Towards an Adaptive, Fully Automated Performance Modeling Methodology for Cloud Applications

Ioannis Giannakopoulos*, Dimitrios Tsoumakos§ and Nectarios Koziris*
* *National Technical University of Athens, Greece, {ggian, nkoziris}@cslab.ece.ntua.gr*
§ *Ionian University, Greece, dtsouma@ionio.gr*

***Abstract*—The advent of the Cloud computing era along with the wide adoption of the distributed paradigm has enabled applications to increase their performance standards and greatly extend their scalability limits. Nevertheless, the ability of modern applications to be deployed in numerous different ways has complicated their structure and enormously increased the difficulty of extracting accurate performance models for them. This capability is crucial for many operations, such as the identification of the most appropriate execution setups for an anticipated workload, finding bottlenecks, etc. In this work, we propose a fully automated performance modeling methodology that aims at the creation of highly accurate performance models for a given maximum number of deployments. The main idea of the proposed methodology lies on the "smart" exploration of the application configuration space, the selection and deployment of a set of representative application configurations and the construction of the performance model through the adoption of Machine Learning techniques. Moreover, taking into consideration the often unstable and error-prone nature of the cloud, the proposed system attempts to overcome transient cloud failures that occur during the application deployment phase through the re-execution of the parts that failed. Our evaluation, conducted for a set of real-world applications frequently deployed over cloud infrastructures, indicates that our system is capable of both constructing performance models of high accuracy and doing so in an efficient manner, fixing application deployments that present errors without requiring human intervention.**

I. Introduction

Performance modeling, i.e., the ability of predicting the performance of an application given a set of parameters that affect it, is a well researched problem [1], [2] that dates back to the era of the first computer programs. A performance model encapsulates the knowledge of the application's behavior under different circumstances, something that is crucial during its lifetime. Identifying the optimal setups [3], recognizing the bottlenecks [4], predicting the impact of a cloud elasticity action [5] and facilitating the choice of a suitable engine in multi-engine environments [6] are some of the many use cases in which the utilization of a performance model is of key importance.

However, the wide adoption of the cloud computing paradigm, as indicated by the ever increasing number of applications that migrate to the cloud [7], in combination with the extensive utilization of distributed application architectures, in order for them to fully utilize the merits of the Cloud (e.g., seemingly infinite scalability, fault-tolerance, etc.), has lead to significantly more complex application configuration spaces. As a consequence, the difficulty of estimating accurate performance models within realistic time and cost constraints has greatly increased. The ability of a cloud user to programmatically determine the exact application configuration in great detail (e.g., how many nodes should be employed for each application component, how many cores, memory, disk, etc.) results in configuration spaces of high dimensionality and an enormous amount of possible configurations, much harder to be modeled.

In order to tackle this challenge, several approaches have been suggested which fall in two categories: the *analytical* or "white-box" and "black-box" approaches. Approaches of the former category assume a well-known application structure and target to model it in an analytical way. The analyst needs to know the data flow, which components are used and under which circumstances in order to identify which are the dominant application parameters. Such an approach is presented in [8] for Hadoop whereas in [9] the authors focus on the memcached cache. Albeit such "white-box" approaches can be extremely useful and accurate for simpler application structures, their accuracy is proportional to the level of detail employed during the application analysis: The more complex the application, the more difficult it becomes to capture all the parameters that affect its performance.

On the contrary, "black-box" approaches make no assumption on application structure and view the modeling problem as a function approximation problem. The main idea behind them lies on the generation of a multidimensional configuration space, the points of which represent the possible configurations of the application, and view the performance model as a function that projects this space to a performance metric. In order to construct the performance model, one needs to *sample* the configuration space and extract a subspace of configurations, deploy the application for each of them and *model* the untested configurations using statistical and Machine Learning techniques. Such an approach is presented in [10] for capacity planning, whereas in [11], [12], similar black-box approaches are presented using different Machine Learning models.

Despite being considered more generic than their "white-box" counterparts, "black-box" approaches pose some limitations that hinder their efficiency for applications with complex structures. First, massive configuration spaces with a large number of parameters each of which may receive multiple values, require an increasing number of samples in order to be approximated with satisfying accuracy, something that entails enormous computation power, time and cost. Although the selection policy of the configurations, i.e., *which* configurations should be chosen for deployment, greatly impacts the accuracy of the final model and it could greatly help with reducing the number of deployments, this dimension is not thoroughly investigated in the literature.

Second, any black-box algorithm requires the fully automated deployment of the selected configurations, something that, in practice, may prove more difficult than anticipated. For a wealth of reasons that include, but are not limited to, the complexity of the cloud software and hardware stack, the big number of components that need to cooperate in order to assist in cloud services (e.g., identity, compute, networking services), unexpected events (e.g., power outages [13]), cloud infrastructures often tend to present *transient* errors that threaten application execution and vanish after a short period of time. Such errors may be tolerable during the application's lifetime, e.g., if a DNS request fails, chances are that a second DNS request will succeed. However, during the application deployment phase, which requires the coordination between multiple components, such errors may lead a deployment to failure, requiring manual intervention for fixing it or dropping it in order to avoid stale resources.

In order to tackle the aforementioned challenges, in this work we propose a system that takes a holistic view in the performance modeling problem, addressing both problems in a unified way. In order to increase modeling accuracy, we propose an adaptive performance modeling algorithm that relies on partitioning the configuration space, based on the approximated performance, and focusing on the partitions the behavior of which is harder to be modeled. Through the adoption of the mechanics of Decision Trees [14], the proposed methodology partitions the configuration space, obtains sample configurations from each region according to their size and accuracy and constructs a model as a composition of linear models for different regions of the configuration space. Intuitively, the suggested methodology is an attempt to adaptively "zoom-in" to regions of the configuration space that present complex patterns, without overlooking to equally explore the configuration space.

In order to tackle the second challenge, we integrate the adaptive performance modeling algorithm with AURA [15], a system that automates application deployment with the extra feature of overcoming transient cloud failures during the deployment phase. AURA analyzes the application description and serializes it into a dependency directed acyclic graph (DAG). A deployment effectively means the traversal of this DAG. In case of failure, AURA isolates the DAG components (scripts) that failed and re-executes them, employing a filesystem snapshot mechanism to guarantee that the deployment scripts always have the same effects and, hence, can be re-executed as many times as needed.

The concrete contributions of this work can be, thus, summarized as follows:

- We propose an adaptive, accuracy-driven performance modeling methodology that relies on recursively partitioning the application configuration space, sampling each region according to its approximation error and space coverage and generalizing its findings using Decision Trees,
- We integrate our adaptive algorithm with AURA, a system used to automate the cloud deployment process with error-recovery enhancements,
- We conduct a thorough experimental evaluation using popular, real world applications and complex configuration spaces of varying dimensionality.

Our extensive evaluation showcases that the suggested methodology outperforms other, state-of-the-art methodologies, as it generates performance models up to $3\times$ more accurate in the best case, also achieving to ignore configuration parameters with extremely low importance. Moreover, the suggested algorithm is proven to be extensible as it can successfully consider deployment parameters such as the monetary cost of the deployments, reducing the budget of the modeling process, introducing an affordable modeling error. Finally, the combination of our modeling methodology with AURA renders our system capable of deploying applications in the most unstable environments.

II. Problem Formulation

In this section we provide a mathematical description of the problem that this work tackles, also providing the necessary notation followed throughout the paper.

Assume an application A, affected by a set of parameters denoted with d_1, d_2, $\cdots$, d_n and producing an output that represents a performance metric P. Each d_i, $1 \leq i \leq n$ will be referred to as the application's *dimension* i. Each dimension may receive values within a pre-defined set, i.e., $d_i \in \{c_1, c_2, \cdots, c_m\}$. For example, if d_1 refers to the number of nodes of a Hadoop cluster, then the set $d_1 \in \{1, \cdots, 20\}$ means that this Hadoop cluster enumerates 1 up to 20 nodes. The Cartesian product of all d_i, $1 \leq i \leq n$ produces an upper bound of the applicaton *Deployment Space* D, i.e., $D = d_1 \times d_2 \times \cdots \times d_n$. Note that, $|D| = \prod_{i=1}^{n} |d_i|$ which means that an increase to the number of dimensions leads to an exponential increase to D's configurations.

We also define the performance function F of an application as a projection from D to P, i.e., $F : D \rightarrow P$. Moreover, using a subset $D_s \subseteq D$, we deploy A for all $c \in D_s$ and based on the obtained performance values, we construct an *approximate* function $F' : D \rightarrow P$, that approximates the original function F. The construction of F'

occurs through applying regression techniques (e.g., linear regression), based on the performance points obtained from the deployment of the application for all $c \in D_s$. Note that, the estimation of F is prohibitive as it entails the deployment of the entirety of D. Using this notation, the problem that this paper attempts to tackle is the following: *Given a maximum number of deployments B, find a set $D_s \subseteq D$, $|D_s| = B$, such that the trained function F' produced using D_s best approximates the original performance function F.* Intuitively, we seek for a set of *representative* configurations that provide a good overview of the unknown F function, in order to train a Machine Learning model that accurately approximates it.

The optimal solution of the problem is NP-Hard, as one should exhaustively calculate all the possible D's subsets of size B and identify the one that generates a performance model F' with the lowest approximation error. Note that, even if such calculation were possible, one should first obtain the performance values of all D points, something that entails prohibitive time and cost, as this procedure requires the deployment of the application to a cloud infrastructure, and the execution of a workflow for each possible combination of parameters. This exhaustive approach is cumbersome, especially when the application structure is complex and D comprises many dimensions, each with multiple values.

It should be noted that this problem formulation implies that the deployment of a given configuration always provides the same performance metrics, i.e., the deployments are *reproducible*. For this to apply, the following precondition must be met: The cloud infrastructure in which the application is deployed to must provide resources with consistent behavior, i.e., there should exist no interference and performance degradations due to the noisy-neighbor effect [16]. Albeit this assumption may sound too rigid, in practice it means that if the cloud provider respects the offered SLAs, the interference factor generates a minimal noise to the model without seriously affecting its accuracy.

Finally, we should note that the previous description makes no assumption regarding the nature of the input application dimensions. In general, the dimensions can fall under three categories: (a) resource-related dimensions (e.g., number of nodes of a cluster, number of cores, amount of RAM, etc.), (b) workload-related dimensions (e.g., dataset size, request throughput, etc.), (c) application-level dimensions (e.g., amount of cache of a DBMS, replication of HDFS, etc.). The definition of the dimensions is conducted by the user who must define which dimensions should be profiled and which are the possible values for each one. Note that this does not violate the "black-box" principle: The user is invited to define as many dimensions as one desires, and our system is responsible for filtering out the unimportant ones, focusing only on the impactful dimensions.

III. Adaptive Performance Modeling Methodology

A. Methodology overview

As mentioned before, the problem of identifying a *representative* set of application configurations $D_s \subseteq D$ is NP-Hard. Moreover, it requires the deployment of the application for the entire configuration space, something that makes it impractical both in terms of time and monetary cost. However, taking into consideration the nature of the applications that are typically deployed over cloud platforms, we make two crucial observations, through which we propose an efficient heuristic algorithm that focuses on dynamically constructing D_s, avoiding unnecessary deployments.

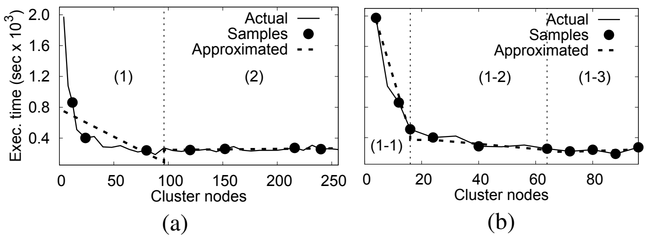


Figure 1. Algorithm execution for the Wordcount examplea

First, by virtue of their design, distributed applications are largely affected by resource-related dimensions (e.g., number of nodes, number of cores, etc.) that tend to impact their performance on a linear or partially linear manner. For example, take the Wordcount operator executed on a Hadoop cluster of fixed resources (cores/node, dataset size, etc.) and assume that one executes the operator changing only the number of Hadoop slaves. Wordcount's execution time, is presented in Figure 1 (a) by the *Actual line*. Note that, the operator clearly presents two regions: Region (1) in which a cluster size increase results in a linear time decrease and Region (2) in which the operator has reached its scalability limits and any further cluster increase results in no time reduction. One could easily approximate this performance function using a partially linear function. Second, the previous example also highlights that the input dimensions do not have a monotonic impact on the application performance throughout the entire D, something that effectively means that the same dimensions may exhibit completely different behavior to different regions of D.

With these two observations under consideration, we now propose the adaptive performance modeling methodology, presented in Algorithm 1. Exploiting the previous observations, the algorithm attempts to iteratively partition the deployment space D to smaller regions, distribute the deployment budget B (expressed as the number of allowed deployments) to each region separately (consuming b samples at each iteration as discussed in the following sections), according to the behavior of the predicted performance function and the region's size and, finally, deploy the selected configurations. Finally, based on the obtained samples, the algorithm constructs an approximation of the performance function and returns it to the user. In essence, the algorithm initially attempts to explore the Deployment Space, obtaining a first "feel" of the performance function and then adaptively focuses on or "zooms-in" parts of the space where the performance function remains obscure.

Algorithm 1 DT-based Adaptive Profiling Algorithm

```
1: procedure DTADAPTIVE($D$, $B$, $b$)
2:     $tree \leftarrow$ TREEINIT($\emptyset$), $samples \leftarrow \emptyset$
3:     while $|samples| \leq B$ do
4:         $tree \leftarrow$ PARTITION($tree$, $samples$)
5:         $s \leftarrow$ SAMPLE($D$, $tree$, $samples$, $b$)
6:         $d \leftarrow$ DEPLOY($s$)
7:         $samples \leftarrow samples \cup d$
8:     $model \leftarrow$ CREATEMODEL($samples$)
9:     return $model$
```

Note that, the output of the algorithm is the approximate function (F'), which can be used by the user for many purposes. For example, if one wants to identify the optimal execution application configuration, one has to search for the point $c = argmin_x F'(x)$, assuming F' expresses the execution time (and, hence, lower is better). If one wants to consult F' in order to predict the application performance for a given configuration c, one has to call $F'(c)$, which is pre-calculated and instantly returned to the user. We now discuss the algorithm's steps in detail.

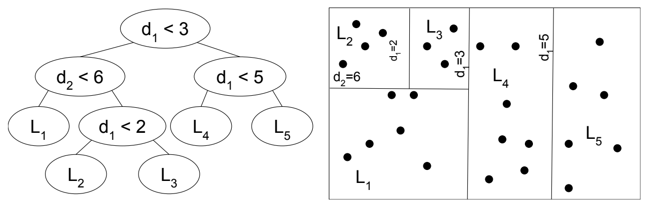
B. Deployment Space Partitioning

The two observations regarding the partial linearity of the performance function and the space locality, lead us to consider a recursive space partitioning scheme where the space is adaptively partitioned in smaller regions, according to the behavior of the performance function. A popular data structure with this property is the *Classification and Regression Tree* [14], or Decision Tree (DT). DTs are tree structures that consist of two nodes: The intermediate (or *test*) nodes that represent a boundary of the space and the terminal (or *leaf*) nodes that represent a region of the space along with the label of a class (classification) or a linear model (regression). An example of a DT can be found in Figure 2. Figure 2 (a) depicts the tree structure and Figure 2 (b) demonstrates the space partitioning this tree represents. In our example, we assume a 2-dimensional Deployment Space (d_1 being the horizontal and d_2 the vertical axis) and the dots represent the chosen samples.

The problem of space partitioning with a DT can be reduced to constructing a DT from scratch. The main idea behind the construction algorithm lies on recursively partitioning the DT's leaves, through the identification of a new boundary line with respect to maximizing the *homogeneity* of the samples that belong to the same newly-created leaf and, minimizing the homogeneity between the samples that belong to different leaves, accordingly. The samples' homogeneity can be measured in many different ways: When the DT is used for classification, Gini Impurity [14], Entropy and Information Gain [17] are extensively used, whereas the Variance Reduction [14] metric is used for regression. In order to take into account the previous observation regarding the (partially) linear impact of many resource related dimensions to the application performance, we design a new homogeneity metric, presented in Equation 1. Specifically, our metric expresses whether a line can separate a set of samples (within a DT leaf) in two new sets (leaves) so that the two new sets fit into linear models, constructed one for each new leaf, accurately enough. This is expressed as follows:

$$Score(line) = \frac{|L_1|R_1^2 + |L_2|R_2^2}{|L_1| + |L_2|} \quad (1)$$

where $L_{\{1,2\}}$ represent the sets generated by a set of samples if $line$ is used for partitioning and $R^2_{\{1,2\}}$ represent the coefficients of determination [18] for each of the two sets which, in turn, reflects the suitability of a linear model for each newly generated leaf. R^2 receives values in the interval $[0, 1]$, in which 1 indicates total linear fitness and 0 indicates that the samples cannot be represented by a linear model. From our experimental evaluation, we conclud that the adoption of Equation 1 against other popular homogeneity metrics contributed to the construction of a more accurate model (experiment omitted due to space constraints).



(a) (b)

Figure 2. Example of a Decision Tree

Given the above, we now provide the algorithm that is responsible to partition the tree's leaves at each iteration. Algorithm 2 iterates over the leaves of the tree. For each leaf, all the possible new split lines are calculated (iterating all the values of all the dimensions of the leaf) and the samples that belong to the specified leaf ($samples \cap l$) are partitioned using the $split$ function. For the two new sets L_1 and L_2, we estimate the linear regression model that best summarizes them (using the Ordinary Least Squares (OLS) method [19]) and the respective R^2 values. The respective score of each is calculated. Finally, when the line with the maximum score is found, a new $testNode$ is generated and replaces the former leaf node (l) of the tree. When this process is repeated for each leaf separately, the new tree is returned.

In order to better illustrate the functionality of Algorithm 2, consider the example depicted in Figure 1 (a). In this one-dimensional Deployment Space (depicted in the horizontal axis), the partitioning algorithm tries all the (vertical) lines that pass from each value of the horizontal axis and finds a line that maximizes Equation 1 for the value *Cluster nodes*

= 96 and it constructs a tree with two leaves: Regions (1) and (2) of Figure 1 (a). Region (2) is extremely accurately approximated by a linear model whereas Region (1) presents higher residuals, i.e., lower R^2 values. The chosen line maximizes Equation 1, as it generates a large (containing 4 samples) region (2) with almost perfect linear fit ($R^2_{(2)} \rightarrow 1$) and a shorter (3 samples) region (1) with non-linear behavior. One dimension not stressed in this one-dimensional example, is the capability of the proposed algorithm to prioritize D's dimensions: Through selecting the most suitable dimensions for partitioning D, the proposed algorithm prioritizes them, only focusing on the ones with the highest impact, ignoring dimensions that present low importance.

Algorithm 2 Deployment Space Partitioning Algorithm

```
1: procedure PARTITION($tree$, $samples$)
2:   $newTree \leftarrow tree$
3:   for $l \in leaves(tree)$ do
4:     $L'_1, L'_2 \leftarrow \emptyset$
5:     $maxScore \leftarrow 0$
6:     for $d \in dimensions(l)$ do
7:       for $v \in values(d)$ do
8:         $L_1, L_2 \leftarrow split(samples \cap l, d, v)$
9:         $R_1, R_2 \leftarrow OLS(L_1), OLS(L_2)$
10:        $score \leftarrow \frac{|L_1|R_1^2 + |L_2|R_2^2}{|L_1| + |L_2|}$
11:        if $score > maxScore$ then
12:          $L'_1, L'_2 \leftarrow L_1, L_2$
13:          $maxScore \leftarrow score$
14:    $testNode \leftarrow \{L'_1, L'_2\}$
15:    $newTree.replace(l, testNode)$
16:  return $newTree$
```

C. Adaptive Sampling

After partitioning the Deployment Space, the next step is to sample the generated regions. At each algorithm iteration, b samples need to be distributed to the existing tree leaves. Note the difference between B and b in our Algorithm: B represents the *total* deployment budget, i.e., the maximum number of deployments conducted by the algorithm, whereas b represents the per-iteration number of deployments. The sample distribution policy is influenced by two factors: (a) The error of the generated linear model for each leaf and (b) the size of the leaf. Specifically, as shown in Algorithm 3, the sampling algorithm iterates over the leaves of the tree (Line 3). For the samples of each leaf, a new linear regression model is calculated (Line 5) and its residuals are estimated using Cross Validation [20]: The higher the residuals, the worse the fit of the points to the linear model. The size of the specified leaf is then estimated. After storing both the error and the size of each leaf into a map, the maximum leaf error and size are calculated (Lines 8-11). Subsequently, a score is estimated for each leaf (Lines 13-15). The score of each leaf is set to be proportional to its scaled size and error. This normalization is conducted so as to guarantee that the impact of the two factors is equivalent. Two coefficients w_{error} and w_{size} are used to assign different weights to each measure. These scores are accumulated and used to proportionally distribute b to each leaf (Lines 16-19). In that loop, the number of deployments of the specified leaf is calculated and new samples from the subregion of the Deployment Space are randomly drawn with the *RANDOMSELECT* function, in a uniform manner. Finally, the new samples set is returned.

Algorithm 3 Sampling algorithm

```
1: procedure SAMPLE($D$, $tree$, $samples$, $b$)
2:   $errors, sizes \leftarrow \emptyset$, $maxError, maxSize \leftarrow 0$
3:   for $l \in$ leaves($tree$) do
4:     $points \leftarrow \{d | d \in samples, d.in \in l\}$
5:     $m \leftarrow$ regression($points$)
6:     $errors[l] \leftarrow$ crossValidation($m$,$points$)
7:     $sizes[l] \leftarrow |\{e | e \in D \cap l\}|$
8:     if $maxError \leq errors[l]$ then
9:       $maxError \leftarrow errors[l]$
10:    if $maxSize \leq sizes[l]$ then
11:      $maxSize \leftarrow sizes[l]$
12:  $scores, newSamples \leftarrow \emptyset$, $sumScores \leftarrow 0$
13:  for $l \in$ leaves($tree$) do
14:    $scores[l] \leftarrow w_{error} \cdot \frac{errors[l]}{maxError} + w_{size} \cdot \frac{sizes[l]}{maxSize}$
15:    $sumScores \leftarrow sumScores + scores[l]$
16:  for $l \in$ leaves($tree$) do
17:    $leafNoDeps \leftarrow \lceil \frac{scores[l]}{sumScores} \cdot b \rceil$
18:    $s \leftarrow$ RANDOMSELECT($\{d | d \in D \cap l\}$, $leafNoDeps$)
19:    $newSamples \leftarrow newSamples \cup s$
20:  return $newSamples$
```

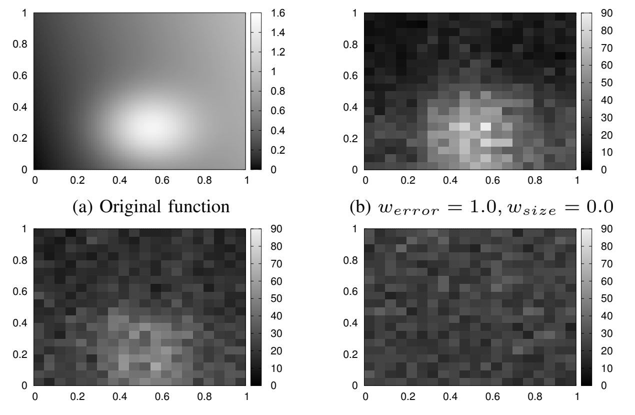

(a) Original function
(b) $w_{error} = 1.0, w_{size} = 0.0$
(c) $w_{error} = 1.0, w_{size} = 0.5$
(d) $w_{error} = 0.0, w_{size} = 1.0$

Figure 3. Sample distribution for different weights

To further analyze on the sampler's functionality, we provide an example of execution for a linear performance function of the form $y = 0.8x_1 + 0.2x_2$. On the randomly selected point $(0.58, 0.22)$, an abnormality is introduced, modeled by a Gaussian function. In Figure 3 (a), a projection of the performance function is provided. The horizontal and vertical axes represent x_1 and x_2 respectively and the colors represent y values, where the lighter colors demonstrate higher y values. Algorithm 1 is executed for different w_{error}, w_{size} during the SAMPLE step. We assume a maximum number of deployments B of 100 points out of the 2500 available points and a per-iteration number of deployments b of 10 points. In Figures 3 (b), (c) and (d) we provide the distribution of the selected samples, for different weight values. Each dimension is divided in 20 intervals and for each execution we keep count of the samples that appear inside each region. The color of the regions demonstrate the number of the samples within the region (lighter colors imply more samples).

The adaptiveness of the proposed methodology leads the samples to immediately identify the abnormal area. In Figure 3 (b) the score of each leaf is only determined by its error. Most samples are gathered around the Gaussian distribution: The first leaves that represent the area of the Gaussian function produce less accurate models since they cannot express the performance function with a linear model. Since the score of each leaf is only determined by its error, these leaves claim the largest share of b at each step, thus the samples are gathered around the abnormality. On the contrary, when increasing w_{size} as in Figure 3 (c), the gathering of the samples around the abnormality is neutralized as more samples are now distributed along the entire space, something that is intensified in Figure 3 (d), where the abnormality is no longer visible.

The consideration of two factors (error and size) for deciding the number of deployments spent at each leaf targets the trade-off of *exploring* the Deployment Space versus *exploiting* the obtained knowledge, i.e., focus on the abnormalities of the space and allocate more points to further examine them. This is a well-known trade-off in many fields of study [21]. In our approach, one can favor either direction by adjusting the weights of leaf error and size, respectively. Note that this scheme enables the consideration of other parameters as well, such as the deployment cost through the extension of tre score function (Line 14). This way, more "expensive" deployment configurations, e.g., ones that entail multiple VMs with many cores, would be avoided in order to regulate the profiling cost.

D. Modeling

After B samples are returned by the profiling algorithm, they are utilized by the *CREATEMODEL* (Algorithm 1, Line 8) function to train a new DT. The choice of training a new DT instead of expanding the one used during the sampling phase is made to maximize the accuracy of the final model. When the first test nodes of the former DT were created, only a short portion of the samples were available to the profiling algorithm and, hence, the original DT may have initially created inaccurate partitions. Moreover, in cases where the number of obtained samples is comparable to the dimensionality of the Deployment Space, the number of constructed leaves is extremely low and the tree degenerates into a linear model that covers sizeable regions of the Deployment Space with reduced accuracy. To overcome this limitation, along with the final DT, a set of Machine Learning classifiers are also trained, keeping the one that achieves the lowest Cross Validation error. However, when the DT is trained with enough samples, it outperforms all the other classifiers. This is the main reason for choosing the DT as a base model for our scheme: The ability to provide higher expressiveness by composing multiple linear models in areas of higher unpredictability, make them a perfect choice for modeling a performance function. Note that, the linearity of this model does not compromise its expressiveness: Non-linear performance functions can also be accurately approximated by a piecewise linear model, through further partitioning the Deployment Space. For example, one can observe that the non-linear Region (1) of Figure 1 (a), was further analyzed through dedicating more samples to it in the next algorithm step (depicted in Figure 1 (b)). Note that, only Region (1) claimed new samples from the deployment budget: Region (2) needed no more samples as the existing ones indicated that the region is accurately approximated. This leads the proposed algorithm to better focus on the non-linear region and better approximate it through the utilization of more samples and leaves.

IV. System Architecture

A. Overview

Figure 4 provides the architecture of the system that realizes this methodology. The main part of the methodology is executed through the Sampler, the Partitioner and the Modeler components. These components are orchestrated appropriately in order to execute Algorithm 1. When the sampler selects new configurations from D, it issues new deployment requests to the *Deployment* module. The latter is responsible for contacting the cloud provider to instantiate new application instances (*D1 – D4*). Upon the execution of the selected configurations, the results (i.e., performance metric) are pushed back to the *Modeler* module, that updates the performance model accordingly and stores them to an internal database.

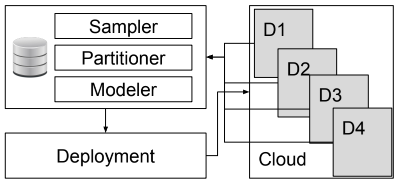


Figure 4. System Architecture

Note that the main components of the modeling system do not directly communicate with the cloud provider, but access it through the *Deployment* module. This way, the modeling system remains agnostic of the underlying infrastructure and the APIs that need to use in order to contact the provider.

B. Deployment

In our work, we utilize AURA [15] in order to deploy applications in an automated way. AURA[1] models an application deployment as a set of scripts, each of which refers to a software component, executed in a particular order. Each script may require input from another script or produce output(s) that need to be forwarded to other software modules, e.g., passwords, IP addresses, SSH keys, etc. This mechanism facilitates the coordination between different software components and enforces specific tasks to be executed in a particular order, e.g., a Web Server must

[1]https://github.com/giagiannis/aura

attempt to connect to a Database Server, only if the latter has successfully finished configuration. Using this model, AURA extracts a Directed Acyclic Graph (DAG) of dependencies between the actions that need to take place: The deployment execution is, essentially, the traversal of the DAG. Figure 5 showcases such a DAG for a Hadoop cluster that comprises one master and two slave nodes.

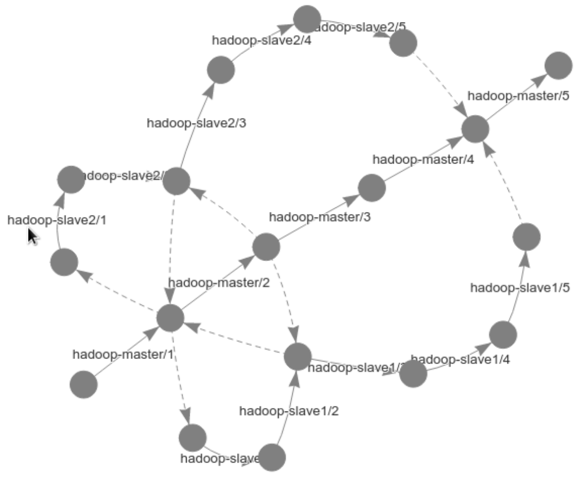


Figure 5. Hadoop deployment DAG

The reason behind utilizing AURA instead of another popular deployment tool, lies on the error-recovery enhancements AURA introduces. Specifically, AURA attempts to overcome transient failures that appear during the deployment phase of an application, through the re-execution of the deployment parts that failed. Furthermore, in order to guarantee that the configuration scripts have the same side effects, it introduces a lightweight filesystem snapshot mechanism that guarantees that the failed scripts do not leave filesystem related resources in an inconsistent state.

Finally, in order to enforce deployment with different characteristics, e.g., different number of VMs, CPUs, amount of RAM, etc., we have implemented a parser that receives a Deployment Space point, i.e., a vector containing the chosen values for each D dimension and modifies the application description provided to AURA in `json` format accordingly. This way, the main modeling system is not closely integrated with the deployment system, as it does not rely on a specific application description. This means that if one wants to utilize a different deployment tool, or even apply the proposed methodology to container-based platforms (orchestrated by Kubernetes [22], Docker Swarm [23], etc.), one does not need to modify the system but should only provide a new application description parser that knows how to translate Deployment Space points to concrete application descriptions.

V. Experimental Evaluation

Experimental Setup: All experiments are conducted on a 8-node private Openstack cluster, each of which contains two Intel Xeon E5645 processors running at 2.40GHz, 96G of main memory and 2TB of hard disk (2 $\times$ 1TB 2.5" SATA HDD in RAID-0), running Ubuntu Server 14.04.2 LTS with Linux kernel 3.13.05. The partitioning, sampling and modeling parts of our prototype is implemented in Java (v.1.8.0_144), whereas the deployment component (AURA) is implemented in Python (v.2.7.13).

Applications: We have deployed four popular real-world Big Data operators/applications which are commonly encountered in cloud installations. The applications along with their dimensions chosen to represent D as well as their domains are summarized in Table I. We opted for applications with diverse characteristics with Deployment Spaces of varying dimensionality (3 – 5 dimensions).

Table I
Applications under profiling

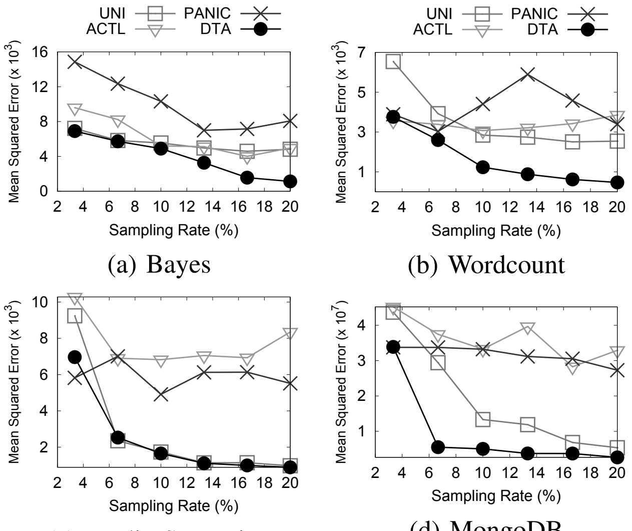
Application (perf. metric)	**Dimensions**	**Values**
Spark Bayes (execution time)	YARN nodes	4–20
	# cores per node	2–8
	memory per node	2–8 GB
	# of documents	0.5–2.5 ($\times 10^6$)
	# of classes	50–200 classes
Hadoop Wordcount (execution time)	YARN nodes	2–20
	# cores per node	2–8
	memory per node	2–8 GB
	dataset size	5–50 GB
Media Streaming (throughput)	# of servers	1–10
	video quality	144p–1440p
	request rate	50–500 req/s
MongoDB (throughput)	# of MongoD	2–10
	# of MongoS	2–10
	request rate	5–75 ($\times 10^3$) req/s

The first two operators are implemented in Spark (Bayes) and Hadoop (Wordcount) and they are deployed to a YARN cluster. In all cases, the performance metric corresponds to the execution time. The Media Streaming application [24] consists of two components: The backend is an NFS server that serves videos to the Web Servers. A number of lightweight Web Servers (nginx) are setup to serve the videos to the clients. The NFS server retains 7 different video qualities and 20 different videos per quality. MongoDB is deployed as a sharded cluster and it is queried using YCSB [25]. The sharded deployment of MongoDB consists of three components: (a) A configuration server that holds the cluster metadata, (b) a set of nodes that store the data (MongoD) and (c) a set of load-balancers that act as endpoints to the clients (MongoS).

Methodology: We compare our profiling methodology against other end-to-end profiling schemes, measuring both modeling accuracy and profiling time. The accuracy of the profiling algorithms is measured using the *Mean Squared Error* (MSE) metric. Our approach is referred to as DT-based Adaptive methodology (*DTA*). Active Learning [26] (*ACTL*) is a Machine Learning field that specializes on exploring a performance space by obtaining samples assuming that finding the class or the output value of the sample is computationally hard. We implemented *Uncertainty Sampling* that prioritizes the points of the Deployment Space with the highest uncertainty, i.e., points for which a Machine Learning model cannot predict their class or continuous value with high confidence. PANIC [27] is an adaptive approach that favors points belonging into steep areas of the performance function, utilizing the assumption that the abnormalities of the performance function characterize it best. Furthermore, since most profiling approaches use a randomized sampling algorithm [10], [28], [11], [12] to sample the Deployment Space and different Machine Learning models to approximate the performance, we implement a profiling scheme where we draw random samples (*UNI*) from the performance functions and approximate them using the models offered by WEKA [29], always keeping the most accurate one. In all but a few cases, the Random Committee [30] algorithm prevailed, which uses Multi-Layer Perceptron as a base classifier.

A. *Modeling Accuracy*

We first compare the four methods against a varying Sampling Rate (SR), i.e., the portion of the Deployment Space utilized for approximating the performance function ($SR = \frac{|D_s|}{|D|} \times 100\%$). SR varies from 3% up to 20% for the tested applications. In Figure 6, we provide the accuracy of each approach measured in terms of MSE.



(a) Bayes (b) Wordcount

(c) Media Streaming (d) MongoDB

Figure 6. Accuracy vs sampling rate (MSE)

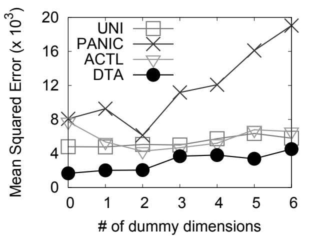
Figure 6 showcases that DTA outperforms all the competitors for increasing SR, something indicative of its ability to distribute the available number of deployments accordingly so as to maximize the modeling accuracy. In more detail, all algorithms benefit from an increase in SR since the error metrics rapidly degrade. In Bayes, when the SR is around 3% UNI and DTA construct models of the highest accuracy. As mentioned in Section III-D, for very low SR the linearity of the DT would fail to accurately represent the relationship between the input and the output dimensions, thus a Random Committee classifier based on Multi-Layer Perceptrons is utilized for the approximation. The same type of classifier also achieves the highest accuracy for the rest of the profiling algorithms (ACTL, PANIC) that also present higher errors due to the less accurate sampling policy at low SR. As SR increases, DT obtains more samples and creates more leaves, which contributes in the creation of more linear models that capture a shorter region of the Deployment Space and, thus, producing higher accuracy. Specifically, for $SR \geq 3\%$, DT created a more accurate prediction than other classifiers and was preferred.

In the rest of the cases, DTA outperforms its competitors for the Wordcount application and, interestingly, this is intensified for increasing SR. Specifically, DTA manages to present $3\times$ smaller modeling error compared to UNI when $SR = 20\%$. Media Streaming, on the other hand, exhibits an entirely different behavior. The selected dimensions affect the performance almost linearly, thus the produced performance function is smooth and easily modeled by less sophisticated algorithms than DTA, explaining UNI's performance which is similar to DTA's. PANIC and ACTL try to identify the abnormalities of the space and fail to produce accurate models. Finally, for the MongoDB case, DTA outperforms the competitors increasingly with SR. In almost all cases, DTA outperforms its competitors and creates models even 3 times more accurate (for Bayes when $SR = 20\%$) from the best competitor. As an endnote, the oscillations in PANIC's and ACTL's behavior are explained by the aggressive exploitation policy they implement. PANIC does not explore the Deployment Space and only follows the steep regions, whereas ACTL retains a similar policy only following the regions of uncertainty, hence the final models may become overfitted in some regions and fail to capture most patterns of the performance function. Our work identifies the necessity of both exploiting the regions of uncertainty but also exploring the entire space. This trade-off is only addressed by DTA and explains its dominance for difficult to approximate applications.

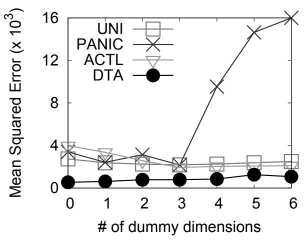
B. *Deployment Space Dimensionality*

A key property of any black-box profiling methodology is its ability to isolate impactful from meaningless dimensions, in order to avoid wasting the deployment budget on unimportant samples. To this end, we design the following experiment: Utilizing the performance functions of the four real-world applications, we introduce a varying number of "dummy" dimensions, i.e., dimensions that have no impact to the output performance. We assume that each new dimension receives two possible values and, thus, the addition of a dummy dimension doubles the number of possible configurations. Using the same *number* of deployments ($B = 300$ for each case) we run our algorithm and compare its accuracy against PANIC, UNI and ACTL. The results for the four applications are provided in Figure 7. Note that, we utilize the *entire* space for testing the accuracy of the approximated models, something which means that the number of test points doubles with the addition of new dimensions.

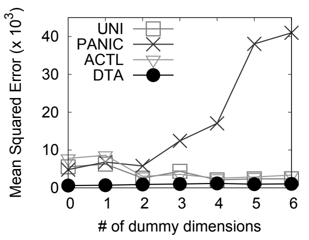
Figure 7 demonstrates that the addition of new unimportant dimensions has, practically, no impact to DTA, UNI and ACTL as all of them present zero MSE increase. DTA, in particular, presents the lowest error for all cases, indicative of its ability to, not only, model the tested applications with higher accuracy, as seen before, but also isolate the most important dimensions without being affected by the unimportant ones. One can observe the importance of this finding: During the application definition, a user need not be aware of the dimensions that have an impact. One can simply provide as many dimensions as needed, letting DTA to decide which are the most impactful ones. Finally, PANIC is the only method that appears to be strongly affected by the space dimensionality, as an increase in the number of dimensions leads to a rapid error growth. This is attributed to the fact that PANIC favors areas of the Deployment Space that present the highest oscillations. The insertion of new dimensions exponentially increases the number of points that belong to these areas, something which means that PANIC dedicates an increasing number of its deployment budget for them, and, hence, poorly approximates the rest of the space.



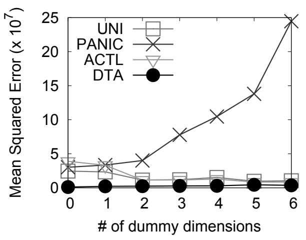
(a) Bayes



(b) Wordcount



(c) Media Streaming



(d) MongoDB

Figure 7. Modeling error vs dimensionality(MSE)

C. Cost-aware Profiling

So far, we have assumed that all Deployment Space points are equivalent, in the sense that we have not examined the deployment configuration they represent: A point representing a deployment configuration of 8 VMs, each of which has 8 cores, is equivalent to a point representing 1 VM with 1 core. However, since the choice of a point results in its actual deployment, it is obvious that this choice implicitly includes a (monetary) cost dimension that has not been addressed. We now examine DTA's ability to adapt when such a cost consideration exists. Let us define the following cost models:

- Bayes: $|\text{nodes}| \times |\text{cores}|$
- Wordcount: $|\text{nodes}| \times |\text{cores}|$
- Media Streaming: $|\text{servers}|$
- MongoDB: $|\text{MongoS}| + |\text{MongoD}|$

We have chosen realistic cost models, expressed as functions of the allocated resources (VMs and cores). Let us recall that Media Streaming and MongoDB utilize unicore VMs. Hence, their cost is only proportional to the number of allocated VMs.

We execute DTA, extending the leaf score function, presented in Algorithm 3 of Section III-C, where the error and size parameters positively influence the score of each leaf and cost is a negative factor, i.e.:

$$\begin{aligned} Score(leaf) = w_{error}\frac{error(leaf)}{maxError} &+ w_{size}\frac{size(leaf)}{maxSize} \\ &- w_{cost}\frac{cost(leaf)}{maxCost} \end{aligned} \tag{2}$$

For $w_{error} = 1.0$ and $w_{size} = 0.5$, we alter w_{cost} between 0.2 and 1.0 for SR of 3% and 20%. We provide our findings in Table II, in which we present the percentage difference in the profiling error (measured in MSE) and cost for each case, against the case of $w_{cost} = 0.0$.

Table II
MSE AND COST FOR DIFFERENT COST WEIGHTS

App/tions	**SR**	**MSE**			**Cost**		
		0.2	**0.5**	**1.0**	**0.2**	**0.5**	**1.0**
Bayes	3%	+1%	-1%	-2%	-1%	-1%	-1%
	20%	+4%	+10%	+5%	-7%	-9%	-12%
Wordcount	3%	-1%	-5%	-1%	-4%	-6%	-1%
	20%	0%	+13%	+19%	-6%	-8%	-18%
Media Str.	3%	-1%	-3%	+3%	-1%	-5%	-6%
	20%	-6%	-2%	-8%	-7%	-11%	-26%
MongoDB	3%	+6%	+12%	+13%	-2%	-3%	-4%
	20%	-1%	-7%	-7%	-6%	-9%	-12%

For low SR , increasing values for w_{cost} does not heavily influence the profiling cost. Specifically, for the MongoDB case, the cost reduces by a marginal factor (around 4% in the most extreme case) whereas the error increases by 13%. On the contrary, for high SR, the consideration of the cost increases its impact as the application profiles are calculated with even 26% less cost than the case of $w_{cost} = 0$, e.g., in the Media Streaming case for $w_{cost} = 1.0$. Furthermore, increasing w_{cost} seems to intensify cost degradation, as the cost becomes a more important factor for the leaf score. Regarding the profiling accuracy, in most cases the error remains the same or its increase does not exceed 10%. A notable exception from this is the Wordcount case, where MSE reaches a growth of 19% in the case where $w_{cost} = 1.0$. From this analysis, we can conclude that cost-aware sampling becomes particularly effective for higher SR values, which is also desirable since high SR entail many deployments, i.e., increased cost. In such cases, the cost-aware algorithm has more leverage to improve the profiling cost whereas, at the same time, the accuracy sacrifice is totally dependent on the nature of the performance function. However, from our evaluation, we conclude that the accuracy degradation is analogous to the cost reduction, allowing the user to choose between higher accuracy or reduced deployment cost.

VI. Related Work

Performance modeling is a vividly researched area. The challenge of accurately predicting the performance of a distributed application is hindered by the virtualization overhead inserted from the cloud software (hypervisors, virtual hardware, shared storage, etc.). To allow the problem decomposition, the proposed solutions are mainly focused on two directions: (a) modeling the performance of the cloud provider itself through cloud benchmarks, and (b) modeling the application performance in an infrastructure-agnostic way. The first approach is mainly focused on executing sets of typical cloud applications such as [31], [24], [32], [33] over different cloud providers and identifying the relationships between the cloud offerings and the respective application performance. Each benchmark is deployed over different clouds with different configurations and the relation between the performance and the deployment setups is obtained. Industrial solutions such as [34], [35], [36] retain statistics and perform the analysis, providing comparisons between different public clouds, identification of the resources' impact to the applications, etc. These results can then be generalized into custom user applications in order to predict their performance into different cloud providers.

The distinct approaches used to model the behavior of a given application can be further graded in two categories: (a) simulation/emulation based and (b) "black-box" approaches. In the first case, the approaches are based on known models of the cloud platforms [37] and enhance them with known performance models of the applications under profiling. CDOSim [38] is an approach that targets to model the Cloud Deployment Options (CDOs) and simulate the cost and performance of an application. CloudAnalyst [39] is a similar work that simulates large distributed applications and studies their performance for different cloud configurations. Finally, WebProphet [40] is a work that specializes in web applications. These works assume that performance models regarding both the infrastructure and the application are known, as opposed to our approach that makes no assumptions neither for the application nor for the infrastructure. CloudProphet [41] is an approach used for migrating an application into the cloud. It collects traces from the application running locally and replays them into the cloud, predicting the performance it should achieve over the cloud infrastructure. //Trace [42] is an approach specializing in predicting the I/O behavior of a parallel application, identifying the causality between I/O patterns among different nodes. In [43] a similar approach is presented, in which a set of benchmark applications are executed in a cloud infrastructure, measuring microarchitecture-independent characteristics and evaluating the relationship between a target and the benchmarked application. According to this relationship, a performance prediction is extracted. Finally, [44] specializes to I/O-bound BigData applications, generating a model of the virtualized storage through microbenchmarking and generalizing it to predict the application performance.

Finally, the "black-box" approaches attempt a statistical approach on the modeling problem, considering the application as a black-box that receives a number of inputs and produces a single output (performance metric) and tries to model the relationship between them, through deploying the application for some configuration combinations. In [10], [28] a generic methodology is proposed used to infer the application performance of based on representative deployments of the configuration space. The approach tackles the problem of generalizing the performance for the entire deployment space, but does not tackle the problem of picking the most appropriate samples from the deployment space, as the suggested approach. PANIC [27] is a similar work, that addresses the problem of picking representative points during sampling. This approach favors the points that belong to the most steep regions of the Deployment Space, based on the idea that these regions characterize most appropriately the entire performance function. However it is too focused on the abnormalities of the Deployment Space and the proposed approach outperforms it. Similarly, the problem of picking representative samples of the Deployment Samples is also addressed by Active Learning [26]. This theoretical model introduces the term of uncertainty for a classifier that, simply put, expresses its confidence to label a specific sample of the Deployment Space. Active Learning favors the regions of the Deployment Space that present the highest uncertainty and, as PANIC, fail to accurately approximate the performance function for the entire space, as also indicated by our experimental evaluation. Finally, in [11] and [12] two more generic black-box approaches are provided, utilizing different machine learning models for the approximation, without considering, though, the problem of minimizing the number of samples.

VII. Conclusions

In this work, we revisited the problem of application performance modeling deployed over cloud infrastructures. The complexity of their structure has radically increased the difficulty of generating an accurate performance model using an affordable number of deployments. We proposed a methodology that utilizes Decision Trees to partition the application configuration space in disjoint regions, sample each one separately based on the approximation accuracy and size. Our approach manages to adaptively "zoom-in" to areas of the configuration space that require more detailed sampling, achieving superior accuracy under a small number of deployments.

References

[1] S. Lavenberg, *Computer performance modeling handbook*. Elsevier, 1983.

[2] M. H. MacDougall, *Simulating computer systems: techniques and tools*. MIT press, 1989.

[3] J. Cao, K. Hwang, K. Li, and A. Y. Zomaya, "Optimal multiserver configuration for profit maximization in cloud computing," *ieee transactions on parallel and distributed systems*, vol. 24, no. 6, pp. 1087–1096, 2013.

[4] C. Roser, M. Nakano, and M. Tanaka, "A practical bottleneck detection method," in *Proceedings of the 33nd conference on Winter simulation*. IEEE Computer Society, 2001, pp. 949–953.

[5] I. Giannakopoulos, N. Papailiou, C. Mantas, I. Konstantinou, D. Tsoumakos, and N. Koziris, "Celar: automated application elasticity platform," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 23–25.

[6] K. Doka, N. Papailiou, V. Giannakouris, D. Tsoumakos, and N. Koziris, "Mix nmatch multi-engine analytics," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 194–203.

[7] "RightScale 2017 State of the Cloud Report," https://www.rightscale.com/lp/2017-state-of-the-cloud-report.

[8] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang, "Hadoop performance modeling for job estimation and resource provisioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 441–454, 2016.

[9] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[10] M. Gonçalves, M. Cunha, N. C. Mendonca, and A. Sampaio, "Performance Inference: A Novel Approach for Planning the Capacity of IaaS Cloud Applications," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015.

[11] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao, "Application performance modeling in a virtualized environment," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010.

[12] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *ACM SIGPLAN Notices*, vol. 47, no. 7. ACM, 2012.

[13] "Google App Engine Incident," https://goo.gl/ICI0Mo.

[14] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[15] I. Giannakopoulos, I. Konstantinou, D. Tsoumakos, and N. Koziris, "Aura: Recovering from transient failures in cloud deployments," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2017, pp. 762–765.

[16] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, 2014.

[17] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, 1986.

[18] S. A. S. Glantz *et al.*, *Primer of applied regression and analysis of variance*, 1990, no. Sirsi) i9780070234079.

[19] C. Dismuke and R. Lindrooth, "Ordinary least squares," *Methods and Designs for Outcomes Research*, vol. 93, 2006.

[20] S. Geisser, *Predictive inference*. CRC press, 1993, vol. 55.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 1998.

[22] "Kubernetes," https://kubernetes.io/.

[23] "Docker Swarm," https://docs.docker.com/engine/swarm/.

[24] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010.

[26] B. Settles, "Active learning literature survey," *University of Wisconsin, Madison*, vol. 52, no. 55-66, 2010.

[27] I. Giannakopoulos, D. Tsoumakos, N. Papailiou, and N. Koziris, "PANIC: Modeling Application Performance over Virtualized Resources," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015.

[28] M. Cunha, N. Mendonça, and A. Sampaio, "Cloud Crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds," *Concurrency and Computation: Practice and Experience*, 2016.

[29] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, 2009.

[30] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1-2, 2010.

[31] "PerfKitBenchmarker," https://goo.gl/b4Xcij.

[32] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *New Frontiers in Information and Software as Services*. Springer, 2011.

[33] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010.

[34] "CloudHarmony," https://cloudharmony.com/.

[35] "CloudSpectator," http://cloudspectator.com/.

[36] "LoadImpact," https://loadimpact.com/.

[37] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, 2011.

[38] F. Fittkau, S. Frey, and W. Hasselbring, "CDOSim: Simulating cloud deployment options for software migration support," in *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*. IEEE, 2012.

[39] B. Wickremasinghe, R. N. Calheiros, and R. Buyya, "Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. IEEE, 2010.

[40] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. G. Greenberg, and Y.-M. Wang, "WebProphet: Automating Performance Prediction for Web Services." in *NSDI*, vol. 10, 2010.

[41] A. Li, X. Zong, S. Kandula, X. Yang, and M. Zhang, "CloudProphet: towards application performance prediction in cloud," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011.

[42] M. P. Mesnier, M. Wachs, R. R. Simbasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. R. O'hallaron, "//Trace: parallel trace replay with approximate causal events." USENIX, 2007.

[43] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006.

[44] I. Mytilinis, D. Tsoumakos, V. Kantere, A. Nanos, and N. Koziris, "I/O Performance Modeling for Big Data Applications over Cloud Infrastructures," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015.