Holistic Workload Scaling: A New Approach to Compute Acceleration in the Cloud

Juan F. Pérez
Universidad del Rosario
Bogotá, Colombia

Lydia Y. Chen
IBM Research Zurich
Rüschlikon, Switzerland

Massimo Villari
University of Messina, Italy

Rajiv Ranjan
Newcastle University, UK

Editor:
Rajiv Ranjan
rranjans@gmail.com

Workload scaling is an approach to accelerating computation and thus improving response times by replicating the exact same request multiple times and processing it in parallel on multiple nodes and accepting the result from the first node to finish. This is not unlike a TV game show, where the same question is given to multiple contestants and the (correct) answer is accepted from the first to respond. This is different than traditional strategies for parallelization as used in, say, MapReduce workloads, where each node runs a subset of the overall workload. There are a variety of strategies that trade off metrics such as cost, utilization, performance, and interprocessor communication requirements. Performance modeling can help determine optimal approaches for different environments and goals. This is important, because poor performance can lead to application and domain-specific losses, such as e-commerce conversions and sales.[1] Performance modeling and analysis plays an important role in designing and driving the selection of resource scaling mechanisms. Such modeling and analysis is complex due to time-varying workload arrival rates and request sizes, and even more complex in cloud environments due to the additional stochastic variation caused by performance interference due to resource sharing across co-located tenants. Moreover, little is known on how to multi-scale, i.e., dynamically and simultaneously

Copublished by the IEEE CS and IEEE ComSoc
2325-6095/18/$33.00 ©2018 IEEE

scale resources vertically, horizontally, and through workload scaling. In this article, we first demonstrate the effectiveness of multi-scaling in reducing latency, and then discuss the performance modeling challenges, particularly for workload scaling.

A study from Amazon estimates[1] a latency delay of 100 ms can cause a one percent sales drop. A recent study from Akamai[1] shows that a one second delay in page response can result in 7% loss in e-commerce conversions. These numbers illustrate the relevance of finding novel solutions to the long standing and critical challenge of reducing and/or guaranteeing the latency of interactive applications, such as web services, in a cost-effective way. Traditionally, the fundamental difficulty lies in the workload volatility, i.e., time-varying request arrival rates and varying request sizes being served by resources that may already be partially loaded. Cloud computing, with its unique ability to scale resources on demand, offers a powerful engineering solution to tackle the workload variability, but possibly by incurring additional costs due to pay-per-use pricing. The advancement of virtualization technologies makes a wide range of resources readily available upon users' requests, e.g., virtual machines, CPU cores, and docker images, and "serverless compute APIs" which can be triggered to react to changes in the workload.

However, the downside of the cloud is that the performance of virtual resources may not be stable[2] due to the underlying hardware, colocated applications, virtualization solutions, and network congestion spikes. For example, the performance of web services can be significantly degraded by CPU or network hungry neighbor VMs due to the resource contention.[3] Such issues become even more prominent when moving into multitenant clouds, where the degree of resource sharing increases significantly. The impact of this problem is more tangible for the tail latency, e.g., 95th or 99th percentile, which can grow much larger than the average latency, hindering the users' quality of experience significantly. The pitfall of resource sharing presents itself as a challenge to manage and model interactive applications[4,5] as the service rate per virtual resource is no longer constant making the service times become even more volatile.

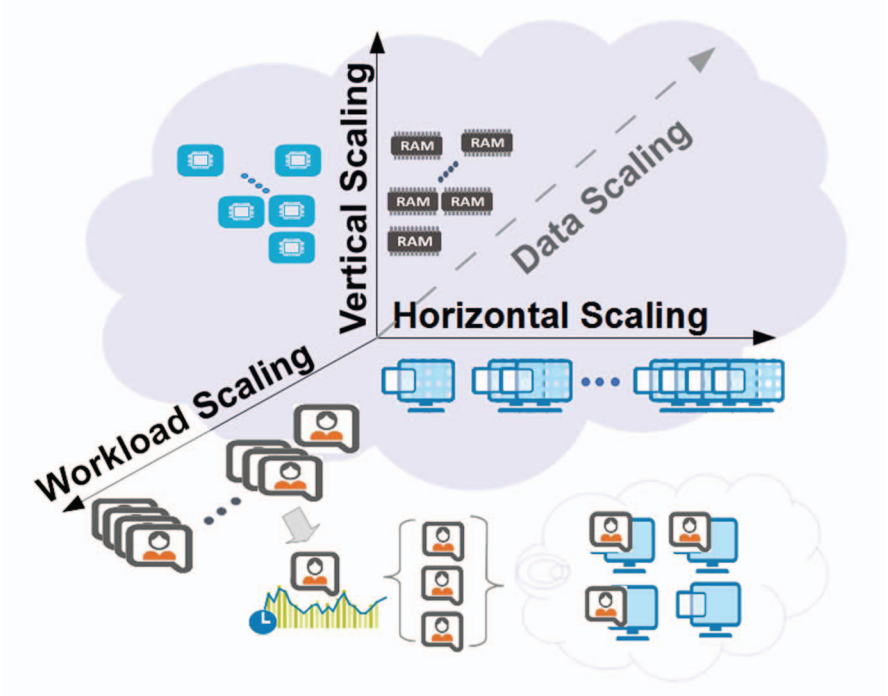

Figure 1. Scaling choices for interactive applications on the cloud.

To defend the latency against the workload variability, particularly the arrival pattern, a plethora of prior art[2,3] develops auto-scaling solutions along the vertical and horizontal directions. Vertical scaling increases/decreases virtual resources per virtual machine, e.g., the number of virtual cores, whereas horizontal scaling increases/decreases the number of virtual machines. The core principle behind these solutions is to scale the resources according to the workload demands and the latency target, often with a focus on the average value. While resource scaling can cost-effectively fulfill the average latency target for interactive services, it falls short in curtailing the *tail* latency due to the high variability per virtual resource.

Consequently, workload scaling has emerged as an alternative to manage the tail performance at scale,[6] specifically targeting scenarios where servers have time-varying processing speeds as in the cloud. Quite differently from resource scaling, workload scaling deliberately increases the workload by cloning incoming requests and simultaneously processing them on different virtual or physical servers. A reply can thus be sent to the user as soon as the *first* clone completes execution, that is, the fastest clone determines the request processing time. The *advantage* of workload scaling thus lies in making the most of the available resources by executing the same request on several servers and using the fastest to respond. The *drawback* of workload scaling is the additional load introduced by the clones, requiring underutilized servers or elastic resources to be available. This solution thus requires careful use as it could potentially *harm* the application performance if used in peak-load conditions.

In this article, we first explain the advantages and limitations of scaling resources and workloads through empirical examples of web services. We then discuss key modeling challenges found when capturing vertical, horizontal, and workload scaling, with a focus on the latter as it is the least studied.

THE LANDSCAPE OF SCALING SOLUTIONS

In this section we show the limits of vertical and horizontal scaling and how these can be combined with workload scaling to robustly manage the tail latency. We focus on the latency mean and 95th percentile as key performance metrics, and employ web applications hosted in the cloud, modifying the number of cores per VM (vertical scaling) and the number of VMs (horizontal scaling). We close this section by highlighting the limitations of these solutions.

Vertical Resource Scaling. In Figure 2, we show the latency 95th percentile observed for RUBiS, a web shopping benchmark, on a single VM at a private cloud where a neighboring VM workload is injected to emulate inference. Requests are generated at a constant arrival rate, i.e., 100 requests per second, while the number of virtual cores increases from two to six cores, one at a time every 6 minutes. One can clearly see that the tail latency decreases with the increasing number of virtual cores, but with a decaying marginal gain. This is because the processing time, i.e., latency, follows a $1/n$ rule where n is the number of cores. It will be apparent that the latency improvement between 5 and 6 virtual cores is quite small, compared to the difference between 2 and 3 cores. The effective capacity per VM is not linearly proportional to the number of virtual cores per VM, even though the price does increase linearly with the number of cores. In fact, even the largest relative gain in latency (obtained when moving from 2 to 3 cores) is not proportional to the increase in cost. Thus, increasing the number of cores may have a limited impact and may not be cost effective. Further, increasing the number of cores does not offer a solution to the larger latency due to the interference caused by the neighboring VMs.

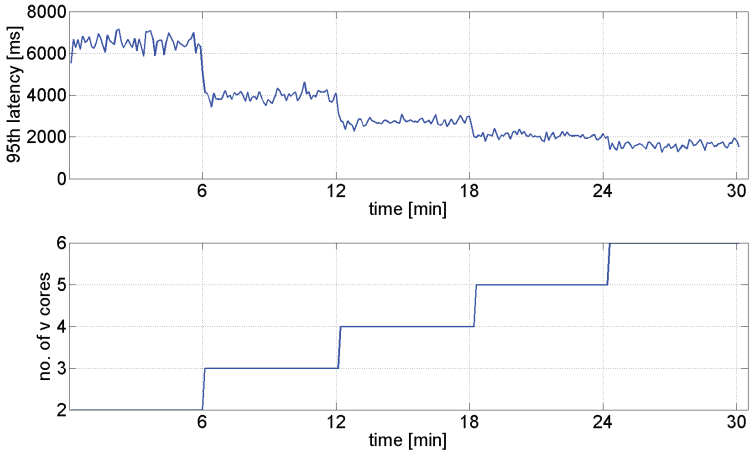

Figure 2. The tail latency of RUBiS request when increasing the number of allocated virtual cores, under a constant arrival rate of 100 requests per second.

Horizontal Resource Scaling. In Figure 3, we show the improvement in the average latency of a popular web knowledge application, MediaWiki,[7] as the number of VMs increases. Two types of VMs are considered, namely wimpy and brawny, which consist of 2 and 4 virtual cores, with 2 and 4 GB RAM, respectively, to serve a load of 10 requests per second. We observe a similar trend as for vertical scaling: the latency decreases with the increasing number of VMs but the marginal gain decreases even as the cost increases. The best latency provisioning wimpy VMs (5) is around 138 ms, whereas provisioning 5 brawny instances results in an average latency as low as 120 seconds. Moreover, brawny VMs achieve lower latency than wimpy VMs for any given number of instances. However, the improvement is definitely less than half even though the number of virtual resources doubles and the prices also doubles according to the standard market practice, e.g., Amazon EC2. One may thus conclude that using a sufficient number of brawny instances, i.e., 5, one can achieve the target latency of 120 ms, whereas even a high number of wimpy instances fails to achieve such a target. If we combine horizontal scaling of wimpy VMs with a workload scaling factor of two a surprising result occurs. Upon arrival of requests, we replicate them once and send each replicas to two different VMs. The latency is determined by the fastest request among the two replicas. One can see that a sufficiently large number of wimpy instances, i.e., 5 VMs, can achieve an average latency as low as 115 ms, which is lower than the target and better than the best performance achieved with brawny instances.

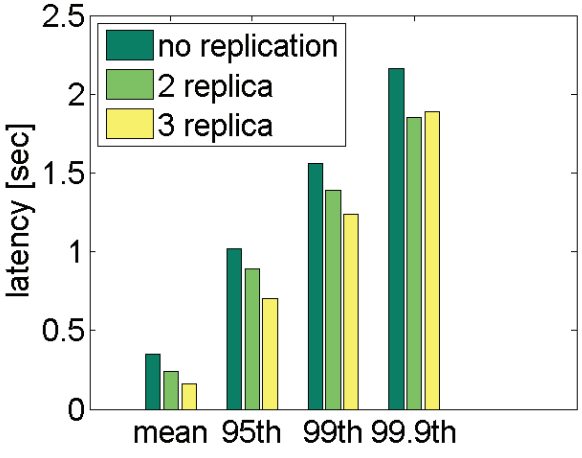

Figure 3. The average latency of hosting MediaWiki in cloud under three scaling strategies: (i) adjusting wimpy VMs only, (ii) adjusting brawny VMs only, and (iii) adjusting wimpy VMs with a replication factor of two.

At a first order of analysis, the latency of the system can be modeled by the theory of order statistics, which describes among other things the expected value of the minimum of k samples taken from a given distribution. For example, the expected value of the minimum of k samples taken from a uniform distribution on [0,1] is $1/(k+1)$. In the real world, such a simple model is insufficient and performance modeling is required, because we also note that when the number of wimpy instances is low, 2 instances or less, replicating queries results in a latency worse than without request replication. This can be explained by the extra load introduced by the

replicated requests, and indicates that enabling workload scaling without sufficient resources can result in more harm than performance advantages.

Workload Scaling. As shown in the previous example, and in agreement with prior art, speculatively replicating requests is an effective strategy to strengthen system dependability[8] and to improve the latency,[9] particularly its high percentiles. Workload scaling policies in interactive systems can be grossly classified by the issuing time of the replicated requests and by the canceling policy on the remaining redundant requests. Replicated requests can be issued proactively upon the arrival of requests or reactively after observing performance degradation so as to minimize the processing overhead. Upon receiving the first result from replicated requests/jobs, the majority of replication policies leave the rest of replicas in the system due to the overhead of terminating requests, while a few studies show the benefits of terminating requests for certain benchmarks.[6] Additional details on these policies are provided in the section on Research Challenges and Open Issues.

Table 1. Comparisons of the analytical models on replication.

Article	Assumption		Replication		Res Scaling		Testbed
	Arr.	Proc.	Cancel	Latency	Verti.	Horiz.	Validation
[9]	Static	Ind.	√	mean			√
[4]	Static	Ind.		mean			
[10]	Static	Corr	√	mean			
[11]	Static	Ind.		dis.		√	
[5]	Dynamic	Ind.	√	mean		√	√

Optimal Workload Scaling. Though the prior art demonstrates the effectiveness of workload scaling, little is known on determining the optimal level of scaling or replication. The overhead of a high replication level can overturn the benefit of returning the first-completed request. Moreover, it also depends on the metrics of interests, i.e., the optimal level for mean latency may not be the same for the 95th or 99th percentiles. Herein, we show the performance of a MediaWiki cluster hosted on Amazon EC2, subject to different levels of workload scaling. This cluster has seven t1.micro instances: six running the MediaWiki stack and one used as central queue to dispatch the requests. Figure 4 depicts the mean and tail latency with different percentiles when applying between 1 and 3 replicas for an arrival rate of 1.33 requests per second. Clearly, replication is effective in reducing the latency tail, with reductions close to 15% with 2 replicas. However, introducing a third replica hurts the tail even if it improves the mean latency. This highlights the importance of developing analytic models that are able to compute key latency metrics, both mean and distribution, under workload scaling.

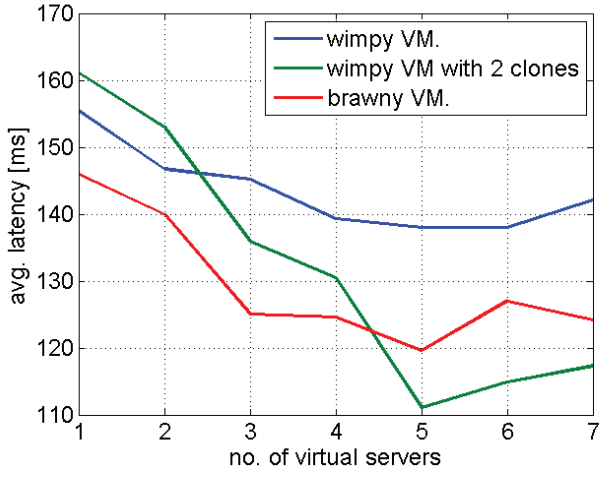

Figure 4. Different latency metrics under workload scaling only: no replication, two replicas and three replicas.

Principle and Limitation of Multi-Scaling. The main idea behind workload scaling is to increase the likelihood that requests are processed on the fastest servers. This could be determined *a priori* by a scheduling and load balancing strategy that sniffs the speed of servers via different performance indicators. In contrast, workload scaling defeats the variability by trying out multiple servers simultaneously, determining the fastest server *a posteriori*. A common criticism is therefore the high processing overhead introduced by redundant requests. In fact, the prerequisites for workload scaling to be an effective solution are (i) high variability of server speeds across VMs over time, (ii) somewhat low system load, and (iii) a sufficient number of servers available on a cost-effective basis. In the case where all the request clones are processed in full, the total load introduced by proactive replication is simply the baseline load multiplied by the replication factor. Being able to cancel the request clones (after the first one finishes) and reactively spawning the replicas can reduce the overhead. The number of available servers not only constraints the maximum number of workload scaling levels, i.e., the number of replicas cannot exceed the number of available servers, but also implicitly requires data availability. For example, when deploying the MediaWiki stack on multiple VMs, one shall ensure the all database entries are replicated on each VM, otherwise requests can only be served by a subset of VMs that contain the requested data. Hence, in addition to workload and resource scaling, data content scaling and data access network scaling are additional considerations yet to be addressed the multi-scaling in the cloud. Further details on this and other key aspects in modeling workload and multi-scaling are treated in the next section.

RESEARCH CHALLENGES AND OPEN ISSUES

In this section we delve into the challenges of modeling multi-scaling for software applications in the cloud, existing solutions and open problems. Rather than considering problems that are common to the modeling of software applications in general we focus on those key features that are more prominent when modeling multiscaling solutions. We provide a summary of these key features in related work in Table I. In the following we detail those features from the perspective of workload scaling that implicitly requires resource scaling.

Request Canceling Policy

When introducing workload scaling, multiple clones of a request are issued and processed. Since it is sufficient to receive a single response from any of these clones, it is in principle possible (and desirable) to kill or cancel all remaining clones when the first one completes service. Canceling is particularly appealing as it limits at least a portion of the additional load introduced by the workload scaling mechanism. As a result, most modeling works,[4,10] assume that request canceling is adopted, and in many cases the cancellation is assumed to have zero cost.

Canceling is also appealing from a modeling point of view and it is central in the few analytical results available for workload scaling. For instance, canceling allows Joshi and colleagues[10] to reduce a multi-server setup to a single-server one and to provide approximate solutions. Also, the solution to the Markov chain model proposed by Gardner et.al.[4] relies on the canceling assumption. One key reason why canceling simplifies the analysis is that all clones of a request finish at the same time, such that the minimum service time among all clones of a request determines the service and departure times of *all* clones.

However, canceling request clones is not always feasible as it requires incorporating into the application a functionality that is not necessarily available. Also, in "fast" distributed applications, where the request processing times are very short, canceling requests may be infeasible as the signaling delays may be too long for the signals to arrive before the processing completes. From a modeling perspective there may be stochastic variation in the delays for a cancellation signal to arrive across variably congested or distant network links to different nodes. Thus, in many applications one needs to account for the possibility that clones cannot be canceled or that the cancellation has a non-negligible cost.

Leaving all clones to execute until each of them if fully processed means that the system remains busy a much larger fraction of time than when canceling is in place. In other words, if each request is cloned (on average) *r* times the offered load increases r times. Thus, workload scaling without canceling will often require more resources than with canceling, sometimes many more, and its performance advantage is therefore limited to systems facing a low-to-moderate load, as has been shown for instance in Vulimiri et.al.[9], Qiu et.al.[8], Pérez et.al.[5].

The operation without clone canceling has also proven more difficult for modeling and analysis. In fact, the analytical results available are limited to scenarios with fairly strict assumption, such as the 2-server system with a centralized queue, Poisson arrivals, and exponential processing times considered in Lee et.al.[12]. An approximate analysis method proposed in Vulimiri et.al.[9] consists of assuming that all servers operate independently, receiving a fraction of the total traffic augmented by the workload scaling factor *r*, computing the latency for each server and obtaining the latency of a request as the minimum of the latency of *r* independent random variables, each holding the latency of a single server. The proposal in Vulimiri et.al.[9] exploited the fact that the latency offered by a single server with exponentially-distributed processing times and Poisson arrivals is itself exponentially distributed, whereas in Qiu et al.[13] a similar argument is used for phase-type distributed processing times.

Load Balancer Scaling Awareness

In a distributed setup the load balancer plays the key role of forwarding the incoming requests to the available servers. When horizontal scaling is introduced, the load balancer needs to stay aware of the changing set of resources, and appropriately modify its routing policy to incorporate and remove target resources in real time. In addition, if workload scaling is introduced, the load balancer can be further involved in properly allocating the clones of a request to the target resources. For instance, it is desirable if not mandatory that clones of the same request are processed by *different physical* servers to exploit the benefits of workload scaling, as otherwise copies of the same request would simply contend for resources at the same server.

As a result, from a modeling standpoint, the awareness of the load balancer needs to be considered explicitly as this vastly affects the performance of the scaling mechanism[14]. In the case of a scaling-unaware load balancer, a distributed server may receive multiple copies of the same request, which impacts its arrival stream creating batch arrivals and therefore impacts the latency negatively. Instead, this scenario is not possible in a scaling-aware load balancer, where arrivals to a server occurs as singletons and with the same timestamps as the request arrivals to the load balancer.

One further consideration in a distributed setup is that, in case all servers are busy, the incoming requests could either queue at the load balancer or immediately dispatched to an appropriately chosen server (e.g., following the least-connections rule). This decision typically depends on the type of application and the ratio between the transfer delays and the request processing times. If transfer times are significant, immediately dispatching the request is preferred to avoid additional delays. In either case, this choice must be capture by the model, especially when workload scaling is introduced. With the queueing-at-the-load-balancer option and workload scaling, it is possible for the load balancer to dispatch the clones of a request to the next available server as long as the reply for that request has not been received. In other words, if a request is quickly processed and copies of that request are still queueing at the load balancer, it is possible to remove those copies from the queue and avoid unnecessary processing. This is not possible if the load balancer immediately dispatches all copies to the target servers. Whereas exact numerical models exist for the former case[11], the latter option has proven more difficult due to the multi-dimensionality of the problem (many servers each with its own queue) and the close connection that exists among them since the request latency is given by the clone that finished first, in any of the servers.

The final issue to consider, mainly due to evolution of container-based microservices application architectures, is that the underlying load-balancer can either operate at transport layer (L4, request agonistic) or application layer (L7, request aware). In the case of transport layer (L4) load balancer, the incoming requests are distributed across web and/or app servers without necessarily analyzing the composition of the request type. On the other hand, the application layer (L7) load

balancer balances the requests across web and/or app servers more intelligently after analyzing the HTTP and/or HTTPS request headers. In the case of a scaling servers connected to the request agnostic (L4) load balancer, the load balancer only needs to be aware of the contact end-point of new servers as these have exact replicas of the data-sets and files. Instead, this scenario is not possible in a request aware (L7) load balancer, as the servers manage different data-sets and files (e.g., an image server vs. an account server). As a result of this, scaling in the context of L7 the load balancer will require both the contact end-point of the new servers as well as the type of request that they can serve, to be configured within the load-balancer at run-time. In summary, such diversities in load-balancing approaches further complicate the scaling and replication of web services in the cloud for improving end-to-end request latency.

Request Processing Times

As in general software application modeling, appropriately capturing the request processing times has a large impact in the accuracy of the latency predictions obtained by the model. The more general models are however more limited in the analyses that can be performed, in many cases requiring approximated methods. Thus, for multi-scaling, as in general software application modeling, many more results are available for simpler assumptions on the processing times (exponentially distributed) than for more general assumptions.

Beyond the distribution of the processing times, one key issue in modeling workload scaling is recognizing that clones of the same request may have similar processing times. For instance, if the processing time is dominated by the search time of data items and each request has a list of data items to retrieve, it is natural that all its copies have similar data retrieval and therefore overall processing times. This behavior can be captured by representing the processing times of the clones of a request as *correlated* random variables, in contrast to the usual assumption of independent processing times.

Introducing this correlation is however difficult as standard queueing models used to represent software applications assume independent request processing times. Even those models that assume correlated processing times typically assume a general correlation structure for all requests processed by the system. Instead, in the case of workload scaling the processing times of the clones of the same request can be heavily correlated whereas the processing times of different requests may still be fairly independent. The majority of modeling work still assumes independent service times, except for Qiu.et.al.[8].

Application Topology and tier-specific workload multi-scaling

The vast majority of existing works in workload scaling focuses on relatively simple applications where the whole functionality is contained in a single box. As a number of these boxes is available, a request clone can be forwarded to and processed by any of these boxes, simplifying the modeling and analysis. However, many applications do not fall within these simple assumptions. This and the following subsection consider two common cases where it is not possible to assume that a request clone can be processed by any application server.

Here we focus on the case where the application servers are actually split among groups (e.g. web tier, app tier, database tier) that provide different functionality, as in the common multi-tier architecture. In this setup a number of new questions arise regarding to workload scaling and modelling. For instance, if a request is replicated at the first tier (the one first hit by any incoming request) its clones generate additional processing requirements at all the application tiers. Further, clones could be generated at any tier and their impact therefore affects all tiers downstream from the one that implements the cloning, and the number of clones grows as the product of the replication factor applied at each tier. For instance, in a 3-tier application, if tier 2 uses cloning factor 2 (creates 1 additional clone) and tier 3 uses cloning factor 3 (creates 2 additional clones), tier 1 will not see any additional load, tier 2 will see its load doubled, while tier 3 will see its load multiplied by a factor of 6 since each of the 2 clones in tier 2 generates requests to tier 3, each of which is cloned 3 times. Moreover, as each application tier has different workload

behavior and resource requirements, it is important to develop tier specific workload scaling models.

To the best of our knowledge, the only work that explicitly considers workload scaling and modelling for a multi-tier application is sPARE[14], which proposes an algorithm to determine the best cloning levels at each tier explicitly considering the multiplicative impact of cloning at multiple tiers.

Performance Characterization

The vast majority of the existing work tends to focus on modelling the performance (e.g, request processing latency, memory utilization, CPU utilization, request processing throughput) of multi-tier applications (as noted above) on cloud datacenters that provide hypervisor-based virtualization technologies. Hypervisors enable cloud providers to create unique virtual machines (VMs) that share a set of physical hardware resources (CPU, memory, network, and disk). The hypervisor-based virtualization has many advantages (such as stronger security context), but at the cost of performance overhead, as each VM has its own Operating System (OS) image. To narrow the gap between performance of applications on virtualized and non-virtualized physical resources, container-based (e.g., Docker, LXC) cloud virtualization solutions have evolved in the last 5 years. Although containers are becoming an attractive choice for deploying applications in the cloud, very limited modelling techniques exist[15] in the literature that can appropriately distinguish and reason about the differences in the performance overhead between hypervisor-based and container-based infrastructures. For instance, hypervisors provide each VM with its own resources, while Container engine (equivalent of hypervisor) share a single host's resources among multiple containers (roughly equivalent to VMs) via *cgroup* and *namespace* mechanisms. Considering another example, the run-time performance (e.g. request processing latency) of a containerized web or database microservice not only depends on the resource configurations (e.g., CPU Cores, memory size) allocated to its *cgroup* but also on whether the underlying physical host is hypervisor-based or is hypervisor-free. In the case where containerized web services are deployed on hypervisor-based cloud resources, one would need to undertake following multi-level modelling across each application-tier including: (i) container-level and (ii) VM-level.

Data Availability

As described in the previous subsection, when modeling workload scaling it is common to assume that a replica clone can be forwarded to any of the application servers. Even if the model explicitly considers the application tiered architecture, or if the application has a single tier, a specific request may only be processed by a subset of all servers. This is the case when the application considered, or one of its tiers, serves data items stored across a number of distributed nodes such that (each of) the items required to fulfill a request are available in just a few of the nodes.

Incorporating data availability in modeling workload scaling introduces too the issue of data popularity, as this makes the request arrival rates to be much larger for popular data items than for non-popular ones. Thus, whereas introducing cloning for unpopular data items may have little to no effect, cloning requests for popular data items must consider the availability of enough servers holding that item to fully exploit the benefits of workload scaling as well as to avoid potential server overloads. As of now, this topic has not been explicitly considered in the literature on modeling workload scaling.

Reliability

Most analyses of workload scaling have focused on the performance benefits of this strategy, as it has proven effective in reducing latency. However, workload scaling has the potential of improving reliability in failure. Consider for instance a request that is cloned and its two copies are sent to two different servers. If one of the servers fails, the request clone sent to the other server can still complete processing and the reply can be sent back to the user without requiring any additional steps. Or, for error-prone systems, a voting mechanism can determine the correct result

from, say, a majority of the correct and incorrect results received. Workload scaling is thus able to boost both performance and reliability. Evaluating workload scaling across both of these perspectives simultaneously still remains largely unexplored, with few exceptions such as Qiu and Pérez[16].

Performance Metrics

As workload scaling is mainly put forward as a method of reducing response time at the possible expense of higher resource requirements, models are typically geared towards obtaining latency metrics. In many cases the metric of interest is the average latency[9,4,5]. However, workload scaling can be particularly effective to limit the tail latency, which makes obtaining the latency distribution (or its high percentiles) necessary to assess this potential benefit. Several works[11,5] have therefore focused on deriving models to obtain the latency distribution. Finally, one metric that has proven very useful[9,8] to assess when to activate a workload scaling mechanism is the *threshold load*, that is, the maximum load at which introducing one additional clone remains beneficial without congesting or overloading the total system. Obtaining this metric allows for the design of adaptive controllers that increase/decrease the cloning intensity according to the observed load, such that cloning is deactivated under peak loads to avoid overloads whereas time periods with low loads (valleys) can benefit from heavy cloning.

CONCLUSION

As we have seen, a number of recent works consider workload scaling a mechanism to latency management. A few of them further attempt to answer the fundamental question of the optimal number of replicas under various system assumptions (request canceling policy, load-balance awareness, etc.). Although the number of servers is a usual input to the proposed models, almost no work explores the resource and workload scaling jointly, except Qiu and Pérez[11], Pérez et.al.[5] . Among the latter, Qiu and Pérez[11] consider a stationary workload and is validated via simulations only. Instead, DuoScale[5] considers both horizontal resource and workload scaling and its results are experimentally validated on a testbed subject to a dynamic workload. To the best of our knowledge, there is no modeling work jointly exploring the three dimensions of horizontal resource scaling, vertical resource scaling, and workload scaling.

ACKNOWLEDGEMENTS

The research presented in this paper has been supported by the Swiss National Science Foundation National Research Programme “Big Data” (NRP 75) (project 407540 167266).

REFERENCES

1. *K. Metrics*, blog; https://blog.kissmetrics.com/loading-time.
2. M. Björkqvist, L.Y. Chen, and W. Binder, “Opportunistic Service Provisioning in the Cloud,” *Proceedings of the IEEE 5th International Conference on Cloud Computing* (CLOUD 12), 2012, pp. 237–244.
3. R. Nathuji, A. Kansal, and A. Ghaffarkhah, “Q-clouds: managing performance interference effects for QoS-aware clouds,” *Proceedings of the 5th European conference on Computer Systems* (EuroSys 10), 2010, pp. 237–250.
4. K. Gardener et al., “Reducing latency via redundant requests: Exact analysis,” *ACM SIGMETRICS*, vol. 43, no. 1, 2015, pp. 347–360.
5. J.F. Pérez et al., “Dual scaling vms and queries: Cost-effective latency curtailment,” *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems* (ICDCS 18), 2017, pp. 988–998.
6. J. Dean and L.A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, no. 2, 2013, pp. 74–80.

7. S. Melnik et al., "Dremel: Interactive analysis of web-scale datasets," *Proceedings of the 36th International Conference on Very Large Data Bases* (PVLDB 10), 2010, pp. 330–339.
8. Z. Qiu et al., "Cutting latency tail: Analyzing and validating replication without canceling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, 2017, pp. 3128–3141.
9. A. Vulimiri et al., "Low latency via redundancy," *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies* (CoNEXT 13), 2013, pp. 283–294.
10. G. Joshi, E. Soljanin, and G.W. Wornell, "Proceedings of the 53rd Annual Allerton Conference on Communication, Control, and Computing," *Efficient replication of queued tasks for latency reduction in cloud systems* (Allerton 15), 2015, pp. 107–114.
11. Z. Qiu and J.F. Pérez, "Evaluating the effectiveness of replication for tail-tolerance," *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (CCGrid 15), 2015, pp. 443–452.
12. K. Lee, R. Pedarsani, and K. Ramchandran, "On scheduling redundant requests with cancellation overheads," *Proceedings of the 53rd Annual Allerton Conference on Communication, Control, and Computing* (Allerton), 2015, pp. 99–106.
13. Z. Qiu, J. Pérez, and P. Harrison, "Variability-aware request replication for latency curtailment," *Proceedings of the 35th Annual IEEE International Conference on Computer Communications* (INFOCOM 16), 2016.
14. R. Birke et al., "sPARE: Partial replication for multi-tier applications in the cloud," *IEEE Transactions on Services Computing*, 2017; doi.org/10.1109/TSC.2017.2780845.
15. M. Fazio et al., "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, 2016, pp. 81–88.
16. Z. Qiu and J.F. Pérez, "Enhancing reliability and response times via replication in computing clusters," *Proceedings of the 34th Annual IEEE International Conference on Computer Communications* (INFOCOM 15), 2015, pp. 1355–1363.

AUTHOR BIOS

Juan F. Pérez is an assistant professor at Universidad del Rosario, Colombia, Department of Applied Mathematics and Computer Science. He received a PhD degree in Computer Science from University of Antwerp, Belgium. His research interests center around the performance analysis of computer systems, especially on cloud and cluster computing and optical networking. Contact him at juanferna.perez@urosario.edu.co.

Lydia Y. Chen is a research staff member at the IBM Zürich Research Lab, Switzerland. She received a PhD degree in Operations Research and Industrial Engineering from the Pennsylvania State University. Her research interests include performance evaluation for datacenters and big data systems. She has served on several technical program committees in various performance and network conferences. She is an IEEE senior member. Contact her at yic@zurich.ibm.com.

Massimo Villari an associate professor of computer science at the University of Messina. His research interests include cloud computing, Internet of Things, big data analytics, and security systems. Villari has a PhD in computer engineering from the University of Messina. He's a member of IEEE and IARIA boards. Contact him at mvillari@unime.it.

Rajiv Ranjan is a reader in the School of Computing Science at Newcastle University, UK; chair professor in the School of Computer, Chinese University of Geosciences, Wuhan, China; and a visiting scientist at Data61, CSIRO, Australia. His research interests include grid computing, peer-to-peer networks, cloud computing, Internet of Things, and big data analytics. Ranjan has a PhD in computer science and software engineering from the University of Melbourne. Contact him at raj.ranjan@ncl.ac.uk or http://rajivranjan.net.