Scalable Metering for an Affordable IT Cloud Service Management

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 commodities, cloud service providers have to carefully choose the metering and rating tools and scale their infrastructure to effectively process the collected metering data. In this paper, we focus on the metering and rating aspects of the revenue management and their adaptability to business and operational changes. We design a framework for IT cloud service providers to scale their revenue systems in a cost-aware manner. The main idea is to dynamically use existing or newly provisioned SaaS VMs, instead of dedicated setups, for deploying the revenue management systems. At onboarding of new customers, our framework performs off-line analysis to recommend appropriate revenue tools and their scalable distribution by predicting the need for resources based on historical usage. This allows the revenue management to adapt to the ever evolving business context. We evaluated our framework on a testbed of 20 physical machines that were used to deploy 12 VMs within OpenStack environment. Our 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. By dynamically scaling the setup, we were able to reduce the metering data processing time by many folds without incurring any additional cost.**

I. Introduction

A crucial challenge, especially for a sustainable IT business model, is how to adapt the cloud service management, and implicitly its cost, e.g., impact of associated monitoring overhead, to dynamically accommodate the changes in service requirements and data centers [17]. As the cloud services journey through their lifecycle towards commodities, a challenging change to their management, specifically to the revenue management, is the demand for more granular pricing model such as pay-as-you-go and usage-based, rather than the extant coarse grain model that uses metrics such as VM hours [6].

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 usage based pricing [21]. As a particular technology or service becomes more of a commodity (e.g., Infrastructure Services, or IaaS), more and more customers are interested in fine-grained pricing based on their actual usage or "pay as you go" model. For instance, from the perspective of a Software as a Service (SaaS) customer, it is more advantageous to be charged based on the usage of the platform (e.g., the number of http transactions or volume of the db2 queries) instead of a fixed monthly fee, especially when the usage is low. Lack in providing the usage based pricing policy for commodity services may result in losing customers and eventually the market share [9]. Hence, from the cloud service provider perspective, maintaining the competitive advantage by effectively adapting to versatile pricing policies has become a matter of high priority.

Usage based pricing policies bring a new set of service management requirements for the service providers, particularly for their revenue management [28]. 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. This is because, the service resources and applications need to be monitored at the appropriate level to provide the usage that has to be charged for, which may result in collecting a large amount of metered data. Furthermore, this metered data needs to be processed in order to perform: (1) the mediation, i.e., transformation into the desired units of measure expected by the usage price policy, e.g., average, maximum or minimum usage; (2) the rating based on the price policy for generating the invoice for the customers, e.g., multiplying usage by per unit rate; and (3) the calculation required to generate custom reports regarding the customers' usage trends. Hence, additional resources are required not only to store data, but also to process it for supporting such finer-grained service management.

Cloud service providers that align their services price plan to usage based pricing have to carefully choose the metering, mediation, and rating tools and infrastructure to minimize the cost of the resource requirements for performing them. Thus, the first step in performing this cost benefit analysis is to accurately estimate the cost associated with monitoring, storing and processing the data for the various metering and rating tools [8]. The cost of fine grained monitoring depends on the volume of data collected for the purpose of metering [8]. The extant 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

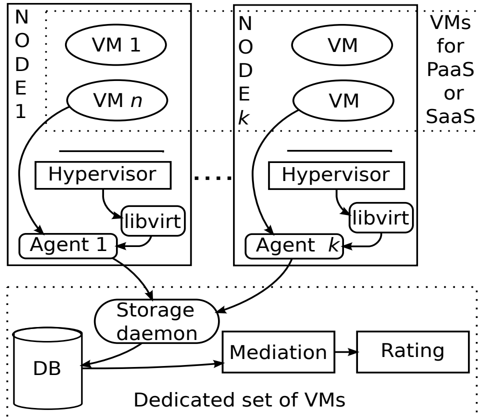

Figure 1. Current approach for metering cloud services.

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 from OpenStack 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.

Finer-grain pricing necessitates dynamically modifying the price plans offered to the customers based on the market demand [25]. In this context, the key challenge is how to provide a scalable metering setup that can adapt with the price policy updates and the load in the data center, while minimizing the additional resources, performance impact and interference so as to avoid toll on the business outcome. As 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 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. Depending on the cloud service type, for instance SaaS, cloud service providers may themselves be customers of IaaS or PaaS (Platform as a Service). As such, they are charged 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 [5], [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.

II. Enabling Technologies

In this section, we discuss the enabling technologies and provide the background for our proposed approach.

A. *OpenStack and Ceilometer*

We have designed our protocol on well-established cloud ecosystem of OpenStack—an open source project that provides a massively scalable cloud operating system. OpenStack adopts a modular design and has become the de facto cloud computing platform for managing large pools of compute, storage, and networking resources in modern data centers. 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 aim of Ceilometer is to deliver a unique point of contact for billing systems to acquire all of the measurements needed to establish customer billing, across all OpenStack core components [4]. Ceilometers' primary targets are monitoring and metering, but the framework is flexible and can be extended to collect data to support other needs as well.

The main components of ceilometer can be divided into two categories, namely agents, e.g., compute agents, central agents, etc., and services, e.g., collector service, API service, etc. The compute agents poll the local `libvirt` daemon to fetch resource utilization of launched VMs and emit this data as AMQP [29] notifications on the message bus called Ceilometer bus. Similarly, central agents poll the public RESTful APIs of OpenStack services, such as Cinder and Glance, to track resources and emit this data onto the OpenStack's common message bus called Notification bus. On the other hand, the collector service collects the AMQP notifications from the agents and other OpenStack services, and dispatches the collected information to the metering database. Finally, the job of the API service is to present aggregated metering data to the billing engine.

In Ceilometer, resource usage measurement, e.g., CPU utilization, Disk Read Bytes, etc., is done by meters or counters. Typically there exists a meter for each resource being tracked, and there is a separate meter for each instance of the 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 Solution or Software, this feature allows the cloud service providers to track the application usage as well.

B. MongoDB

OpenStack allows integration of multiple databases with Ceilometer for the purpose of storing metering data, e.g., MySQL, MongoDB, etc. However, MongoDB is recommended and is the default database in OpenStack because of features such as flexibility and allowing the structure of documents in a collection to be changed over time. In the following, we discuss MongoDB and the features that allow us to scale up or scale down the proposed setup.

MongoDB is a cross platform document-oriented NOSQL [11] database. MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas, making the integration of metering data easier and faster. MongoDB offers two key features of sharding and replication, which make it a perfect candidate for our approach [12].

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. Another feature of MongoDB is replication, which 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 [16], [3] framework for batch processing of data and aggregation options.

III. DESIGN

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

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. The information is then used to estimate the expected change in the size of metering data that will be collected by Ceilometer. Meanwhile, the resource profiler module keeps track of the resources that are already in use, and profile their usage for mediation and rating purposes. The resource predictor module uses the profiled and newly requested resources information to estimate the additional resources that would be required for mediation and rating for the provisioning request. The estimate is then used to scale the metering store, and the setup is finally launched along with the requested provisioning.

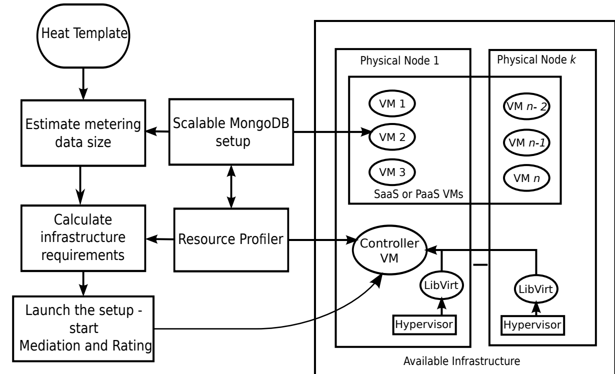

Figure 2. Overview of the fine-grained metering and rating architecture.

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. List 1 shows a sample collected from the metering store to measure the overall average object size. Finally, the expected size of metering data (2664719556 bytes) is determined by multiplying the number of objects (2239713) with the average event object size (1189.759382 bytes).

List 1. Sample of average event object size collected from metering store

```
> db.stats()
{
        "db" : "ceilometer",
        "collections" : 6,
        "objects" : 2239713,
        "avgObjSize" : 1189.759382,
        "dataSize" : 2664719556,
        "storageSize" : 3217911760,
        ...
}
```

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 provide an approximate estimate of the resources that would be required for the associated metering setup. A possible trade-off faced in such estimation of the needed resources is whether use less revenue management resources at the expense of performance degradation in terms of average time taken to process the collected metering data. We allow the managers to manage this trade-off by specifying the expected processing query time, query rate, and average load on the setup, as an input. Based on the provided input this module outputs a recommended mediation and rating setup to achieve an effective estimate for driving our framework decisions.

D. Auto-Scalable Metering Store

The metering related data is collected and stored in the metering store that is typically provided using a 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 the monitoring data can be reused to determine the OS-level info needed for this purpose as well. This information, coupled with periodically collected MongoDB statistics, is 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 enact scaling of the metering store. For this purpose, our framework exploits creation of additional MongoDB replica sets. These replica sets are added as shards to achieve further partitioning of data, which in turn realized 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. System Controller

Finally, we provide a system controller module to control and fine-tune the scalable metering store and the resource profiler. The module acts also 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.

F. Discussion

By default, OpenStack installs a standalone instance of MongoDB to store metering data. In order to perform mediation and rating, cloud service providers usually use a separate set of dedicated physical machines for 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 data ingestion into HDFS is identified as a major performance bottleneck [20], not to mention expensive data copying. Our approach has the advantage that it does not require such data redistributed. Rather, our approach collects data in a distributed setup to begin with and avoid extra copying and ingestion challenges and overheads. Another advantage of our framework is that it allows cloud service providers to offer not only the fine-grained metering information, but also customizable price plans, e.g., charging customers only on CPU utilization, etc.

Furthermore, our approach can also be applied for metering IaaS. However, this would require extending the framework and modifications such as: (i) launching the

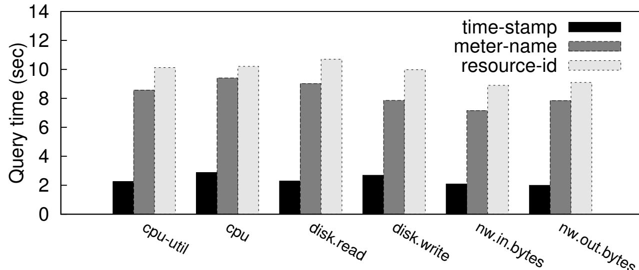

Figure 3. Effect of using different shard keys on query times. The query calculated variance in utilization (standard deviation) of Ceilometer counters using MapReduce.

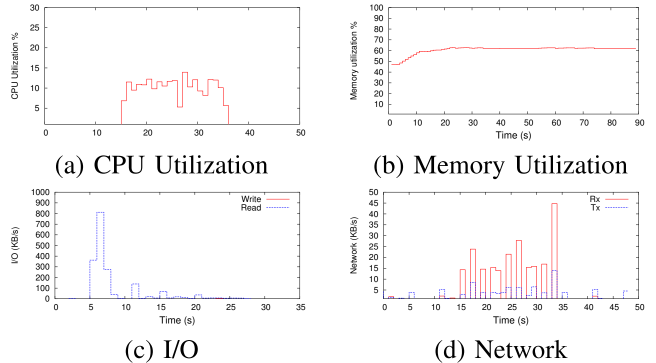

Figure 5. Resource usage while calculating variance in utilization (standard deviation) of various Ceilometer counters using MapReduce.

metering setup on physical nodes instead of VMs so that customers do not get access to the collected metered data; and (ii) enabling monitoring of the physical nodes within Ceilometer for tracking infrastructure utilization per physical node instead of per VM.

IV. Experimental Setup and Evaluation

For proof of concept evaluation, we have implemented the fine-grained metering approach as discussed in Section III on top of OpenStack and MongoDB. We used Java to implement data size estimator whereas Python was used to implement the resource profiler, resource predictor and controller modules. 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 provide 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.

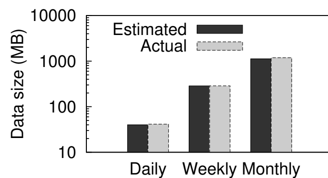

Figure 4. Comparison of estimated and actual collected metering data size.

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. Furthermore, the replica sets were added as shards to scale the MongoDB deployment. For testing purposes, we launched three configuration servers but only one query router that was deployed on the controller VM. All the performance related 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 out tests. Figure 3 shows the effect of using different sharding keys on the query timings for a MongoDB setup consisting of 4 shards. It can be seen that the query time is affected more by the choice of the sharding key for 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, instead of counter name, user-id or resource-id.

Figure 4 shows the comparision of estimated and actual collected metering data size. The framework predicted that 254 events will be collected from the launched 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 seen in the Figure, compared to the actual observed values, the data estimator module predicted metering data size with 99% accuracy.

Figure 5 shows that 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 needed data is already distributed to the various VMs, so mediation is expected to generate reads but not writes. This is confirmed by the I/O usage shown in the Figure, with observed written data at almost zero and also with low average data read. Another key observation here is that due to the computation being mostly performed locally, the network usage is also negligible. These results validate our claim that, if handled properly as in our approach, existing VMs can be used to perform mediation and rating tasks without affecting the performance of the provided SaaS.

V. Related Work

The focus of several recent works [26], [7], [13], [23], [15], [27] 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.

A pay-as-you-go scheme has also been proposed [21], which employs a machine-learning-based prediction model of the relative cost of interference between metering/rating and SaaS applications. Similarly, [22] 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 [10] proposes a toolkit to enable modelling 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 [19] 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.

Several recent works employ a database for enabling a monitoring system. An elastically-scalable database management system is designed in [14], 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 [24], but lacks the ability to scale the setup and use existing VMs. Similarly, a scalable metering architecture is developed in [18]. 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.

VI. Conclusion

We designed a framework for IT cloud service providers to scale their revenue management systems in a cost-aware manner. We evaluated the ability of our framework to use existing SaaS VMs for the purpose of metering, for analysis purposes we used 20 physical node setup to launch variable number of virtual machine ranging from 3 to 12 within the OpenStack environment. The results show that our approach is promising, and has small impact on the co-located SaaS while providing for dynamic scaling at minimal cost.

Acknowledgment

This work is sponsored in part by the NSF under the CNS-1405697 and CNS-1422788 grants.

References

[1] Survey: Cloud still underutilized, 2011. http://windowsitpro.com/article/cloud-business-issues/Survey-Cloud-still-underutilized-129534.
[2] Ceilometer Quickstart, 2014. https://openstack.redhat.com/CeilometerQuickStart.
[3] MongoDB MapReduce, 2014. http://docs.mongodb.org/manual/core/map-reduce/.
[4] Openstack docs, 2014. http://docs.openstack.org/.
[5] Survey: Cloud still underutilized, 2014. http://www.datacenterdynamics.com/focus/archive/2014/06/companies-wasting-%C2%A31bn-year-underused-cloud-capacity.
[6] M. Armbrust, O. Fox, R. Griffith, A. D. Joseph, Y. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. M.: Above the clouds: A berkeley view of cloud computing. 2009.
[7] 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.
[8] 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.
[9] 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.
[10] 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.
[11] R. Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 2011.
[12] K. Chodorow. *Scaling MongoDB*. " O'Reilly Media, Inc.", 2011.
[13] 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.
[14] 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.
[15] S. A. De Chaves, R. B. Uriarte, and C. B. Westphall. Toward an architecture for monitoring private clouds. *Communications Magazine, IEEE*, 2011.
[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.
[17] 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.
[18] E. Elmroth, F. G. Marquez, D. Henriksson, and D. P. Ferrera. Accounting and billing for federated cloud infrastructures. In *GCC*. IEEE, 2009.
[19] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *CNSM 2010*. IEEE.
[20] 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*.
[21] S. Ibrahim, B. He, and H. Jin. Towards pay-as-you-consume cloud computing. In *IEEE SCC*, 2011.
[22] R. Iyer, R. Illikkal, L. Zhao, D. Newell, and J. Moses. Virtual platform architectures for resource metering in datacenters. *ACM SIGMETRICS*, 2009.
[23] 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.
[24] L. Kai, T. Weiqin, Z. Liping, and H. Chao. Scm: A design and implementation of monitoring system for cloudstack. In *CSC 2013*. IEEE.
[25] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi. Cloud computingthe business perspective. *Decision Support Systems*, 51:176–189, 2011.
[26] 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*.
[27] Y. Sun, Z. Xiao, D. Bao, and J. Zhao. An architecture model of management and monitoring on cloud services resources. In *ICACTE*. IEEE, 2010.
[28] 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.
[29] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 2006.