Cost-Aware Cloud Metering with Scalable Service Management Infrastructure

Ali Anwar†, Anca Sailer‡, Andrzej Kochut‡, Charles O. Schulz‡, Alla Segal‡, Ali R. Butt†

†*Virginia Tech* ‡*IBM T.J. Watson Research Center*

{ali, butta}@cs.vt.edu, {ancas, akochut, cschulz, segal}@us.ibm.com

***Abstract*—As the cloud services journey through their lifecycle towards becoming commodities, the demand is increasing for "pay-per-use" pricing model. In this model, users are charged for the amount of resources, e.g., volume of transactions, CPU usage, etc., being consumed during a given time period. Software as a Service (SaaS) providers charging their customers via pay-per-use (e.g., Microsoft Azure Web Services) and facing Infrastructure as a Service (IaaS) costs per VM per month (e.g., SoftLayer) have to carefully choose and scale their non-revenue generating service management infrastructure to penetrate and stay in the market. In this paper, we focus on the metering and rating aspects of cloud service management, and their scalability with the SaaS business and operational changes. We design a framework for cloud service providers to scale their revenue management systems in a cost-aware manner, where the deployment of these revenue systems dynamically uses existing or newly provisioned SaaS VMs, instead of the extant approach of using dedicated setups. Our experimental analysis shows that service management related tasks can be offloaded to the existing VMs with at most 15% overhead in CPU utilization, 10% overhead for memory usage, and negligible overhead for I/O and network usage. We used traces from IBM production servers to mimic the load on VMs. By dynamically scaling the service management setup, we were able to adapt to increasing metering data processing requirements without incurring additional cost, while preserving the infrastructure footprint.**

I. Introduction

Until recently, cloud service providers could afford to charge their customers only on a flat-rate basis, e.g., in the form of a monthly subscription fee. Although this pricing methodology is straight forward and involves little management and performance overhead for the cloud service providers, it does not offer the competitive advantage edge of the pay-per-use or usage based pricing [25].

From the perspective of the cloud service provider, maintaining the competitive advantage by effectively adapting to versatile pricing policies has become a matter of high priority [12]. However, usage based pricing policies bring a new set of service management requirements for the service providers, particularly for their revenue management [32]. The revenue management aspects impacted by the pricing policy change are the collection of new metered data and its rating according to the new detailed price plan. This entails finer-grain metering, which may impact the performance of resources due to the need to monitor service resources and applications at the appropriate level to provide the usage to be charged for. This may result in collecting large amounts of metered data. Furthermore, this metered data needs to be processed in order to perform: (1) mediation, i.e., transformation into the desired units of measure expected by the usage price policy, e.g., average, maximum or minimum usage; and (2) rating based on the price policy for generating the invoice for the customers, e.g., multiplying usage by per unit rate. The capacity requirement for these non-revenue generating resources fluctuates with the service demand (e.g., the number of subscriptions), service price policy updates (e.g., from single metric based charge to complex multi-metric based charge), while their unit cost changes depending on the operational infrastructure solution (e.g., on premise, traditional outsourcing or IaaS).

The cloud services profit is the difference between the revenue driven from charging the customers that subscribed to the services and the cost of managing the service. A crucial challenge for a sustainable business model, is how to adapt the pay-per-use revenue management, and implicitly its costs, to dynamically accommodate business changes in the pricing model, service demand or operational changes in infrastructure [20] in order to profitably remain in the race for the cloud market. The providers have to carefully choose the metering, mediation, and rating tools and infrastructure to minimize the cost of the resources performing them.

The first step in performing this cost associated with monitoring and collecting the metering data [11]. The existing practice is to use a separate setup for collecting metering data for pricing in addition to the cloud health monitoring setup that collects information such as performance and availability of resources and resource usage contention. The extra resources used for such revenue management place additional burden on the cloud service provider. To this end, recent works such as Ceilometer [2] from OpenStack [30], [4] aim to consolidate metering for multiple purposes and avoid collecting of the same data by multiple agents. In this paper, we propose a framework that leverages such approaches, especially the OpenStack's ecosystem, to efficiently collect and estimate the volume of metering data.

Second, we need to estimate the cost of storing and processing the metering data. As the service demand fluctuation and the selection of different pricing policies will result in different sizes of collected metering data, the setup is expected to store and process data of varying size without

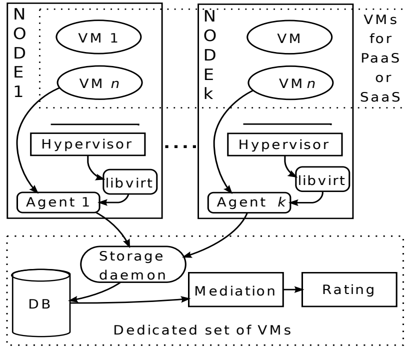


Figure 1. Current approach for metering cloud services.

wasting resources. Typically, cloud service providers use a dedicated set of VMs for their service management as shown in Figure 1, which they manually expand based on the increasing load in their data centers. Cloud service providers such as SaaS may themselves be customers of IaaS or PaaS (Platform as a Service). Thus, they incur monthly charges for this dedicated set of VMs. This infrastructure cost is additional to the cost of the tools (e.g., for license per volume, maintenance etc.). The goal is to minimize the footprint of this nonrevenue-generating infrastructure, thus minimizing service management infrastructure cost; ideally eliminating it.

The typical workloads of the PaaS and SaaS providers clients have been found to use 50% of the IaaS capacity at best [6], [1], leaving the remaining un-utilized 50% for additional workloads. Moreover, SaaS customers can be conveniently given controlled access to the metering data, if such data is collected and maintained at the same set of VMs as that running the workload. Therefore, in our scalable metering solution we adopt this approach. The providers need to comply with their customers SLAs by scaling up their setup according to the load on the systems. To this end, our framework dynamically monitors the resource utilization per VM and scales up or down the tools deployment accordingly. In the worst case scenario, when the workloads on all the customers VMs is about to reach the maximum allowed as per the SLA, our framework automatically launches new VM(s) to adapt to the workload. Note that the traditional customer workload placement is out of the scope of this paper.

II. Enabling Technologies

We have designed our protocol on the well-established cloud ecosystem of OpenStack—an open source project that provides a massively scalable cloud operating system. A key component that we leverage in our project is OpenStack's Ceilometer that provides an infrastructure to collect detailed measurements about resources managed by OpenStack. The use of Ceilometer in our case is to deliver a unique point of contact for the billing systems to acquire all the measurements needed to generate the customer's invoice, across all OpenStack core components [5]. In Ceilometer, resource usage measurement, e.g., CPU utilization, Disk Read Bytes, etc., is done by meters or counters. Typically there is a meter for each resource being tracked, and there is a separate meter for each instance of that resource. It is important to note that the lifetime of a meter is decoupled from the associated resource, and a meter continues to exist even after the resource it was tracking has been terminated [2]. Each data item collected by a meter is referred to as a "sample," and consists of a timestamp to mark the time of collected data, and a volume that records the value. Ceilometer also allows service providers to write their own meters. Such customized meters can be designed to conveniently collect data from inside launched VMs. For a PaaS or SaaS service, this feature allows the service providers to track application usage as well. OpenStack allows integration of multiple databases with Ceilometer for the purpose of storing metering data, e.g., MySQL, MongoDB, etc. We use MongoDB as that is the recommended and the default database in OpenStack because of features such as flexibility and ability to change the structure of documents in a collection over time.

MongoDB offers two key features of sharding and replication, which make it a perfect candidate for our approach [15]. Sharding is a method of storing data across multiple machines (shards) to support deployments with very large datasets and high throughput operations. Sharding helps in realizing scalable setups for storing metering data because the data collected by Ceilometer is expected to increase linearly over time. This is especially true for production servers. Replication allows multiple machines to share the same data. Unlike sharding, replication is mainly used to ensure data redundancy and facilitate load balancing. Finally, MongoDB also allows the use of the MapReduce [19], [3] framework for batch processing of data and aggregation options, which are highly relevant for the purposes of our application.

III. Design

In this section, we present the design of our fine-grained scalable metering framework. Figure 2 illustrates the overall architecture including the key components and their interactions. The main modules are data size estimator, resource profiler, resource predictor, auto-scalable setup for mediation and rating with metering store for Ceilometer, and load balancer.

The framework initiates a new sequence of operations upon receiving a heat template file when OpenStack is servicing a provisioning request. The template is first parsed to extract the information about the requested resources. This information is then used to estimate the expected change in the size of the metering data to be collected by Ceilometer. Meanwhile, the resource profiler module, which keeps track of the resources that are already in use, profiles their capacity for mediation and rating purposes. The resource predictor

module uses the information about the profiled and newly requested resources to estimate the additional resources that would be required for the mediation and rating of the provisioning request. The estimate is then used to scale the metering store, and the setup is finally launched along with the requested provisioning. The dynamic load balancer module comes into action and ensures that resource usage per VM does not exceed a predetermined threshold.

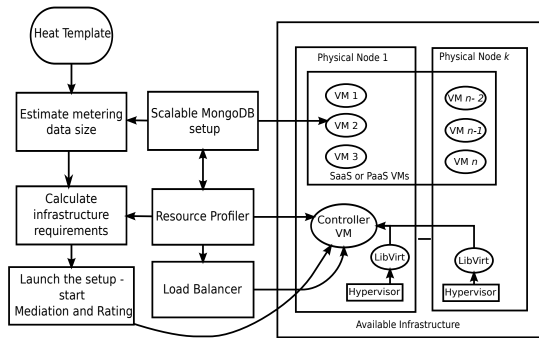


Figure 2. Overview of the fine-grained scalable metering and rating architecture.

A. Data Size Estimator

The data size estimator module calculates the expected changes in the size of metering data. For this purpose, the module uses the resources information from the heat template file of the provisioning request, and determines the set of meters that are required to perform the necessary monitoring and metering. Next, the expected total number of metering events on various polling intervals is calculated along with the average event object size. The number of events are calculated by parsing the pipeline.yaml file to fetch the sampling frequency of each meter. The average object event size is variable and depends on the type of meters launched and their sampling frequency or polling interval. To this end, the data estimator module keeps track of the changes in the event object size per meter and estimates the value by taking the average of the three previously collected values. The module then averages these values across the meters to determine the overall average object size. An alternative approach is to directly track the overall average object event size from the metering store's database.

B. Resource Profiler

Ceilometer launches various meters for monitoring and metering the usage of different resources per VM, e.g., CPU, memory, storage, networking, etc. The resource profiler module intercepts the metering data send to the metering store, and uses it to keep tabs on the per-VM resource utilization. A challenge is that the collected metering data only gives an instantaneous view of a VM's resource usage at a particular time instance, and do not necessarily portray the overall usage. To address this, the resource profiler uses a sliding window across last n metering samples to calculate a moving average and uses that as an estimate of the current per-VM resource utilization. An alternative would be that instead of intercepting the data, we query the metering store for overall utilization. However, this would unnecessarily burden the database and impact overall efficiency. Thus, we do not adopt the querying approach. The resource profiler also maintains queues of resources sorted based on estimated utilization. This information can be used to determine free resources within each VM, which in turn supports effective scaling of the metering setup.

C. Offline Resource Predictor

The job of offline resource predictor module is to analyze the data collected by the resource profiler and the data size estimator output to provide an assessment of the resources that would be required for the associated metering setup. A possible trade-off faced in such assessments of the needed resources is whether to underprovision or not the revenue management resources at the expense of performance degradation in terms of average time taken to process the collected metering data. We allow the providers to manage this trade-off by specifying the expected processing query time, query rate, and average load on the setup, as an input. Thus, based on the provided input, the resources predictor first determines if the current metering store setup will hold under the newly predicted incoming data without breaching the established thresholds. Maintaining the current setup avoids deployment oscillations. If case the threshold is breached, the replicator will calculate the optimum data partition emulating the addition of replica sets as shards. Only in the worst case scenario, when a partition cannot be reached due to high per-VM resource utilization on all the VMs, our predictor will resort to introduce new VM(s) to handle the management workload. This module outputs a recommended mediation and rating setup to achieve a potential minimum footprint.

D. On-line Fine Tuning Auto-Scaling of the Metering Store

Once the new request provisoning completed and the metering setup was updated as recommended by the predictor module, the metering data starts being collected and stored in the metering store database. The growing volume of the metering data entails that the database setup is scalable and efficient, and can handle complex queries in a timely fashion. This is crucial as the overall goal of our framework is to provide fine-grained pricing plans that require high-frequency querying. To this end, we have engineered an auto-scalable setup for MongoDB to act as the metering store for Ceilometer. Our setup is instantiated on the same set of VMs that are used to provide SaaS—as the VMs have been observed to be not fully utilized as stated earlier in Section I.

1) When to Scale?: The first step in realizing our auto-scalable MongoDb setup is to determine when scaling is needed. For this purpose, we use two kinds of metrics: i) OS-level metrics, e.g., CPU, memory, disk usage, etc.; and ii) MongoDB performance statistics, e.g., query time, writes/s, reqs/s, etc. Since the MongoDB instances are running on the same VMs as those providing user services, the VMs are already being monitored and this monitoring data can be reused to also determine the OS-level info needed for the scaling purpose. This information, coupled with periodically collected MongoDB statistics, are then used to determine if the metering store is loaded beyond a pre-specified high threshold or below a low threshold, and scaling decisions are made accordingly.

2) How to Scale?: The next step is to perform the scaling of the metering store. For this purpose, our framework exploits the creation of additional MongoDB replica sets. These replica sets are added as shards to achieve further partitioning of data, which in turn realizes the desired scalability of the setup. An important design decision while performing sharding is to carefully choose the sharding key. To this end, we keep track of the speedup achieved with various sharding keys and choose the best option. Note that replication and sharding are not mutually exclusive, and can be scaled individually based on the monitored reads/s or writes/s throughput observed through the MongoDB performance monitor.

E. Load Balancer and SLAs Enforcement

The selection of VMs for launching replicas for scaling-up the metering store is critical, as the additional management load may affect a VM's SaaS performance. Over time, this can lead to a point where the VM can no longer provide sufficient resources for the supported SaaS workload. Typically, cloud service providers set out service level agreements (SLA) for the performance of their services. When faced with a potential SLA violation, the cloud providers perform traditional workload re-placement or launch additional resources. To account for the additional management workload on the VMs, our framework complements the traditional load re-placement by using a load balancer that actively tracks the management workload on each VM. This is achieved by coordinating with the resource profiler. Thus, if the management workload on any of the VMs exceeds a certain threshold, that workload is either transferred to an existing VM with lower load or a new VM is launched to handle the management overload. The load balancer uses the process shown in Algorithm 1 to handle shards or replica sets. We exploit the MongoDB's internal load balancer, which upon creation of a new shard transfers chunks of 64 MB of data from other machines to the newly added shard(s) to evenly distribute the total number of chunks.

How to select the threshold for triggering load balancing operations?: The management workload threshold is set as to ensure that each resource is not over-utilized by the metering framework to an extent where the performance of the provided SaaS is affected. As the nature of the SLAs varies with the type of SaaS, as well as with the type of resources and configurations, the threshold is not fixed. Instead, it varies from solution to solution and we provide means for the resource managers to determine the appropriate thresholds.

Algorithm 1 The algorithm used for load balancing.

```
for each resource r monitored by Resource Profiler do
    rl ← List of VMs sorted by usage for r
    for each virtual machine vm in rl do
        cl ← current load on vm for r
        el ← Extra load on vm for r
        t ← Threshold for r
        if cl ≥ t then
            el ← cl − t
            REPLACE_VM(vm, el, r)
        else
            break
        end if
    end for
end for
function REPLACE_VM(vm, el, r)
    rl ← List of VMs sorted by usage for r
    for each virtual machine uvm in inverted rl do
        cl ← current load on uvm for r
        if cl + el ≤ t then
            Check threshold for rest of the resources
            use_this_vm ← uvm
            break
        else
            use_this_vm ← 0
        end if
    end for
    if use_this_vm ≤ 0 then
        use_this_vm ← launch a new virtual machine
    end if
    Transfer load from vm to use_this_vm
end function
```

F. System Controller

Finally, we provide a system controller module to control and fine-tune the scalable metering store, the resource profiler, and the load balancer. The module also acts as a facilitator for the various module operations by providing access to the collected data. We run the controller in a dedicated VM to ensure that it is not affected by the performance and workload dynamics of the resources.

G. Discussion

By default, OpenStack installs a standalone instance of MongoDB to store metering data. In order to perform mediation and rating, cloud service providers traditionally use a separate set of dedicated physical machines for a standalone installation of MongoDB. In case of huge data sizes, a distributed setup, e.g., Hadoop, is used for data processing. This approach requires redistribution of metering data from the metering store to the Hadoop Distributed File System (HDFS). This is burdensome as the data import into HDFS is identified as a major performance bottleneck [24], in addition to the expensive data copying. Our

Table I
SPECIFICATION OF VMS USED FOR EXPERIMENTATION

CPU	RAM (GB)	Write BW (MB/s)	Read BW (MB/s)	NW BW (MB/s)
8 Cores 3.0 GHz	8	380	533	100

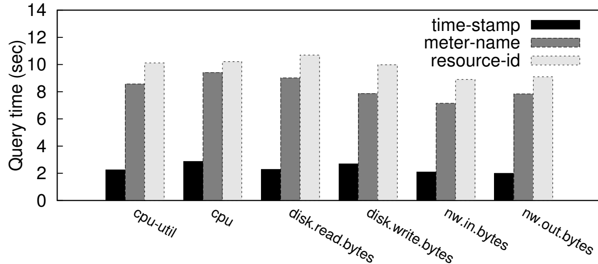


Figure 3. Effect of using different shard keys on query times. The query calculated variance in utilization (standard deviation) of Ceilometer counters using MapReduce.

approach has the advantage that it does not require such data redistribution. Instead, our approach collects data in a distributed setup to begin with and avoids the extra copying and import challenges and overheads. Another advantage of our framework is that it allows cloud service providers to support in addition to the fine-grained metering information, customizable price plans, e.g., charging from single metric based charge to complex multi-metric based charge.

Furthermore, our approach can also be applied for metering IaaS. However, this would require extending the framework and modifications such as: (i) launching the metering setup on physical nodes instead of VMs so that customers do not get access to the collected metered data; (ii) enabling monitoring of the physical nodes within Ceilometer for tracking infrastructure utilization per physical node instead of per VM; and (iii) updating the load balancer to effectively perform in heterogeneous environments so that cores not used by Nova can be used to launch metering services. Such modifications are focus of our ongoing and future research.

IV. EXPERIMENTAL SETUP

We deployed OpenStack Icehouse version 2014.1.1 on 20 physical machines, where each machine has six cores and 32 GB of RAM. We varied the number of VMs from 3 to 12 to emulate a SaaS. The metering data was collected from these VMs using variable sampling interval. We tracked the usage of VMs for a period of one month. We launched both default as well as customized meters to collect the resource usage. Table I shows the specifications of each VM.

We performed tests using both a standalone as well as a scalable MongoDB setup. In our scalable setup, each replica set consisted of only one node that acted as a primary copy of the data. The replica sets were added as shards to scale the MongoDB deployment. For testing purposes, we launched three configuration servers. One query router was deployed on the controller VM. All the performance related

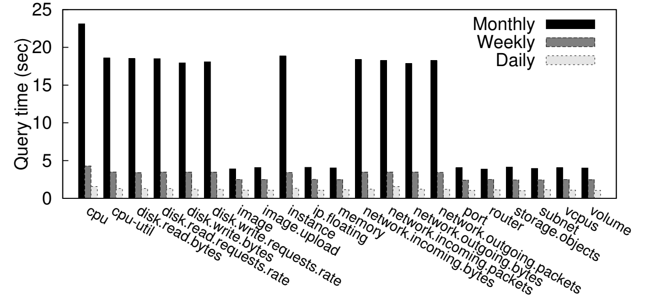


Figure 4. Comparison of query time at different granularity levels for various meters when processing user level data.

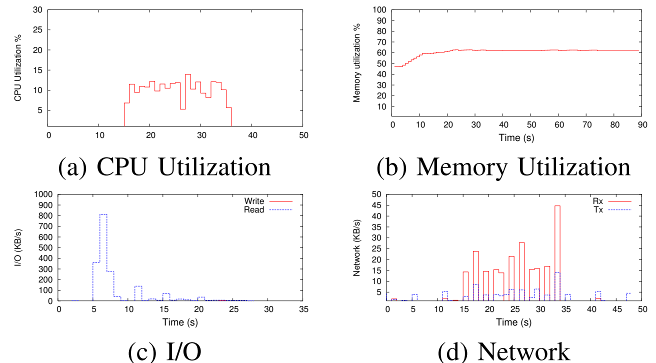


Figure 5. Resource usage while calculating variance in utilization (standard deviation) of various Ceilometer counters using MapReduce.

experiments were done on the actual collected metering data of more than 11 GB from the deployed OpenStack setup over the period of one month.

We used different sharding keys for the Ceilometer database in our tests. Figure 3 shows the effect of using different sharding keys on the query timings for a MongoDB setup consisting of 4 shards. As illustrated, the query time is affected more by the choice of the sharding key in the distributed setup compared to the standalone setup. Further investigation revealed that chunks greater than 64 MB were created in all cases except when timestamp of metering events was used as a shard key. This resulted in the MongoDB internal load balancer distributing chunks unevenly, with most of the chunks assigned to just one machine. This created a bottleneck and caused a significant increase in the query time. Consequently, the best sharding key to use in the target environment is timestamp.

V. EVALUATION

In this section, we evaluate the various aspects of our framework and demonstrate its effectiveness.

A. Data Size Estimation

In our first experiment, we compare the size of estimated and actual collected metering data associated with the 12 VMs launched within the OpenStack deployment with the default set of meters.

Figure 6 shows the results. The framework predicted that 254 events will be collected from the VMs every 10 minutes. The estimated average event object size was 1150 bytes, 1134 bytes, and 1188 bytes for per day, per week and per month calculations, respectively. As illustrated, the data estimator module predicted metering data size with 99% accuracy compared to the actual observed values.

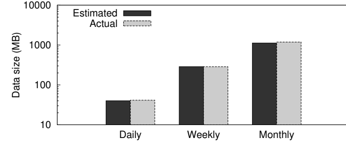


Figure 6. Comparison of estimated and actual collected metering data size.

B. Resource Profiler

In our next test, we measured the effect of performing mediation at different granularity levels. Figure 4 shows the time taken to perform mediation on the data of a single user using the statistics API provided by Ceilometer. We measured Maximum, Minimum, Average, Sum, and Count for the considered meters at three different granularity levels, namely, daily, weekly, and monthly. The results reveal that the meters that collect samples continuously at a fixed sampling interval took $4\times$ to $6\times$ more time to perform mediation on one month of data compared to one week of data. Next, we measured the increase in average resource utilization per VM due to mediation. Figure 5 (a) shows that the CPU utilization in the observed VMs did not increase above 15%. Similarly, the increase in memory utilization was observed to be less than 10%. As the collected data is inherently distributed over various VMs, the mediation is expected to generate reads but not writes. This is confirmed by the I/O usage shown in the Figure 5 (c), where the observed written data is almost zero, while the data read has a low average. Another key observation here is that due to the locally performed computation, the network usage is also negligible (see Figure 5 (d)). These results validate our claim that, in our approach, existing SaaS VMs can be used to perform mediation and rating tasks without affecting the performance of the provided SaaS.

C. Scalable MongoDB

In our next set of experiments, we analyze the effect of scaling our metering store, i.e., the distributed MongoDB setup, on the mediation duration. Figure 7 shows the reduction of the time to calculate the Variance (standard deviation) of various Ceilometer meters when using MongoDB's MapReduce [3] functionality, as we scale up the metering store. As illustrated, the standalone installation of MongoDB performs better than the single shard distributed MongoDB setup; this is due to the networking overhead. However, as we increase the number of shards, the mediation duration decreases. For the case of two replica sets acting as shards, the average query time is half of that of the standalone setup. Further increasing the number of shards shows increasingly better performance in terms of query time. We also observe that the reduction in the query time is not linear and after reaching a certain threshold, the networking overhead actually causes performance degradation. Figure 8 shows the impact of scaling MongoDB on query execution time for calculating the Average, Sum, Maximum, and Minimum when using the aggregation functionality [2] provided by MongoDB. Here, a more linear trend was observed when performing mediation using the aggregation.

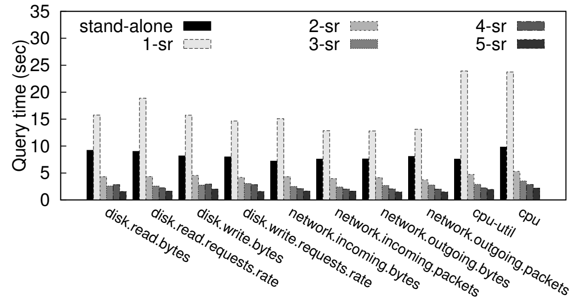


Figure 7. The impact on execution time of a query to calculate variance in the utilization of various Ceilometer counters using MapReduce, due to scaling of MongoDB. "sr" represents the number of sharded replica sets used.

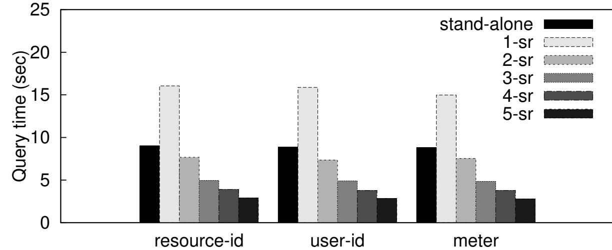


Figure 8. The impact of scaling MongoDB on query execution time when calculating Average, Sum, Maximum, and Minimum using aggregation for different levels.

Table II
TIME TAKEN FOR SCALING THE TEST MONGODB SETUP.

Scaled from x to y shards	Time to scale (minutes)	Total chunks transferred	Chunks transferred per shard
1 to 2	10	93	93
2 to 3	6	62	31
3 to 4	3.5	47	15
4 to 5	2	37	9

D. Load Balancer and Effect of Scaling

In our next test, we analyze the load balancer, the effect of scaling the setup, and the role of load balancing. Figure 9 shows the overall behavior of the framework. We used traces from IBM production servers to mimic SaaS load on VMs and launched our framework management load on top of those VMs. The framework launches initially one sharded replica on a set of five VMs with 0 GB collected metering data. At point A, as the size of collected data reaches 1.1 GB, the load balancer triggers the launching of another sharded replica to avoid SLA violation. At point B, a new set of five SaaS VMs is added to the setup. Based on the calculation

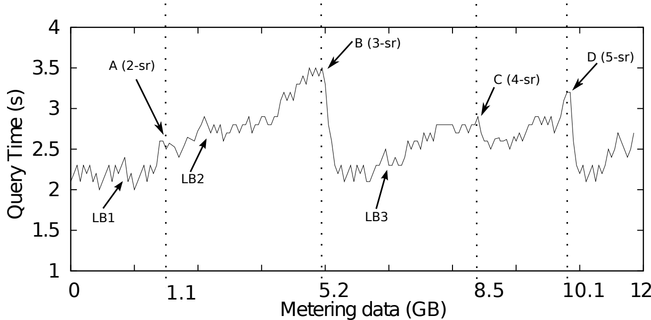
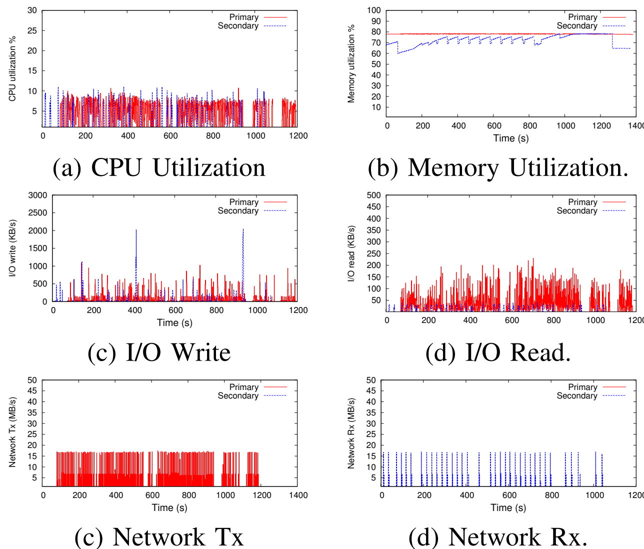


Figure 9. Evaluation of overall framework with Load Balancer.



(a) CPU Utilization (b) Memory Utilization.

(c) I/O Write (d) I/O Read.

(c) Network Tx (d) Network Rx.

Figure 10. Resource usage under MongoDB setup scaling from one to five shards.

performed by the offline resource predictor, a third sharded replica is added to keep the query time under 3.5 sec. At point C, as the collected data size reaches 8.5 GB, the load balancer launches another shareded replica to reduce the load on the VMs. At point D, two new SaaS VMs are added. The offline resource predictor, after calculating the expected metering data size, recommends adding another sharded replica. At the points labeled LB1, LB2, and LB3, the load balancer successfully shifted load from one VM to another to avoid both an SLA violation and launching a new VM.

Table II shows the time taken to scale the metering store, the total number of chunks transferred and the chunks transferred per shard. MongoDB, by default, transfers only one chunk at a time, which slows down the transferring process. The numbers show that it is important to take the transfer time into consideration while making setup scaling decisions. Furthermore, transferring chunks while scaling the setup also requires additional resources and adds an observable overhead to the SaaS VMs. Figure 10 shows this overhead when scaling from one shard to five, in terms of resource usage per VM for both the primary VM, i.e., the source of a chunk transfer, and the secondary VM, i.e., the destination of the chunk. We observe that while the CPU utilization is high on the primary VM, it never exceeds 10% of the CPU before the chunk transfer. Similarly, the memory utilization remains constant for the primary VM but goes up by 5% to 10% for the secondary VM compared to the pre-transfer memory usage.

The amount of data written in both the primary and the secondary VMs remains almost unaffected, although high spikes of up to 2 MB/s are observed in the write I/Os. In contrast, the read I/O is higher for the primary VM as compared to secondary VM. The average write rate on the primary VM is observed to be 0.5 MB/s while for the secondary VM it is 0.4 MB/s. Lastly, the network transmission and reception rates stay below 20 MB/s. The above results confirm our expectations on the low impact of our approach on the SaaS VMs.

VI. Related Work

The focus of several recent works [29], [8], [10], [16], [27], [18], [31] is on providing an efficient and scalable cloud monitoring setup, however, these works do not consider or discuss scalability of the mediation and rating systems. In contrast, our approach is designed for scalable deployment. Furthermore, our approach is also unique in that it uses existing VMs and only launches additional VMs rarely, thus incurring little additional cost. [7] provides analysis of cloud monitoring by discussing motivations and basic concepts, and pointing out open research issues. Similarly, [14] focuses on clustering VMs depending on the resource usage to simplify the monitoring requirements.

A pay-as-you-go scheme has also been proposed [25], which employs a machine-learning-based prediction model of the relative cost of interference between metering/rating and SaaS applications. Similarly, [26] describes a metering and pay-as-you-go model and proposes a solution to meter resources. However, unlike our approach, the focus of this model is to come up with a metering approach for enabling monitoring of cache space and memory bandwidth.

CloudSim [13] proposes a toolkit to enable modeling and simulation of cloud computing environments and perhaps is the closest to our work in terms of profiling and predicting the resources required for supporting cloud applications. Similarly, PRESS [22] proposes a PRedictive Elastic reSource Scaling scheme for cloud systems. The significant difference between our approach and these works is that they predict load as a standalone applications, whereas in our case we predict the additional load that can be added to the existing VMs that are already loaded. Moreover, there has been a lot of work done on SLA based resource allocation, such as Oceano [9] that is a prototype for resource allocation for enabling flexible service level agreements (SLAs). Similarly, [23] provides an algorithm for SLA based resource allocation for multi-tier cloud computing systems. However, the problem addressed by our framework is unique

in that we have to place and shift only the prioritized part of the overall load to ensure SLA.

Several recent works employ a database for enabling a monitoring system. An elastically-scalable database management system is designed in [17], based on the argument that in spite of the elasticity offered by the cloud infrastructure, the backend database still is the scalability bottleneck for cloud applications. A monitoring system that collects and stores the metering data in distributed database is presented in [28], but lacks the ability to scale the setup and use existing VMs. Similarly, a scalable metering architecture is developed in [21]. These works are orthogonal to our work, and we leverage the techniques therein when possible to achieve a scalable and flexible metering and rating system for the SaaS applications in the cloud.

VII. Conclusion

We designed a framework for SaaS and PaaS cloud service providers to scale their management systems in a cost-aware manner that minimizes the non-revenue generating management resources. We evaluated the ability of our framework to achieve this goal by co-locating the revenue management tools on VMs used to provide SaaS. The results of our experiments show that our efficient approach to deploying revenue systems has small impact on the co-located SaaS performance while providing dynamic scaling at minimal cost. Although we focused on the revenue management only, our methodology can be applied to other aspects of the service management.

Acknowledgment

This work was sponsored in part by the NSF under Grants CNS-1405697 and CNS-1422788.

References

[1] Survey: Cloud still underutilized, 2011. http://goo.gl/CVuLuo.

[2] Ceilometer Quickstart, 2014. https://openstack.redhat.com/CeilometerQuickStart.

[3] MongoDB MapReduce, 2014. http://docs.mongodb.org/manual/core/map-reduce/.

[4] OpenStack, 2014. htt://http://www.openstack.org/.

[5] Openstack docs, 2014. http://docs.openstack.org/.

[6] Survey: Cloud still underutilized, 2014. http://goo.gl/oXU1tE.

[7] G. Aceto, A. Botta, W. de Donato, and A. Pescapè. Cloud monitoring: Definitions, issues and future directions. *CLOUDNET*, 2012.

[8] A. Anwar, A. Sailer, A. Kochut, C. O. Schulz, A. Segal, and A. R. Butt. Scalable metering for an affordable it cloud service management. In *IEEE IC2E'15*.

[9] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano-sla based management of a computing utility. In *IM, IEEE/IFIP*, 2001.

[10] A. Brinkmann, C. Fiehe, A. Litvina, I. Luck, L. Nagel, K. Narayanan, F. Ostermair, and W. Thronicke. Scalable monitoring system for clouds. In *IEEE/ACM UCC*, 2013.

[11] R. Buyya, R. Ranjan, and R. N. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *Algorithms and architectures for parallel processing*, pages 13–31. Springer, 2010.

[12] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[13] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. 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*, 41:23–50, 2011.

[14] C. Canali and R. Lancellotti. Automated clustering of vms for scalable cloud monitoring and management. In *SoftCOM*. IEEE, 2012.

[15] K. Chodorow. *Scaling MongoDB*. " O'Reilly Media, Inc.", 2011.

[16] W.-C. Chung and R.-S. Chang. A new mechanism for resource monitoring in grid computing. *Future Generation Computer Systems*, 25(1):1–7, 2009.

[17] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM TODS*, 38(1):5, 2013.

[18] S. A. De Chaves, R. B. Uriarte, and C. B. Westphall. Toward an architecture for monitoring private clouds. *Communications Magazine, IEEE*, 2011.

[19] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[20] B. Di Martino, S. Venticinque, D. Kyriazis, and S. V. Gogouvitis. A comparison of two different approaches to cloud monitoring. In *Inter-cooperative Collective Intelligence: Techniques and Applications*, pages 69–91. Springer, 2014.

[21] E. Elmroth, F. G. Marquez, D. Henriksson, and D. P. Ferrera. Accounting and billing for federated cloud infrastructures. In *GCC*. IEEE, 2009.

[22] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *CNSM 2010*. IEEE.

[23] H. Goudarzi and M. Pedram. Multi-dimensional sla-based resource allocation for multi-tier cloud computing systems. In *IEEE CLOUD*, 2011.

[24] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *IEEE CDE 2011*.

[25] S. Ibrahim, B. He, and H. Jin. Towards pay-as-you-consume cloud computing. In *IEEE SCC*, 2011.

[26] R. Iyer, R. Illikkal, L. Zhao, D. Newell, and J. Moses. Virtual platform architectures for resource metering in datacenters. *ACM SIGMETRICS*, 2009.

[27] X. Jiang and X. Wang. out-of-the-box monitoring of vm-based high-interaction honeypots. In *Recent Advances in Intrusion Detection*, pages 198–218. Springer, 2007.

[28] L. Kai, T. Weiqin, Z. Liping, and H. Chao. Scm: A design and implementation of monitoring system for cloudstack. In *CSC 2013*. IEEE.

[29] W. Richter, C. Isci, B. Gilbert, J. Harkes, V. Bala, and M. Satyanarayanan. Agentless cloud-wide streaming of guest file system updates. In *IEEE IC2E'14*.

[30] O. Sefraoui, M. Aissaoui, and M. Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[31] Y. Sun, Z. Xiao, D. Bao, and J. Zhao. An architecture model of management and monitoring on cloud services resources. In *ICACTE*. IEEE, 2010.

[32] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 2008.