Auto-scaling of Web Applications in Clouds: A Tail Latency Evaluation

Mohammad S. Aslanpour*†, Adel N. Toosi*, Raj Gaire† and Muhammad Aamir Cheema*
*Faculty of Information Technology, Monash University, Clayton, Australia
†CSIRO's Data61, Australia

***Abstract*—Mechanisms for dynamically adding and removing Virtual Machines (VMs) to reduce cost while minimizing the latency are called auto-scaling. Latency improvements are mainly fulfilled through minimizing the "average" response times while unpredictabilities and fluctuations of the Web applications, aka flash crowds, can result in very high latencies for users' requests. Requests influenced by flash crowd suffer from long latencies, known as outliers. Such outliers are inevitable to a large extent as auto-scaling solutions continue to improve the average, not the "tail" of latencies. In this paper, we study possible sources of tail latency in auto-scaling mechanisms for Web applications. Based on our extensive evaluations in a real cloud platform, we discovered sources of a tail latency as 1) large requests, i.e. those data-intensive; 2) long-term scaling intervals; 3) instant analysis of scaling parameters; 4) conservative, i.e. tight, threshold tuning; 5) load-unaware surplus VM selection policies used for executing a scale-down decision; 6) cooldown feature, although cost-effective; and 7) VM start-up delay. We also discovered that after improving the average latency by auto-scaling mechanisms, the tail may behave differently, demanding dedicated tail-aware solutions for auto-scaling mechanisms.**

***Index Terms*—cloud computing; auto-scaling; tail latency; resource provisioning; performance evaluation**

I. Introduction

With the advent of cloud computing, an increasing number of services are now hosted and deployed on the cloud. Application providers often host web applications along with other applications in Virtual Machines (VMs) running in cloud service infrastructures like Amazon EC2 and Microsoft Azure to provide services to end-users. They pay the cost of Web server, Application server, Database server, Storage and so on as much as they actually use, in contrary to traditional solutions requiring considerable up-front cost [1]. The elasticity feature of cloud, by which the numbers of VMs can dynamically be added/removed using an auto-scaling mechanism, has further encouraged application providers to use cloud VMs [2]. Auto-scaling's main goal is to prevent both over- and under-provisioning of computing resources [3]. These mechanisms are utilized for many purposes including cost reduction, VM performance optimization, and most importantly guaranteeing Quality of Experience (QoE) for end-users interacting with the application [4]. One of the critical QoE metrics for the Web users is latency [5]. The latency for a user's request is calculated from the issue time to the time the response is returned back to the user.

High latency can severely impact businesses. Every 100ms increase in the page load latency decreases sales by 1% [6]. An expert from Google believes that "users really respond to speed" [7], meaning that the higher the latency the lower the visits to a Website. Therefore, studying the causes of latencies is important.

Latency optimization using auto-scaling mechanisms in terms of the *a*verage is well-understood in the cloud community. Researchers mostly take the results of averages of latencies as a major factor for evaluating their auto-scaling solutions [1]. Looking at the literature [4], [8], it is clear that the improvement of latency, especially in average, by resource auto-scaling has been rigorously investigated. However, auto-scaling using the average is not sufficient for achieving low latency.

It is reported that, while optimization solutions can reduce the average latency, one out of a thousand user requests will still suffer from an unacceptable delay [9]. In other words, when a flash crowd happens, i.e. a surge in the workload occurs, it is more likely that a number of users' requests experience very high latency (even 10X higher) even though the average latency is being optimized. These requests are called outliers. Exploring and studying flash crowds and subsequently such outliers require understanding the *Tail Latency* concept [5].

The term tail latency originates from statistics and refers to the distribution and frequency of outliers in a distribution (i.e., the values that are really far from the mean). Note that in statistics, tail refers to outliers on both sides of the mean. In this paper, we only consider the outliers with high latency since the main concern is mostly in this area [9].

The efficiency of an auto-scaling mechanism from the viewpoint of mitigating such outliers can be evaluated using tail latency metrics. It is also apparent that common solutions for average latency, although effective, cannot guarantee shorter tail latency [9]. There is a considerable recent discussion about tail latency optimization in cloud environments, ranging from task scheduling [10] to load balancing [11], [12], but the impact of auto-scaling has not been investigated yet, to the best of our knowledge. Note that the latency for outliers—with 99.9th percentile latency—is orders of magnitude worse than the average [9]. Hence, the study of tail latency for auto-scaling mechanisms deserves more attention.

In this paper, the issue of tail latency in auto-scaling mechanisms is extensively investigated. First, we answer the following **questions**:

- How does an auto-scaling system work?
- What is the tail of latency in statistics?
- What are potential components and features of the web applications auto-scaling mechanisms which can be the sources of tail latency?
- What are the solutions for these tail latency sources?

Then, we **contribute** to the literature by the following:

- Investigating and discovering potential sources of tail latency in cloud auto-scaling mechanisms,
- evaluating if the tail has different behavior than average latency, and
- conducting a series of experiments to evaluate the sources of tail latency under real-world workload.

Methodology/Roadmap. To answer the aforementioned questions, we first dissect auto-scaling mechanisms to know how they work and what are the possible causes of tail latency, thereby raising several questions about possible tail latency sources by reviewing related work (Section II); in summary, discovered sources are sevenfold: the Web requests' size, scaling interval, analyzing method, threshold tuning, surplus VM selection policy, cooldown feature, and VM start-up delay. Then, the sevenfold concerns are comprehensively evaluated through real implementation on OpenStack and utilizing Mediawiki as the Web application along with real-world Wikipedia traces as workload (Section III). Tail-related metrics such as Percentiles, Skew and Kurtosis to discover sources of tail latency are evaluated (Section III). After the evaluations, the significance of such sources are measured using statistical approaches and discussed in Section IV. Finally, we draw our conclusions in Section V.

II. Sources of Tail Latency in Auto-scaling

In the cloud environments, cloud providers, application providers, and end-users are three main stakeholders (see Fig. 1). The application provider utilizes two main components to serve the Web application: data management for balancing the load and resource management for resource auto-scaling (with four phases), which will be discussed in the following.

A. Data Management

Data management mainly depends on the type of incoming workload (e.g. transaction or batch mode) and load balancing methods. Workload for the Web applications is transactional, meaning that there are many requests that iteratively arrive and the load balancing component is responsible for submitting them to the corresponding VMs.

1) Load Balancing: Research [9] has claimed that the size of requests may be a source of tail latency; that is, they believe if requests are large and require more resource/data to be returned back to the user, they may cause tail latency. However, the impact of this matter when investigating auto-scaling mechanisms is unclear yet. *Q1: Is the size of requests a matter of concern for tail latency?*

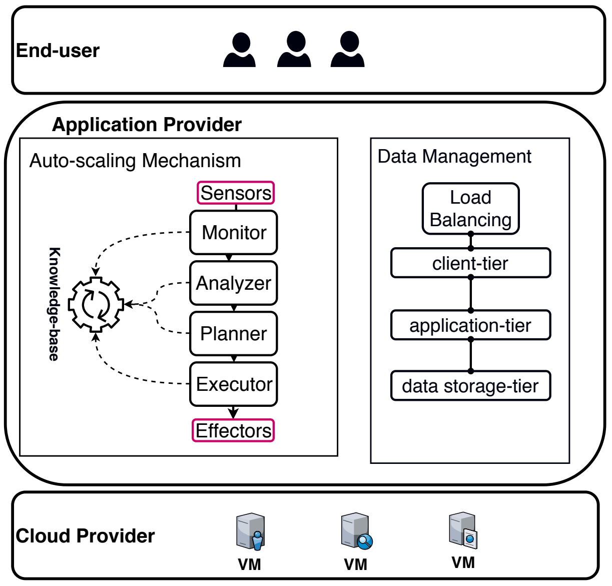

Fig. 1. The architecture of auto-scaling systems in cloud.

B. Auto-scaling

To systematically study the potential sources of tail latency in an auto-scaling mechanism for Web application, it is important to categorize necessary activities into distinguished phases to make the process understandable. MAPE-K concept introduced by IBM [13], which refers to the mechanism acting based on Monitoring, Analyzing, Planning, Executing and putting the data into a Knowledge-based repository, can perfectly fulfill this goal. In the following sections, the lifecycle of an auto-scaling mechanism is fully reviewed and the potential sources of latency are discussed.

1) Monitoring: When it comes to an auto-scaling mechanism, the first phase is monitoring which is responsible for collecting necessary information from the whole environment, ranging from the VMs' status to end-users' QoE status, through sensors. It is apparent that the more the sensors, the more the possibility for accurate decision-making in the mechanism. The sensors performance and elements involving in the monitoring demand specific investigations which are out of the scope of this work. However, running an auto-scaling mechanism, we must decide on the period of time on which the whole mechanism should be executed which is called scaling interval (SI). The SI indicates the frequency, usually in minutes, at which the auto-scaling cycle should be repeated. The monitoring interval is not necessarily equal to SI. The SI equals 1 to 12 minutes were used in recent works [14], but they never investigated as to whether this time can have any relationship with tail latency. *Q2: Is there any connection between SI and tail latency?*

2) Analyzing: In the second phase, the analyzer analyzes the data collected by the monitor to be used as decisions-making parameters. These parameters include response time, resource utilization, number of VMs, incoming load, Service Level Agreement's status, etc [15]. Some researchers who mainly focus on the decision-making, not analyzing, reactively pass the raw data (i.e. the latest observation), to the decision-making phase while others adopt proactive analyzing methods like Moving Average (MA), Weighted Moving Average (WMA), Single/Double Exponential Smoothing (SES/DES) [14], or even Artificial Neural Networks, pass the prediction of observations to the decision-maker [2], [14]. Those not considering complex/proactive analyzing justify that their solution is more agile, is independent of the previous observations, and has low overhead. Advocates of utilizing proactive methods, however, claim that they will reach more accurate and reliable decisions. Proactive methods for analyzing parameters, like response time, can predict the future growing trend for the parameter, helping the auto-scaler to provision, or at least be prepared for provisioning resources in advance, resulting in preserving QoE for end-users. *Q3: Does the way we analyze the monitored data can impact the tail latency?*

3) Planning: Having analyzed the parameters, the scaling decision-maker, i.e. planner, comes into play to decide on scaling-up or down. It evaluates if the resources are less than required or higher. This phase of MAPE is also known as resource planning or capacity planning. If the resource, i.e. VMs, are less, the planner will announce a scale-up decision, and vice versa. In horizontal scaling, we focus on the number of VMs while, in vertical scaling, the resources of a VM are manipulated, in accordance to the scaling decision. In case of horizontal scaling, the step-size parameter indicates how many VMs should be added/removed as a result of a scaling decision [2].

Importantly, to reach scaling decisions there are different methodologies including: analytical modelling, machine learning and rule-based [2]. The first two approaches can result in proactive and provident decisions. For instance, Learning Automata used by [15] and artificial neural networks used by are common, wherein the planner can regard past, current, and future status of parameters. Rule-based approaches, however, are most popular in the community wherein a set of rules, like if-else, are defined, and their conditions at every time of scaling can reactively result in a decision [16]. As examples, distinguished parameters like CPU utilization and response time, the combination of parameters through Fuzzy-based or Decision-Tree models can be evaluated. Technically, there is a need for threshold tuning to specify when we must interpret the situation as the time to make a scale-up, scale-down, or none decision. For example, if the planner sets a rule for evaluating CPU utilization, it needs a pair of predefined thresholds for scale-ups and scale-downs. This is to define conditions for triggering scaling decisions. Looking at related works [2], [17], there is no consensus on the pair of thresholds to be used, as different range of thresholds for scale-up, e.g. 70, 80, 85, or 90%, and scale-down, e.g. 40, 30, 25, 20, or 10% has been assigned. Although threshold tuning analysis was performed by [18], they did not analyze it for tail latency. *Q4: Does the threshold tuning has any impact on tail latency? How?*

4) Executing: Executor plays the role of a broker between application provider and cloud provider to perform the announced decision using provided APIs in cloud. This process is done by requesting to release or launch new VMs in horizontal scaling, or allocating or de-allocating resources, like CPU cores, to an already rented VM in vertical scaling. There are three technical points residing in this phase that potentially can be causes of tail latency: surplus VM selection policy, cooldown time, and VM start-up delay; the first is related to executing scale-down decisions and others to scale-up decisions.

Surplus VM selection policy comes to play when executing a scale-down decision and seeks for the due VM to be selected as the surplus. Assume that the application is hosted on the cloud using 8 VMs and now the executor has decided to execute a scale-down decision, so it must select a VM among eight as surplus to be released. Here, the question of which VM to select is raised. Amazon EC2 default policy takes the age of VMs into account, that is selection of the oldest [4]; in [17], selection of the youngest is proposed. Further studies discuss load-aware, selecting the VM with the lowest load to reduce failed requests, or cost-aware, selecting the VM whose renting time is equal or close to its billing period [1]. Cost-aware policy has the root to the billing cycle defined by cloud providers where they count a partial time as a full billing cycle. For instance, Amazon EC2 has an hourly billing cycle and partial usages, e.g. 10 min, are considered one hour, so a cost-aware policy seeks for the most cost-efficient VM. Studies consider only the role of surplus selection policy in either request failure or cost reduction. *Q5: Can the surplus VM selection policy influence the tail of latency?*

Cooldown time, or cooling time [2], is another feature resided in the executor to reduce the overhead of the mechanism. Since the planner may, because of inefficient design, iteratively decide on scaling-up while unnecessary, a cooldown time between decisions is required. Such decisions can stem from the lack of awareness about the VM start-up delay. For instance, assume the planner decides on scaling-up now and the VM is to be launched 6 min later (because of start-up delay); at the next SI, if shorter than 6 min, the planner will again decide on scaling-up because it is unaware that the requested VM is about to be launched. Hence, application should keep track of requested VMs. Such iterative decisions can result in contradictory decisions, negatively affecting the cost, so intelligent mechanisms are equipped with a feature called cooldown which is responsible for neutralizing the subsequent scale-up decision. Cooldown has a time, usually equal to SI or VM start-up time, and during this period prevents the executor from any scale-up decision. While having positive impact on the cost, its effect on tail latency needs to be investigated. *Q6: Is cooldown time a source of tail latency?*

VM start-up delay is a technical obstacle in the real-world usage of cloud resources [19]. When the executor asks the cloud provider to launch a VM, the VM requires to pass initial stages like allocating physical resources, launching,

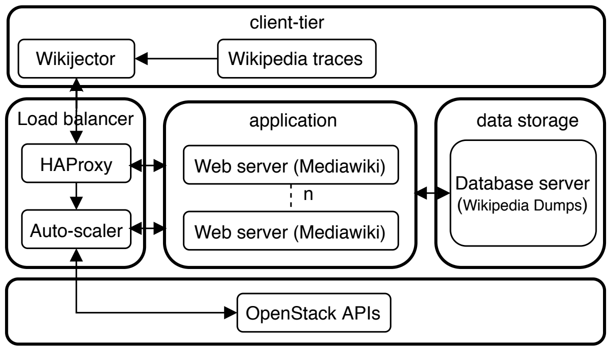

Fig. 2. The architecture of our custom test-bed.

and deploying the application to be then ready and available. This is called VM start-up delay which is either neglected in simulation-based research [20], or assigned a static value as in [21]. In a real-world scenario, this delay is dynamic and considered the time of day the VM is requested, the step-size, the configuration of the VM, the renting type (on-demand, reserved, or spot), etc. For example, the start-up delay for a Large VM with 8 cores in Amazon Ec2 is by far longer than for a Micro VM with just 1 core. This is the case for step-size as well; cloud resource allocator can surely find resources for 1 VM faster than for 3 VMs, for instance. Start-up delay is rarely considered to be dynamic, e.g. in [1], in simulation-based experiments. Regardless of the experimental test-bed, real cloud or simulator, the role of this technical obstacle in tail latency has never been investigated. *Q7: Can VM start-up delay, either static or dynamic, be a source of tail latency?*

III. Tail Latency Evaluation

The discussed auto-scaling features are evaluated in the following sub-sections, to investigate their possible impact on tail latency. We conduct extensive experiments using an OpenStack platform which is deployed on a cluster of physical hosts in DisNet laboratory at Monash University. Architecture of the test-bed is shown in Fig. 2.

In cloud provider layer, OpenStack is providing VMs in different sizes and allowing us to launch/destroy VMs using its APIs from our auto-scaler to communicate with. Each communication requires the auto-scaler to establish the authentication and receive a valid token to be able to use OpenStack APIs.

In application provider layer, a three-tier web application (Client, Application, and Data storage) is deployed as well as the auto-scaling mechanism. Each tier is deployed on a separated VM while the VMs in application-tier are supposed to be horizontally scaled up/down. The client-tier is directly in connection to the end-user where it receives HTTP requests and distributes them among the VMs in application-tier to be processed and then delivers responses to end-users. The HAProxy (version 1.6) installed on an m1.medium VM is acting as the load balancer. The load balancer distributes requests in a load-aware manner (prioritizing the VM with the least connections, leastconn). In the application-tier, an installed Linux, Apache, MySQL, PHP (LAMP) Server hosts the Mediawiki (version 1.30) application on its Apache Server (version 2.4.18). Since the auto-scaler is supposed to instantiate VMs representing application-tier, we created a snapshot of our pre-configured application server which has the Mediawiki already installed and requires the *m1.small* flavor at least. Utilizaing *m1.small* VMs with low computing power results in more auto-scaling decisions, whereby the auto-scaler will be well-stressed. Technically, each newly instantiated VM in the application tier needs to reconfigure its Mediawiki by adding its IP to the configuration file in advance. This is done by placing a bootstrapping shell script in the snapshot which is being run at start-up time. Another shell script running at the startup is the CPU utilization logger that at every single second will collect the system-level CPU usage for all cores using the *sar* tool (i.e. $sar -u\ 1\ 1$) and will append it to a file to be read by the monitoring phase of the auto-scaler. The data storage-tier is realized using an *m1.xlarge* VM storing the dumps of English Wikipedia in its MYSQL, containing roughly 7 million wiki pages.

Our auto-scaler program, written in Java (available on GitHub[1]), will continuously monitor requests' rate, and response time of the application-tier VMs measured using HAProxy on the client-tier VM by calling HAProxy APIs. Also, CPU utilization of the VMs are collected. Note that CPU usage is highly variable and making decisions only by relying on just one observation would not be justifiable. To make the measurements more accurate, firstly, for each VM, the average of all cores are measured; secondly, at least the average value for last 30 seconds is included; and thirdly, the monitor passes the CPU usage average for all VMs to the analyzer phase. After monitoring, the analyzing and planning phases are called to analzye the monitored data and to make a scaling decision with the step-size of 1 using load-aware rule-based algorithms. Upper and lower thresholds are assigned for CPU and workload according to the collected profiling data. Then, executor performs scale up/down decisions by calling the OpenStack APIs. The executor also re-configures HAProxy to update, i.e. add/remove records, the list of application-tier VMs after each decision. Once a scale up decision is performed, a new VM, which is to be added to the application-tier, is created from a snapshot, having installed the Mediawiki.

At the top layer, an *m1.medium* VM having Wikibench [22] installed is emulating end-users' behavior. Wikibench is a Java-written benchmark tool which reads the traces of access to Wikipedia website and sends to the Mediawiki application. In the literature, experiments with the workloads at the scale of seconds and minutes are mostly performed [23] while the web applications with sever fluctuation would not be well-stressed at that scales. We collect a 4-hour trace containing 2% of all requests sent to the Wikipedia website worldwide (roughly 1.4 million HTTP requests) to ensure the reliability

[1]https://github.com/aslanpour/auto-scaling-in-openstack

of our results. The characteristics of all Linux-based VMs (Ubuntu Xenial 16.04) used in this research are summarized in Table II.

Having prepared the test-bed, the tail analysis of seven distinct features of auto-scaling is performed here. Metrics of importance, which are considered here, include: the percentile, average, standard deviation (SD), skewness, and kurtosis of latency. To make our analysis more understandable, we categorize the percentiles and name them as: Low, medium, high, and very high percentile (see Table I). The very high percentile has recently been considered very important, yet challenging to minimize, for cloud latency optimization. To supplement the discussions, parameters like CPU utilization, cost and total scale up decisions are evaluated as well (by briefly reporting without visualizing).

TABLE I: Naming each measure of percentiles.

Name/Degree	Low	Medium	High	Very High
Percentile Rank	50*th*	75*th*	90–95*th*	99–100*th*

TABLE II: The characteristics of the VMs

Flavor	**VCPUs**	**Memory (GB)**	**Disk (GB)**
m1.small	1	2	20
m1.medium	2	4	40
m1.xlarge	8	16	160

A. Effect of Size of Web Requests

Initially, we investigate the tail latency changes when the size—or data requirements—of user's requests varies. It has been claimed that larger requests can cause more tail latency [12] in general. To evaluate the consequences of requests' size, we manipulate Wikibench so it substitutes all HTTP request addresses within the Wikipedia traces with a specific *tiny*, *small*, *large*, and *very large* request per experiment, requesting 5000, 18000, 40000, and 90000 bytes of data, respectively. The larger the requests are, the higher the tail latency will probably be. The necessity of such evaluations for auto-scaling mechanisms has been elaborated in [1].

According to Fig. 3 and 4, the general pattern of the results demonstrates that larger requests deteriorate the tail of latency although the mechanism is taking care of all requests with the same configuration. This is because the larger requests do not only require fetching more data from the data storage-tier, but also demand more processing time in the application-tier. The *tiny* requests are answered promptly. The only exception in this pattern was seen where the low, medium and high percentiles for *large* requests are higher than that of *very large*. This might be because of the processing requirements of the HTTP requests which can be CPU-intensive. In other words, assuming two requests, both requesting the same amount of data but one is connecting to the data storage-tier and another to both data storage- and application-tiers, being CPU-intensive as well as data-intensive. The second would certainly sustain more latency. Although out of the scope, it is worth mentioning that the *tiny* and *small* requests increased the number of scaling decisions than others. Looking at Fig. 4, the increasing trend, once the requests' size goes up, is seen as well. Noteworthy, we clearly see that the tail (Fig. 3) and the average (Fig. 4) can behave differently, particularly for the *large* requests, reflecting the paramount importance of tail specific evaluations. More interestingly, the SD's values are also behaving differently when compared to the tail and average.

FINDING 1: Large requests influence the tail latency, especially when they are CPU-intensive as well, since they demand the communication with the whole application tiers.

B. Effect of Scaling Interval

Scaling interval (SI) is a challenging parameter to be set because each application workload and even VM type may require specific interval period to be set for its auto-scaling mechanism [2]. Leaving aside the way we can set the optimal value for SI, here we study the tail-related impact of different intervals including: *short-term* (2 min—close to the VM start-up delay), *mid-term* (4 min), and *long-term* (8 min). This range of SI is based on the VM start-up times studied by [19].

Fig. 5 and 6 illustrate the impact of SIs on latency. Once the SI increases, the latency in terms of the tail, avg and SD goes up, regardless of rare exceptions. This trend becomes stronger when it comes to very high percentiles. Weak performance of *long-term* SIs has the root to the late reaction of the auto-scaler to workload variation. Also, when the auto-scaler makes irrelevant decision by mistake, the consequences would be alleviated by *short-term* SIs while *long-terms* fail to. More than 30 scaling decisions were made for the *short-term* SI while it was only 15 and 10 decisions for the *mid-term* and *long-term* SIs, respectively. Furthermore, late reaction of auto-scalers with *long-term* SIs is revealed by investigating the average CPU utilization where the results showed 70, 80, and 84% of CPU usage for *short-*, *mid-*, and *long-term* SIs, respectively. This witnesses that with *long-term* SIs the workload is more accumulated on VMs resulting in a higher CPU usage. Analyzing the shape of distributions, they were rather equally long-tailed (positively-skewed), but the kurtosis value reveals the reason why *long-term* SIs failed. Kurtosis for the *short-term* SI ($= 3$) represents a distribution with the tendency to be more light-tailed while for *mid-terms* and *long-terms* was 7 and 8, respectively, representing heavy-tailed distributions with many outliers who are the main cause of their weakness in very high percentiles. While the long-term SI is representing the largest latencies in terms of the tail, it stands in the second rank when it comes to the average latency evaluations (see Fig.6). This is true in terms of the SD evaluations, particularly indicating the need for tail-specific solutions for SI selections. It teaches us that when selecting the SI, the engineer has to decide on its main performance goal. If they plan to avoid the long tail latencies, they have to avoid the *long-term* SIs; otherwise, if they plan to avoid the longer average latencies, they obviously have to avoid the *mid-term* SIs. One may argue that why not always selecting the *short-term* SIs? The clear answer is that the overhead they impose on the mechanism can be significant. Moreover, technical designs may force the engineers to adopt the longer SIs.

FINDING 2: SIs can be a major cause of latency variation, although valid reasons for other applications generating batch workload, not transactional, are worth investigating.

C. Effect of Analyzing Method

The parameters contributing to the scaling decision-making can either reactively obtain their values from the current monitored data (instant analysis), or proactively obtain a bunch of monitored data from both current and past observations. The real impact of each method on the tail latency is still an unanswered question. Simply considering the latest value, for example current CPU utilization or request rate, is less aware of the fluctuations. However, it is expected that more intelligent data analyzing methods like MA, WMA and SES methods can positively improve the auto-scaling, thereby reducing the tail latency. Hence, this issue is investigated here to compare the impact of raw data (just considering the latest observation) and some analytical methods on the tail of latency.

Looking at Fig. 7 and 8, analyzing methods like MA, WMA, and SES fulfilled lower latency in terms of the average, SD, and tail, respectively with minor exceptions compared to the simple approach that does not consider these methods. The difference gets very large when it comes to very high percentiles. Simple analyzing method expands the tail, and the data confirms the importance of adopting more complex analyzing methods, instead of just considering current variable observations. Looking deeper at the mechanism's performance, we observed 22, 14, 11, 8 scaling decisions when using the SIMPLE, MA, WMA and SES methods, respectively. This once again confirms that the more complex the analysis method is, the less scaling decisions it has to take, subsequently lees incorrect decisions will happen. Such an improvement happens even when our selected analyzing methods are not that intelligent. This would be expected to increase even more when more complex statistical and machine learning methods are used as analyzers. Our further analysis revealed that the direction and shape of the distributions are positively-skewed (long-tailed) and heavy-tailed ($Kurtosis > 0$) under all analyzing methods.

FINDING 3: The more accurate the analyzing method is, the less tail and even average and SD for latency will be.

D. Effect of Threshold Tuning

Thresholds for parameters contributing to auto-scaling decision would be other possible sources of tail latency if not properly tuned. Here, we investigate the role of three commonly-used [1], [4], [15] strategies including *loose*, *moderate*, and *tight* thresholds. According to our profiling data each VM can handle a maximum of 10 requests per second simultaneously with an acceptable performance. The *loose*, *moderate* and *tight* thresholds are set as 7-4, 8-3, and 9-2 running requests per VM, respectively, for scale-up and scale-down decisions. For instance, *tight* thresholds result in scale-up decisions if the observed value for requests' rate is higher than 9 and result in scale-down if it is lower than 2.

Fig. 9 and 10 demonstrate that the *tight* strategy reduces the latency at low and medium percentiles but the improvement is not significant. However, the *loose* and *moderate* strategies can remarkably reduce the tail of latency when it comes to high and very high percentiles. This success is largely because of the early reaction to the growing behavior of scaling parameters like requests' rate. For instance, when the incoming workload is growing, the *loose* strategy would result in a scale up decision once the requests' rate is more than 7 while a *tight* strategy would wait until it reaches 9 requests. Scaling up after reaching 9 requests per VM would make the under-test VMs saturated and subsequently suffering from the delay in the starting-up of the requested VM increases the tail of latency. The average and SD of latency also support this claim that the *tight* strategy imposes higher tail latency. It is noticeable that, while effective in tail, the *loose* one has repercussions in terms of the cost of renting VMs as, being sensible to workload variation, it acts to provision and deprovision more resources. The *loose* strategy resulted in provisioning 19 VMs while this was only 8 for the *tight*, which resulted in 15% increase in the cost by applying *loose* strategies. Fourteen VMs were provisioned when using the *moderate* thresholds. Note that this analysis demonstrates that looser tuning of thresholds is more tail efficient, not merely the studied thresholds, as these are selected according to profiling investigations. Again, the direction and shape of the distribution for response times were of positively long-tailed and heavy-tailed families.

FINDING 4: A conservative, i.e. *tight*, threshold tuning can considerably increase the tail of latency, particularly the very high percentiles, while *loose* tuning decreases it.

E. Effect of Surplus VM Selection Policy

Surplus VM selection comes to play once a scale-down decision is to be executed. This selection may affect the performance of auto-scaling mechanism, at least in terms of the cost which is confirmed by [1]. However, its effect on the tail latency needs more investigation. Here, four distinguished and commonly used policies are studied including: *the Youngest*, *Resource-aware*, *Load-aware*, and *Cost-aware*. To calculate the relative cost for VMs, the Amazon on-demand pricing model is used [4].

The obtained results, in Fig. 11 and 12, overall witness that this feature can considerably impact the tail, particularly for the high and very high percentiles. This selection can technically influence the latency if aware of the load on VMs as the load will be postponed as a result of scale-down actions. Hence, it is obvious that the *Load-aware* policy performs better. The *Resource-aware* policy is also indirectly aware of the load and reduces the tail. This is because the *Resource-aware* is selecting the VM with the lowest CPU utilization. The CPU utilization here represents the number

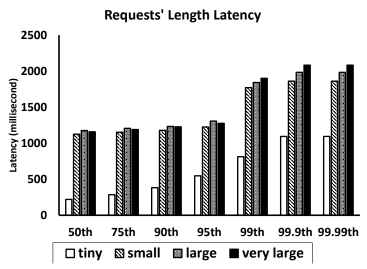

Fig. 3. The impact of requests' length on tail latency.

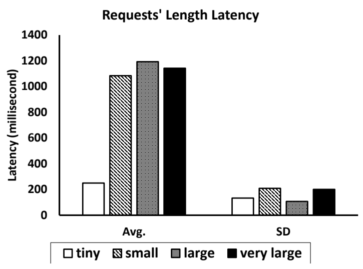

Fig. 4. The impact of requests' length on average and SD (Standard Deviation) latency.

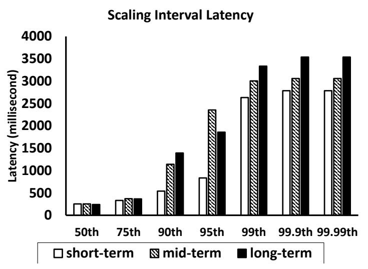

Fig. 5. The impact of scaling intervals on tail latency.

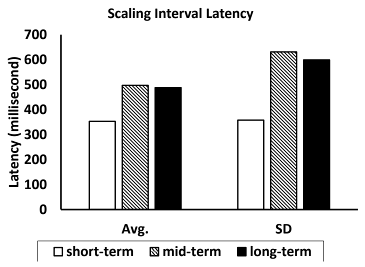

Fig. 6. The impact of scaling intervals on average and SD (Standard Deviation) latency..

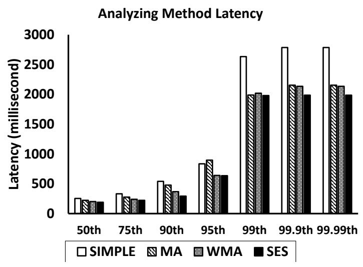

Fig. 7. The impact of analyzing methods on tail latency.

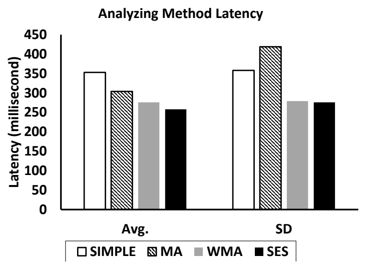

Fig. 8. The impact of analyzing methods on average and SD (Standard Deviation) latency.

of running requests as the major application running on the Web servers is the Mediawiki. The *Cost-aware* and *the Youngest* fail to reduce the tail as they are unconscious of the load on VMs although the *Cost-aware* managed to fulfill its promises as cost-reduction. Another observation is that while the tail (in very high percentiles) and SD evaluations confirm the same order for the policies' performance, the average evaluation's results (see Fig. 12) are behaving differently, indicating the need for average specific solutions for the optimization in this component. Precisely, in order for the tail of latency optimization, the *Load-aware* and *Resource-aware* work well while for the average of latency optimization the *Load-aware* and *the Youngest* policies work well. The SD is behaving the same as the tail in this experiment.

FINDING 5: Surplus VM selection policies can influence the tail of latency when not aware of the load on the VMs.

F. Effect of Cooldown Time

We here compare auto-scaling mechanisms with enabled and disabled cooldown times to investigate cooldown impact on tail latency. It is expected that enabled cooldown can be cost-saving, but its effect on the tail latency is unknown.

The results in Fig. 13 and 14 show that cooldown-enabled mechanisms can deteriorate the tail, average and SD of latency. The tail of latency increased remarkably when cooldown feature was enabled, i.e., ON. This gap becomes wider for very high percentiles, around 4x longer tails. The reason for this is that once a surge in the incoming workload occurs, the mechanism may need to perform subsequent scale-ups although the cooldown feature prevents. Hence, until it allows, although profitable for the application provider, the users' requests may suffer from resource shortage. Noticeably, as expected, an enabled cooldown time could reduce the total number of scale actions: 23 and 15 actions for the cooldown OFF and ON, respectively. Note that we considered a minimal value for our cooldown component that mitigates only the next possible scale-up decision, i.e. the cooldown value is equal to 121 seconds while SI is set to 120. Hence, referring back to the SI investigations (Section III-B), it can be claimed that increasing cooldown time value always increases the tail.

FINDING 6: The cooldown time feature in auto-scaling mechanisms is a major cause of tail latency.

G. Effect of VM Start-up Delay

Last parameter to be analyzed for the tail latency is the start-up time of a VM. Once the start-up time/delay becomes longer, the user's requests must wait longer to have the requested VM launched. Studies show that this delay time is approximately between 2 and 10 minutes while some simulation-based research neglect considering this delay time [17]. Hence, we here examine three types of VM start-up delays included: a) no start-up delay whereby the start-up delay is omitted; b) static start-up delay whereby a fix value, here 5 min, is set for the start-up time; and c) dynamic start-up delay which, in addition to the static

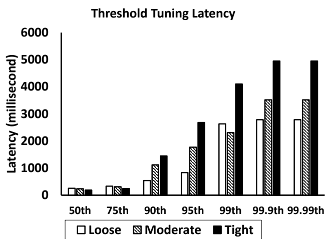

Fig. 9. The impact of threshold tuning on tail latency.

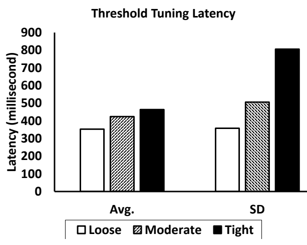

Fig. 10. The impact of threshold tuning on average and SD (Standard Deviation) latency.

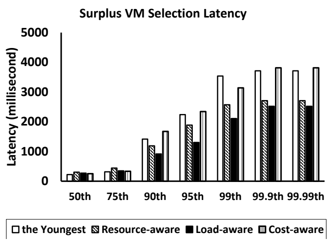

Fig. 11. The impact of surplus VM selection policy on tail latency.

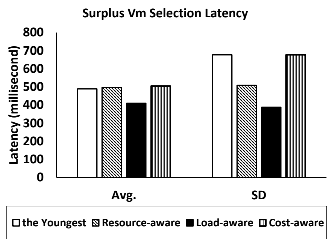

Fig. 12. The impact of surplus VM selection policy on average and SD (Standard Deviation) latency.

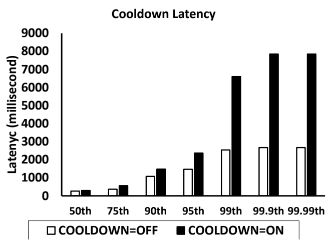

Fig. 13. The impact of cooldown time on tail latency.

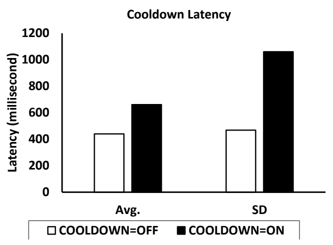

Fig. 14. The impact of cooldown time on average and SD (Standard Deviation) latency.

5 min, takes the external conditions like the busy time of the day into account. Such experiments are not applicable unless using a cloud simulator. We developed our simulator available on GitHub[2], extending CloudSim [24] and develop an auto-scaling mechanisms based on the MAPE-K concept. The widely-used real-world dataset, NASA [25], containing the Web traces of NASA servers for a day was used to stress the auto-scaler. These three approaches are examined under different conditions of scaling rules: Resource-aware, SLA-aware, and Hybrid and SIs: 5 and 10 min to mitigate the interference ($3 \times 3 \times 2 = 18$ tests).

Figures 15 and 16, show that once the delay time in starting-up a VM increased, the tail, average and, SD of latency were negatively impacted. The increasing trend continues to exist even for very high percentiles. As expected, dynamic start-up delay shows more impact on latency than static. Our supplementary results show that SLA violation, time to adaptation, and the number of failed requests increased once the start-up delay are considered more.

FINDINGS 7: The start-up delay of VMs is a major cause of tail latency, so not considering—or considering only statically—this delay in simulation-based investigations may influence the correctness of results. The higher the VM start-up delay is, the longer the tail will be.

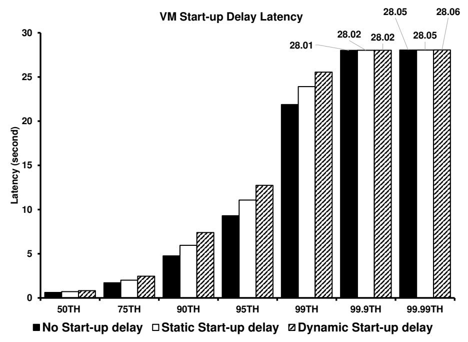

Fig. 15. The impact of VM start-up delay on tail latency.

IV. Discussion

In each previously studied tail sources, 3 to 4 different tests were conducted, depending on the type of latency factor, to evaluate the tail latency under different circumstances (except for the cooldown time that we used t-Test as it had only two tests i.e., On or Off). Those tests showed that how and under which conditions the seven investigated factors can cause/deteriorate the tail of latency. For instance, we learnt that if we pick longer SIs, the tail latency gets larger. The results may give the implication that all the studied factors are sources of tail latency, but the reality is that they are showing that only if not well-tuned, can they become a tail source. In other words, since there is no baseline for the tail latency evaluation in auto-scaling mechanisms (e.g., latency larger than 1 second or 10 seconds), one cannot target any

[2]https://github.com/aslanpour/AutoScaleSim

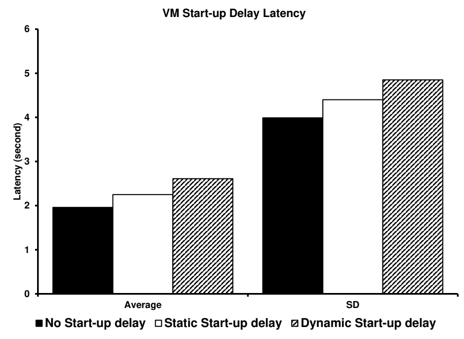

Fig. 16. The impact of VM start-up delay on average and SD (Standard Deviation) latency.

sharp value to categorize the factors as shorter or larger than the baseline to reach a unique conclusion. Hence, we particularly demonstrated under which conditions each factor is causing/deteriorating the tail of latency.

It is clear that the mean and latency distribution for tests in each experiment (Section III-A to III-G) may differ, indicating the positive or negative impact of tuning the corresponding latency factor (e.g., threshold tuning, scaling interval, etc.). The main question is that "the impact of tuning which factor is statistically significant and which is not?" Here, using ANOVA (Analysis of Variance) Single Factor method, we investigated the significance level for every factor. We here feed those latency-related results to ANOVA tool, for each set of experiments, to evaluate if the difference between means is statistically significant. A p-value less than 0.05 is considered statistically significant. The F value higher than the F-Critical value is interpreted that the tests in that set of experiments are representing significantly different behavior. To clearly explore which test had exactly different average, we utilize Bonferroni approach, whereby the difference between tests are evaluated using pairwise t-Test, i.e., two-sample assuming equal variances tool on a pair-by-pair basis. Due to space limitation we briefly report the findings. Our statistically significant investigation demonstrates that the impact of tuning each factor on the latency in auto-scaling mechanisms is as follow. Factors with significant impact are fourfold: The size of the Web requests, analysis method, cooldown time and VM start-up. In the Web requests' tests, the *tiny* and *small* requests resulted in significant difference in comparison to the *very large* requests. This is the case for the instant and proactive analysis when it comes to analysis methods. The significant effect of the enabled cooldown time was already clear with the percentile results and here was confirmed as well. In the VM start-up delay evaluations, the difference between only No-start-up delay and Dynamic start up delay is significant. Now that significant sources are discovered, one may strictly ask which factors do produce more outliers? To answer this question, Fig. 17 delivers a broader view of the sources (i.e. factors) of tail latency in terms of their contribution in producing outliers. Fig. 17 draws a distinction between all tests' results, excluding simulation-based tests in the interest of consistency. Obviously, turning on the cooldown feature seems to be the most outlier producer in auto-scaling mechanisms, imposing more than 7000 ms of latency for very high percentiles, i.e. double the other factors. Then comes poor threshold tuning. Poor scaling interval and surplus VM selection tuning stand at the third place, with the potential to produce outliers more than requests' length and poor analysis method. Hence, another lesson we can learn with this boxplot is that although tuning a parameter might not have statistically significant effect on tail latency, still this tuning can play the role of a huge outlier producer. For example, threshold tuning did not have statistically significant effect on tail latency, according to the earlier ANOVA reports, but Fig. 17 meaningfully illustrates that such poor thresholds can produce more outliers when compared to those tail sources with statistically significant effect. Overall, it is obvious that developers have to decide on which primary objective they are looking to meet: tuning parameters to achieve less statistically significant impact on tail latency or those to achieve less outliers. It seems that these two objectives are conflicting and demand accepting a compromise, although reaching a trade-off may be an interesting research objective for the future works.

One may be curious about the factors actively presented to be a source of tail latency in order for shifting greater focus on only those factors. Hence, another concluding analysis would be to discover the relationship between all evaluations and findings in this research. After evaluating percentiles, average, SD, statistically significant and outliers measures, we can see which tail factors are actively present. This further analysis led us discover the presence of the analysis method and cooldown time components in all evaluations as the major sources of tail latency in auto-scaling mechanisms. Practical solutions include: 1) turning off the cooldown component; 2) improving the cooldown functionality by adopting dynamic and variable time slots [16]; and 3) undoubtedly leveraging more complicated analysis methods such as machine learning, instead of just instant analysis [17].

Another critical lesson we learnt is that, while the average of latency may be alleviated/deteriorated for an auto-scaling mechanism by tuning a specific component (e.g. the SI), the opposite may happen for the mechanism in terms of the tail latency, simultaneously. The experiments for the size of web requests, SI and surplus VM selection policy confirm this inconsistency. For instance, in the size of request's experiments, the *large* requests' length showed larger average than that of the *very large* while the opposite is true for their tail latency. In the SI's experiments also the *mid-term* SI had a slightly larger average latency than that of the *long-term* while the opposite is true when it comes to the tail latency for this tuning. This observation, once again, warns us of the importance of designing mechanisms specifically focusing on the tail latency minimization as the average and tail would not necessarily behave the same way.

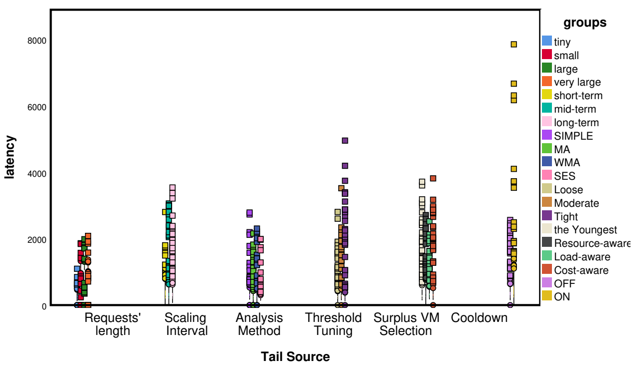

Fig. 17. A basis for tail sources comparison.

V. Conclusions and Future Work

Auto-scaling of Web applications in clouds is a widely-investigated topic which aims at minimizing the cost and the latency of responses for hosting Web applications. Looking at available solutions, it is obvious that researchers mainly take the average of latency into account and neglect the tail. In this paper, seven possible sources of tail latency were investigated. Having empirically evaluated, we found that sources of tail latency in auto-scaling web applications include: (1) large requests; (2) long-term scaling intervals; (3) instant analysis of scaling parameters; (4) conservative, i.e. tight, threshold tuning; (5) load-unaware surplus VM selection policies; (6) cooldown time feature; and (7) VM start-up delay. We learnt that the length of requests, analysis method, cooldown feature and VM start-up delay can result in significant impact on tail latency. We visualized the latency distribution for all factors in order to discover which factors produce more outliers. These evaluations demonstrated that the cooldown feature, poor threshold tuning, poor scaling interval and poor surplus VM selection produce more outliers. Overall, poor analysis method and cooldown feature are the most critical sources of tail latency.

One cannot claim that having all tail sources at the same time in a mechanism would multiply the tail. Furthermore, our investigations are for transactional Web applications and cannot be generalised to other applications which is the focus of our future work. We also plan to investigate tail latency in multi-tenant scenarios with multiple running applications. Finally, having discovered the contrary behavior of average and tail latency, we are interested to find solutions which will jointly minimize the average and tail.

References

[1] M. S. Aslanpour, M. Ghobaei-Arani, and A. Nadjaran Toosi, "Auto-scaling web applications in clouds: A cost-aware approach," *Journal of Network and Computer Applications*, vol. 95, pp. 26–41, 2017.

[2] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 73, 2018.

[3] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza, "Elasticity in cloud computing: a survey," *annals of telecommunications-annales des télécommunications*, vol. 70, no. 7-8, pp. 289–309, 2015.

[4] M. S. Aslanpour and S. E. Dashti, "SLA-Aware resource allocation for application service providers in the cloud," in *2016 2nd International Conference on Web Research, ICWR 2016.* Tehran: IEEE, apr 2016, pp. 31–42.

[5] M. S. Aslanpour, S. S. Gill, and A. N. Toosi, "Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research," *Internet of Things*, p. 100273, 2020.

[6] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding Long Tails in the Cloud." in *NSDI*, vol. 13, 2013, pp. 329–342.

[7] G. Linden, "Marissa Mayer at Web 2.0," 2006.

[8] T. Chen, R. Bahsoon, and X. Yao, "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 61, 2018.

[9] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[10] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles.* ACM, 2017, pp. 325–341.

[11] C.-F. Liu, M. Bennis, and H. V. Poor, "Latency and reliability-aware task offloading and resource allocation for mobile edge computing," in *Globecom Workshops (GC Wkshps), 2017 IEEE.* IEEE, 2017, pp. 1–7.

[12] D. Sun, G. Li, Y. Zhang, L. Zhu, and R. Gaire, "Statistically managing cloud operations for latency-tail-tolerance in IoT-enabled smart cities," *Journal of Parallel and Distributed Computing*, 2018.

[13] A. Computing, "An architectural blueprint for autonomic computing," *IBM White Paper*, vol. 31, 2006.

[14] J. Huang, C. Li, and J. Yu, "Resource prediction based on double exponential smoothing in cloud computing," in *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on.* IEEE, 2012, pp. 2056–2060.

[15] M. S. Aslanpour, M. Ghobaei-Arani, M. Heydari, and N. Mahmoudi, "LARPA: A learning automata-based resource provisioning approach for massively multiplayer online games in cloud environments," *International Journal of Communication Systems*, p. e4090, 2019.

[16] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[17] M. S. Aslanpour and S. E. Dashti, "Proactive Auto-Scaling Algorithm (PASA) for Cloud Application," *International Journal of Grid and High Performance Computing*, vol. 9, no. 3, pp. 1–16, jul 2017.

[18] F. Al-Haidari, M. Sqalli, and K. Salah, "Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 2. IEEE, 2013, pp. 256–261.

[19] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on.* IEEE, 2012, pp. 423–430.

[20] A.-F. Antonescu and T. Braun, "Simulation of SLA-based VM-scaling algorithms for cloud-distributed applications," *Future Generation computer systems*, 2015.

[21] M. Beltrán, "Automatic provisioning of multi-tier applications in cloud computing environments," *The Journal of Supercomputing*, vol. 71, no. 6, pp. 2221–2250, 2015.

[22] E.-J. van Baaren, "Wikibench: A distributed, wikipedia based web application benchmark," *Master's thesis, VU University Amsterdam*, 2009.

[23] E. Casalicchio and L. Silvestri, "Mechanisms for SLA provisioning in cloud-based service providers," *Computer Networks*, vol. 57, no. 3, pp. 795–810, 2013.

[24] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[25] M. F. Arlitt and C. L. Williamson, "Internet web servers: Workload characterization and performance implications," *IEEE/ACM Transactions on Networking (ToN)*, vol. 5, no. 5, pp. 631–645, 1997.