How A Consumer Can Measure Elasticity for Cloud Platforms

Sadeka Islam
National ICT Australia,
University of New South Wales
sadeka.islam@nicta.com.au

Kevin Lee
National ICT Australia,
University of New South Wales
kevin.lee@nicta.com.au

Alan Fekete
University of Sydney,
National ICT Australia
alan.fekete@sydney.edu.au

Anna Liu
National ICT Australia,
University of New South Wales
anna.liu@nicta.com.au

ABSTRACT

One major benefit claimed for cloud computing is elasticity: the cost to a consumer of computation can grow or shrink with the workload. This paper offers improved ways to quantify the elasticity concept, using data available to the consumer. We define a measure that reflects the financial penalty to a particular consumer, from under-provisioning (leading to unacceptable latency or unmet demand) or over-provisioning (paying more than necessary for the resources needed to support a workload). We have applied several workloads to a public cloud; from our experiments we extract insights into the characteristics of a platform that influence its elasticity. We explore the impact of the rules used to increase or decrease capacity.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: PERFORMANCE OF SYSTEMS—*Measurement techniques, Modeling techniques*; D.2.8 [**SOFTWARE ENGINEERING**]: Metrics—*Complexity measures, Performance measures*

Keywords

Cloud Computing, Elasticity, Performance Measures

1. INTRODUCTION

Cloud computing platforms are already used extensively by some companies with huge and rapidly growing computational needs, and traditional enterprises are looking closely at the cloud as an addition or even an alternative to running their own IT infrastructure. Many features of cloud platforms are attractive; e.g., cloud platforms may be low-cost and they may achieve high availability. One feature that is commonly a powerful selling point for cloud platforms is the claim that they are *elastic*; the user can pay only for what they need at a given time. The (US) National Institute of Standards and Technology (NIST)[1] defines elasticity:

> Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out, and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time.

We note here two fundamental elements of elasticity: (1) Time (e.g., a platform is more elastic if resources are available sooner after a request is made), and (2) Cost (e.g., a platform charging on a per-hour basis is less elastic than one charging on a per-second basis).

Elasticity is a desirable property for many businesses. A startup facing rapid growth wishes that its costs start small, and grow as and when the income arrives to match. In contrast, traditional data processing requires a large up-front capital expenditure to buy and install IT systems. Traditionally, the cost must cover enough processing for the anticipated and the hoped-for growth; this leaves the company bearing much risk from uncertainty in the rate of growth. If growth is slower than expected, the revenue won't be available to pay for the infrastructure, while if growth is too fast, the systems may reach capacity and then a very expensive upgrade or expansion is needed. Also, it is common in web-based companies for demand to be periodic or bursty (e.g., the Slashdot effect). The workload may grow very rapidly when the idea is "hot", but fads are fickle and demand can then shrink back to a previous level. Traditional infrastructure must try to provision for the peak, and so it risks wasting resources after the peak has passed. In summary, elasticity can remove risk from a startup or an enterprise, by allowing "pay-as-you-grow" computing infrastructure where the costs adjust smoothly to rising (and perhaps falling) workload.

While all cloud offerings claim elasticity as a virtue that they possess, we can be sure that none are perfect in letting a customer pay for exactly what they need, and no more. There may be a minimum charge (e.g., 1 instance), there may be delays in adapting the platform to sudden increase in workload, and so on. Thus we would want to know

[1]See `http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf`

how elastic is each system. As yet, the literature does not contain any explicit measurement to quantify the amount of elasticity in a platform. This is the gap that our paper addresses. We are *not* proposing a mechanism for minimising resource usage. Rather, we are proposing elasticity as a *property* of a platform with time and cost as its essential elements, and we demonstrate how elasticity can be measured with respect to applications with certain Quality of Service (QoS) requirements. We envisage that different mechanisms for minimising resource usage can be evaluated using our elasticity measure.

Just as with traditional IT infrastructure, the company seeking to use a cloud platform needs a basis for comparing different offerings, and choosing the one that will be best for its needs. The usual approach is to follow a benchmark, which includes a standardised workload, and defines exactly how to measure the behavior of any system when subjected to this workload. The benchmark gives one (or a few) summary numbers that represent the value to the chooser of the system; it is then reasonable to select the system with best benchmark results. Organisations like TPC and SPEC have a range of benchmarks for hardware and/or software systems. This paper is part of an effort in the research community to define appropriate measures that will allow comparing the desirableness of competing cloud platforms.

We draw attention to the difference between elasticity and scalability, since the two notions are often mixed up. Scalability is defined as the ability of a system to meet a larger workload requirement by adding a proportional amount of resources. It means that the system must be able to handle a high workload in a graceful manner (i.e., maintaining its performance). However, scalability is a time-free notion and it does not capture how long it takes for the system to achieve the desired performance level. In contrast, time is a central aspect in elasticity, which depends on the speed of response to changed workload.

One important feature of our work is that we regard benchmarking as a process done by the consumer of cloud services. We limit ourselves to running a benchmark as a consumer - this includes considering her application and workload profile, incorporating the business objectives, and taking observations that are available to the consumer through the platform's API or inside the user's application code. This makes our task harder, since we do not have access to arbitrary measurements of the infrastructure itself, but this viewpoint is necessary for our work to give the consumer a reasonable basis for choosing between competing platforms. In contrast, the provider's viewpoint of elasticity can be completely different because they have access to the measurements from the underlying physical infrastructure (hardware configuration and specification) and virtualisation environment. This enables them to do performance modelling and looking at interactions between components to deduce the performance outcome for a class of applications. For the consumer's viewpoint, we try to understand the elasticity behavior of the platform from its response to a suite of workload patterns.

A key idea in our proposal is to come up with a number that measures elasticity as a property of a cloud platform (though the actual figure-of-merit will vary depending on the application's business model, workloads, etc). To achieve this, we use workloads that vary over time in different ways. Some workloads will rise and fall repeatedly, others will rise rapidly and then fall back slowly, etc. For each workload, we examine the way a platform responds to this, and we quantify the effect on the consumer's finances. That is, we use a cost measure in cents per hour, with a component that captures how much is wasted by paying for resources that are not needed at the time for the workload (overprovisioning), and a component to see how much the consumer suffers (opportunity cost) when the system is underprovisioned, that is, the platform is not providing enough resource for a recent surge in workload.

From our measurements, we have discovered several characteristics of a cloud platform that are important influences on the extent of elasticity. Some of these (such as the speed of responding to a request for increased provisioning) have already been discussed by practitioners, but others seem to have escaped attention. For example, we find in Amazon EC2 that there is a large improvement of elasticity from relatively simple changes in the set of rules that the system uses to control provisioning and deprovisioning. There is a set of rules (based on recent utilisation rates) that is widely followed, perhaps because it is done that way in tutorial examples. We find that this leads to rapid deprovisioning when load decreases, which leaves the system underprovisioned if a future upswing occurs. Because the financial impact from poor QoS (when demand can't be handled) is generally much more severe than the cost of running some extra resources for a while, this is a poor strategy. What is worse, on typical platforms one pays for an instance in quanta that represent a significant period of time (say 1 hour), so eagerness to deprovision can leave the consumer paying for a resource without the ability to use it.

The key contributions that this paper makes are (i) a novel framework for measuring elasticity, that can be run by a consumer and which takes account of the consumer's particular business situation (ii) a specific set of case studies, using one set of workload patterns and financial assumptions, to show that this approach can be carried out in practice, (iii) insights that we gained from the case studies, especially concerning the main internal characteristics of a platform that impact on the elasticity experience of a consumer.

This paper is structured as follows: Section 2 identifies relevant work on elasticity and cloud performance measurements. Section 3 presents our proposed benchmark for measuring elasticity, first as a general framework based on penalties that are expressed in monetary units, and then with specific choices for penalty functions, workload curves etc. Section 4 describes the details of the experimental setup, including the tools and specific cloud technologies used in our case studies. Section 5 presents empirical case studies that show the elasticity of particular platforms, given by different choices of rulesets that control provisioning decisions in a widely-used public cloud. We close the paper with a discussion of the lessons learnt and a conclusion.

2. RELATED WORK

The main objective of this paper is to provide a way for a consumer to measure how well (or not) each cloud platform delivers the elasticity property. The most relevant prior research is concerned with understanding the elasticity concept, and with measuring cloud platform performance. This work has appeared in formal academic forums, and/or in trade press or practitioner blogs.

2.1 Elasticity : Definition and Characteristics

It is important to get common usage of marketable terms such as "elastic". Several efforts explain the meaning of terms that are used about cloud platforms, and along with explanation, they point to aspects of the platform's performance that can be important for elasticity. Unlike our work, these studies do not give explicit measurement proposals.

Armbrust et al. [1] discuss relevant use cases and potential benefits for cloud platforms. Among their points, they draw attention to the value of cloud elasticity as compared to the conventional client-server model in the context of perceived risks due to over- and under- provisioning.

The prestigious National Institute of Standards and Technology (NIST) has pointed out rapidity in resource provisioning and de-provisioning as an important aspect of elasticity. Two other discussions from David Chiu[2] and Ricky Ho[3] have pointed to an additional important factor of elastic behavior, that of the granularity of usage accounting. That means that elasticity when load declines is not only a function of the speed to decommission a resource, but also it depends on whether charging for that resource is stopped immediately on decommissioning, or instead is delayed for a while (say till the end of a charging quantum of time).

2.2 Cloud Performance and Benchmarks

Recent research efforts have conducted in-depth performance analysis on the virtual machine instances offered by public cloud providers. For example, Stantchev et al. [14] introduce a generic benchmark to evaluate the nonfunctional properties (e.g., response time) of individual cloud offerings for web services from cost-benefit perspective. Dejun et al. [5] and Schad et al. [11] examine the performance stability and homogeneity aspects of VM instances over time. These studies are useful to understand the underlying performance characteristics of the cloud infrastructure, however they do not consider the responsiveness of the platform during scaling with the variation in workload demand. A group at HP Labs [2] has defined provider-done measurements for a range of quality features of cloud platforms, focusing on environmental factors such as energy use.

Cloud service providers adopt dynamic VM migration strategies to balance application workloads among different physical machines. Several groups [13, 6] have presented benchmarking solutions to quantify the comparison of live VM migration techniques for data center scenarios. We evaluate cloud platform's elasticity from the consumers' viewpoint, whereas their work takes the service providers' perspective. They define a set of performance measures for assessing the overheads associated with dynamic VM migration techniques. In contrast, we consider the impact of imperfect elasticity based on consumers' business situation.

Several performance benchmarks have been proposed to quantify many important cloud performance metrics, among them the resource spin-up (spin-down) delay. Yigitbasi et al. [16] present a framework to determine the performance overheads associated with the scaling latency of the virtual machine (VM) instances in the cloud. Li et al. [9] developed CloudCmp to analyse customer perceived performance and cost effectiveness (e.g., scaling latency, cost per operation) of public cloud offerings. However, they do not combine their discrete performance metrics into a macroscopic overview of the platform's adaptability behavior. We propose a single summary measure for elasticity, which is influenced by several factors that were used in these earlier studies.

Yahoo! Cloud Serving Benchmark (YCSB) [4] evaluates performance of cloud databases (e.g., Cassandra, HBase) under load for a variety of workload scenarios as well as scale-up and elastic speed-up measures (that is, they consider workloads that grow and grow). Their work is valuable when seeking to analyse the performance implications of large database-intensive applications in the cloud; however, we also consider de-provisioning and resource granularity aspects. Furthermore, our elasticity model captures the financial implications as well as traditional performance.

Donald Kossmann's group at ETHZ has a research project on benchmarking cloud platforms. An initial workshop discussion [3] proposed that it would be useful to take the ratio of the throughput achieved by operations with acceptable response time, to the rate of requests, in workloads with successive peaks and troughs. The later conference paper [7] presents an extensive evaluation of the end-to-end scalability aspect of existing cloud database architectures for OLTP workloads. Here they define a set of performance and cost metrics to compare the throughput, performance/cost ratio and cost predictability of existing cloud database systems for larger and larger loads. They look at a much wider variety of measures than we do, but they omit to look at the speed of responding to change in workloads, nor do they consider workloads that can shrink as well as grow.

2.3 Elasticity Measurement Model

Weinman has proposed a numeric measurement of elasticity [15] using a conceptual model: consider a resource (e.g., computational capacity), then there is a demand curve $D(t)$ that indicates, as a function of time, how much resource is needed for the application to work properly. A function $R(t)$ shows how much resource is allocated to the application at each time. Perfect elasticity would be shown if $R(t) = D(t)$ for all t. Weinman identifies the situation where $R(t) > D(t)$ as "excess resource" (we say "overprovisioning"), and assigns it a cost which (in the simplest case) is linear in the quantity of resource that is allocated above that needed. Similarly, he considers "unserved demand" (which we call "underprovisioning") when $D(t) < R(t)$, and measures this by opportunity cost, linear in the difference. The constant of proportionality is much higher for unserved demand than for excess resource. Figure 1 illustrates this for a resource of CPU capacity; the curves show a hypothetical situation with a sine wave variation in demand (solid blue line), and linearly increasing supply (dotted black line). A value of 150% for demand indicates that the application could use one-and-a-half times the capability of the standard instance, and 150% as the supply indicates that the application has been allocated instruction execution from cycles that were one-and-a-half as frequent as those on a standard instance. Weinman's measure is a weighted combination of the areas between the curves (with higher weight for the areas of under-provisioning).

We propose improvements over Weinman's model in several respects. First, we design our workload suite to resemble complex real-world scenarios, while his workload types (e.g., constant one) are limited to theoretical analysis only. Sec-

[2]See Crossroads, Vol. 16, No. 3. (2010), pp. 3-4.

[3]See `http://horicky.blogspot.com/2009/07/between-elasticity-and-scalability.html`

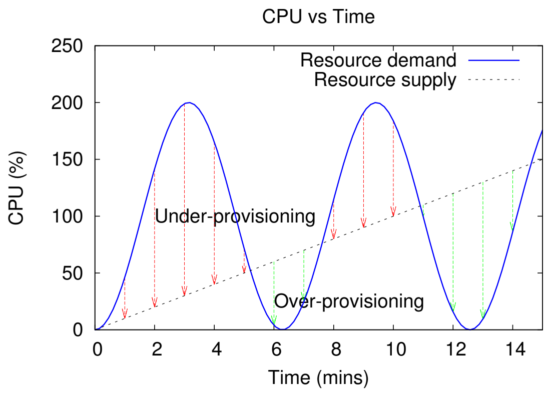

Figure 1: Elasticity Explained

ond, Weinman's area-based under-provisioning model can only accommodate unmet demand and is not able to include penalties resulting from unsatisfactory performance (e.g., high latency). We propose shifting our focus to a QoS-based under-provisioning model. Our under-provisioning model allows industry-typical SLAs, where, for example, the opportunity cost from high latency is not linear in the delay, but rather depends on whether the latency has breached a threshold, and furthermore we allow a small number of requests to see excessive latency. Also, we distinguish between the resources that are allocated, and those that are charged to the consumer (as we will see, the difference between these can be significant). In addition, we define a unified metric to summarize the financial penalties of a set of workload demands for a particular platform. Finally, we consider pragmatic issues needed to produce a figure-of-merit for a platform, by choosing explicit workloads and carrying out measurements, while Weinman's paper is entirely theoretical. Unlike Weinman, we explore the impact of the scaling rules used in the platform, to provision or deprovision resources.

3. ELASTICITY MEASUREMENT

This section defines our proposal, to determine a figure that expresses "how elastic is a given cloud platform". We explain a general framework to measure the cost of imperfect elasticity when running a given workload, with penalties for overprovisioning and underprovisioning; the sum of these is the penalty measurement for the workload. By considering a suite of workloads, and combining penalties calculated for each, we can define a figure-of-merit for a cloud platform. Next we discuss choices that we have made for an explicit measurement - that is, taking concrete decisions on the Service-Level Agreement (SLA) aspects that are evaluated, charging rates, and the particular suite of workloads.

3.1 Penalty model

We present our approach to measuring imperfections in elasticity for a given workload in monetary units. We assume that the system involves a variety of resource types. For example, the capacity of an EC2 instance can be measured by looking at its CPU, memory, network bandwidth, etc. We assume that each resource type can be allocated in units. We assume that the user can learn what level of resourcing is allocated and the relevant QoS metrics for their requests (such as distribution of latency). Amazon CloudWatch[4] is an example of the monitoring functionality we expect.

Our elasticity model combines penalties for over-provisioning and for under-provisioning. The former captures the cost of *provisioned* but *unutilised* resources, while the latter measures opportunity cost from the performance degradation that arises with under-provisioning.

3.1.1 Penalty for Over-Provisioning

In existing cloud platforms, it is usual that resources are temporarily allocated to a consumer from a start time (when the consumer requests the resource based on observed or predicted needs, or when the system proactively allocates the resource) until a finish time. This is represented by a function we call available supply and denote by $R_i(t)$ for each resource i. In current platforms it can also happen that a resource may be charged to a consumer even without being available. For example, in Amazon EC2, an instance is charged from the time that provisioning is requested (even though there is a delay of several minutes before the instance is actually running for the consumer to utilise). Similarly, charging for an instance is done in one-hour blocks, so even after an instance is deprovisioned, the consumer may continue to be charged for it for a while. Thus we need another function $M_i(t)$ that represents the *chargeable supply* curve; this is what the consumer is actually paying for. These curves can be compared to the demand curve $D_i(t)$.

The basis of our penalty model is that the consumer's detriment in overprovisioning (when $R(t) > D(t)$) is essentially given by the difference between chargeable supply and demand; as well, we charge a penalty even in underprovisioned periods whenever a resource is charged for but not available (and hence not used). These penalties are computed with a constant of proportionality c_i that indicates what the consumer must pay for each resource unit. In real systems, resources of different types are often bundled, and only available in collections (e.g., an EC2 instance has CPU, bandwidth, storage etc.). We assume that some weighting is used to partition the actual monetary charge for the bundle between its contained resources.

Formally, we define the overprovision penalty $P_o(t_s, t_e)$ for a period starting at t_s and ending at t_e. We assume a set of resources indexed by i, and we use functions $D_i(t)$, $R_i(t)$, and $M_i(t)$ for the demand, available supply, and charged supply, respectively, of resource i at time t. Our definition aggregates the penalties from each resource, and for each resource we integrate over time.

Definition 1.

$$P_o(t_s, t_e) = \sum_i P_{o,i}(t_s, t_e)$$

$$P_{o,i}(t_s, t_e) = \int_{t_s}^{t_e} c_i \times d_i(t)dt$$

$$d_i(t) = \begin{cases} M_i(t) - D_i(t) & \text{if } R_i(t) > D_i(t), \\ M_i(t) - R_i(t) & \text{if } M_i(t) > R_i(t) \\ & \text{and } D_i(t) \geq R_i(t), \\ 0 & \text{otherwise.} \end{cases}$$

[4]See http://aws.amazon.com/cloudwatch/

3.1.2 *Penalty for Under-Provisioning*

Next, we turn to the penalty model for under-provisioning, when resources are insufficient and performance is poor. We measure the opportunity cost to the consumer, using SLAs that capture how service matters to them.

We assume that the consumer has used their business environment to determine a set of performance or Quality of Service (QoS) objectives, and that each is the foundation for an SLA-style quantification of unsatisfactory behavior. For example, the platform's failure to meet the objective of availability can be quantified by counting the percentage of requests that are rejected by the system. In many cases, such SLA quantification might reflect a wide variety of causes, not only those that arise from underprovisioning, but also some from e.g., network outage. We assume that the customer also knows how to convert each measurement into an expected financial impact. For example, there might be a dollar value of lost income for each percent of rejected requests. In many cases, the financial impact may be proportional to the measurement, but sometimes there are step functions or other nonlinear effects (for example, word-of-mouth may give a quadratic growth of the damage from inaccurate responses). To provide a proper baseline for the penalties, we also consider the ideal value that occurs when resources are unlimited (in practice, we measure with such a large amount of overprovision that any additional allocation would not change the SLA measurement).

Formally, we let Q be a non-empty set of QoS measures, and for each $q \in Q$, we consider a function $p_q(t)$ that reflects the amount of unsatisfactory behavior observed on the platform at time t. The consumer provides also, for each QoS aspect q, a function f_q that takes the observed measurement of unsatisfactory behavior and maps this to the financial impact on the consumer. Let $p_q^{opt}(t)$ denote the limit (as $K \leftarrow \infty$) of the amount of unsatisfactory behavior observed in a system that is statically allocated with K resources.

Thus we define the underprovision penalty $P_u(t_s, t_e)$ for a period starting at t_s and ending at t_e

Definition 2.

$$P_u(t_s, t_e) = \sum_{q \in Q} P_{u,q}(t_s, t_e)$$

$$P_{u,q}(t_s, t_e) = \int_{t_s}^{t_e} (f_q(p_q(t)) - f_q(p_q^{opt}(t)))dt$$

3.1.3 *Total Penalty Rate for an Execution*

We calculate the overall penalty score $P(t_s, t_e)$ accrued during an execution from t_s till t_e, by taking the sum of the penalties from both over- and under-provisioning; note that both are expressed in units of cents. We then calculate the total penalty rate P in cents per hour. A lower score for P indicates a more elastic response to the given workload.

Definition 3. The penalty score over a time interval $[t_s, t_e]$ is defined as follows:

$$P(t_s, t_e) = P_o(t_s, t_e) + P_u(t_s, t_e)$$

$$P = \frac{P(t_s, t_e)}{t_e - t_s}$$

3.2 Single Figure of Merit for Elasticity

The definitions above measure the elasticity of the system's response to a single demand workload. Different features of the workload may make elastic response easier or harder to achieve; for example, if the workload grows steadily and slowly, a system may adjust the allocation to match the demand, but a workload with unexpected bursts of activity may lead to more extensive under-provisioning. Thus, we consider a suite of different workloads, and determine the penalty rate for each of these.

In order to draw simple conclusion about the worthiness of one platform's elasticity over another, we wish to summarize the penalty rates for the entire workload collection into a single score, as usual in benchmarks. To combine measured penalty rates from several workloads into a single summary number, we follow the approach used by the SPEC family of benchmarks. That is, we choose a reference platform, and measure each workload on that platform as well as on the platform of interest. We take the ratio of the penalty rate on the platform we are measuring, to the rate of the same workload on the reference platform, and then we combine the ratios for the different workloads by the geometric mean. That is, if $P_{x,w}$ is the penalty rate for workload w on platform x, and we have n workloads in our suite, then we measure the elasticity of platform x relative to reference platform x_0 by

$$E = \sqrt[n]{\prod_{i=1}^{n} (P_{x,w_i}/P_{x_0,w_i})}$$

3.3 Choices for an Elasticity Benchmark

The approach to elasticity described above is flexible, and could be adapted to the needs of each consumer, through the choices available. We can set particular SLA objectives and metrics that reflect the business situation, workloads that are representative of that consumer's patterns of load variation, etc. To actually determine an elasticity score, we need to make one set of choices for all these parameters. For the purpose of this paper, we use the following. Our workload consists of requests that follow the TPC-W application design.

To calculate the overprovisioning penalty, we deal with a single resource (CPU capacity, relative to a standard small EC2 instance) and measure the financial charge as \$0.085 per hour per instance. This reflects the current charging policy of AWS. To calculate the underprovisioning penalty, we set the QoS constraints based on the existing user behavior studies in usability engineering literature [10]. In particular, each user in our workload pattern lands on the homepage first and then searches for newly released books. We expect at least 95% of these requests generated by the users will see a response within 2 seconds. So we have used the following two QoS aspects with associated penalties over an hourly evaluation period. The cost penalty for latency violation is a simplified version of the cost function mentioned in [8]. As e-commerce websites lose more revenue when latency is slow than for application down-time[5], we associate a lower cost penalty for unserved requests.

- (Latency) In each hour of measuring, there is no penalty as long as 95% of requests have response time up to 2 seconds; otherwise, a cost penalty, 12.5¢ will apply for each 1% of additional requests (beyond the allowed 5%) that exceed the 2 seconds latency.

[5]See `http://blog.alertsite.com/2011/02/online-performance-is-business-performance/`

- (Availability) Cost penalty of 10¢ per hour will apply for each 1% of requests that fail completely (they are rejected or timed out).

Note that the penalty for unmet demand is very high compared to the cost of provisioning; this is accurate for real consumers. As Weinman [15] points out, the cost of resources should be much less than the expected gain from using them (and the latter is what determines the opportunity cost of unmet demand).

The appropriate SLAs and their penalties may vary largely based on the business situation of the consumer of cloud services. In this paper we use a penalty corresponding to a rather small business (the penalty is only \$10 in case the service is completely unavailable for an hour, when all requests are rejected). For a large e-commerce business application (e.g., e-bay), the appropriate penalty for down-time might be much higher[6], say \$2000/second, and similarly the appropriate workloads would be much greater. The SLA penalty specified here should be considered as an illustration, to be adjusted based on the application's business context.

We have developed a workload suite to explore the platform's elasticity behavior for a range of patterns of demand change. We consider various workload characteristics (e.g., periodicity, growth and decay rate, randomness) to understand how the platform's elastic response varies across the workload space. In our measurements, we use a set of 10 different workloads, which grow and shrink in a variety of shapes, though (to make benchmarking manageable) all are fairly small, peaking with less than 10 instances, and lasting across 3 hours. Across time, some workloads show recurring cycles of growth and decrease, such as an hourly news cycle. Others have a single burst, such as when news breaks or during a marketing campaign[7]. We explore some trends as the length of cycles changes, etc., however work is still needed to consider the behavior with longer cycles such as daily ones, or longer-lasting one-off events. Further research is also needed on whether conclusions from small loads will be valid for much larger ones, as expected by large customers.

- Sinusoidal Workloads: These loads can be expressed as $D(t) = A(sin(2\pi t/T + \phi) + 1) + B$, where A is the amplitude, B is the base level, T is the period and ϕ is the phase shift. For the benchmark suite, we use three different examples, whose periods are 30 minutes, 60 minutes and 90 minutes, respectively. All have peak demand of 450 req/sec, and trough at 50 req/sec. A load of 150 req/sec is about what one VM instance can support.

- Sinusoidal Workload with Plateau: This workload type modifies the sinusoidal waveform, by introducing a level (unchanging) demand for a certain time, at each peak and trough. Thus the graph has the upswings and downswings, with flat plateau sections spacing them out. In the suite we have three workloads like this, each starting from the sinewave with period of 30 minutes; in one case the plateau at each peak lasts 10 minutes, in another it lasts 40 minutes, and in the last of this type, the peak plateaus last 70 minutes each. In all cases, the plateaus at troughs last 45 minutes (and there is always a 10 minutes plateau at the start of the experiment and also at the end).

- Exponentially Bursting Workload: This workload type exhibits extremely rapid buildup in demand (rising U-fold each hour), followed by a decay (declining D-fold each hour). We provide two workloads of this type, one with $U = 18$ and $D = 2.25$; the other has $U = 24$ and $D = 3$ (so this rises and falls more quickly).

- Linearly Growing Workload: This workload represents a website whose popularity rises consistently. It can be stated as $D(t) = mt + c$, where m is the slope of the straight line and c is the y-axis intercept. We have one example of this type, with workload that starts at 50 requests/second (and stays here for 10 minutes to warm the system up), then the load rises steadily for 3 hours, each hour increasing the rate by an extra 240 requests/second. Thus we end up with 770 requests/second.

- Random Workload: The generation of requests is ongoing and independent. We have one example of this type, with requests produced by a Poisson process.

We note that the demand curves described above are expressed in terms of the rate requests are generated; in practice, performance variation in identical instances means that this does not cause the utilisation of CPU resources to track the desired demand pattern exactly.

4. IMPLEMENTATION

In this section, we illustrate the implementation details of our elasticity measurement environment. We first describe the architectural components of our experimental testbed and then outline the key steps to configure the testbed to fit the consumer-specific scenarios.

4.1 Experimental Setup

In the high level view, the architecture of our experimental setup can be seen as a client-server model. The client side is a workload generator implemented using JMeter[8], which is a Java workload generator used for load testing and measuring performance. The sole purpose of JMeter in this experiment is to generate workload based on our predefined workload patterns.

We chose TPC-W [12] as the application in all our suite of workloads, because it has easy-to-obtain code examples and it is most often used in the literature. It can be substituted with other applications if desired. TPC-W emulates user interactions of a complex e-commerce application (such as an online retail store). In our experiment, we adopt the online bookshop implementation of TPC-W application and deploy it into EC2 small instances. Instead of having the TPC-W workload generator at the client side, we use JMeter to specify our pre-defined workload patterns.

The server side is considered to be the System Under Test (SUT), which consists of a single load-balancer facing the client side, and a number of EC2 instances behind the load-balancer. We hosted the web server, application server

[6]See `http://www.raritan.com/resources/case-studies/ebay.pdf`

[7]See `ecn.channel9.msdn.com/o9/pdc09/ppt/SVC54.pptx`, `http://www.mediabistro.com/alltwitter/osama-bin-laden-twitter-record_b8019`

[8]See `http://jakarta.apache.org/jmeter/`

and database on the same EC2 m1.small instance at US-East Virginia region (the cost of each instance is 8.5¢ per hour, matching the penalty we apply for overprovisioning); as some of the database queries consumed more CPU, we had to restrict the processing rate for TPC-W server to 150 requests/second to achieve satisfactory performance. The number of instances is not fixed, but rather it is controlled by an autoscaling engine which dynamically increases and decreases the number of instances based on the amount of workload. The behavior of an autoscaling engine follows a set of rules that must be defined. Each ruleset produces a different "platform" for experimental evaluation, with different elasticity behavior.

An autoscaling rule has the form of pair consisting of an antecedent and a consequence. The antecedent is the condition to trigger the rule (e.g., CPU utilisation is greater than 80%) and a consequence is the action to trigger when the antecedent is satisfied (e.g., create one extra instance). In our experiments we consider three platforms, because we run with three different rulesets. The detailed configurations of the autoscaling engine (configured via AutoScaling[9] library) is shown in Table 1.

We have here explored rulesets that scale-out and -in by changing the number of instances, all of the same power. Some cloud platforms, including EC2, also allow one to provision instances of different capacity, vary bandwidth, etc.; how such rules alter the elasticity measures is an issue for further research, although our definitions will still apply.

We measure available supply $R(t)$ by using the reports from CloudWatch showing the number of instances that are allocated to our experiment; we treat k instances as $R(t) = 100 \times k\%$ of supply, so this function moves in discrete jumps. Chargeable supply $M(t)$ is determined from the launch time and termination time of the allocated EC2 instances, given by AWS EC2 API tools. For demand, our generator is defined to produce a given number of requests, rather than in the measure of CPU capacity, that is needed for our measurements. Thus we use an approximation: we graph $D(t)$ from what CloudWatch reports as the sum of the utilisation rates for all the allocated instances. As will be seen in the graphs in Section 5, this is quite distorted from the intended shape of the demand function. One distortion is that measured $D(t)$ is capped at the available supply, so under-provisioning does not show up as $D(t) > R(t)$. This inaccuracy is not serious for our measurement of elasticity, since the use of $D(t)$ in measurement is only for cases of over-provisioning; during under-provisioning, the penalty is based on QoS measures of latency and lost requests, and these do reveal the growth of true demand. Another inaccuracy is from the system architecture, where requests that arrive in a peak period may be delayed long enough that they lead to work being done at a later period (and thus measured $D(t)$ may be shifted rightwards from the true peak). As well, there is considerable variation in the performance of the supplied instances [5], so a given rate of request generation with 450 req/s can vary from 350% to 450% when we see the measured demand. Future work will find ways to more accurately measure demand in units of CPU capacity.

4.2 Configuration and Measurement Procedure

Here is the procedure for setting up the elasticity measurement environment. First, a VM image is prepared by installing necessary components for the target application (e.g., TPC-W). Then the load-balancer is launched and autoscaling configuration for the dynamically scalable server farm is set up. A monitoring agent (e.g., CloudWatch) is also configured to measure utilisation and performance data for each workload run.

Next, the scripts for all workload demands are distributed to the client-side load generator (e.g., JMeter). Each workload demand is applied to a fresh set-up of the server farm. At the end of each workload run, utilisation and performance data are collected from the monitoring agent and load generator respectively. The penalty rate for each workload demand is computed with the help of the penalty model, described in Section 3.1. Same procedure is repeated to measure the penalty rate for other workload demands. Finally a single elasticity score is derived by taking the geometric mean of the penalty rates of all workload demands in the collection.

5. CASE STUDIES

We describe in some detail the observations made when we run our workloads against Amazon EC2. These case studies serve two purposes. (1) As a means of sanity checking the elasticity model in Section 3. That is, we can see that the numerical scores, based on our elasticity model, do in fact align with what is observed in over- or underprovisioning. For example, in reducing the steepness of a workload increase we shall observe that supply tracks more closely to demand, and the penalty calculated is lower. (2) We demonstrate the usefulness of our elasticity benchmark in exposing situations where elasticity fails to occur as expected, and other interesting phenomena can be observed.

5.1 Explore Workload Patterns

To begin, we applied each of the 10 workload patterns from our elasticity benchmark, in EC2 with a fixed scaling ruleset 1 as defined in Table 1. This ruleset is common in tutorial examples, and it seems widespread in practice[10]. With this ruleset, the number of instances increases by one when average CPU utilisation exceeds 70%, and one instance is deprovisioned when CPU utilisation drops below 30%.

We illustrate first how to derive the penalty rate from the raw measurement data. For each workload demand, we compute over-provisioning amount by taking the difference between the charged resource supply and used-up resource demand for the entire workload duration (e.g., 110 minutes interval for sine workload with 30 minutes period). For these case studies, we consider only CPU resource and assume that its pricing is equal to that of an EC2 instance (i.e., 8.5¢/hour). Thus we calculate the unit price for CPU resource, assuming that each hour consists of 60 time units (i.e., 60 minutes) and supplied CPU at each time unit is $k \times 100\%$, where k is the number of charged VM instances. We work out the over-provisioning penalty by multiplying the unit CPU price with the over-provisioned quantity. For the under-provisioning penalty, we measure the percentage of latency violations and dropped requests and evaluate the opportunity cost of the degraded performance based on the SLA definition, described in Section 3.3. Finally, we aggregate the penalty values for over- and under-provisioning and

[9]See `http://aws.amazon.com/autoscaling/`

[10]See `http://mtehrani30.blogspot.com/2011/05/amazon-auto-scaling.html`

Table 1: AutoScaling Engine Configuration

Ruleset	Monitoring Interval	Upper Breach Duration	Lower Breach Duration	Upper Threshold	Lower Threshold	VM Increment	VM Decrement	Scale-out Cool down Period	Scale-in Cool down Period
1	1 min	2 mins	2 mins	70% CPU Average	30% CPU Average	1	1	2 mins	2 mins
2	1 min	2 mins	15 mins	70% CPU Average	20% CPU Average	2	2	2 mins	10 mins
3	1 min	4 mins	10 mins	1.5 sec Maximum Latency	20% CPU Average	1	1	2 mins	5 mins

normalise it to compute the penalty rate per hour, giving the penalty rate for a particular workload demand.

5.1.1 Effect of Over- and Under-provisioning

Figure 2 shows behavior of the platform in response to an input sinusoidal workload with a period of 30 minutes. The CPU graph shows the available supply, chargeable supply and demand curves over a 110-minute interval. Initially, there was only one instance available to serve the incoming requests. As workload demand increases (after 15 minutes), the rule triggers provisioning a new instance, but we observed a delay of about 6 minutes until that is available (however it is charged as soon as the launch begins). As workload generation is rising fast during this delay, the system experiences severe effects of under-provisioning: latency spikes and penalties accrue at about 20¢/min. In our implementation, demand is measured on the instances and so the curve shown is capped at the available supply, rather than showing the full upswing of the sinewave. The lag between charging for the instance and it being available, is reflected in a penalty for over-provisioning of about 0.14¢/min during this period.

Between timepoints 50 and 55, we see high penalties for both over-provisioning and under-provisioning. This may sound counter-intuitive as one might wonder how "excess resource" and "insufficient resource" co-exist at the same time. Looking at the CPU graph, we find significant difference between charged and available supply during that interval; the consumer continues paying for 3 extra unusable instances (2 de-provisioned instances from the previous cycle and 1 yet-to-be-provisioned instance in the current cycle) which contribute to over-provisioning penalty. Also, the available instance supply (i.e., 2 instances) is not enough to meet the increasing demand for that duration, thus resulting in high under-provisioning penalty.

5.1.2 Deprovisioning of Resources

Our work's inclusion of cases where workload declines is different from most previous proposals for cloud benchmarks. These situations show interesting phenomena. We saw situations where a lag in releasing resources was actually helpful for the consumer. In Figure 2 we see, on the downswing of the demand curve, that most of the resources claimed on the upswing were kept; this means that the next upswing could utilise these instances, and so the latency problems (and underprovisioning penalty) were much less severe than in the first cycle.

We also observe in downswings that the difference between chargeable supply and available supply is significant, with a deprovisioned instance continuing to attract charge till the end of the hour-long quantum. We see in the CPU graph of Figure 2 that the chargeable supply is simply not following the demand curve at all, and indeed there are extensive periods when the consumer is paying for 4 instances, even though they never have more than 2 available for use.

Considering the evolution of the supply led us to discover an unexpected inelasticity phenomenon, where the cloud-hosted application is never able to cut back to its initial state after a temporary workload burst. The average utilisation may not drop below 30% which triggers deprovisioning, even though several instances are not needed. To demonstrate this fact, we ran a sinusoidal workload pattern with peak at 670 req/s and trough at 270 req/s, and 40 minutes plateau at each peak and trough; the resultant graphs are shown in Figure 3. The peak workload triggered the creation of 6 instances. The long-lasting trough workload (about 136% CPU) could easily be served by 2 instances, however, the number of VM instances remained at 4.

5.1.3 Trends In Elasticity Scores

Looking at the penalty scores in Table 2, we can see how the calculated penalty varies with the type of workload. In all workloads (except the linear one), the overall penalty is dominated by the loss in revenue due to under-provisioning. This is appropriate to business customers as the opportunity cost, from unmet requests or unsatisfactory response that may annoy users, is much more than the cost of resources.

For pure sinusoidal workload patterns, the overall penalty declines with the increase in waveperiod. This demonstrates that, with these rulesets, the EC2 platform is better at adapting to changes that are less steep. Here underprovisioning penalties will be less severe as the demand will not have increased too much in the delay from triggering a new instance, until it is available to serve the load.

The sinusoidal workload with plateaus has higher overall penalty compared to the basic sinusoidal workload where the cycles are sharper. We attribute this to the insertion of a 45 minutes plateau at the trough which wipes out the resource-reuse phenomena in subsequent cycles. With a trough plateau, the system has time to deprovision and return to its initial state before the next cycle; therefore, each cycle could not take advantage of the resources created in the previous cycle and it pays a similar underprovisioning penalty. As the length of the plateau at the peak increases (from 10 minutes to 70 minutes), overall penalty gradually

Table 2: Penalty for Benchmarking Workloads - Ruleset 1

Workload	$P_o(t_s, t_e)/hr$	$P_u(t_s, t_e)/hr$	$P(t_s, t_e)/hr$
sine_30	27.51¢	374.88¢	402.39¢
sine_60	28.84¢	133.65¢	162.49¢
sine_90	22.17¢	52.82¢	74.99¢
sine_plateau_10	22.08¢	554.44¢	576.52¢
sine_plateau_40	18.52¢	292.96¢	311.48¢
sine_plateau_70	23.81¢	174.19¢	198.0¢
exp_18_2.25	24.83¢	528.61¢	553.44¢
exp_24_3.0	17.65¢	1093.05¢	1110.7¢
linear_240	35.01¢	0.0¢	35.01¢
random	29.31¢	129.14¢	158.45¢

moves down. The system has time to adapt to the peak demand, and serve it effectively for longer.

For the exponential burst workloads, we observe large penalty values in Table 2. In general, under-provisioning penalty tends to rise as the growth rate increases; that means, the underlying cloud platform is not elastic enough to grow rapidly with these fast-paced workloads, thus resulting in sluggish performance as they head towards the peak. Figure 4 explains the performance implications of an exponential workload with growth 24-fold per hour and decay 3-fold per hour. The high under-provisioning cost in the penalty graph also confirms EC2 platform's inelasticity in coping up with this fast-paced workload pattern. The under-provisioning penalties incurred were much higher than in the sinusoidal workloads, indicating that EC2 platform is not so adaptive to traffic surges with high acceleration rate. Again, looking at the over-provisioning penalty graph, we observe large over-provisioning cost (around 0.43¢/min) right after the peak is over; as some of the VM instances launched during the peak load were available at the off-peak period, they just accrued more penalty due to over-provisioning with no significant reduction to under-provisioning penalty.

Unlike the above workloads, linear workload yields less overall penalty. This suggests that EC2 platform can easily cope up with workloads with lower and consistent growth. This is not surprising as slowly growing workload pattern is not affected by the provisioning delay of the underlying platform and therefore incur less under-provisioning penalty. However, we expect that as the slope becomes steeper, the overall penalty will show a rising trend, as the resources are not provisioned rapidly enough.

5.2 Explore Impact of Scaling Rules

In our experiments with the widely-used ruleset 1, under-provisioning penalty dominates the overall score. Sometimes the system took too long in adjusting to rapid growth in demand. When demand drops, there is a tradeoff: slow response increases the duration of overprovisioning charges, but it can help if an upswing follows that might reuse the retained resources. To improve the elasticity, one can try changing the scaling rules so that they are aggressive in provisioning extra resources and conservative enough in deprovisioning those resources. The initial (Ruleset 1) and adjusted (Rulesets 2 and 3) scaling rulesets are shown in Table 1. Ruleset 3 is distinctive by making scale-out decisions based on the monitored values of the application level performance metric (response time) instead of considering a resource utilisation metric. This performance-based approach has been adopted by some practitioners for autoscaling cloud applications[11]. Ruleset 3 explores the tradeoffs in these different approaches to scaling.

Ruleset 2 performed markedly better than ruleset 1. Two major factors contribute to this improvement in ruleset 2: adding multiple instances at each trigger in the upswing and lazy deprovisioning in the downswing. The ruleset 1 is less responsive to the rapidly increasing sinusoidal and exponential workloads because it only adds one instance at each rule trigger and there is a cooling period which stops it from immediately creating another instance (even if the condition is again met). On the other hand, ruleset 2 increases two instances at each trigger thus it responds quicker to sharp workload increase.

On the downswing, ruleset 1 responds much quicker to the drop of demand by deprovisioning its resources. Though the trend of available supply follows closely with the demand, the chargeable supply does not follow as well: resources are being charged but not used. In contrast, as we intended, ruleset 2 (with an increased lower breach duration of 15 minutes and scale-in cool-down period of 10 minutes) keeps the resources from the previous upswing so that they are reused in the subsequent cycles of the workload demand.

This benefit from resource reuse holds as long as workloads come in periodic bursts and the inter-arrival time of bursts are short enough to retain some resources from the previous ones; otherwise, subsequent bursts will not be able to enjoy the resource reuse phenomenon as the resources are likely to be released by that time. For this reason, sine-plateau workloads could not make use of the resources from the previous cycle because of their long plateau at the trough (45 minutes). Same holds for exponential workload with growth rate 18 and decay rate 2.25 per hour; it could not improve that much with ruleset 2 as the duration between the bursts is long enough to set the number of instances back to the initial state (1 EC2 instance).

Results for all workloads of our suite are shown in Table 3, which should be compared to Table 2. One clear disadvantage of ruleset 2 is that it is likely to overprovision too much in the case where workload does not increase quickly. This is reflected in the experiment with the linear workload pattern. The modest pace of growth in demand here means that ruleset 1 was sufficient to align the resource supply with its resource demand. Ruleset 2 rather worsened the overall penalty by increasing the over-provisioning cost.

We observe a lower penalty score for ruleset 3 than for ruleset 1. Since the under-provisioning penalty is mostly dominated by high latency, we designed this ruleset to add an extra instance when the maximum latency goes beyond 75% of the allowed threshold. This ruleset triggers an instance provisioning request as soon as the observed latency starts rising due to request-buffering at the server. The results show that this ruleset ensures higher SLA conformance and hence lower under-provisioning penalty for most of the workloads. An increased lower-breach duration and scale-in cool-down period in the deprovisioning rule also promote resource reuse from previous cycles and thus contribute to the improvement in the penalty score.

The only short-coming of ruleset 3 is that it sometimes

[11]See `http://blog.tonns.org/2011/01/autoscaling-revisited.html`

Table 3: Penalty for Benchmarking Workloads - Ruleset 2

Workload	$P_o(t_s, t_e)/hr$	$P_u(t_s, t_e)/hr$	$P(t_s, t_e)/hr$	Ratio to Rule 1
sine_30	40.33¢	127.50¢	167.83¢	0.41
sine_60	38.49¢	1.98¢	40.47¢	0.24
sine_90	38.03¢	1.24¢	39.27¢	0.52
sine_plateau_10	33.94¢	335.56¢	369.5¢	0.64
sine_plateau_40	32.27¢	138.86¢	171.13¢	0.54
sine_plateau_70	33.24¢	44.52¢	77.76¢	0.39
exp_18_2.25	33.27¢	428.09¢	461.36¢	0.83
exp_24_3.0	60.62¢	416.47¢	477.09¢	0.42
linear_240	39.83¢	0.0¢	39.83¢	1.13
random	61.35¢	35.13¢	96.48¢	0.60
Geometric Mean	N/A	N/A	N/A	0.52

Table 4: Penalty for Benchmarking Workloads - Ruleset 3

Workload	$P_o(t_s, t_e)/hr$	$P_u(t_s, t_e)/hr$	$P(t_s, t_e)/hr$	Ratio to Rule 1
sine_30	25.22¢	181.29¢	206.51¢	0.51
sine_60	33.78¢	106.28¢	140.06¢	0.86
sine_90	60.23¢	0.0¢	60.23¢	0.80
sine_plateau_10	22.68¢	408.92¢	431.6¢	0.74
sine_plateau_40	20.93¢	223.88¢	244.81¢	0.78
sine_plateau_70	21.97¢	173.92¢	195.89¢	0.98
exp_18_2.25	27.0¢	538.42¢	565.42¢	1.02
exp_24_3.0	37.76¢	577.52¢	615.28¢	0.55
linear_240	15.68¢	11.19¢	26.87¢	0.76
random	36.63¢	108.75¢	145.38¢	0.91
Geometric Mean	N/A	N/A	N/A	0.77

results in excessive over-provisioning because of "rippling effect". This ruleset assumes resource bottleneck as the only cause for latency violation and therefore provisions extra instances to improve the observed latency. However, there might be several other reasons for high latency even after an instance is available, for example, the warm-up period of the newly provisioned instance, request-queuing in other instances or problems in third party web service calls[12]. If the rule sets too small a "cool-down" period (how long after a rule is triggered till it can be triggered again), then provisioning requests for new instances might be triggered repeatedly based on latency violation information that has not yet reflected the earlier provisionings. Figure 5 demonstrates this rippling phenomenon for one workload.

We computed a single figure of merit based on the SPEC family of benchmarks as defined in Section 3.2. We used the platform with ruleset 1 as the reference when evaluating ruleset 2 and 3. The last column in Table 3 and 4 shows the ratio of the total penalties for the two rulesets with respect to ruleset 1 for each of the 10 workload patterns. All but one of these ratios are smaller than one, indicating that both ruleset 2 and 3 are generally more elastic than ruleset 1 for the benchmark workload patterns. We calculated the geometric mean of these ratios (0.52 for ruleset 2 and 0.77 for ruleset 3), which quantifies the improvement in elasticity. Thus we demonstrated that our single figure for elasticity can be effectively used to compare different rule configurations. It can also used to detect variation in elasticity level between platforms from different cloud providers, as well as variation over time within a cloud provider due to the consistently evolving underlying infrastructures of cloud.

[12]See `http://aws-musings.com/choosing-the-right-metrics-for-autoscaling-your-ec2-cluster/`

6. DISCUSSION

Our case studies have given us insights into how a cloud platform can be better or worse at elasticity when following a varying workload. Identifying the importance of these characteristics should be of independent value to consumers who want to choose a platform, and they may also help a cloud provider to offer better elasticity in his platform.

It is well understood that the granularity of instances is important in elasticity. If a substantial PC-like (virtual) machine is the smallest unit of increased resource, this is less elastic than a platform which can allow each customer to have whatever percentage of the cycles that they need, as in the case of GAE[13]. Similarly, the time delay between a request for provisioning, until one can actually run on the new instance, can be significant. We observed that this delay varies unpredictably, but can be over 10 minutes. If the workload is increasing fast enough, by the time 10 minutes have elapsed, the previous configuration may have become badly overloaded. Ongoing changes in implementation by platform providers may lessen the provisioning delay, and thus improve observed elasticity. Finally, the delay to decommission an instance is also important. Being too slow to

[13]See `http://code.google.com/appengine`

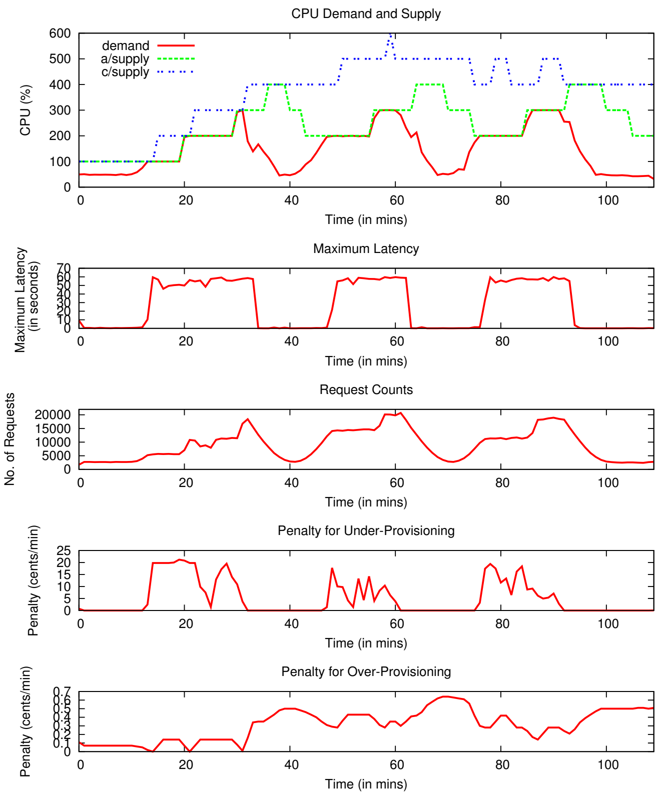

Figure 2: Results of Sinusoidal Workload with 30 Minutes Period

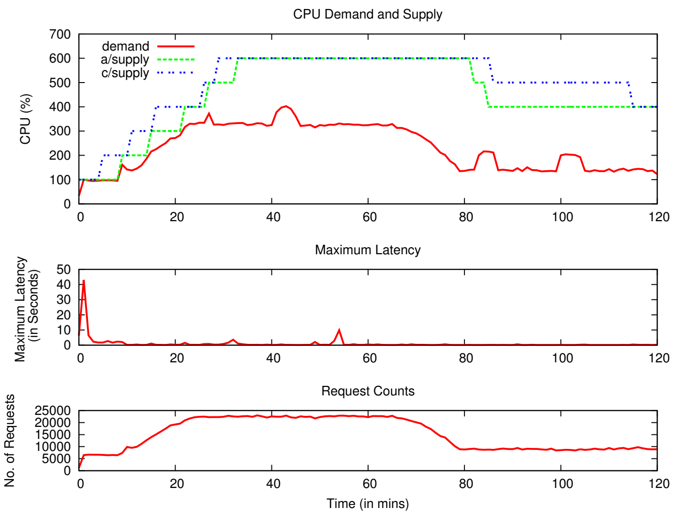

Figure 3: Results for the Trapping Scenario

give up resources is wasteful, but being too eager can leave the consumer without resources if/when workload recovers to previous levels.

We have seen how important it is to understand the way the consumer is charged for resources, We have seen that this can be quite different from the actual access to those resources, and the difference is important for the consumer's perception of elastic behavior. When charging runs till the

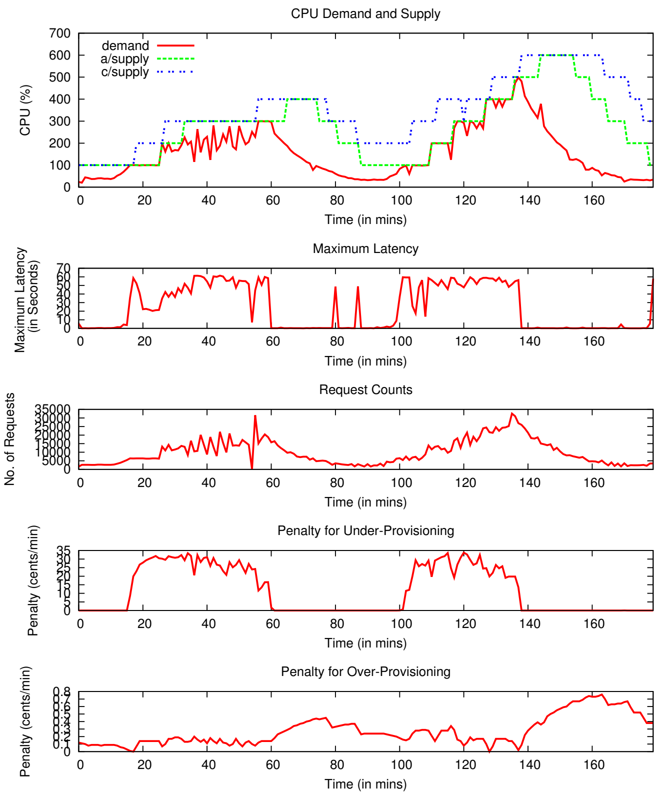

Figure 4: Results for Exponential Workload with Growth 24/hour and Decay 3/hour

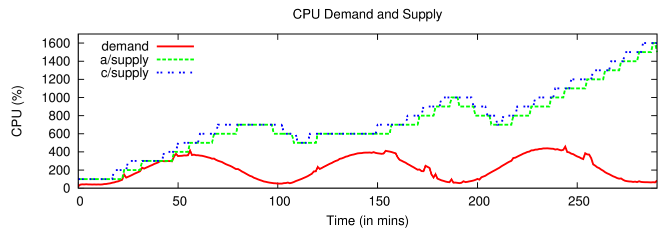

Figure 5: Rippling Effect for Sinusoidal Workload with 90 Minutes Period

end of a substantial quantum (e.g., an hour, for EC2), we can see financial losses from too rapid response to changed load. In particular, if the fluctuating load leads a consumer to give up an instance, and then they need to request it back, they may end up paying for it twice over.

Our experiments have shown how changes to the provisioning and deprovisioning rules can alter the elasticity of the platform. This seems to deserve much more attention from consumers, and we haven't found useful guidelines in research or tutorial literature. In particular, many applications seem to follow sample code, and use a default policy where instances are created or given up based on utilisation levels holding for a fairly short time (e.g., 2 minutes). By being less eager to deprovision, we saw a different ruleset gave significant improvement (about 50% for ruleset 2) in the elasticity measure. We also observed better SLA conformance for rules based on QoS (i.e., latency-based ruleset 3) as compared to the utilisation-based one (CPU-based ruleset 1); however, ruleset 3 is less reliable for autoscaling as it causes excessive over-provisioning when a QoS threshold is breached because of external factors (e.g., latency spike in other tiers or web services) instead of resource scarcity.

Running our benchmark has been informative for us. We now reflect directly on the advantages and disadvantages of the decisions we made in proposing this benchmark, that is, how exactly we decided to measure elasticity.

Having workloads with diverse patterns of growth and decline in demand is clearly essential. Those that rise rapidly (that is, fast compared to the provisioning delay in the platform) reveal many cases of poor elasticity. When demand declines and rises again, we see effects of charging quanta.

We followed the SPEC approach to combine information from several workloads into one number. It gives consistent relative scores no matter which platform is the reference. It is very robust in that it does not change depending on subtle choice of weights, nor on the scale chosen for each workload.

Our calculated penalty for overprovisioning is based on the charged level of resources, rather than on the resources that are actually allocated (as in Weinman's discussion of elasticity [15]). We have seen that there can be a considerable difference between these quantities. By this decision, we properly give a worse score for a system if it keeps charging over a longer quantum. Our penalty calculation for underprovision is based on observed QoS, and using consumer-supplied functions to convert each observation into the opportunity cost. We do not assume a constant impact of each unmet request. This clearly fits with widespread practice, where SLAs with penalty clauses are enshrined in contracts.

Overall, our approach fits the decision-making of a consumer selecting a suitable cloud platform for their needs.

7. CONCLUSION

Small and medium enterprises are heading towards the cloud for many reasons, including varying workload. To choose appropriately between platforms, a consumer of cloud services needs a way to measure the features that are important, one of which is the amount of elasticity of each platform. This paper offers a concrete proposal giving a numeric score for elasticity. We suggested specific new ways to use SLAs to determine penalties for underprovisioning. We have defined a suite of workloads that show a range of patterns over time. We carried out a case study showing that our approach is feasible, and that it leads to helpful insights into the elasticity properties of the platform. In particular, we have shown that one gets poor elasticity when following a widespread ruleset for provisioning and deprovisioning.

In future, we will extend our measurements to other platforms with a wider range of features. We hope to consider workloads that grow much further than our current set (which peak at demand for about half-a-dozen instances). We also will try examples with a greater range of SLAs and opportunity cost functions. We would like to make the benchmark running as automatic as possible. We see this paper as an important step towards allowing consumers to make informed choices between cloud platforms.

Acknowledgement

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. A grant from Amazon provided access to AWS.

8. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] C. Bash, T. Cader, Y. Chen, D. Gmach, R. Kaufman, D. Milojicic, A. Shah, and P. Sharma. HPL-2011-148: Cloud Sustainability Dashboard, Dynamically Assessing Sustainability of Data Centers and Clouds. Technical report, Hewlett-Packard Labs, 2011.

[3] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *Proc DBTest'09*, 2009.

[4] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc SoCC'10*, pages 143–154, 2010.

[5] J. Dejun, G. Pierre, and C. Chi. EC2 performance analysis for resource provisioning of service-oriented applications. In *ICSOC Workshops (Springer LNCS 6275)*, pages 197–207, 2009.

[6] D. Huang, D. Ye, Q. He, J. Chen, and K. Ye. Virt-LM: a benchmark for live migration of virtual machine. In *Proc ICPE'11*, pages 307–316, 2011.

[7] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proc SIGMOD'10*, pages 579–590, 2010.

[8] S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper. Quality of service enabled database applications. In *ICSOC*, pages 215–226, 2006.

[9] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *Proc IMC'10*, pages 1–14, 2010.

[10] F. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[11] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1):460–471, 2010.

[12] W. Smith. TPC-W: Benchmarking an ecommerce solution. White paper, Transaction Processing Performance Council, 2000.

[13] K. Srinivasan, S. Yuuw, and T. Adelmeyer. Dynamic VM migration: assessing its risks & rewards using a benchmark. In *Proc ICPE'11*, pages 317–322, 2011.

[14] V. Stantchev. Performance evaluation of cloud computing offerings. In *Proc IEEE AdvComp'09*, pages 187–192, 2009.

[15] J. Weinman. Time is Money: The Value of "On-Demand". `www.joeweinman.com/Resources/Joe_Weinman_Time_Is_Money.pdf`, Jan. 2011.

[16] N. Yigitbasi, A. Iosup, D. Epema, and S. Ostermann. C-meter: A framework for performance analysis of computing clouds. In *Proc CCGrid'09*, pages 472–477, 2009.