

Dynamic Selection of Virtual Machines for Application Servers in Cloud Environments*

Nikolay Grozev and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory,
Department of Computing and Information Systems,
The University of Melbourne, Australia

ngrozev@student.unimelb.edu.au, rbuyya@unimelb.edu.au

February 9, 2016

Abstract

Autoscaling is a hallmark of cloud computing as it allows flexible just-in-time allocation and release of computational resources in response to dynamic and often unpredictable workloads. This is especially important for web applications whose workload is time dependent and prone to flash crowds. Most of them follow the 3-tier architectural pattern, and are divided into presentation, application/domain and data layers. In this work we focus on the application layer. Reactive autoscaling policies of the type “*Instantiate a new Virtual Machine (VM) when the average server CPU utilisation reaches X%*” have been used successfully since the dawn of cloud computing. But which VM type is the most suitable for the specific application at the moment remains an open question. In this work, we propose an approach for dynamic VM type selection. It uses a combination of online machine learning techniques, works in real time and adapts to changes in the users’ workload patterns, application changes as well as middleware upgrades and reconfigurations. We have developed a prototype, which we tested with the CloudStone benchmark deployed on AWS EC2. Results show that our method quickly adapts to workload changes and reduces the total cost compared to the industry standard approach.

*To cite this technical report, please use the following: Nikolay Grozev and Rajkumar Buyya, “Dynamic Selection of Virtual Machines for Application Servers in Cloud Environments,” Technical Report CLOUDS-TR-2016-1, Cloud Computing and Distributed Systems Laboratory, The University of Melbourne, February 7, 2016.

1 Introduction

Cloud computing is a disruptive IT model allowing enterprises to focus on their core business activities. Instead of investing in their own IT infrastructures, they can now rent ready-to-use preconfigured virtual resources from cloud providers in a “pay-as-you-go” manner. Organisations relying on fixed size private infrastructures often realise it can not match their dynamic needs, thus frequently being either under or overutilised. In contrast, in a cloud environment one can automatically acquire or release resources as they are needed — a distinctive characteristic known as *autoscaling*.

This is especially important for large scale web applications, since the number of users fluctuates over time and is prone to flash crowds as a result of marketing campaigns and product releases. Most such applications follow the 3-tier architectural pattern and are divided in three standard layers/tiers [1–3]:

- **Presentation Layer** — the end user interface.
- **Business/Domain Layer** — implements the business logic. Hosted in one or several Application Servers (AS).
- **Data Layer** — manages the persistent data. Deployed in one or several Database (DB) servers.

A user interacts with the presentation layer, which redirects the requests to an AS which in turn can access the data layer. The presentation layer is executed on the client’s side (e.g. in a browser) and thus scalability is not an issue. Scaling the DB layer is a notorious challenge, since system architects have to balance between consistency, availability and partition tolerance following the results of the CAP theorem [4, 5]. This field has already been well explored (Cattel surveys more than 20 related projects [6]). Furthermore, Google has published about their new database which scales within and across data centres without violating transaction consistency [7]. Hence data layer scaling is beyond the scope of our work.

In general, autoscaling the Application Servers (AS) is comparatively straightforward. In an Infrastructure as a Service (IaaS) cloud environment, the AS VMs are deployed “behind” a load balancer which redirects the incoming requests among them. Whenever the servers’ capacity is insufficient, one or several new AS VMs are provisioned and associated with the load balancer and the DB layer — see Figure 1.

But what should be the type of the new AS VM? Most major cloud providers like Amazon EC2 and Google Compute Engine offer a predefined set of VM types with different performance capacities and prices. Currently, system engineers “hardcode” preselected VM types in the autoscaling rules based on their intuition or at best on historical performance observations. However, user workload characteristics vary over time leading to constantly evolving AS capacity requirements. For example, the proportion of browsing, bidding and buying requests in an e-commerce system can change significantly during a holiday season, which can change the server utilisation patterns. Middleware and operating

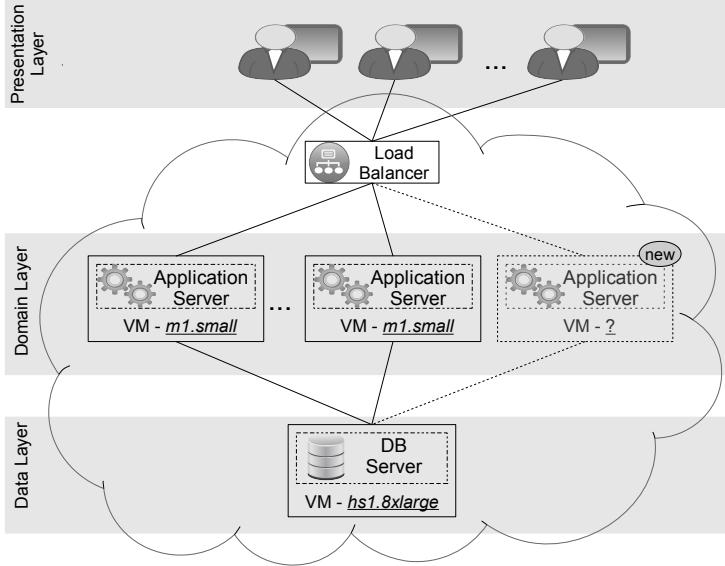


Figure 1: A 3-tier application in Cloud. Whenever the autoscaling conditions are activated, a new application server should be provisioned. In this work we select the optimal VM type for the purpose.

system updates and reconfigurations can lead to changes in the utilisation patterns as well [8]. This can also happen as a result of releasing new application features or updates.

Moreover, VM performance can vary significantly over time because of other VMs collocated on the same physical host causing resource contentions [9–11]. Hence even VM instances of the same type can perform very differently. From the viewpoint of the cloud’s client this can not be predicted.

To illustrate better, let us consider a large scale web application with hundreds of dedicated AS VMs. Its engineers can analyse historical performance data to specify the most appropriate VM type in the autoscaling rules. However, they will have to reconsider their choice every time a new feature or a system upgrade is deployed. They will also have to constantly monitor for workload pattern changes and to react by adjusting the austoscaling rules. Given that VM performance capacities also vary over time, the job of selecting the most suitable VM type becomes practically unmanageable. This can result in significant financial losses, because of using suboptimal VMs.

To address this, the key **contributions** of our work are (i) a machine learning approach which continuously learns the application’s resource requirements and (ii) a dynamic VM type selection (DVTS) algorithm, which selects a VM type for new AS VMs. Since both workload specifics and VM performance vary over time, we propose an online approach, which learns the application’s behaviour and the typical VM performance capacities in real time. It relieves

system maintainers from having to manually reconfigure the autoscaling rules.

The rest of the paper is organised as follows: In Section 2 we describe the related works. Section 3 provides a succinct overview of our approach. Section 4 discusses the machine learning approaches we employ to “learn” the application’s requirements in real time. Section 5 describes how to select an optimal VM type. Section 6 details the architecture of our prototype and the benchmark we use for evaluation. Section 7 describes our experiments and results. Finally, Section 8 concludes and defines pathways for future work.

2 Related Work

The area of static computing resource management has been well studied in the context of grids, clouds, and even multi-clouds [12]. However, the field of dynamic resource management in response to continuously varying workloads, which is especially important for web facing applications [12], is still in its infancy. Horizontal autoscaling policies are the predominant approach for dynamic resource management and thus they have gained significant attention in recent years.

Lorido-Botran et al. classify autoscaling policies as *reactive* and *predictive* or *proactive* [13]. The most widely adopted *reactive* approaches are based on threshold rules for performance metrics (e.g. CPU and RAM utilisation). For each such characteristic the system administrator provides a lower and upper threshold values. Resources are provisioned whenever an upper threshold is exceeded. Similarly, if a lower threshold is reached resources are released. How much resources are acquired or released when a threshold is reached is specified in user defined autoscaling rules. There are different “flavours” of threshold based approaches. For example in Amazon Auto Scaling [14] one would typically use the average metrics from the virtual server farm, while RightScale [15] provides a voting scheme, where thresholds are considered per VM and an autoscaling action is taken if the majority of the VMs “agree” on it. Combinations and extensions of both of these techniques have also been proposed [16–18]. *Predictive* or *proactive* approaches try to predict demand changes in order to allocate or deallocate resources. Multiple methods using approaches like reinforcement learning [19, 20], queuing theory [21] and Kalman filters [22] to name a few have been proposed.

Our work is complementary to all these approaches. They indicate at what time resources should be provisioned, but do not select the resource type. Our approach selects the best resource (i.e. VM type) once it has been decided that the system should scale up horizontally.

Fernandez et al. propose a system for autoscaling web applications in clouds [23]. They monitor the performance of different VM types to infer their capacities. Our approach to this is different, as we inspect the available to each VM CPU capacity and measure the amount of “stolen” CPU instructions by the hypervisor from within the VM itself. This allows us to normalise the VMs’ resource capacities to a common scale, which we use to compare them and for

further analysis. Furthermore, their approach relies on a workload predictor, while ours is usable even in the case of purely reactive autoscaling.

Singh et al. use k-means clustering to analyse the workload mix (i.e. the different type of sessions) and then use a queueing model to determine each server’s suitability [24]. However, they do not consider the performance variability of virtual machines, which we take into account. Also, they do not select the type of resource (e.g. VM) to provision and assume there is only one type, while this is precisely the focus of our work.

A part of our work is concerned with automated detection of application behaviour changes through a Hierarchical Temporal Memory (HTM) model. Similar work has been carried out by Cherkasova et al. [8], who propose a regression based anomaly detection approach. However, they analyse only the CPU utilisation. Moreover they consider that a set of user transactions’ types is known beforehand. In contrast, our approach considers RAM as well and does not require application specific information like transaction types. Tan et al. propose the PREPARE performance anomaly detection system [25]. However, their approach can not be used by a cloud client, as it is built on top of the Xen virtual machine manager to which external clients have no access.

Another part of our method is concerned with automatic selection of the *learning rate* and *momentum* of an artificial neural network (ANN). There is a significant amount of literature in this area as surveyed by Moreira and Fiesler [26]. However, the works they overview are applicable for static data sets and have not been applied to learning from streaming online data whose patterns can vary over time. Moreover, they only consider how the intermediate parameters of the backpropagation algorithm vary and do not use additional domain specific logic. Although our approach is inspired by the work of Vogl et al. [27] as it modifies the *learning rate* and *momentum* based on the prediction error, we go further and we modify them also based on the *anomaly score* as reported by the Hierarchical Temporal Memory (HTM) models.

3 Method Overview

Figure 2 depicts an overview of our machine learning approach and how the system components interact. Within each AS VM we install a monitoring program which periodically records utilisation metrics. These measurements are transferred to an *autoscaling component*, which can be hosted either in a cloud VM or on-premises. It is responsible for (i) monitoring AS VMs’ performance (ii) updating machine learning models of the application behaviour and (iii) autoscaling.

Within each AS VM the *utilisation monitors* report statistics about the CPU, RAM, disk and network card utilisations and the number of currently served users. These records are transferred every 5 seconds to the *autoscaling component*, where they are normalised, as different VMs have different de facto resource capacities. In the machine learning approaches we only consider the CPU and RAM utilisations, as disk and network utilisations of AS VMs are

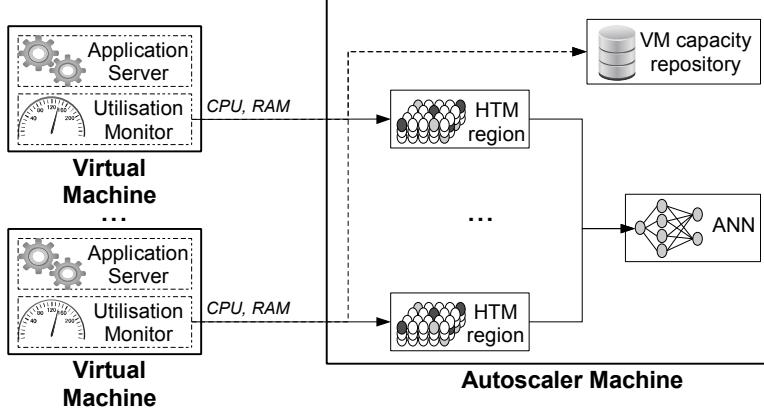


Figure 2: System components and their interaction.

typically small [28, 29].

For each AS VM the *autoscaler* maintains a separate single-region Hierarchical Temporal Memory (HTM) model [30], which is overviewed in a later section. In essence we use HTMs to detect changes in the application behaviour of each AS VM. We prefer HTM to other regression based anomaly detection approaches, as it can detect anomalies on a stream of multiple parameters (e.g. CPU and RAM). Whenever monitoring data is retrieved from an AS VM, the *autoscaler* trains its HTM with the received number of users, CPU and RAM utilisations and outputs an *anomaly score* defining how “unexpected” the data is.

As a next step we use these utilisation measurements to train a 3-tier artificial neural network (ANN) about the relationship between the number of served users and resource consumptions. We choose to use an ANN because of its suitability for online data streams. Other “sliding window” approaches operate only on a portion of the data stream. As a system’s utilisation patterns can remain the same for long time intervals, the window sizes may need to become impractically large or even be dynamically adjusted. On the contrary, an ANN does not operate on a fixed time window and is more adept with changes in the incoming data stream, as we will detail in a later section.

There is only one ANN and training samples from all AS VMs are used to train it. In essence the ANN represents a continuously updated regression model, which given a number of users predicts the needed resources to serve them within a single VM without causing resource contentions. Thus, we need to filter all training samples, which were taken during anomalous conditions (e.g. insufficient CPU or RAM capacity causing intensive context switching or disk swapping respectively). Such samples are not indicative of the relationship between number of users and the resource requirements in the absence of resource contentions. Furthermore, we use the *anomaly score* of each training sample (extracted from HTM) to determine the respective *learning speed* and

momentum parameters of the back propagation algorithm so that the ANN adapts quickly to changes in the utilisation patterns.

Training the ANN and the HTMs happens online from the stream of VM measurements in parallel with the running application. Simultaneously we also maintain a *VM capacity repository* of the latest VM capacity measurements. When a new VM is needed by the autoscaling component, we use this repository to infer the potential performance capacity of all VM types. At that time the ANN is already trained adequately and given the predicted performance capacities can be used to infer how many users each VM type could serve simultaneously. Based on that we select the VM type, with minimal cost to number of users ratio.

4 Learning Application Behaviour

4.1 Utilisation Monitoring

To measure VM performance utilisation, we use the *SAR*, *mpstat*, *vmstat* and *netstat* Linux monitoring tools. We use the *mpstat %idle* metric to measure the percentage of time during which the CPU was idle. The *%steal* metric describes the percentage of “stolen” CPU cycles by a hypervisor (i.e. the proportion of time the CPU was not available to the VM) and can be used to evaluate the actual VM CPU capacity. Similarly, *SAR* provides the *%util* and *%ifutil* metrics as indicative of the disk’s and network card’s utilisations.

Measuring the RAM utilisation is more complex as operating systems keep in memory cached copies of recently accessed disk sectors in order to reduce disk access [29]. Although in general this optimisation is essential for VM performance, web application servers (AS) are not usually I/O bound, as most of the application persistence is delegated to the data base layer. Hence, using the *vmstat* RAM utilisation metrics can be an overestimation of the actual memory consumption as it includes rarely accessed disk caches. Thus, we use the “*active memory*” *vmstat* metric to measure memory consumption instead. It denotes the amount of recently used memory, which is unlikely to be claimed for other purposes.

Lastly, we need to evaluate the number of concurrently served users in an AS VM. This could be extracted from the AS middleware, but that would mean writing specific code for each type of middleware. Moreover, some proprietary solutions may not expose this information. Therefore, we use the number of distinct IP addresses with which the server has an active TCP socket, which can be obtained through the *netstat* command. Typically, the AS VM is dedicated to running the AS and does not have other outgoing connections except for the connection to the persistence layer. Therefore, the number of addresses with active TCP sockets is a good measure of the number of currently served users.

4.2 Normalisation and Capacity Estimation

Before proceeding to train the machine learning approaches, we need to normalise the measurements which have different “scales”, as the VMs have different RAM sizes and CPUs with different frequencies. Moreover, the actual CPU capacities within a single VM vary over time as a result of the dynamic collocation of other VMs on the same host.

As a first step in normalising the CPU load, we need to evaluate the actual CPU capacity available to each VM. This can be extracted from the `/proc/cpuinfo` Linux kernel file. If the VM has n cores, `/proc/cpuinfo` will list meta information about the physical CPU cores serving the VM including their frequencies fr_1, \dots, fr_n . The sum of these frequencies is the maximal processing capacity the VM can get, provided the hypervisor does not “steal” any processing time. Using the `%steal` mpstat parameter we can actually see what percentage of CPU operations have been taken away by the hypervisor. Subtracting this percentage from the sum of frequencies gives us the actual VM CPU capacity at the time of measurement. To normalise we further divide by the maximal CPU core frequency fr_{max} multiplied by the maximal number of cores n_{max_cores} of all considered VMs in the cloud provider. This is a measure of the maximal VM CPU capacity one can obtain from the considered VM types. As clouds are made of commodity hardware, we will consider $fr_{max} = 3.5\text{GHz}$. This ensures that all values are in the range $(0, 1]$, although for some cloud providers all values may be much lower than 1, depending on the underlying hardware they use. This is formalised in Eq. 1.

$$cpuCapacityNorm = \frac{(100 - \%steal) \sum_{i=0}^n fr_i}{100 n_{max_cores} fr_{max}} \quad (1)$$

Having computed the VM CPU capacity, we store it into the *VM capacity repository*, so we can use it later on to infer the capacities of future VMs. Each repository record has the following fields:

- *time* - a time stamp of the capacity estimation;
- *vm-type* - an identifier of the VM type - e.g. “m1.small”;
- *vm-id* - a unique identifier of the VM instance - e.g. its IP or elastic DNS address;
- *cpuCapacityNorm* - the computed CPU capacity.

If we further subtract the `%idle` percentage from the capacity we will get the actual CPU load given in Eq. 2.

$$cpuLoadNorm = \frac{(100 - \%idle - \%steal) \sum_{i=0}^n fr_i}{100 n_{max_cores} fr_{max}} \quad (2)$$

Normalising the RAM load and capacity is easier, as they do not fluctuate like the CPU capacity. We divide the *active memory* by the maximal amount of memory RAM_{max} in all considered virtual machine types in the cloud - see Eq. 3.

$$ramLoadNorm = \frac{active_memory}{RAM_{max}} \quad (3)$$

Whenever a new AS VM is needed, we have to estimate the CPU and RAM capacities of all available VM types based on the *capacity repository* and their performance definitions provided by the provider. The normalised RAM capacity of a VM type is straightforward to estimate as we just need to divide the capacity in the provider's specification by RAM_{max} . To estimate the CPU capacity of a VM type we use the mean of the last 10 entries' capacities for this type in the *capacity repository*. If there are no entries for this VM type in the repository (i.e. no VM of this type has been instantiated) we can heuristically extrapolate the CPU capacity from the capacities of the other VM types. Typically IaaS providers specify an estimation of each VM type's CPU capacity - e.g. Google Compute Engine Units (GCEU) in Google Compute Engine or Elastic Compute Units (ECU) in AWS. Hence given an unknown VM type vmt we can extrapolate its normalised CPU capacity as:

$$\begin{aligned} cpuCapacity(vmt) = & \\ & \frac{1}{|V|} \sum_{vmt_i \in V} \frac{cpuCapacity(vmt_i) \times cpuSpec(vmt_i)}{cpuSpec(vmt)} \end{aligned} \quad (4)$$

Where V is the set of VM types present in the *capacity repository* and whose CPU capacity can be determined from previous measurements, $|V|$ is its cardinality, and $cpuSpec(vmt_i)$ defines the cloud provider's estimation of a VM type's capacity - e.g. number of GCEUs or ECUs.

4.3 Anomaly Detection Through HTM

The Hierarchical Temporal Memory (HTM) model is inspired by the structure and organisation of the neocortex. It has been developed and commercialised by the Grok company [31] (formerly Numenta [32]), and follows the concepts from Jeff Hawkins' book "On Intelligence" [33]. The model creators build upon the seminal work of Mountcastle [34] that the neocortex is predominantly uniform in structure and function even in regions handling different sensory inputs - e.g. visual, auditory, and touch. The HTM model tries to mimic this structure in a computational model. There are several differences compared to the biological structure of the neocortex in order to be computationally viable as described in the implementation white paper [30]. Grok's implementation is available as an open source project called NuPIC [35]. In this section, we provide only a brief overview of HTM to introduce the reader to this concept. The interested reader is referred to the official documentation [30].

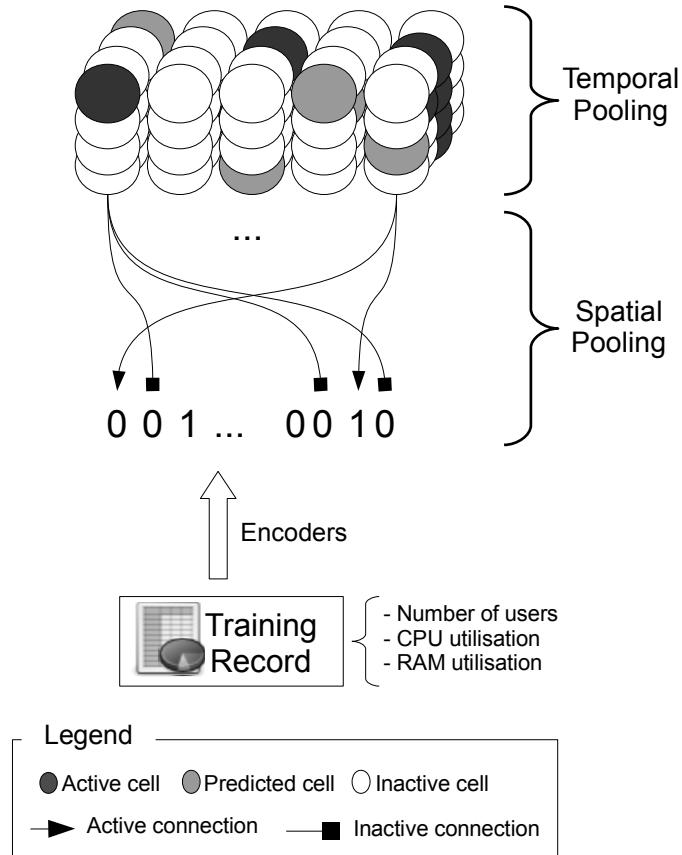


Figure 3: HTM region structure.

HTMs consist of one or several stacked regions. During inference, input arrives into the lowest region, whose output serves as input to the successive one and so forth until the topmost region outputs the final result. The purpose of a region is to convert noisy input sequences to more stable abstract representations. Conceptually, the different regions represent different levels of abstraction in the learning process - i.e. the lowest level recognises low-level patterns, while each higher level layer recognises more complex ones based on the result of the previous one. In this work, we use single-region HTMs and we will focus on them in the rest of the section.

A HTM region consists of columns of cells, which are most often arranged in a three dimensional grid - see Figure 3. Each cell can be in one of three possible states: (i) active from feed forward input, (ii) active from lateral input (i.e. predicted), or (iii) inactive. Conceptually, active cells represent the state of the last input and predicted cells represent the likely state after future inputs. A HTM region receives as input a bit sequence. Special *encoders* are used to

convert input objects into bitwise representations, so that objects which are “close” in the sense of the target domain have similar bit representations. Upon receiving new binary input the HTM changes the states of the columns based on several rules summarised below.

As a first step the HTM has to decide which columns’ cells will be activated for a given input - an algorithm known as *Spatial Pooling*. It nullifies most of the 1 bits, so that only a small percentage (by default 2%) are active. Each column is connected with a fixed sized (by default 50% of the input length) random subset of input bits called the *potential pool*. Each column’s connection to an input bit has a ratio number in the range [0,1] associated with it known as the *permanence*. HTM automatically adjusts the *permanence* value of a connection after a new input record arrives, so that input positions whose value have been 0 or 1 and are members of the *potential pool* of a selected column are decreased or increased respectively. Connections with *permanences* above a predefined thresholds are considered active. Given an input, for each column the HTM defines its *overlap score* as the number of active bits with active connections. Having computed this for every column, HTM selects a fixed sized (by default 2%) set of columns with the highest *overlap score*, so that no two columns within a predefined radius are active.

As a second step, HTM decides which cells within these columns to activate. This is called *Temporal Pooling*. Within each of the selected columns the HTM activates only the cells which are in *predicted* state. If there are no cells in predicted state within a column, then all of its cells are activated, which is also known as *bursting*.

Next, the HTM makes a prediction of what its future state will be - i.e. which cells should be in predicted state. The main idea is that when a cell activates it establishes connections to the cells which were previously active. Each such connection is assigned a weight number. Over time if the two nodes of a connection become active in sequence again, this connection is strengthened, i.e. the weight is increased. Otherwise, the connection slowly decays, i.e. the weight is gradually decreased. Once a cell becomes active, all non-active cells having connections to it with weights above a certain threshold are assigned the predicted state. This is analogous to how synapses form and decay between neurons’ dendrites in the neocortex in response to learning patterns.

The presence of predicted cell columns allows a HTM to predict what will be its likely state in terms of active cells after the next input. However, it also allows for the detection of anomalies. For example, if just a few predicted states become active this is a sign that the current input has not been expected. Thus the *anomaly_score* is defined as the proportion of active spatial pooler columns that were incorrectly predicted and is in the range [0, 1].

In our environment every 5 seconds we feed each HTM with a time stamp, the number of users and the CPU and RAM utilisations of the respective VM. We use the standard NuPIC scalar and date encoders to convert the input to binary input. As a result we get an *anomaly score* denoting how expected the input is, in the light of the previously described algorithms.

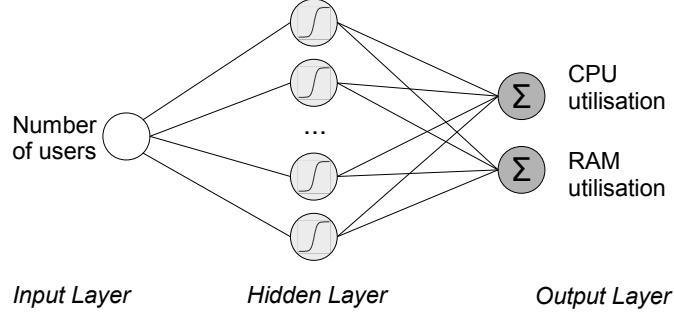


Figure 4: ANN topology.

4.4 ANN Training

Figure 4 depicts the topology of the artificial neural network (ANN). It has one input — the number of users. The hidden layer has 250 neurons with the sigmoid activation function. The output layer has two output nodes with linear activation functions, which predict the normalised CPU and RAM utilisations within an AS VM.

Once a VM’s measurements are received and normalised and the *anomaly score* is computed by the respective HTM region, the ANN can be trained. As discussed, we need to filter out the VM measurements which are not representative of normal, contention free application execution, in order to “learn” the “right” relationship between number of users and resource utilisations. We filter all VM measurements in which the CPU, RAM, hard disk or network card utilisations are above a certain threshold (e.g. 70%). Similarly, we filter measurements with negligible load — i.e. less than 25 users or less than 10% CPU utilisation. We also ignore measurements from periods during which the number of users has changed significantly — e.g. in the beginning of the period there were 100 users and at the end there were 200. Such performance observations are not indicative of an actual relationship between number of users and resource utilisations. Thus, we ignore measurements for which the number of users is less than 50% or more than 150% of the average of the previous 3 measured numbers of users from the same VM.

Since we are training the ANN with streaming data, we need to make sure it is not overfitted to the latest training samples. For example if we have constant workload for a few hours we will be receiving very similar training samples in the ANN during this period. Hence the ANN can become overfitted for such samples and lose its fitness for the previous ones. To avoid this problem, we filter out measurements/training samples, which are already well predicted. More specifically, if a VM measurement is already predicted with a *root mean square error* (RMSE) less than 0.01 it is filtered out and the ANN is not trained with it. We call this value $rmse^{pre}$ because it is obtained for each training sample before the ANN is trained with it. It is computed as per Eq. 5, where $output_i$ and $expected_i$ are the values of the output neurons and the expected values

respectively.

$$rmse^{pre} = \sqrt{\sum (output_i - expected_i)^2} \quad (5)$$

With each measurement, which is not filtered out, we perform one or several iterations/epochs of the back-propagation algorithm with the number of users as input and the normalised CPU and RAM utilisations as expected output. The back-propagation algorithm has two important parameters — the *learning rate* and the *momentum*. In essence, the *learning rate* is a ratio number in the interval $(0, 1)$ which defines the amount of weight update in the direction of the gradient descent for each training sample [26]. For each weight update, the *momentum* term defines what proportion of the previous weight update should be added to it. It is also a ratio number in the interval $(0, 1)$. Using a *momentum* the neural network becomes more resilient to oscillations in the training data by “damping” the optimisation procedure [26].

For our training environment we need a low *learning rate* and a high *momentum*, as there are a lot of oscillations in the incoming VM measurements. We select the *learning rate* to be $lr = 0.001$ and the *momentum* $m = 0.9$. We call these values the *ideal parameters*, as these are the values we would like to use once the ANN is close to convergence. However, the low *learning rate* and high *momentum* result in slow convergence in the initial stages, meaning that the ANN may not be well trained before it is used. Furthermore, if the workload pattern changes, the ANN may need a large number of training samples and thus time until it is tuned appropriately. Hence the actual *learning rate* and *momentum* must be defined dynamically.

One approach to resolve this is to start with a high *learning rate* and low *momentum* and then respectively decrease/increase them to the desired values [26, 27]. This allows the back-propagation algorithm to converge more rapidly during the initial steps of the training. We define these parameters in the initial stages using the asymptotic properties of the sigmoid function, given in Eq. 6.

$$s(x) = \frac{1}{1 - e^{-x}} \quad (6)$$

As we need to start with a high *learning rate* and then decrease it gradually to lr , we could define the learning rate lr_k for the k -th training sample as $s(-k)$. However, the sigmoid function decreases too steeply for negative integer parameters and as a result the learning rate is higher than lr for just a few training samples. To solve this we use the square root of k instead and thus our first approximation of the *learning rate* is:

$$lr_k^{(1)} = max(lr, s(-\sqrt{k})) \quad (7)$$

As a result $lr_k^{(1)}$ gradually decreases as more training samples arrive. Figure 5 depicts how it changes over time.

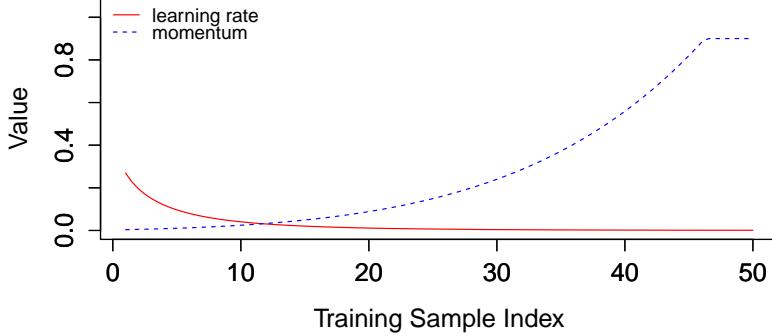


Figure 5: The $lr_k^{(1)}$ approximation of the *learning rate* and the respective momentum during the initial ANN training stages.

We also need to ensure that it increases in case unusual training data signalling a workload change arrives and thus we need to elaborate $lr_k^{(1)}$. For this we keep a record of the last 10 samples’ *anomaly scores* and errors (i.e. $rmse^{pre}$). The higher the latest anomaly scores, the more “unexpected” the samples are and therefore the *learning rate* must be increased. Similarly, the higher the sample’s $rmse^{pre}$ compared to the previous errors, the less fit for it the ANN is and thus the *learning rate* must be increased as well. Thus our second elaborated approximation of the *learning rate* is:

$$lr_k^{(2)} = lr_k^{(1)} \max\left(1, \frac{rmse_k^{pre}}{\overline{rmse}}\right) \prod_{i=0}^9 2s(an_{k-i}) \quad (8)$$

where an_k and $rmse_k^{pre}$ are the *anomaly score* and the error of the k -th sample and \overline{rmse} is the average error of the last 10 samples. Note that we use the sigmoid function for the anomaly scores in order to diminish the effect of low values.

In some cases the *learning rate* can become too big in the initial training iterations, which will in fact hamper the convergence. To overcome this problem, for each sample k we run a training iteration with $lr_k^{(2)}$, compute its RMSE $rmse_k^{post}$ and then revert the results of this iteration. By comparing $rmse_k^{pre}$ and $rmse_k^{post}$ we can see if training with this $lr_k^{(2)}$ will contribute to the convergence [27]. If not, we use the ideal parameter lr instead. Thus we finally define the *learning rate* parameter lr_k in Eq. 9:

$$lr_k = \begin{cases} lr_k^{(2)} & \text{if } rmse_k^{pre} > rmse_k^{post} \\ lr & \text{otherwise} \end{cases} \quad (9)$$

Similarly we have to gradually increase the *momentum* as we decrease the *learning rate* until the ideal *momentum* is reached. If a workload change is present we need to decrease the *momentum* in order to increase the learning

speed. Hence, we can just use the ratio of the ideal learning rate lr to the current one as shown in Eq. 10.

$$m_k = \min(m, \frac{lr}{lr_k^{(2)}}) \quad (10)$$

Figure 5 depicts how the *learning rate* and *momentum* change during the initial training stages, given there are no anomalies, accuracy losses and $\forall k : rmse_k^{pre} > rmse_k^{post}$ — i.e. when $\forall k : lr_k^{(1)} = lr_k^{(2)} = lr_k$. Figure 7 shows the actual lr_k given realistic workload.

Furthermore, to speed up convergence it is beneficial to run multiple *epochs* (i.e. repeated training iterations) with the first incoming samples and with samples taken after a workload change. The ideal *learning rate* lr and its approximation $lr_k^{(2)}$ already embody this information and we could simply use their ratio. However, $\frac{lr_k^{(2)}}{lr}$ can easily exceed 300 given $lr = 0.001$, resulting in over-training with particular samples. Hence we take the logarithm of it as in Eq. 11:

$$e_k = \left\lfloor 1 + \ln\left(\frac{lr_k^{(2)}}{lr}\right) \right\rfloor \quad (11)$$

5 Virtual Machine Type Selection

When a new VM has to be provisioned the ANN should be already trained so that we can estimate the relationship between number of users and CPU and RAM requirements. The procedure is formalised in Algorithm 1. We loop over all VM types VT (line 3) and for each one we estimate its normalised CPU and RAM capacity based on the *capacity repository* as explained earlier (lines 5-6). The VM cost per time unit (e.g. hour in AWS or minute in Google Compute Engine) is obtained from the provider's specification (line 7).

Next we approximate the number of users that a VM of this type is expected to be able to serve (lines 10-18). We iteratively increase n by Δ starting from $minU$, which is the minimal number of users we have encountered while training the neural network. We use the procedure *predict* (defined separately in Algorithm 2) to estimate the normalised CPU and RAM demands that each of these values of n would cause. We do so until the CPU or RAM demands exceed the capacity of the inspected VM type. Hence, we use the previous value of n as an estimation of the number of users a VM of that type can accommodate. Finally, we select the VM type with the lowest cost to number of users ratio (lines 20-23).

Algorithm 2 describes how to predict the normalised utilisations caused by n concurrent users. If n is less than the maximum number of users $maxU$ we trained the ANN with, then we can just use the ANN's prediction (line 5). However, if n is greater than $maxU$ the ANN may not predict accurately. For example if we have used a single *small* VM to train the ANN, and then we try to

Algorithm 1: Dynamic VM Type Selection (DVTS)

```
    input :  $VT$ ,  $ann$ ,  $\Delta$ ,  $minU$ ,  $maxU$ 
1  $bestVmt \leftarrow null;$ 
2  $bestCost \leftarrow 0;$ 
3 for  $vmt \in VT$  ;                                // Inspect all VM types
4 do
5    $cpuCapacity \leftarrow vmt$ 's norm. CPU capacity ;
6    $ramCapacity \leftarrow vmt$ 's norm. RAM capacity;
7    $vmtCost \leftarrow vmt$ 's cost per time unit;
8    $userCapacity \leftarrow 0;$ 
9    $n \leftarrow minU;$ 
10  while  $True$  ;                               // Find how many users it can take
11  do
12     $cpu, ram \leftarrow predict(ann, n, minU, maxU);$ 
13    if  $cpu < cpuCapacity$  and  $ram < ramCapacity$  then
14      |  $userCapacity \leftarrow n;$ 
15    else
16      | break;
17    end
18     $n \leftarrow n + \Delta;$ 
19  end
20  // Approximate the cost for a user per time unit
21   $userCost \leftarrow \frac{vmtCost}{userCapacity};$ 
22  // Find the cheapest VM type
23  if  $userCost < bestCost$  then
24    |  $bestCost \leftarrow userCost;$ 
25    |  $bestVmt \leftarrow vmt;$ 
26  end
27 end
28 return  $bestVmt;$ 
```

predict the capacity of a *large* VM, n can become much larger than the entries of the training data and the regression model may be inaccurate. Thus, we extrapolate the CPU and RAM requirements (lines 7-11) based on the range of values we trained the ANN with and the performance model we have proposed in a previous work [29].

6 Benchmark and Prototype

There are two main approaches for experimental validation of a distributed system's performance — through a simulation or a prototype. Discrete event

Algorithm 2: Resource Utilisation Estimation

```
input : ann, n, minU, maxU
1  cpu ← 0;
2  ram ← 0;
3 if n < maxUsers ;                                // If within range - use ANN
4 then
5   cpu, ram ← ann.run(n);
6 else
7   // If outside range - extrapolate
8   minRam, minCPU ← ann.run(minU);
9   maxRam, maxCPU ← ann.run(maxU);
10  cpuPerUser ←  $\frac{(maxCPU - minCPU)}{(maxU - minU)}$ ;
11  ramPerUser ←  $\frac{(maxRam - minRam)}{(maxU - minU)}$ ;
12  cpu ← maxCPU + cpuPerUser(n - maxU)
13  ram ← maxCPU + ramPerUser(n - maxU)
14 end
15 return cpu, ram;
```

simulators like CloudSim [36] have been used throughout industry and academia to quickly evaluate scheduling and provisioning approaches for large scale cloud infrastructure without having to pay for expensive test beds. Unfortunately, such simulators work on a simplified cloud performance model and do not represent realistic VM performance variability, which is essential for testing our system. Moreover, simulations can be quite inaccurate when the simulated system serves resource demanding workloads, as they do not consider aspects like CPU caching, disk data caching in RAM and garbage collection [29]. Therefore, we test our method through a prototype and a standard benchmark deployed in a public cloud environment.

We validate our approach with the CloudStone [37, 38] web benchmark deployed in Amazon AWS. It follows the standard 3-tier architecture. By default CloudStone is not scalable, meaning that it can only use a single AS. Thus we had to extend it to accommodate multiple servers. Our installation scripts and configurations are available as open source code. For space considerations we will not discuss these technical details and will only provide an overview. The interested readers can refer to our online documentation and installation instructions.¹

The benchmark deployment topology is depicted in Figure 6. CloudStone uses the *Faban* harness to manage the runs and to emulate users. The *faban driver*, which is deployed in the client VM communicates with the *faban agents* deployed in other VMs to start or stop tests. It also emulates the incoming

¹<http://nikolaygrozev.wordpress.com/2014/06/02/advanced-automated-cloudstone-setup-in-ubuntu-vms-part-2/>

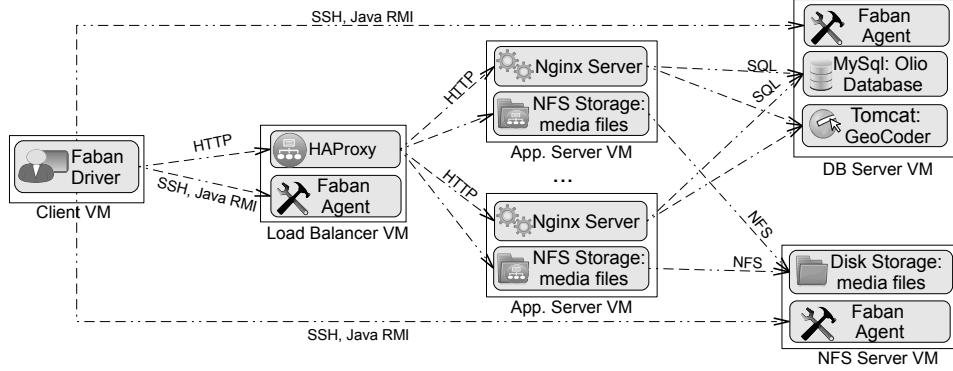


Figure 6: CloudStone benchmark’s extended topology.

user requests to the application. These requests arrive at a HAProxy *load balancer* which distributes them across one or many application servers (AS). CloudStone is based on the Olio application, which is a PHP social network website deployed in a Nginx server. In the beginning we start with a single AS “behind” the *load balancer*. When a new AS VM is provisioned we associate it with the *load balancer*. We update its weighted round robin policy, so that incoming request are distributed among the AS VMs proportionally to their declared CPU capacity (i.e. ECU).

The persistent layer is hosted in a MySQL server deployed within a separate DB VM. CloudStone has two additional components - (i) a geocoding service called *GeoCoder*, hosted in an Apache Tomcat server and (ii) a shared *file storage* hosting media files. They are both required by all application servers. We have deployed the geocoding service in the DB VM. The file storage is deployed in a Network File System (NFS) server on a separate VM with 1TB EBS storage, which is mounted from each AS VM.

We use “m3.medium” VMs for the client, load balancer and DB server and “m1.small” for the NFS server. The types of the AS VMs are defined differently for each experiment. All VMs run 64bit Ubuntu Linux 14.04.

Our prototype of an autoscaling component is hosted on an on-premises physical machine and implements the previously discussed algorithms and approaches. It uses the JClouds [39] multi-cloud library to provision resources, and thus can be used in other clouds as well. We use the NuPIC [35] and FANN [40] libraries to implement HTM and ANN respectively. We ignore the first 110 *anomaly scores* reported from the HTM, as we observed that these results are inaccurate (i.e. always 1 or 0) until it receives initial training. Whenever a new AS VM is provisioned we initialise it with a deep copy of the HTM of the first AS VM, which is the most trained one. The monitoring programs deployed within each VM are implemented as bash scripts, and are accessed by the autoscaling component through SSH. Our implementation of Algorithm 2 uses $\Delta = 5$.

Previously we discussed that the number of current users could be approximated by counting the number of distinct IP addresses to which there is an

Table 1: AWS VM type definitions.

VM type	ECU	RAM	Cost per hour
m1.small	1	1.7GB	\$0.058
m1.medium	2	3.75GB	\$0.117
m3.medium	3	3.75GB	\$0.098

active TCP session. However, in CloudStone all users are emulated from the same client VM and thus have the same source IP address. Thus, we use the number of recently modified web server session files instead.

Our autoscaling component implementation follows the Amazon Auto Scaling [14] approach and provisions a new AS VM once the average utilisation of the server farm reaches 70% for more than 10 seconds. Hence, we ensure that in all experiments the AS VMs are not overloaded. Thus, even if there are SLA violations, they are caused either by the network or the DB layer, and the AS layer does not contribute to them. We also implement a *cool down* period of 10 minutes.

7 Validation

In our experiments, we consider three VM types: *m1.small*, *m1.medium* and *m3.medium*. Table 1 summarises their cost and declared capacities in the Sydney AWS region which we use.

In all experiments we use the same workload. We start by emulating 30 users and each 6 minutes we increase the total number of users with 10 until 400 users are reached. To achieve this we run a sequence of CloudStone benchmarks, each having 1 minute ramp-up and 5 minutes steady state execution time. Given CloudStone’s start-up and shut-down times, this amounts to more than 5 hours per experiment. The goal is to gradually increase the number of users, thus causing the system to scale up multiple times.

To test our approach in the case of a workload characteristic change we “inject” such a change 3.5 hours after each experiment’s start. To do so we manipulate the *utilisation monitors* to report higher values. More specifically they increase the reported CPU utilisations with 10% and the reported RAM utilisation with 1GB plus 2MB for every currently served user.

We implement one experiment, which is initialised with a *m1.small* AS VM and each new VM’s type is chosen based on our method (DVTS). We also execute 3 baseline experiments, each of which statically selects the same VM type whenever a new VM is needed, analogously to the standard AWS Auto Scaling rules.

First we investigate the behaviour of DVTS before the workload change. It continuously trains one HTM for the first AS VM and the ANN. In the initial stages the ANN *learning rate* and *momentum* decrease and increase respectively to facilitate faster training. For example, the *learning rate* lr_k (defined in Eq. 9) during the initial stages is depicted in Fig 7. It shows how lr_k drastically reduces

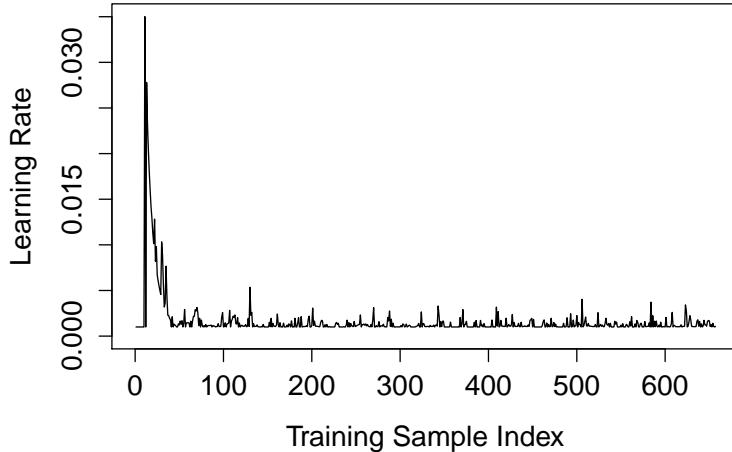


Figure 7: Learning rate lr_k during initial stages of training the ANN.

as the ANN improves its accuracy after only a few tens of training samples. Once the AS VM gets overloaded we select a new VM type. At this point we only have information about $m1.small$ in the *capacity repository* and therefore we infer the other CPU capacities based on Eq. 4. Finally using Algorithm 1 we select $m3.medium$ as the type for the second VM.

After the new VM is instantiated, the autoscaling component starts its monitoring. It trains the ANN and a new dedicated HTM with its measurements. It also updates the *capacity repository* with the CPU capacity of the new VM. Surprisingly, we observe that on average its CPU capacity is about 35% better than the one of the $m1.small$ VM, even though according to the specification $m3.medium$ has 3 ECUs and $m1.small$ has 1. Therefore, the previous extrapolation of $m3.medium$'s capacity has been an overestimation. Hence, when a new VM is needed again, the algorithm selects $m1.small$ again.

3.5 hours after the start of the experiment the workload change is injected. This is reflected in the HTMs' anomaly scores an_k and the ANN's errors. Consequently, the *learning rate* lr_k , the *momentum* m_k and the *epochs* e_k also change to speed up the learning process as per equations 9, 10 and 11 and as a result the ANN adapts quickly to the workload change. As discussed for each sample we compute its error (RMSE-pre) before updating the ANN. Figure 8 depicts how these errors increase when the change is injected and decrease afterwards as the ANN adapts timely.

Eventually the load increases enough so the system needs to scale up again. Due to the injected change, the workload has become much more memory intensive, which is reflected in the ANN's prediction. Hence $m1.small$ can serve just a few users, given it has only 1.7GB RAM. At that point the CPU capacity of $m1.medium$ is inferred from the capacities of $m1.small$ and $m3.medium$ as per Eq. 4, since it has not been used before. Consequently Algorithm 1 selects $m1.medium$ for the 4th VM just before the experiment completes.

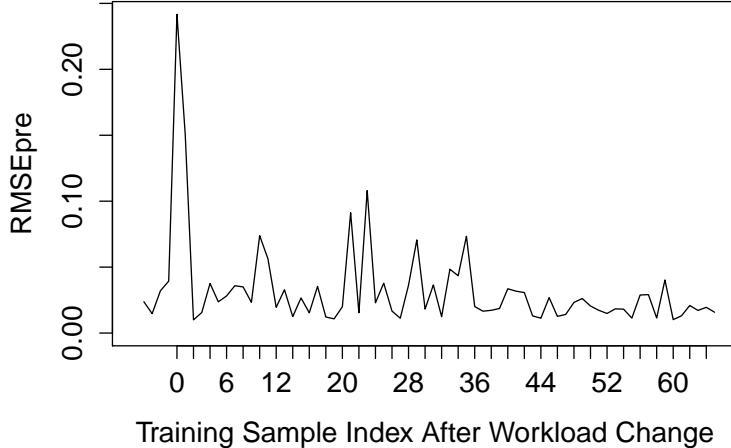


Figure 8: RMSE-pre in the presence of a workload change. The 0 index corresponds to the first sample after the workload change.

For each experiment, Figure 9 depicts the timelines of the allocated VMs and the total experiment costs. For each VM the type and cost are specified to the right. Our selection policy is listed as *DVTS*. The baseline policy which statically selects *m1.small* allocates 8 new VMs after the workload change as *m1.small* can serve just a few users under the new workload. In fact, if there was no *cool down* period in the autoscaling, this baseline would have exceeded the AWS limit of allowed number of VM instances before the end of the experiment. The baselines which select *m1.medium* and *m3.medium* fail to make use of *m1.small* instances before the change injection, which offers better performance for money.

Admittedly, in the beginning DVTS did a misstep with the selection of *m3.medium*, because it started with an empty *capacity repository* and had to populate it and infer CPU capacities “on the go”. This could have been avoided by prepopulating the *capacity repository* with test or historical data. We could expect that such inaccuracies are avoided at later stages, once more capacity and training data is present. Still, our approach outperformed all baselines in terms of incurred costs with more than 20% even though its effectiveness was hampered by the lack of contextual data in the initial stages.

Our experiments tested DVTS and the baselines with a workload, which is lower than what is observed in some applications. While our tests did not allocate more than 12 VMs (in the baseline experiment, which statically allocates *m1.small*) many real world systems allocate hundreds or even thousands of servers. We argue that in such cases, DVTS will perform better than demonstrated, as there will be much more training data and thus the VM types’ capacity estimations will be determined more accurately and the machine learning approaches will converge faster. As discussed, that would allow some of the initial missteps of DVTS to be avoided. Moreover, as the number of AS VMs

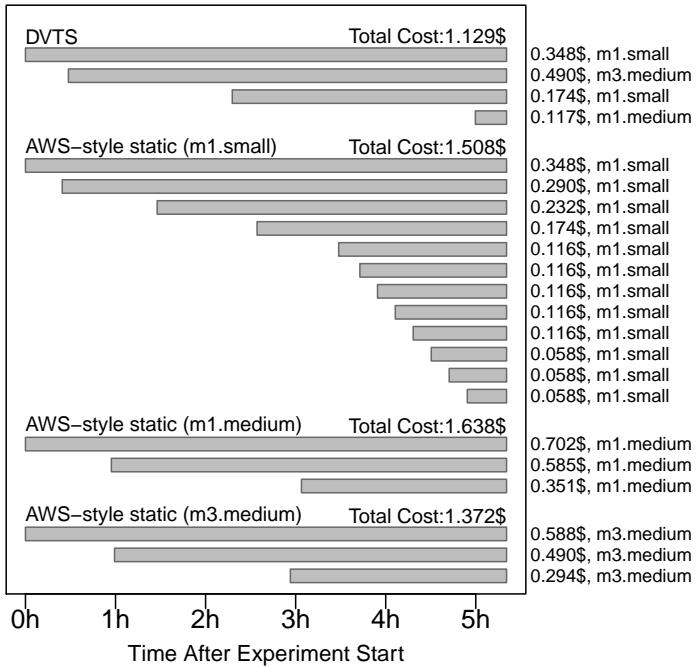


Figure 9: Timelines and costs of all VMs grouped by experiments. DVTS is our approach. The AWS-style policies are the baselines, which statically select a predefined VM type.

grows, so does the cost inefficiency caused by the wastage of allocated resources, which can be reduced by DVTS.

Finally, the response times in the DVTS experiment and all baseline experiments were equivalent. All experiments scale up once the AS VMs’ utilisations exceed the predefined thresholds, and thus never become overloaded enough to cause response delays. The load balancer is equally utilised in all experiments, as it serves the same number of users, although it redirects them differently among the AS VMs. Similarly, the DB layer is equally utilised, as it always serves all users from all AS VMs.

8 Conclusions and future work

In this work we have introduced an approach for VM type selection when autoscaling application servers. It uses a combination of heuristics and machine learning approaches to “learn” the application’s performance characteristics and to adapt to workload changes in real time. To validate our work, we have developed a prototype, extended the CloudStone benchmark and executed experiments in AWS EC2. We have made improvements to ensure our machine learning techniques train quickly and are usable in real time. Also we have intro-

duced heuristics to approximate VM resource capacities and workload resource requirements even if there is no readily usable data, thus making our approach useful given only partial knowledge. Results show that our approach can adapt timely to workload changes and can decrease the cost compared to typical static selection policies.

Our approach can achieve even greater efficiency, if it periodically replaces the already running VMs with more suitable ones in terms of cost and performance, once there is a workload change. We will also work on new load balancing policies, which take into account the actual VM capacities. Another promising avenue is optimising the scaling down mechanisms — i.e. selecting which VMs to terminate when the load decreases. Also, we plan to extend our approach, which currently optimises cost, to also consider other factors like energy efficiency. This would be important when executing application servers in private clouds. Finally, we plan to incorporate in our algorithms historical data about VM types' resource capacity and workload characteristics.

Acknowledgments

We thank Rodrigo Calheiros, Amir Vahid Dastjerdi, Adel Nadjaran Toosi, and Simone Romano for their comments on improving this work. We also thank Amazon.com, Inc for their support through the AWS in Education Research Grant.

References

- [1] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003.
- [2] A. O. Ramirez, “Three-Tier Architecture,” *Linux Journal*, vol. 2000, no. 75, Jul. 2000.
- [3] A. Aarsten, D. Brugali, and G. Menga, “Patterns for three-tier client/server applications,” in *Proceedings of Pattern Languages of Programs (PLoP '96)*, 1996.
- [4] E. Brewer, “Towards Robust Distributed Systems,” in *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, vol. 19. New York, NY, US: ACM, jul 2000, pp. 7–10.
- [5] ———, “CAP Twelve Years Later: How the “Rules” Have Changed,” *Computer*, vol. 45, no. 2, p. 23, 2012.
- [6] R. Cattell, “Scalable SQL and NoSQL Data Stores,” *SIGMOD Record*, vol. 39, no. 4, pp. 12–27, may 2010.

- [7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s globally distributed database,” *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.
- [8] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, “Automated anomaly detection and performance modeling of enterprise applications,” *ACM Transactions on Computer Systems*, vol. 27, no. 3, pp. 1–32, Nov. 2009.
- [9] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, “Modeling Virtual Machine Performance: Challenges and Approaches,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 3, pp. 55–60, Jan. 2010.
- [10] J. Dejun, G. Pierre, and C.-H. Chi, “EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications,” in *Proceedings of the International Conference on Service-Oriented Computing (ICSO 2009)*, ser. ICSOC/ServiceWave’09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 197–207.
- [11] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, “Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance,” *The Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 460–471, Sep. 2010.
- [12] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, “Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers,” *Future Generation Computer Systems*, vol. 28, no. 2, pp. 358–367, 2012.
- [13] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, “Auto-scaling Techniques for Elastic Applications in Cloud Environments,” Department of Computer Architecture and Technology, University of the Basque Country, Tech. Rep. EHU-KAT-IK-09-12, 2012.
- [14] Amazon. (2016, Jan. 14) Amazon Auto Scaling. [Online]. Available: <http://aws.amazon.com/autoscaling/>
- [15] RightScale. (2016, Jan. 14) RightScale. [Online]. Available: <http://www.rightscale.com/>
- [16] T. Chieu, A. Mohindra, A. Karve, and A. Segal, “Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment,” in *Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE 2009)*. IEEE, oct. 2009, pp. 281–286.

- [17] T. Chieu, A. Mohindra, and A. Karve, “Scalability and Performance of Web Applications in a Compute Cloud,” in *Proceedings of the IEEE International Conference on e-Business Engineering*, 2011, pp. 317–323.
- [18] B. Simmons, H. Ghanbari, M. Litoiu, and G. Iszlai, “Managing a saas application in the cloud using paas policy sets and a strategy-tree,” in *Proceedings of the 7th International Conference on Network and Services Management*, ser. CNSM ’11. Laxenburg, Austria, Austria: International Federation for Information Processing, 2011, pp. 343–347.
- [19] E. Barrett, E. Howley, and J. Duggan, “Applying reinforcement learning towards automating resource allocation and application scalability in the cloud,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.
- [20] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, “Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: Towards a Fully Automated Workflow,” in *Proceedings of the 7th International Conference on Autonomic and Autonomous Systems (ICAS 2011)*, May 2011, pp. 67–74.
- [21] A. Ali-Eldin, J. Tordsson, and E. Elmroth, “An adaptive hybrid elasticity controller for cloud infrastructures,” in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, April 2012, pp. 204–212.
- [22] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, “Adaptive, Model-driven Autoscaling for Cloud Applications,” in *11th International Conference on Autonomic Computing, ICAC ’14*, 2014, pp. 57–64.
- [23] H. Fernandez et al., “Autoscaling Web Applications in Heterogeneous Cloud Infrastructures,” in *Proc. of the IEEE International Conference on Cloud Engineering*, March 2014, pp. 195–204.
- [24] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, “Autonomic Mix-aware Provisioning for Non-stationary Data Center Workloads,” in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC ’10. New York, NY, USA: ACM, 2010, pp. 21–30.
- [25] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, “Prepare: Predictive performance anomaly prevention for virtualized cloud systems,” in *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS)*, June 2012, pp. 285–294.
- [26] M. Moreira and E. Fiesler, “Neural networks with adaptive learning rate and momentum terms,” IDIAP, Martigny, Switzerland, Tech. Rep. 95-04, October 1995.
- [27] T. Vogl, J. Mangis, A. Rigler, W. Zink, and D. Alkon, “Accelerating the convergence of the back-propagation method,” *Biological Cybernetics*, vol. 59, no. 4-5, pp. 257–263, 1988.

- [28] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, “Performance implications of multi-tier application deployments on infrastructure-as-a-service clouds: Towards performance modeling,” *Future Generation Computer Systems*, vol. 29, no. 5, pp. 1254–1264, 2013.
- [29] N. Grozev and R. Buyya, “Performance Modelling and Simulation of Three-Tier Applications in Cloud and Multi-Cloud Environments,” *The Computer Journal*, 2013.
- [30] J. Hawkins, S. Ahmad, and D. Dubinsky, “Hierarchical Temporal Memory including HTM Cortical Learning Algorithm,” Numenta Inc, Tech. Rep., Sep 2011.
- [31] Grok. (2016, Jan. 13) Grok. [Online]. Available: <https://www.groksolutions.com/product.html>
- [32] Numenta. (2014, Feb. 13) Numenta. [Online]. Available: <http://numenta.org/>
- [33] J. Hawkins and S. Blakeslee, *On Intelligence*. New York, USA: Times Books, 2004.
- [34] V. Mountcastle, “An organizing principle for cerebral function: the unit model and the distributed system,” in *The Mindful Brain*, G. Edelman and V. Mountcastle, Eds. Cambridge, MA, US: MIT Press, 1978.
- [35] Numenta. (2014, Feb. 13) Numenta Platform for Intelligent Computing (NuPIC). [Online]. Available: <http://numenta.org/nupic.html>
- [36] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, “CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms,” *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, January 2011.
- [37] CloudSuite. (2016, Jan. 14) CloudSuite’s CloudStone. [Online]. Available: <http://parsa.epfl.ch/cloudsuite/web.html>
- [38] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson, “Cloudstone: MultiPlatform, Multi-Language Benchmark and Measurement Tools for Web 2.0,” in *Proceedings of Cloud Computing and Its Applications (CCA ’08)*, ser. CCA ’08, 2008.
- [39] JClouds. (2016, Jan. 14) JClouds. [Online]. Available: <http://www.jclouds.org/>
- [40] FANN. (2016, Jan. 13) FANN. [Online]. Available: <http://leenissen.dk/fann/wp/>