible to a target performance. Rule-based scaling policies are very popular among cloud providers and their consumers as the simplicity and intuitive nature of these policies make them very appealing.

These scaling policies are expressed with **`IF <Expr> THEN <Action>`** rules. In particular, consumers are usually not restricted to the number of rules they can define, while each rule is comprised of an expression (`<Expr>`) and scaling action (`<Action>`). The expression defines the target metric of interest and the relation which will trigger the scaling policy. Triggering the scaling policy will satisfy the desired action which is pre-selected by the consumer from a finite set of permitted scaling actions supported by the cloud provider. For example, let us consider a web service processing requests for local business outlets in the location defined as a parameter in the served request. In this scenario the load is compute-bound and we assume that two scaling

policies are defined. The first policy, if triggered, will add a new virtual instance to the deployed cluster when the average cluster CPU utilization surpasses 80%, while the second policy, will remove a virtual instance if CPU utilization drops below 20%.

```
RULE$1 := IF AVG(cpuUsage(clusterID)) > 80% THEN addVM
RULE$2 := IF AVG(cpuUsage(clusterID)) < 20% THEN removeVM
```

While the simplicity is highly evident from the above example (although selecting appropriate thresholds is a profiling challenge of its own [6]) this approach ignores the volatility of monitoring data. To be precise, monitoring data can be bursty introducing sudden and short-lived spikes which may cause control oscillations. A control oscillation, often dubbed as *a "ping-pong" effect, is defined as the phenomenon where an unexpected and short-lived burst in the monitoring data (even a single datapoint) triggers a scaling action which will be subsequently annulled when the system stabilizes* [10]. As an illustrating example, let us consider the aforementioned web service with its CPU usage depicted in Figure 1. From this, we observe that a sudden burst from a background cleanup process immediately triggers `RULE$1`, thus a VM is added, and after the VM is provisioned and fully integrated to the deployment, the CPU utilization, quickly, drops to the point where `RULE$2` is triggered, returning the application deployment to the previous state. Hence, a single spike in the monitoring data causes a series of elastic control actions, accounting for direct and indirect costs, as users are charged for these unjustified actions and the provisioned resources (e.g., for AWS a VM booted even for 1s is charged for whole hour), and the application may suffer performance-wise if data movement and coordination is required while in a transitioning state.

To compensate with control oscillations, cloud providers (e.g., AWS), extend the rule-based decision model to include a time window, denoted as a decision timeframe, where the scaling policy expression (`<Expr>`) must evaluate to `true` and persist for the length of the timeframe. In particular, AWS, provides its consumers with the option to set a decision timeframe, with the default option being `5min`, while other options are also available [11]. The assumption here is that if a threshold violation persists in time, as depicted in Figure 2, then a scaling action is justified and the larger the decision timeframe, the smaller the possibility of introducing a ping-pong effect. Obviously, absolute guarantees can never be given unless auto-scaling is disabled. However, the downside with introducing a large decision timeframe, even the default AWS option of `5min`, is that a significant performance penalty may occur while waiting for a scaling action to be triggered. Therefore, a new challenge rises: *How can a rule-based elasticity controller scale an application deployment without resorting to large decision timeframes in order to avoid ping-pong effects due to sudden bursts in monitoring data?*

3 The AdaFrame Library

To address the aforementioned challenge, we have designed and developed the AdaFrame library. AdaFrame supports the decision-making process of rule-based elasticity controllers so as to timely detect, and notify the elasticity controller, of actual runtime changes in the statistical properties of monitoring data originating from elastic cloud services. To achieve this, the AdaFrame library employs an online, low-cost and proba-

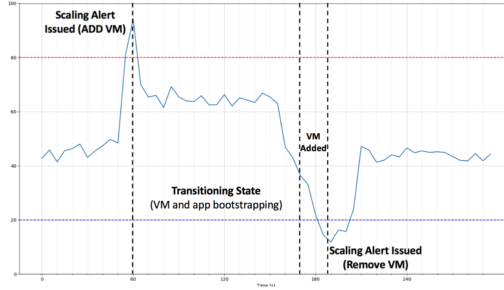

Fig. 1: Ping-Pong Effect on Monitored CPU Utilization of a Web Service

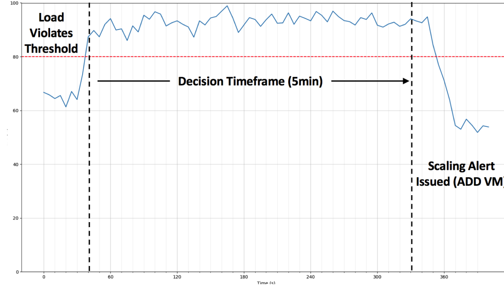

Fig. 2: Auto-Scaling Decision Timeframe to Reduce Control Oscillations

bilistic algorithmic process based on change detection which allows for elasticity controllers to reduce the possibility of falling victims to ping-pong effects without the need to resort to large decision timeframes which inquire significant performance penalties to cloud applications and their owners. Figure 3 depicts the AdaFrame library incorporated in an auto-scaling control loop, resembling AWS, where one can observe that it does not alter the decision-making process, or the control loop in general, as AdaFrame simply acts as a support proxy notifying the Scaling Policy Evaluation of the elasticity controller when a scaling action should be triggered (workload behavior changes) and when not (workload spikes). This completely removes the need of a fixed decision timeframe. In turn, offline profiling to detect a (near-) optimal decision timeframe is not needed, as fixed "optimal" values are only relevant if the properties of the metric stream hold for the entire lifespan of the application which is an assumption far from reality for today's complex cloud applications. In the following, we provide a detail description of the two basic components comprising AdaFrame: the `Adaptive Monitoring Estimation Model` and `Runtime Change Detection`.

3.1 Adaptive Monitoring Estimation Model

At first, let us define a monitoring stream $M = \{d_i\}_{i=0}^{n}$ published by a monitoring source on a cloud application to an auto-scaling entity, as a large sequence of datapoints d_i, where $i = 0, 1, ..., n$ and $n \to \infty$. Each datapoint d_i is a tuple $(s_{id},\ t_i,\ v_i)$ described, at the minimum, by a source identifier s_{id}, a timestamp t_i and a value v_i. We base our approach such that the estimation model is maintained in constant time and space O(1), a requirement for rule-based elasticity control. While AdaFrame supports

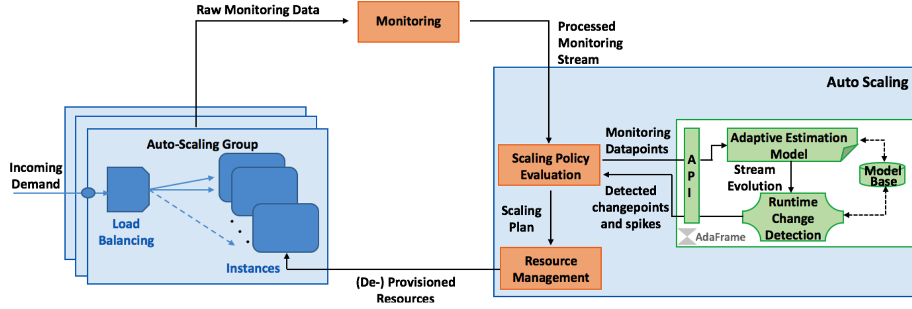

Fig. 3: AdaFrame Incorporated in Auto-Scaling Process

model parameterization, as input it only requires from the user to provide his/her confidence guarantees $\delta \in [0, 1]$, denoting the probability with which estimated datapoints are approximated from sensed datapoints. Now, when a datapoint is made available to the auto-scaling by the monitoring tool, it is passed through `AdaFrame API` to the `Adaptive Monitoring Estimation Model` so as to update the current monitoring stream evolution by using a moving average, denoted as μ_i. This will give an initial estimation for the next datapoint value, denoted as $\hat{v}_{i+1}$. Moving averages provide smoothing and one-step ahead estimations for single dimensional timeseries such as the target metric referenced in the `<Expr>` of a rule-based scaling policy. They are easy to compute, though many types exist, and can be calculated on the fly with only previous value knowledge. A cumulative moving average for streaming data is the Exponential Weighted Moving Average (EWMA), $\mu_i = \alpha\mu_{i-1} + (1 - \alpha)v_i$, where a weighting parameter α, is introduced to decrease exponentially the effect of older values. However, the EWMA features a significant drawback; it is volatile to abrupt transient changes [12]. Thus, we propose adopting a Probabilistic EWMA (PEWMA), which dynamically adjusts the weighting based on the probability density of the given observation. The PEWMA acknowledges sufficiently abrupt transient changes (burstiness), adjusting quickly to long-term shifts in the monitoring stream evolution and when incorporated in our algorithmic estimation process, it requires no parameterization, scaling to numerous datapoints.

$$\mu_i = \begin{cases} v_i, & i = 1 \\ \alpha(1 - \beta P_i)\mu_{i-1} + (1 - \alpha(1 - \beta P_i))v_i, & i > 1 \end{cases} \tag{1}$$

Equation 1 presents the PEWMA where instead of a fixed weighting factor, we introduce a probabilistically adaptable weighting factor $\tilde{a}_i = \alpha(1 - \beta P_i)$. In this equation, the p-value, is the probability of the current v_i to follow the modeled distribution of the metric stream evolution. In turn, β is a weight placed on P_i and as $\beta \to 0$ the PEWMA converges to a common EWMA[1]. The logic behind probabilistic reasoning is that the current value v_i depending on it's p-value will contribute respectively to the estimation process. In turn, if a datapoint falls inside the prediction intervals determined from the given confidence, it is labeled as "expected" or "unexpected" otherwise. Therefore, we

[1] For simplicity in our model we will consider $\beta = 1$

update the weighting by $1 - \beta P_i$ so that sudden "unexpected" spikes are accounted for in the estimation process, however, offer little influence to subsequent estimations, thus restraining the model from overestimating subsequent v_i's. In turn, if an "unexpected" value turns out to be a shift in the monitoring stream evolution, as the probability kernel shifts, subsequent "unexpected" values are awarded with greater p-values, allowing them to contribute more to the estimation process. Assuming, a stochastic and i.i.d distribution as the bare minimum for a monitoring stream, we can adopt a Gaussian kernel $N(\mu, \sigma^2)$, which satisfies the aforementioned requirements. Thus, P_i is the probability of v_i evaluated under a Gaussian distribution, which is computed by Equation 2. Nonetheless, we note that while a Gaussian distribution is assumed, if prior knowledge of the distribution is available and given by the user then only the computation of P_i must change in the estimation process.

$$
\begin{aligned}
P_i &= \frac{1}{\sqrt{2\pi}} \exp(-\frac{Z_i^2}{2}) \\
Z_i &= \frac{v_i - \hat{v}_i}{\sigma_i}
\end{aligned}
\tag{2}
$$

Moreover, in [12] we show how to compute the running variance for the PEWMA, and that the α parameter can take a wide range of values if a small imprecision can be tolerated as most of the error is absorbed by the probabilistic weighting. Thus, with the proposed model we can both estimate the monitoring stream evolution and detect and label bursts in the monitoring stream. However, as mentioned, much of this burstiness is irrelevant for diagnosis of elastic scaling. Nonetheless, significant bursts and long-term trends are useful features for cloud providers and consumers, especially for capacity planning, anomaly detection and quality control.

3.2 Runtime Change Detection

The most prominent functionality of AdaFrame is to detect changes, at runtime, in the statistical properties of the evolution of a monitoring stream, in order to reduce the need of a fixed decision timeframe to avoid ping-pong effects. To achieve this, AdaFrame incorporates change detection based on a variation of the lightweight Cumulative Sum test (CUSUM). The CUSUM, denoted as C_i, is a hypothesis test for detecting shifts in i.i.d timeseries [13]. In particular, there are two hypothesis θ' and θ'' with probabilities $P(M, \theta')$ and $P(M, \theta'')$, where the first corresponds to the statistical distribution of the monitoring stream prior to a shift ($i < t_s$) and the second to the distribution after a shift ($i > t_s$) with t_s denoting the time interval the shift/change occurs. The CUSUM is computed online via sequential probability testing on the instantaneous log-likelihood ratio given for a monitoring stream at the i-th time interval, as follows:

$$
\begin{aligned}
c_i &= \ln \frac{P(M_i, \theta'')}{P(M_i, \theta')} \\
C_{i,\{low,\ high\}} &= C_{i-1,\{low,\ high\}} + c_i
\end{aligned}
\tag{3}
$$

where low and $high$ denote the separation of the CUSUM to identify both positive and negative shifts respectively. The typical behavior of the log-likelihood ratio includes

a negative drift before a shift and a positive drift after the shift. Thus, the relevant information for detecting a shift in the evolution of a monitoring stream lays in the difference between the value of the log-likelihood ratio and the current minimum value. A decision function G_i, is used to determine a shift in the monitoring stream when its outcome surpasses a threshold h, measured in standard deviation units. The time interval at which a shift actually occurs, is computed from the CUSUM as follows:

$$\begin{gathered} G_{i,,\{low,\ high\}} = \{G_{i-1,\{low,\ high\}} + c_i\}^+ \\ t_s = \underset{j \leq s \leq i}{\arg\min}\,(C_{s-1}) \end{gathered} \tag{4}$$

In the above, $G^+ = sup(G, 0)$ and t_i is the time AdaFrame detects the shift. Now, let us consider the particular case of a monitoring stream representing the target metric of a rule-based scaling policy with the monitoring stream supposed to undergo possible shifts in its evolution. Hence, in our case, t_j is considered the time the monitoring stream current value base violates the scaling policy. In turn, we consider the evolution of the monitoring stream in its mean, modelled as the PEWMA moving average previously introduced. Thus, θ' and θ'' can be rewritten as μ' and μ'' respectively, with μ' representing the current evolution, while μ'' the output of the estimation model with $\mu'' = \mu' + \epsilon$, and ϵ denoting the estimated magnitude of change of the monitoring stream evolution. As the monitoring stream evolution is used to provide an estimation for $\hat{v}_i$, the magnitude of change is actually equal to $\epsilon = \hat{v}_i - v_i$. In turn, let $P(M, \mu')$ and $P(M, \mu'')$ be computed from Equation 2. With some calculations, c_i (eq. 4) is rewritten, as follows, to perform the decision-making with only previous value knowledge:

$$c_{i,\{low,\ high\}} = \pm \frac{|\epsilon|}{\sigma_i^2}\,(v_i - \mu' \mp \frac{|\epsilon|}{2}) \tag{5}$$

Nonetheless, the CUSUM test features two drawbacks. First, determining the actual t_s requires linear time. However, exact knowledge of t_s is not required for signalling a scaling action, as t_s is only computed after the shift is detected, with AdaFrame providing an approximate answer (t_i) which is the time it detects the change in the monitoring stream. Second, when the monitoring stream is relatively stable, and thus the stream variance is low ($\sigma_i \rightarrow 0$), the CUSUM is prone to falsely signalling changes [14]. Hence, we follow an adaptive approach where h is updated after a scaling action, based on the number of standard deviations respecting the given user-defined confidence (δ) and an optional positive value (h_{min}) is used to restrict the sensitivity of the CUSUM so as to not oscillate between low values when the monitoring stream is relatively stable.

$$h_i = \max\{h_{min},\ h(\delta)\} \tag{6}$$

4 Evaluation

In this section, we evaluate the accuracy our approach via two real-world testbeds deployed on AWS, the most popular cloud provider. We compare AdaFrame ability to detect changes in the workload and signal the auto-scaling service that a scaling action should be triggered, to (i) using AWS auto-scaling without a decision timeframe,

Algorithm 1 AdaFrame Algorithm

Input: User-provided confidence δ_i at initialization. For every update, datapoint $d(t_i, v_i)$
Output: Label datapoint d_i as "expected", "unexpected" or "changepoint"
Ensure: Monitoring stream M is attached and moving average μ is initialized
compute p– and z–value and then update estimation model
1: $P_i, Z_i \leftarrow$ probDistro($v_i, \hat{v}_i, \sigma_i$) (eq. 2)
2: $\mu_i, \sigma_i \leftarrow$ updPEWMA(P_i, v_i) (eq. 1)
label datapoint as "expected" or "unexpected" based on prediction intervals
3: **if** isDatapointExpected(δ, P_i, Z_i) **then**
4: $label \leftarrow$ `'expected'`
5: **else**
6: $label \leftarrow$ `'unexpected'`
7: **end if**
8: **if** $scaling\ alert\ triggered\ at\ t_{i-1}$ **then**
9: $h_i \leftarrow$ updShiftThres(δ, σ_i) (eq. 6)
10: **end if**
11: $c_i \leftarrow$ updLikelihood($v_i, \hat{v}_i, \mu_i, \sigma_i$) (eq. 5)
12: $C_{i,low},\ C_{i,high} \leftarrow$ updCusum(c_i) (eq. 3)
13: $G_{i,low},\ G_{i,high} \leftarrow$ updDecision(c_i) (eq. 4)
14: **if** $G_{i,\{low,\ high\}} > h_i$ **then**
15: $label \leftarrow$ `'changepoint'`
16: **end if**
17: **return** $label$

and, thus, any scaling policy violation will trigger a scaling action; and (ii) the AWS auto-scaling service with the default decision timeframe of 5min.

To integrate AdaFrame to the AWS control loop, we enabled manual auto-scaling through the AWS API. This provides us with API access to the auto-scaling service so as to immediately trigger the pre-selected scaling action once a changepoint is detected after the initial scaling policy violation occurs. We note that when running AdaFrame with "manual" scaling, auto-scaling, with 5min decision timeframe, is also enabled to see if any scaling action would be detected earlier by AWS. Also, both experiments are conducted with a tight confidence parameter of $\delta = 0.95$ and $h_{min} = 1$. In turn, as AWS monitoring metric collection limits the minimum periodicity to 60s, for the sake of a thorough evaluation we opted to integrate with AWS, JCatascopia monitoring probes [10] in order to collect data every 5s.

4.1 Testbed 1: Scaling a NoSQL Document Store

The first testbed of our evaluation is a NoSQL document store implemented by a Couchbase DB cluster. In particular, couchbase is used for the database backend of the web service described in Section 2, and thus, processes map/reduce-like data requests for local business outlets in the location defined as a parameter in the served request. Initially, we manually provision the cluster to host three database instances which is considered, for Couchbase, as the minimum number of instances for smooth operation. Each instance, and future provisioned instances, are Amazon ubuntu 16.04 LTS medium flavored AMIs (2 VCPU, 4GB Memory, 120GB disk). For this testbed, and with Couchbase cpu-bound, we select the average cpu usage as the target metric. We stress the testbed by generating a stable load of 80 req/s and increase the request rate by 30 req/s

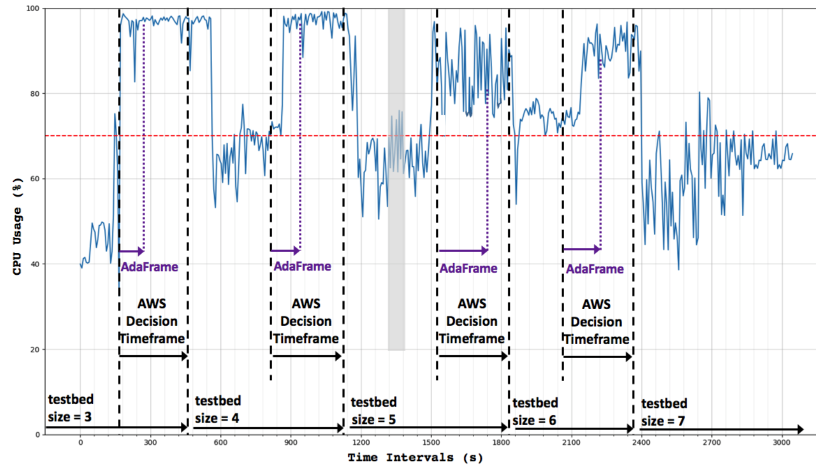

Fig. 4: Couchbase CPU Usage and Scaling Action Detection Delay

every 10 minutes in order for the testbed to scale but not be overwhelmed while the size of the testbed is small. To cope with the workload, the AWS auto-scaling service must provision a new VM, in the first case, if the CPU usage is over 75% and, in the second case, if the same scaling policy is violated but for a timespan of 5 minutes. In the case of embracing AdaFrame, a new instance is only provisioned when a changepoint is detected after the scaling policy is violated.

Figure 4 depicts the testbed CPU usage, the testbed size, the time intervals at which the scaling policy is violated for the first time and the time intervals AWS (with decision timeframe) and AdaFrame trigger each scaling action. From this, we immediately observe that *AdaFrame features the ability to correctly identifying all scaling actions, even for a monitoring stream featuring significant burstiness over the threshold, and does not trigger any false scaling action which could lead to a ping-pong effect*. Also, scaling action detection is performed by AdaFrame in a timely manner (Figure 8). Specifically, in three out of four of the scaling actions, the detection time is significantly less that half (63% less) of the AWS decision timeframe ($112s \pm 16s$) as AdaFrame quickly identifies a change in the statistical properties of the monitoring stream. Nonetheless, for the third action, due to the high volatility of the monitoring stream, AdaFrame requires more than half of the decision timeframe ($196s$), but still significantly outperforms AWS (36% less). In turn, the highlighted period of time in Figure 4 is an example where even if a smaller, but still fixed, decision timeframe (e.g., 2min) is opted for AWS to compete with AdaFrame, then a ping-pong effect will be triggered as oscillations near the threshold are evident, which justifies why ignoring the statistical properties of the monitoring stream to detect actual changes will lead to unexpected and unwanted effects.

Next, to illustrate the importance of timely detecting when to trigger a scaling action, we depict in Figure 5, the performance of the testbed in terms of throughput when a decision timeframe is used. From this, we observe that once a scaling policy violation is detected and for the span of the decision timeframe, throughput suffers and is not able to follow the workload increment. Only after a VM is added and aftergoing a (short) rebalancing phase is the testbed able to surpass the initial saturation. In contrast, *AdaFrame*

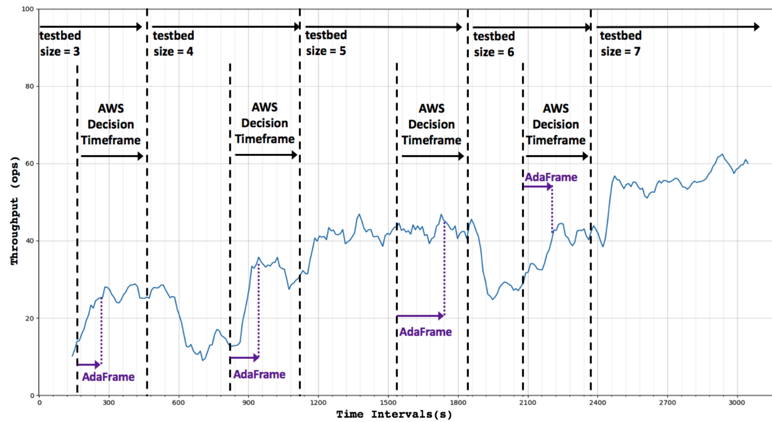

Fig. 5: Couchbase Cluster Throughput and Scaling Action Detection Delay

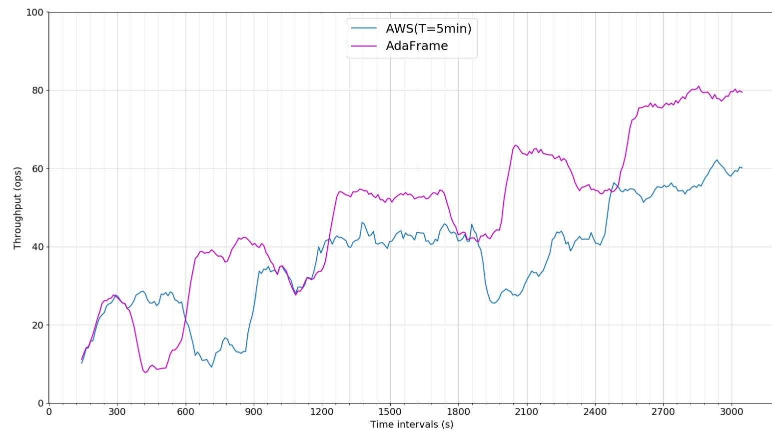

Fig. 6: Couchbase Cluster Throughput with AdaFrame vs 5min Decision Timeframe

is able to correctly and timely detect when a scaling action should be triggered (Figure 5), and we observe that throughput features a significantly larger slop and higher values are achieved with the gains increasing as the workload increases (Figure 6). Finally, we note that for visualization clarity, we omitted depicting cpu usage, throughput and time intervals at which AWS without a decision timeframe triggers a scaling action, and state that in this case correctness suffers as 7 scaling actions were triggered instead of 4 due to the high volatility of the monitoring stream near the threshold.

4.2 Testbed 2: Scaling the Business Logic of a Web Service

The second testbed of our evaluation is an Apache Tomcat cluster implementing the business logic of the aforementioned web service. This cluster was initially provisioned to host a single instance and each provisioned instance, is an Amazon ubuntu 16.04 LTS small flavored AMI (1 VCPU, 2GB Memory). In this set of experiments, we configure our workload generator to adapt the load step-size randomly in order to cause both scale

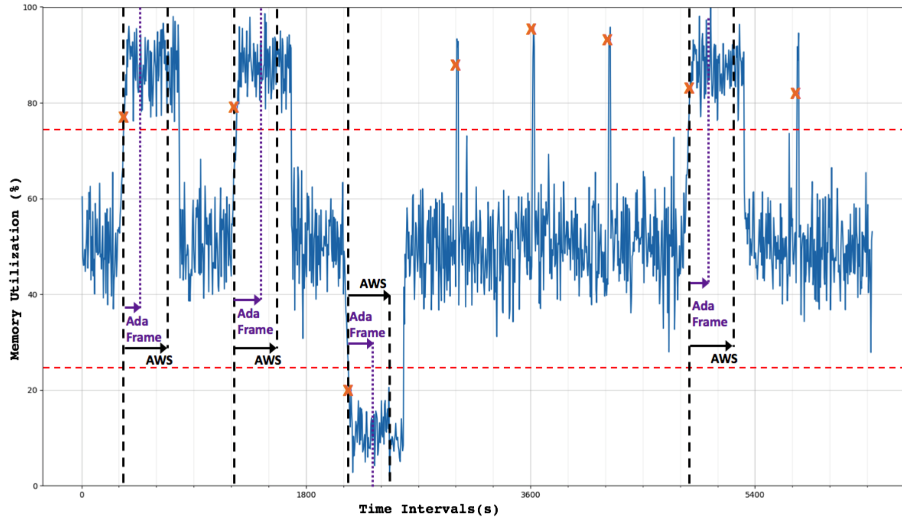

Fig. 7: Apache Tomcat Cluster Memory Utilization and Scaling Action Detection Delay

in and out actions. The number of scaling actions to perform was set to 10. Also, we configure the workload generator, to exhibit at random, 10 periods of time with bursty behavior over the defined thresholds by mixing with the workload gaussian noise to emulate the spiky behavior of a (Tomcat) cleanup background process. For this testbed, and with Apache Tomcat memory-bound, we select the average memory utilization as the targeted metric. Similar to the first testbed, we set the high threshold at 75% and add a scale-in policy to remove a VM if memory utilization drops below 25%.

Figure 7 depicts the Apache Tomcat memory utilization, the VM cluster size, the time intervals at which the scaling policy is violated for the first time and the time intervals AWS (with decision timeframe) and AdaFrame trigger each scaling action. We note that for visualization clarity, the memory plot depicts only the first 4 scaling actions. From this, we first observe that without a decision timeframe, AWS will trigger a scaling action each time the monitoring stream surpasses the defined thresholds which also includes all 10 artificially generated workload spikes. Next, we observe that by adding the 5 minute decision timeframe, AWS auto-scaling is able to restrain from wrongfully triggering a scaling action when burstiness is exhibited. However, *AdaFrame is able to achieve the same results but in contrast to using the fixed 5 minute decision timeframe, detection time for triggering a scaling action is on average* $102s \pm 21s$*, which is significantly lower and, at least, 66% less*.

5 Related Work

A number of sophisticated techniques have been proposed for elastic scaling. Almeida et al. [8] propose a *branch and bound* approach for optimally allocating resources to multi-layer cloud applications during runtime, while Tolosana-Calasanz et al. [9] propose controlling reserved resources for data processing engines by following a shared token bucket approach. A more intuitive approach is proposed by Dustdar et al. [3], defining elasticity as a complex property, having as major dimensions resource, cost

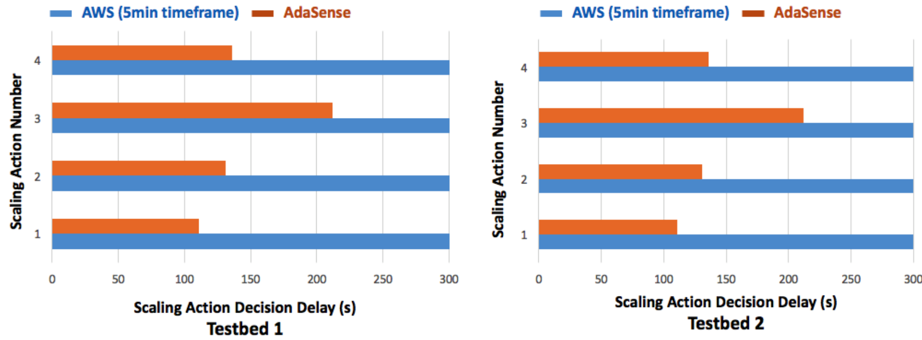

Fig. 8: Scaling Action Decision Delay per Testbed

and quality elasticity. These dimensions reflect not only computing related aspects of application operation, but also business aspects. In turn, Copil et al. [15] introduce an elasticity specification language, denoted as SYBL, which allows the definition of complex and multi-dimensional elasticity policies for rSYBL, an elasticity controller capable of managing cloud elasticity based on SYBL directives. On the other hand, Tsoumakos et al. [6] introduce an open-source elasticity control service, which models the problem of elastic scaling NoSQL databases as a Markovian Decision Process and utilize reinforcement learning to allow the system to adaptively decide the most beneficial scaling action based on user policies and past observations. Naskos et.al. [16] extend this model to resizing clusters of a single generic application hosted on virtual machines. Many queuing theory based approaches have been proposed. For instance, Urgaonkar et al. [17] models servers at each tier as a queue for representing arbitrary arrival and service time distributions. Despite the novelty in all the above approaches, cloud providers, for the time being, refrain from embracing such approaches, and prefer the simplicity of rule-based scaling, as they suffer from practical limitations that derive from the complexity of the algorithmic process in a fully automated environment or the assumption that users have a priori knowledge of optimal parameter configuration. Finally, and to the best of our knowledge, the most notable approach towards attacking ping-pong effects, is ADVISE, a framework supporting and providing "advise" to elasticity controllers to improve the decision-making process by evaluating the outcome of elasticity control actions. However, this approach is an offline approach only labelling past control actions as ping-pong effects.

6 Conclusion

We propose a library, called AdaFrame, which supports the decision-making of rule-based elasticity controllers to timely detect actual runtime changes in cloud services based on an online, low-cost and probabilistic algorithmic process. Our objective is to minimize the time for detecting changes in the targeted monitoring streams of user-defined scaling policies originating from elastic cloud services. AdaFrame can be incorporated in the auto-scaling control loop towards maximizing the profit generated taking into account the monetary cost of the resources as well as the revenue generated by the workload. Results on two real-life testbeds deployed on AWS show that AdaFrame outperforms the AWS auto-scaler and adapts quickly to workload changes.

Acknowledgements. This work is partially supported by the European Commission in terms of Unicorn 731846 H2020 project (H2020-ICT-2016-1).

References

1. Loulloudes, N., Sofokleous, C., Trihinas, D., Dikaiakos, M.D., Pallis, G.: Enabling interoperable cloud application management through an open source ecosystem. IEEE Internet Computing **19**(3) (May 2015) 54–59
2. Willcocks, L., Venters, W., Whitley, E.A. In: Cloud in Context: Managing New Waves of Power. Palgrave Macmillan UK, London (2014) 1–19
3. Dustdar, S., Guo, Y., Satzger, B., Truong, H.L.: Principles of elastic processes. IEEE Internet Computing **15**(5) (Sept 2011) 66–71
4. Trihinas, D., Sofokleous, C., Loulloudes, N., Foudoulis, A., Pallis, G., Dikaiakos, M.D.: Managing and Monitoring Elastic Cloud Applications. In: 14th International Conference on Web Engineering. ICWE 2014 (2014)
5. Copil, G., Trihinas, D., Truong, H., Moldovan, D., Pallis, G., Dustdar, S., Dikaiakos, M.D.: Evaluating cloud service elasticity behavior. International Journal of Cooperative Information Systems (2015)
6. Tsoumakos, D., Konstantinou, I., Boumpouka, C., Sioutas, S., Koziris, N.: Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. IEEE International Symposium on Cluster Computing and the Grid (2013) 34–41
7. Lolos, K., Konstantinou, I., Kantere, V., Koziris, N.: Elastic resource management with adaptive state space partitioning of markov decision processes. CoRR **abs/1702.02978** (2017)
8. Almeida, A., Dantas, F., Cavalcante, E., Batista, T.: A branch-and-bound algorithm for autonomic adaptation of multi-cloud applications. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. (May 2014) 315–323
9. Tolosana-Calasanz, R., ngel Baares, J., Pham, C., Rana, O.F.: Resource management for bursty streams on multi-tenancy cloud environments. Future Generation Computer Systems **55** (2016) 444 – 459
10. D. Trihinas, G. Pallis and M. D. Dikaiakos: Monitoring Elastically Adaptive Multi-Cloud Services. IEEE Transactions on Cloud Computing **4** (2016)
11. Amazon Auto-Scaling Policies. `http://aws.amazon.com/ec2/`
12. Trihinas, D., Pallis, G., Dikaiakos, M.D.: AdaM: an Adaptive Monitoring Framework for Sampling and Filtering on IoT Devices. In: IEEE International Conference on Big Data. (2015) 717–726
13. Luo, Y., Li, Z., Wang, Z.: Adaptive cusum control chart with variable sampling intervals. Computational Statistics & Data Analysis **53**(7) (2009) 2693 – 2701
14. Trihinas, D., Pallis, G., Dikaiakos, M.: ADMin: adaptive monitoring dissemination for the internet of things. In: IEEE INFOCOM 2017 - IEEE Conference on Computer Communications (INFOCOM 2017), Atlanta, USA (May 2017)
15. Copil, G., Moldovan, D., Truong, H.L., Dustdar, S.: SYBL: An Extensible Language for Controlling Elasticity in Cloud Applications. In: 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. (2013) 112–119
16. Naskos, A., Stachtiari, E., Gounaris, A., Katsaros, P., Tsoumakos, D., Konstantinou, I., Sioutas, S.: Dependable horizontal scaling based on probabilistic model checking. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. (May 2015) 31–40
17. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., Wood, T.: Agile dynamic provisioning of multi-tier internet applications. ACM Trans. Auton. Adapt. Syst. **3**(1) (March 2008) 1:1–1:39