Learning Non-Deterministic Impact Models for Adaptation

Francisco Duarte, Richard Gil, Paolo Romano
INESC-ID,
Instituto Superior Técnico,
Universidade de Lisboa

Antónia Lopes
LASIGE,
Faculdade de Ciências,
Universidade de Lisboa

Luís Rodrigues
INESC-ID,
Instituto Superior Técnico,
Universidade de Lisboa

ABSTRACT

Many adaptive systems react to variations in their environment by changing their configuration. Often, they make the adaptation decisions based on some knowledge about how the reconfiguration actions impact the key performance indicators. However, the outcome of these actions is typically affected by uncertainty. Adaptation actions have non-deterministic impacts, potentially leading to multiple outcomes. When this uncertainty is not captured explicitly in the models that guide adaptation, decisions may turn out ineffective or even harmful to the system. Also critical is the need for these models to be interpretable to the human operators that are accountable for the system. However, accurate impact models for actions that result in non-deterministic outcomes are very difficult to obtain and existing techniques that support the automatic generation of these models, mainly based on machine learning, are limited in the way they learn non-determinism.

In this paper, we propose a method to learn human-readable models that capture non-deterministic impacts explicitly. Additionally, we discuss how to exploit expert's knowledge to bootstrap the adaptation process as well as how to use the learned impacts to revise models defined offline. We motivate our work on the adaptation of applications in the cloud, typically affected by hardware heterogeneity and resource contention. To validate our approach we use a prototype based on the RUBiS auction application.

KEYWORDS

Adaptive systems, Runtime models, Uncertainty, Machine Learning

ACM Reference Format:
Francisco Duarte, Richard Gil, Paolo Romano, Antónia Lopes, and Luís Rodrigues. 2018. Learning Non-Deterministic Impact Models for Adaptation. In *SEAMS '18: SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3194133.3194138

1 INTRODUCTION

Software systems are expected to operate in dynamic environments. As a consequence, systems must accommodate different operational conditions and change their configurations as their contexts change. Since manual operation is complex and expensive, some level of automation is desired. *Autonomic computing* [16, 19] proposes to do so via an external controller that automates the operational loop, divided into four processes— *m*onitoring, *a*nalysis, *p*lanning, and *e*xecution— that make use of the available *k*nowledge about the system; a model known as MAPE-K. Systems capable of enabling such mechanisms are known as *self-adaptive*.

To achieve effective self-adaptation, many systems make decisions based on some knowledge about how adaptation actions may impact their key performance indicators. However, it is in general not possible to know all variables that may affect the outcome of adaptation actions and also impractical to include all known variables in the adaptation models. Hence, action impacts are naturally affected by uncertainty. For instance, abstracting from the variables that affect whether the activation of a server succeeds or fails, potentially leads to two different outcomes for such adaptation action. While this uncertainty can be difficult to define, its effects can hinder the adaptation capabilities of the system. The adaptation process may turn out ineffective (e.g., leading to sub-optimal states that do not meet the system's goals) or altogether harmful (e.g., by inducing the violation of the system constraints).

Also important in the adaptation process is the ability to have human operators in-the-loop [4]. Human experts have potentially a better understanding on the system's behavior and its environment and, therefore, they can help validate the knowledge used to make decisions as well as the policies that derive from them. On top of that, key decisions that potentially compromise critical qualities such as availability, may require an expert's approval before execution, for accountability matters.

Models that are used to drive adaptation should: (1) capture non-determinism explicitly so as to be considered in decision-making and (2) be expressed in a human-readable form. Unfortunately, previous techniques that support the automatic generation of interpretable impact models do not deal with non-determinism. On the one hand, there are manually produced models that are expressive andfl exible enough to account for non-determinism. However, models written by humans are typically incomplete and often inaccurate. These models are also hard to maintain as the system evolves (they change with software updates or changes in the underlying infrastructure). On the other hand, there are ML-based techniques that can create knowledge in a fully automated manner (provided a large training set of data) and maintain the derived models as the system evolves and new conditions arise; yet, they do not capture non-determinism explicitly.

In this paper, we employ ML techniques to derive human-readable models that capture non-deterministic impacts explicitly. We leverage on languages that capture *adaptation strategies* with probabilistic *impact models* [3, 5]. We contribute with a method to Learn

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194138

Adaptation Models Under Non-determinism (Lamun) that is able to extract piece-wise linear functions from collected data at runtime, using the K-plane algorithm [1]. Additionally, we propose to use the learned functions to revise existing models and to exploit user-defined models to accelerate the learning process.

We motivate our work on the adaptation of web applications in public cloud environments. Cloud computing is known to be affected by performance variability due to: (1) hardware heterogeneity [23], i.e. a virtual instance's performance depends on the hardware features of the machine where it is deployed, and (2) resource contention [8], i.e. a virtual instance's performance is affected by co-located instances that compete for resources. These factors are external to the application and cause scaling actions to have non-deterministic impacts.

To validate our approach, we perform elastic scaling actions using the RUBiS auction application deployed in a local cloud. After data collection, we compare a model derived with Lamun with one derived with a decision tree learner that does not capture uncertainty explicitly. We evaluate the accuracy, expressiveness, and resilience to noise of the model learned by Lamun.

The rest of this paper is organized as follows. Section 2 motivates adaptation under uncertainty in cloud computing scenarios. Section 3 discusses the limitations of techniques proposed in the literature. We describe how Lamun learns new impact models in Section 4. We describe how to revise existing models with Lamun in Section 5 and how to bootstrap the learning process departing from user-defined models in Section 6. Section 7 discusses the evaluation settings and results. We conclude in Section 8.

2 MOTIVATION

Our research is motivated by a realistic adaptation scenario where scaling actions of web applications are affected by uncertainty in cloud computing.

2.1 Uncertainty in Cloud Computing

Cloud computing has enabled many software applications to dynamically accommodate their resources in response to variations in their workload. Public clouds providers offer a wide variety of *virtual machines* (VMs) types, with distinct processing, memory, storage capacities, and prices. Thus, applications can accommodate their resources by: (1) activating a server, which increases the capacity but incurs additional costs, or (2) terminating a server, which decreases the operational costs but reduces the capacity to handle the load. The goal of these applications is to meet some quality requirements (e.g. keeping the response time experienced by the users low), while minimizing the operational costs.

While applications can decide which VM type to add or remove from their server pools, the underlying physical infrastructure remains hidden. Thus, applications have little control over 'where' their VMs are deployed. This is relevant because cloud services are known to suffer from significant performance variability that is caused by infrastructure-related conditions, such as hardware restrictions and co-residency. Due to the unobservability and uncontrollabilty of these factors, applications can hardly predict the behavior of newly deployed VMs, thus complicating decision-making for scaling resources; a case of *planning under uncertainty*.

The problem of uncertainty in the cloud is widely reported in the literature, e.g. [8, 23]. Performance variability and unpredictability in cloud applications is mostly caused by:

- *Hardware heterogeneity*, which is a primary source of performance variability in CPU-bound applications, where performance data often shows a multi-modal distribution. That is, while the overall variability is large, there are clusters of similarly performing machines that correspond to the hardware types. Heterogeneity remains relevant as the infrastructure evolves and new hardware is installed.
- *Resource contention*, which result in performance variability when VMs placed in the same physical machine compete for limited resources (e.g. CPU). For instance, substantial variability is experienced in individual instances of IO-bound applications due to bandwidth interference. Ergo, there are slow and fast, as well as stable and unstable instances.
- *Failures* that are ubiquitous and may result in performance unpredictability. Failures can be caused by hardware deterioration, software updates, and overall malfunctioning. Faulty machines can also lead to unexpected outcomes. Moreover, failures can affect adaptation actions themselves (e.g. a new VMs can fail to be launched).

These factors do not affect cloud-enabled applications uniformly. Instead, their impact depends on the application type (IO vs CPU bound), the virtual machine type (small or large, general-purpose or not) and the provider's infrastructure (e.g. Google's GCE uses 5 hardware types to deploy $n1$ series machines[1]).

2.2 Adaptation Scenario

Capturing the complexity of such uncertainty is necessary for the system to make effective adaptation. To illustrate this, consider a content-serving web application (CPU-bound) running on a cloud infrastructure with hardware heterogeneity (e.g. Amazon EC2[2]). The application has an initial resource pool composed of a large server L (4 CPUs). In this configuration, the application has enough processing capacity to serve its current maximum workload of 400 rps (requests per second) without violating its latency constraint. As new content is delivered, the maximum load is expected to increase by an additional 200 rps. To serve this extra load, the application could add to its resource pool a medium server M (2 CPUs) or two small servers S (1 CPUs), at equivalent cost.

Assume that M servers are affected by hardware heterogeneity. Thus, these servers can be deployed in two different hardware types: type M_A (with 75% probability) and type M_B (with 25% probability). These hardware types perform differently; M_A can serve up to 220 rps and M_B can serve only 140 rps. Alternatively, assume that S servers are always assigned the same hardware and can serve up to 100 rps. It is worth noticing that M servers offer, on "average", an equivalent performance of the combination of two independent S servers. If an adaptation model ignores non-determinism or averages out the behavior, the application could choose in-distinctively between the two options. This could lead to a harmful decision; e.g., activating a M server to cope with an increase of 200 rps would lead to a violation of the quality requirement with a 25% chance.

[1] https://cloud.google.com/compute/docs/machine-types
[2] https://aws.amazon.com/ec2/

However, if non-determinism in represented explicitly, an adaptation engine can make better decisions considering the system's goals (e.g. latency constraints and reduced costs). Moreover, decisions can derive into separate policies that depend on the workload increase ranges. For instance, between 100 and 140 rps, any M server can cope with the demand (even for type B hardware) at minimum cost. However, between 140 and 200 rps, a M_B server would fail to meet the demand; thus, a safer option is to add two S servers at the same cost. A more interesting condition is when the load increases up to 240 rps. Here a $M + S$ configuration is guaranteed to meet the demand. Still, a M_A server could also meet the demand, at lower cost. Thus, an adaptation engine couldfirst activate M and then activate S only if the M server was assigned a type B hardware.

2.3 Requirements of the Solution

While such scenario clearly illustrates how capturing uncertainty can help the decision-making process to drive self-adaptation, this case oversimplifies the reality. In real life, combinations of hardware heterogeneity, resource contention, and failures would result in more complex models and adaptation situations. Furthermore, adaptation engines often measure and reason about impacts in terms of more "indirect" metrics that can be easily associated to service-level objectives (e.g. response time $\leq$ 10 ms). Thus, impact models must capture how an action would affect the system in terms of its key performance indicators (e.g. adding a S server reduces response time by 10%). Finally, these impacts do not behave uniformly across different system conditions. For instance, the effects of hardware heterogeneity when adding a new server may be visible "only" when the system is saturated; thus, for a range of conditions (e.g. response time $\leq$ 9 ms) the outcome may be a single one (e.g. adding a M server reduces response time by 20%), whilst for a different conditional range (e.g. response time $\approx$ 10 ms) the outcomes could be different, as illustrated before.

Our work focuses on generating models that satisfy the following requirements: (1) accuracy, since inaccuracies may result in ineffective or harmful adaptations, (2) ease to translate into human-readable rules, that reason in terms of (linear) impacts on system metrics (similar to those in [3]), and (3) the ability to represent explicitly non-deterministic impacts within conditional ranges.

3 RELATED WORK

In the following, we discuss the limitations of previous work to deal with the research problem at hand.

3.1 Expert-defined Adaptation Models

The idea that expert's knowledge can be used for adaptation is not new [4, 10, 14]. It is often the case that computer systems are handled by administrators that, over time, acquire knowledge about how the system operates and how it can be adapted in response to expected conditions. Adaptation models exist that can capture such expertise with greatfl exibility to represent the systems behavior while still ensuring human readability. This allows humans to remain *in-the-loop* [4] in order to: verify the system's behavior, react in emergency situations, and testify for the accountability of the adaptation policies [2, 17].

To capture system operations and human-like adaptation, some languages have been proposed. For instance, Stitch [5] departs from an architectural model of the system to define *adaptation strategies*. Strategies follow a decision tree logic, where each step is a conditional execution of some node in the tree, called *tactic*. A tactic, in turn, represents a guarded action that encompass a collection of *operators* (or configuration commands) with their expected effects and impacts on the system metrics. Stitch language, thus, simulates how administrators would adapt a system to reach high-level goals. It follows a condition-action-effect logic and combines it with the possibility of branching out in case that (so far) executed actions are insufficient or not successful at reaching the goal.

Other proposals concentrate on modeling how adaptation actions can impact the system. In particular, Cámara et al. [3] proposes to declare actions with probabilistic impacts, such that one action can lead to different outcomes of the system with an associated probability. By leveraging on models inspired on Discrete-Time Markov Chains (DTMCs), this approach improves on the explicit representation of uncertainty during adaptation (something that Stitch only captures implicitly using the tree branches).

Architectural models and domain-specific languages are commonly accepted as a mean to support reasoning about the system and its adaptation, mainly due to their expressiveness and generality [20]. Yet, expert-defined models are often incomplete, since humans do not consider all possible conditions and effects, and are rather inaccurate, as they are mostly based on perception rather than strict numerical data. Moreover, after the expert has specified a system's model and the adaptation policies, these become hard to keep up-to-date and risk becoming invalid over time, specially as the system evolves and new environmental conditions arise.

3.2 Machine Learning in Adaptation

Techniques based on machine learning derive their models from data collected, typically, from a running system. This has the potential of leading to adaptation models that are accurate and valid, provided that the learners are fed with sufficiently large and representative training sets. Several works in self-adaptive systems that exploit learning techniques are affected by intrinsic limitations that derive from the way learners are trained.

Mechanisms that are trained *online* [12, 22, 33] are exposed to the risk of performing poorly before a correct model is learned, leading to many inefficient and harmful adaptations. Alternatively, mechanisms that are trained *offline* [11] must use a large set of data collected before the system is deployed (which may be actually hard) and require potentially long training phases. As a way to mitigate these limitations, one could exploit expert-defined models to decrease the performance degradation during the learning phase [32]. Pre-defined models can also be used to create synthetic samples to train the learner *offline* and, after, refine the knowledge with samples collected *online*; a technique known as *bootstrapping* that was introduced in [9].

Maybe the most prevalent limitation of current machine learning techniques when evaluated against the requirements of our solution, is the inability to derive models that capture uncertainty explicitly. While some works already adopt machine learning algorithms that result in human-readable models (e.g. event-condition-action (ECA)

rules [15] or decision trees [26, 27]), none of them can explicitly represent the inherent effects of uncertainty during adaptation. Some solutions can capture uncertainty with probabilistic models [13, 31], but do so by considering a discrete state of the world; for instance, actions either succeed or fail to reach a state with a certain probability. These models lack the expressiveness to capture impacts associated to system properties and metrics that are continuous.

Some machine learning techniques are well regarded for their ability to infer precise predictors based on a dataset of samples; namely, artificial neural networks [24], fuzzy inference systems [6] and model trees [27]. However, these do not represent a solution to our research problem. Although artificial neural networks have been successful in accurately approximating non-linear functions, the inferred model is not easily translatable to human-readable rules. Fuzzy inference systems, instead, tend to generate (a large number of) fuzzy rules; in combination, a set of rules can closely approximate non-linear functions within a range, but in isolation these rules may contain little information that can be hard to interpret by humans [29]. Finally, while decision trees can accurately approximate linear functions [7] and output a set of readable rules, these models hide the effects of non-determinism, by averaging the impact of the potential multiple outcomes of the same action.

These techniques have also been used for cloud computing adaptation. For instance, [18] uses a fuzzy controller to decide scaling actions based on two metrics: workload and response time. Fed by expert-defined rules, the controller combines conditional ranges and averages out the proposed number of servers to activate/deactivate. Yet, it does not produce explicit policies that consider the probabilistic impact of these actions on those metrics.

Learning performance-influence models has also been studied in the past. In [30], regression analysis is used to learn influence models in highly-configurable systems. After taming down a large configuration space, an influence function (e.g. linear) is extracted using (few) selected features. Yet, this method is limited to systems that exhibit deterministic performance behavior (i.e. probabilistic outcomes cannot be learned). Similarly, Gaussian Process regression models with mixed experts [28] can identify piece-wise linear and non-linear functions; still, this method is unable to cope with uncertainty such as to identify multiple outcomes within a range.

A technique worth mentioning is the K-plane algorithm [1]. It identifies the K (hyper-)planes that best approximate a set of input data points in a multi-dimensional space. Interestingly, these planes can overlap in the input space, which means that for a given input there may exist more than one corresponding plane, which can be associated with different impact functions. Still, this approach does not explicitly separate the ranges in which a plane is valid.

4 LAMUN

Lamun uses learning techniques to derive human-readable models that capture non-deterministic impacts explicitly. More concretely, it relies on the subspace clustering approach, K-planes, to identify multiple impact functions that may overlap in some segments. Then it uses a method we have developed to extract conditional ranges associated to the system's performance metrics. The learned impacts are then represented in terms of human-readable models.

To illustrate the key concepts and techniques involved in the design of Lamun, we use the adaptation scenario described in Section 2, where a cloud-enabled application scales its resources to keep a low response time while minimizing the cost. We consider an example where all servers have the same "size" but may lead to different system performances, depending on the actual hardware provisioned by the provider.

We assume that a MAPE-K-like control loop is in place and the adaptation engine decides when servers must be activated or terminated, based on the monitored metrics. After the execution of an action, the adaptation engine can collect data samples from the running system, to learn how such action impacts the system's performance. Lamun repeats this process, supporting the improvement of the accuracy of the learned impacts and keeping the learned model up-to-date with the evolution of the system (e.g. when new hardware is installed in the infrastructure, the impacts of adaptations must be updated). A high level description of the main process is presented in Algorithm 1.

Algorithm 1: Learning a model with Lamun

```
dataset ← get_collected_data()
planes ← infer_functions(dataset)
ranges ← get_ranges(dataset, planes)
probabilities ← estimate_probabilities(dataset, planes, ranges)
model ← output_model(planes, ranges, probabilities)
```

For each adaptation action a separate model is learned. In our exposition, we concentrate on deriving the model that captures the effects of activating a new server (i.e. *enlist server*) on the system performance. To simplify our exposition, we consider the impact model over a single metric: the response time. The execution of the action might have different outcomes. For each of them, we consider that the predicted effect of the action in the response time (rsp') is defined as a function of the value of response time before the action is executed (rsp) and this function is linear. In what follows, we refer to these functions as *impact functions*.

More concretely, for each adaptation action Lamun learns a model that captures: (1) all possible outcomes of an action on the system metric (e.g. $rsp' = f(rsp)$ or $rsp' = g(rsp)$), (2) the probabilities associated to each outcome (e.g. with 90% probability, *rsp'* is predicted by the function f), and (3) the parameters of each impact function (e.g. $f(rsp) = 2/3 * rsp + 10$). It is worth mentioning that, in a simple model, the *output* (*rsp'*) is defined in terms of a single *input* (*rsp*). However, in more complex scenarios, the predicted value for a metric may depend on the values of several system properties or metrics, that define a multi-dimensional input space[3].

We assume that impact models use linear functions exclusively; i.e. a linear combination of the values of properties and metrics used to construct such functions can predict the value of the target metric, within a specific range. Thus, our approach is applicable to all functions that can be captured in a piece-wise linear form.

The steps to learn a model with Lamun (c.f., Algorithm 1) are described in detail in the following subsections.

[3]The selection of (a limited number of independent) system properties/metrics in the input space has been studied in the literature and is outside of the scope of our work

4.1 Collecting Samples

In order to learn a model, LAMUN requires a dataset of samples that capture the effect of the adaptation action on the target metrics. For instance, consider that a new server is activated and the observed response time decreases from 70*ms* to 50*ms*; then, a new sample $\langle 70, 50 \rangle$ would be added to the associated dataset.

The way the dataset is collected is orthogonal to the main contribution of our work. The dataset can be obtained by experimenting with the system offline, or by accumulating data points during the normal operation. The dataset should be large enough such that the prediction error of the model is below some application-dependent threshold. Later, we describe how LAMUN can be applied to an evolving underlying system (by combining samples obtained at different points in time) and also how a user-derived model can speed the learning process (creating an initial synthetic dataset).

4.2 Inferring Impact Functions

This step receives as input the dataset collected for a given adaptation action and computes the planes (assuming an n-dimensional input space) that best represents it. To infer the impact functions, we use the K-plane algorithm [1] as a building block, which returns a set of K planes for a given dataset. Note, however, that there is no way to know *a priori* how many impact functions the algorithm is expected tofi nd. In particular, K cannot be derived from a previously learned model. Indeed, one of the purposes is precisely to unveil new impacts that have not been learned yet.

To infer the impact functions we use Algorithm 2. The goal is tofi nd the smallest possible K that captures all the the relevant impacts without cluttering the model with redundant functions. For this purpose, we iterate the K-plane algorithm for different values of K, starting with $K = 1$ and increasing it in each iteration.

In order to assess when the iteration must stop, wefi rst split the dataset into two subsets: training and test subset (90% and 10% of original dataset). The K-plane algorithm is run against the training subset and validated against the test subset.

Algorithm 2: Inferring the Impact Functions

```
Input: dataset: set of samples
Data: ERROR_T: error threshold previously defined
Output: planes: set of planes inferred from the dataset
K ← 0
error ← 1
while error > ERROR_T do
    K ← K + 1
    error_list ← []
    for training, test : 10-Fold-cross-validation(dataset) do
        planes ←K-Plane(K, training)
        // test is a function, which returns error
        error_list ←append(error_list, test(planes, test))
    end
    error ← avg(error_list)
end
planes ← K-Plane(K, dataset)
```

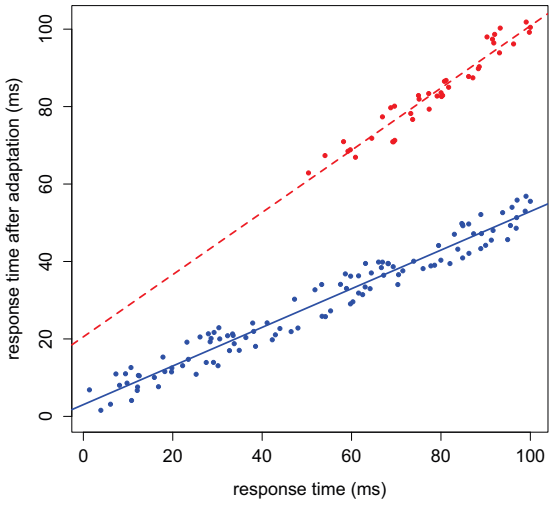

Figure 1: Inferred linear functions

The test subset is used to compute the error of the model, by measuring the average distance between each sample and its closest plane. This process is repeated 10 times, one with a different 10% of the dataset as test set and the error is an average of the 10 runs. The iterative loop stops when the computed error is lower than a given threshold (ERROR_T). After K is chosen, the K-plane algorithm is re-applied to the entire dataset, to obtain a result that is as accurate as possible. The result is a set of planes that represent different linear impact functions.

Figure 1 illustrates the result of applying this algorithm to a dataset of samples (dots) collected monitoring the average response time before and after enlisting a new server. In our example, two linear functions (depicted as lines) are inferred.

4.3 Computing Validity Ranges

In the previous step, K planes were derived capturing K different impact functions for a given adaptation action. However, some of these functions may apply just to a region of the entire input space. For instance, in the running example, we have obtained two impact functions (c.f. Figure 1) but for one of these functions, samples only exist for the interval $[50, 100]$ of the response time prior to the execution of the action. Hence, there is no evidence that this impact function applies when the adaptation is performed in conditions where the response time is outside such interval. Therefore, after computing the impact functions, it is necessary to identify the regions of the input space for which each of the functions is *valid*. Ultimately, this step allows us to capture that multiple outcomes may occur when the response time is in different intervals.

For each plane, only samples that are closer to that plane are used to compute the range for each input dimension. Considering a single dimension at a time, the range will be between the minimum and maximum value taken by samples, which belong to that plane. After this step we will have a range for each dimension and for any given plane, i.e. a hyper-cube surrounding the plane. In our example, one of the functions (planes) is limited by the range $[0, 100]$ and the other one by the range $[0, 50]$. For one dimension,

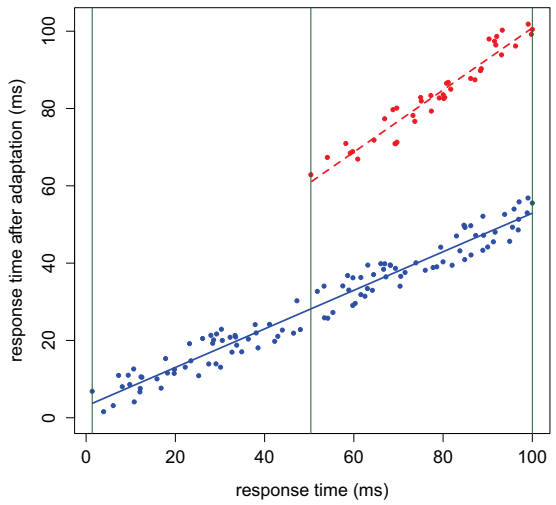

Figure 2: Validity ranges for the impact functions

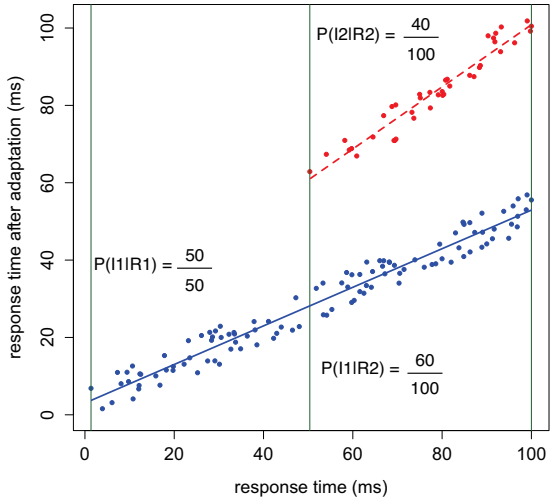

Figure 3: Probabilities for each impact function/region

we split such ranges by joining and ordering the limits of every plane. In our example, this would result in 0, 50, 100. Then, we pair every two consecutive values: [0, 50] and [50, 100]. Here, the second validity range represents an intersection of the sub-regions where the two functions are valid. The validity ranges resulting from this procedure are represented as vertical lines in Figure 2.

For higher dimensions, we check for intersecting planes, one dimension of the input space at a time. If no intersections are found on the input dimension, then the input space does not intersect (meaning that there is no uncertainty). In such case, the space does not need to be split because it is possible to distinguish among planes by, at least, an input variable. Otherwise, we split the input space as it was explained previously considering one dimension at a time. For instance, if there are two planes: ([0, 100], [0, 100]) and ([50, 150], [50, 150]), the result of the splitting for thefi rst dimension is: ([0, 50], [0, 100]), ([50, 100], [0, 100]), ([50, 100], [50, 150]), and ([100, 150], [50, 150]). Then, we repeat the process for the second dimension.

4.4 Estimating Probabilities

This step consists in deriving how likely it is to observe each of the possible impacts, for each region of the space where non-determinism has been detected (i.e. where different impacts can be observed for the same range). That is, the probabilities associated with each impact function are estimated at this stage.

The probability of the result of a given action being the one predicted by the i^{th} impact function takes into account each different region of the input state for which different non-deterministic outcomes are possible. These probabilities are estimated using the formula: $P(f_i|\text{region}) = \#(\text{region}, f_i)/\#(\text{region})$. The probability of i^{th} impact function in a given region is estimated by counting the number of samples that belong to that region (denoted by #(region)) and the number of samples in that region that are closer to that impact function (denoted by #(region, f_i)). Figure 3 depicts probabilities next to their plane.

4.5 Outputting a Human-Readable Model

Although our approach is generic and not tied to a particular language, to make the exposition concrete, we express the learned impact models using a syntax similar to the one proposed in [3].

The model that results from processing our example dataset is presented in Listing 1. The model specifies that, for observed response times (*rsp*) in the range $[50, 100]ms$, the predicted response time after a new server is enlisted can have two outcomes: with 40% probability, the new response time *rsp'* is predicted by function $f(rsp)$, and with 60% probability it is predicted by function $g(rsp)$. Linear functions f and g are also defined in the model.

```
define f(rsp) = (4/5) * rsp + 20
define g(rsp) = (1/2) * rsp + 5
impactmodel enlistServer
  50 < rsp < 100   -> {[0.4]  rsp' = f(rsp) +
                      [0.6]  rsp' = g(rsp) }
```

Listing 1: Resulting Model

5 REVISION OF EXISTING MODELS

We have just described how a new model can be built from scratch based on a given dataset. In many real scenarios the system being modeled is not static and can evolve with time. For instance, a cloud provider may acquire new machines types, whose performance would differ from that of previous hardware. Naturally, LAMUN can be executed periodically, to generate updated models that follow the evolution of the underlying system. In this paper, we do not have space to elaborate on the different strategies that optimize this refinement process, but we simply sketch the general principle that guide the use of LAMUN to refine a model.

Algorithm 3 illustrates how LAMUN would be used inside a loop to revise an existing model. In this loop, the existing dataset is curated, new samples are added to the dataset, and a new model is generated (based on the revised dataset that results from the two operations above). The goal of the *data_curation* step is to remove from the dataset samples that are no longer considered relevant.

Algorithm 3: Revision of existing models

```
while TRUE do
    dataset ← data_curation(dataset)
    dataset ← dataset ∪ get_new_samples()
    model ← LAMUN (dataset)
end
```

As with all similar approaches, there is a trade-off between how fast the model reacts to changes in the underlying system and how prone the model is to be affected by transient perturbations of the system. If the user wants to revise the model quickly in face of changes, he can simply discard the old dataset and build a new model just with the samples collected during the current iteration. On the other extreme, new samples can simply be accumulated with old samples. A middle ground solution consists in eliminating samples that are older than a given age. In any case, the number of points that remain in the dataset, in each iteration, should be large enough to allow the model to be accurate.

6 BOOTSTRAP FROM USER-DEFINED MODEL

In the previous section, we have shown that it is possible to refine a model based on a pre-existing dataset, that can be subsequently enriched with new points. In some scenarios, an user-defined model of the system operation is available. One might be interested in using LAMUN before a large enough dataset is collected from the running system. Thus, to start using LAMUN earlier, it si possible to generate a synthetic dataset form the expert-defined model and then apply the iterative approach of model refinement presented before. Note that departing from a synthetic dataset based on the expert's knowledge can ensure that accuracy is not lost while the new model is learned. Also, even if not enough samples of a given outcome are observed at run time during thefi rst loop iterations, the knowledge captured in the expert-defined model is preserved and taken into account.

User-defined models can be exploited to bootstrap the learning process of LAMUN. The initialization procedure for the dataset is captured in Algorithm 4.

Algorithm 4: Initialize Dataset

```
Input: impacts: a set of impacts
Data: C: constant value
Output: dataset: set of samples
dataset ← {}
for i : impacts do
    samples_per_dim ← i.probability * C
    // get_inputs computes combinations of inputs on a
    // grid with a side of: range / (samples_per_dim - 1)
    input_set ← get_inputs(i.dimensions, samples_per_dim)
    for input : input_set do
        output ← compute(i.function, input)
        dataset ← add(dataset, (input, output))
    end
end
```

As before, to simplify the exposition, let us consider the impact an action has on a single metric. To generate the synthetic samples for this metric, the impact functions from the original model are used. For each impact function f_i, synthetic tuples consist of pairs $\langle input, f_i(input)\rangle$, for some *input* value of the input space. To preserve the information provided by the probabilities associated with each impact function, the number of samples generated for each function is proportional to its probability. This is done by generating a number of samples per dimension that is proportional to the probability associated with the impact. The proportionality factor C must be a positive number, large enough to ensure the creation of multiple samples for each possible impact. Our approach requires that the interval of valid input values is passed to the algorithm. Knowledge on the input ranges is needed to create synthetic points just for the relevant scenarios.

To create the synthetic dataset, LAMUN makes an uniform sampling of the input space by dividing each input dimension in regions of the same size as follows: $range/(samples\_per\_dim - 1)$. This ensures that all samples are within the minimum and maximum value of the dimension's range, and are uniformly spaced. In our example, since we only consider one input variable, assuming that the validity range is $[0, 100]$ and that the number of samples to generate is 1000, then samples will be generated for input values separated by approximately 0.1 units, resulting in 1000 samples.

7 EVALUATION

We now present an experimental evaluation of LAMUN. We experimented with a concrete instantiation of the problem used in Section 4. Namely, we applied LAMUN to learn an adaptation model used to support the elastic scaling of RUBiS deployment in a local cloud. RUBiS is a well-known auction website similar to eBay.

7.1 Experimental Testbed

We have used RUBiS[4] 1.4.3 deployed on virtual machines running Ubuntu 14.04 in a cluster of workstations, each with a 2.13GHz Quad-Core Intel(R) Xeon(R) processor and 32GB of RAM, connected by a private Gigabit Ethernet. We used Autobench[5] 3 as a benchmark workload generator and drive httperf [25] to issue the requests. We run HAProxy[6] 1.6 on a separate virtual machine to distribute the load among servers.

For our experimental usecase, the adaptation action to be modeled is the activation of *one* server (VM). We consider all servers to be of the same "size". Naturally, the models that are learned are slightly more complex than the simplified example presented before. In the deployed system, we monitor several metrics, namely: number of active servers, average request rate, and average response time. We learn a model to estimates the action's impact on a single performance metric: the response time.

We consider that the application is affected by uncertainty caused by resource contention in the underlying infrastructure, composed by physical machines of one unique type. We have considered physical machines that can be in two distinct conditions: non-contentious

[4]Rice University Bidding System: http://rubis.ow2.org/
[5]Autobench: http://www.xenoclast.org/autobench/
[6]HAProxy: http://www.haproxy.org/

or contentious. In thefi rst condition, the physical machine is running RUBiS exclusively. In the second condition, the physical machines runs RUBiS alongside other services. Specifically, we simulate the conditions of a newly activated server of the smallest type in the GoGrid[7] cluster. That is, a new server is guaranteed 1 CPU, but can be assigned 2 CPUs if the physical machine is not being used by other services (with a 60% probability in our usecase). Also, we assume that the provider decides the physical machine where a server is deployed; thus, it cannot be controlled by the application.

7.2 Data Collection

We have collected experimental data for our deployment under different configurations and workloads. We change the number of active servers (between 1 and 4) and the number of requests per second (between 250 and 10000). For simplicity, we ensure that all servers that are active before the adaptation is executed have 2 CPUs each. For each configuration, we generate a load of 10000 requests distributed among four clients.

We then collected samples that measure the impact on the response time. We have executedfi ve different runs of each experiment and used the average behavior to derive thefi nal values to be included in the dataset. Also, samples with observed response time higher than 600*ms* were removed, as we consider that a real system would have to adapt before reaching such conditions.

The size of our dataset is of 688 samples.

7.3 Resilience to Noise

A key parameter in the performance of Lamun is the error threshold (ERROR_T). This parameter is used in Algorithm 2 to decide when to stop adding planes to the model (i.e. to select the value of K). Another factor affecting the accuracy of the learned model is the accuracy of the measurements used to create samples in the dataset. Algorithm 2 assumes that a minimum level of accuracy exists in the dataset, such that the detected number of planes K is valid. Yet, if the dataset samples are created out of (very) noisy measurements, Lamun can falsely detect planes where there is just noise. To test the resilience to noise, we designed a experiment to inject different levels of noise and estimate the value of ERROR_T, such that K is accurately identified in spite of the noise.

In our experiments, we already observed a significant amount of noise when measuring the impact of adaptations. In our case, we were able to smooth the effect of the noise by averaging the outcome of multiple samples, as described previously. However, there may be cases where having enough samples to smooth the noise is unfeasible. To assess the effect of noisy datasets in Lamun, we created "polluted" datasets from the original dataset. This allowed us to control exactly how much noise is present in the dataset, something that is hard to control in real life. The added noise was generated randomly using a normal distribution, with mean equal to the original point's value and standard deviation equal to half the original point's error. To create an instance of a polluted dataset, a percentage of this noise was added to the original dataset values.

To isolate the effects of noise from uncertain impacts, we departed from a dataset that uses dedicated servers exclusively and learns "one" impact function ($K = 1$), i.e. the impact is deterministic.

[7] GoGrid: https://my.gogrid.com/

Table 1: Noise. ERROR_T parameter selection

ERROR_T	0%	10%	20%	30%	40%	50%
0.2	2.3	3.4	3.4	4.5	4.5	6.7
0.3	1.0	1.0	1.0	1.0	2.3	3.0
0.4	1.0	1.0	1.0	1.0	1.0	1.0

We simulated Algorithm 2 running several instances of a "polluted" dataset against different values of ERROR_T and computed the average number of planes detected for each pollution level.

Table 1 show results for a (small but representative) set of pollution levels within the noise range observed when monitoring the original data: 0%, 10%, 20%, 30%, 40%, 50%. We calibrated ERROR_T experimentally and selected three of the tested values (0.2, 0.3, and 0.4) that illustrate the observed trend. We show that as noise levels increase, the detected number of planes increases too.

In particular, the number of planes that should be detected by Lamun in this experiment is $K = 1$. However, if the value of ERROR_T is too small (e.g. 0.2), even data without added noise (0%) can be considered to hold more than one plane: non-determinism is being found where it should not exist. Instead, if the value of ERROR_T is too high (e.g. 0.4) the sensitivity is too low, which may lead to ignoring planes that could exist. For this particular dataset, ERROR_T ≈ 0.3 represents a good trade-off given the noise levels. It is worth mentioning that the absolute values presented here are specific to this setting and should not be applied to other contexts.

7.4 Capturing Uncertainty

An advantage of Lamun is the ability to capture uncertainty explicitly. To demonstrate this feature, we have fed the learner with data capturing the two distinct conditions of the physical machine assigned to a new active RUBiS server. With 60% chance, a fully dedicated server is enlisted (2 CPUs) and, with 40% probability, a server that also runs other services is enlisted (1 CPU).

Listings 2 shows a representation of the learned impact models. In this model, *act* is the initial number of active servers, *req* is the average number of requests per second, and *rsp* and *rsp'* are the average response time before and after the action's execution. As it can be observed, Lamun is able to learn multiple (probabilistic) impact functions. Also, the learned probabilities follow closely the expected numbers as approximately 60% of the dataset corresponds to activating dedicated servers (2 CPUs). Finally, the model can associate each impact function to its validity range. Thus, the derived policies are applicable in different workload conditions, when the number of requests is in three different ranges: $(250, 800]rps$, $(800, 9955]rps$, and $(9955, 10000]rps$.

```
define f(a,rq,rs)=  -34.99*a +0.025*rq +0.36*rs +74.51
define g(a,rq,rs)= -135.08*a +0.014*rq +0.49*rs +502.87
impactmodel enlistServer
(0 < act < 4 & 3.6 < rsp < 595.7 & 250 < req <= 800)
  -> {[1.000] rsp' = f(act,req,rsp)}
(0 < act < 4 & 3.6 < rsp < 595.7 & 800 < req <= 9955)
  -> {[0.623] rsp' = f(act,req,rsp) +
      [0.377] rsp' = g(act,req,rsp)}
(0 < act < 4 & 3.6 < rsp < 595.7 & 9955 < req <= 10000)
  -> {[1.000] rsp'= g(act,req,rsp)}
```

Listing 2: Enlist server impact with Lamun

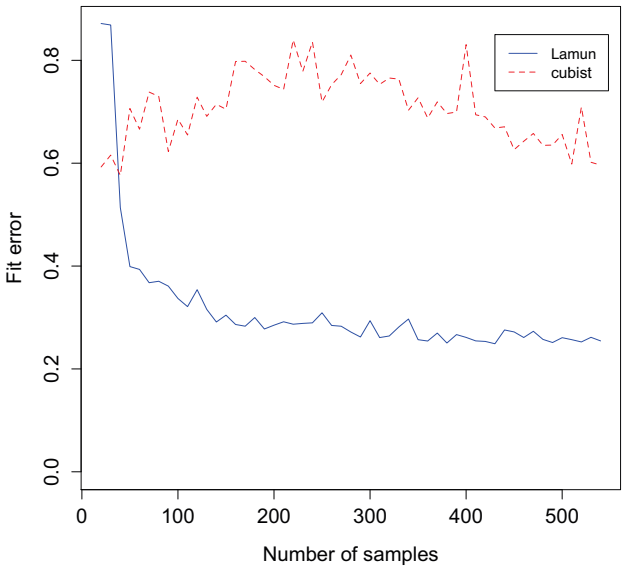

Figure 4: Accuracy: *cubist* and Lamun Errors

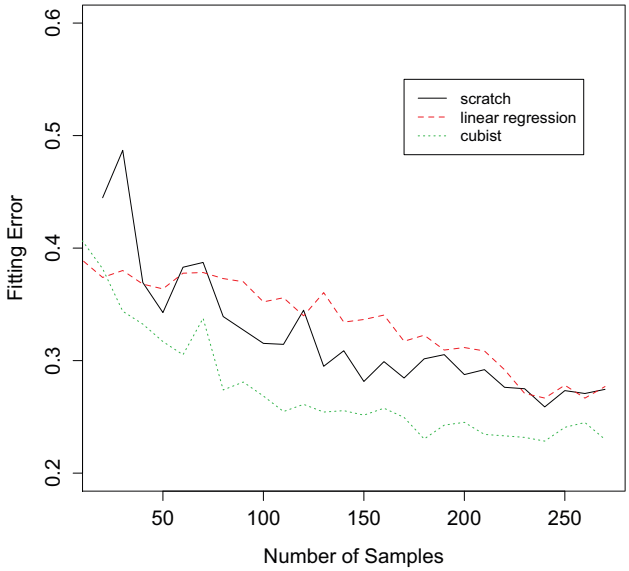

Figure 5: Model convergence after bootstrapping

7.5 Model Accuracy

To evaluate the accuracy of the model learned by Lamun, i.e. how well it captures the behavior of the system, we compare it with the accuracy of a model learned with *cubist* [21]. *cubist* is a rule-based learner that considers continuous states, creating models similar to those learned by Lamun. Yet, *cubist* does not explicitly capture non-determinism; instead, it averages out all outcomes.

To compare both approaches, we have split the collected dataset (688 samples) into a training (80%) and a testing (20%) subset, selecting samples randomly to ensure an unbiased dataset. We built the model using data from the training subset (exclusively) and used data from the testing subset to check the accuracy of the model.

We define thefi t error as the average distance between the (real) value of the test points and the value estimated by the model, according to the following function: $fiterror = \sum pointerror/\#(points)$, where: $pointerror = |real - estimate|/|estimate|$. We variate the size of the training subset (from 0% to 90%) to illustrate how fast the model stabilizes into a "good" model as more samples are added.

The results in Figure 4 show that Lamun achieves a betterfitting to the model w.r.t. *cubist*. The model learned by Lamun stabilizes rapidly after a training set of around 15% of the dataset ($\approx$ 100 samples). At such point, the model captures two distinct outcomes and is able to estimate the real value with $fiterror \leq 0.35$. We can also observe that as the number of samples used to train the model increases, thefi t error of *cubist* increases slightly. We assume this happens because the model grows skewed to a value in between both outcomes, thus increasing its variance as more samples are added.

It is worth mentioning that these results are specific to the experimental dataset, as the number of samples needed to learn an accurate model with Lamun is highly dependent on how samples in the dataset are distributed across multiple outcomes.

7.6 Revision of Existing Models

Lamun can learn models from scratch or departing from a pre-existing model. When departing from a pre-existing model, the quality of such model can have an impact in the learning process. In order to show how the correctness of a pre-existing model affects Lamun, we revise three models with different quality: (1) one inferred using a *linear regression*, (2) one inferred using *cubist*, and (3) one learned from scratch. To train these models we used a partial dataset composed by 50% of the entire dataset (sampled randomly). The remainder of the dataset is split into training and test sets to validate the model through 10-fold cross-validation. For each model, we fed Lamun a growing number of samples from the training set, and test it using the test set. We use bootstrapping to generate the synthetic samples from the two existing models.

Figure 5 shows that, in general, using a pre-existing model to initialize the dataset results in a faster convergence into an accurate model. However, when the original model is inaccurate (see linear regression), the model being learned will behave better at the beginning, but it will be hindered in the long run due to the samples collected from the original (inaccurate) model.

A *data curation* procedure could help reduce the negative impact of outdated samples in the dataset. To evaluate the need of *data curation*, we perform an experiment where the original (synthetic) dataset is created based on an inaccurate prediction of the *new response time* after the *enlist server* action is executed. We generate two inaccurate models for a unique output fo the metric, with a deviation error of 250*ms* (lower error) and 300*ms* (higher error). We then add new samples gathered from the running system and evaluate the system's ability to disregard inaccurate data.

Figure 6 shows how the probability assigned to the inaccurate model decreases with the number of added (accurate) samples. We conclude that data curation is not indispensable; instead, it is only needed if Lamun is slow at capturing new samples from the running

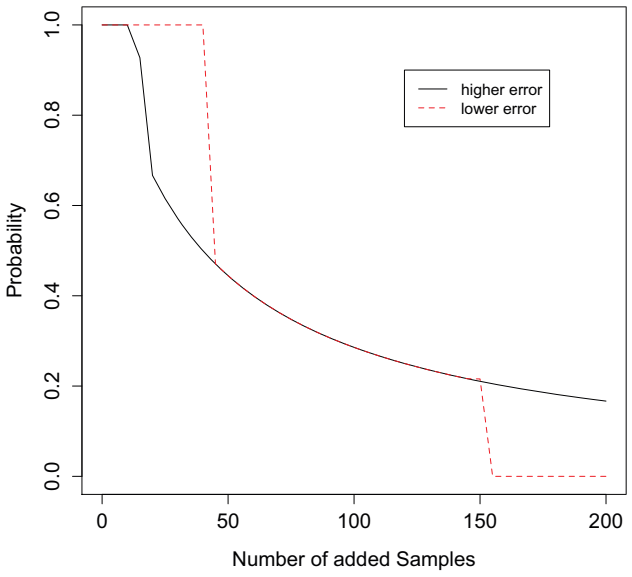

Figure 6: Evolution of the probability of a synthetic impact function as new samples are added to the dataset

system. The definition of the periodicity with which LAMUN updates the dataset online is application-specific and a choice of the user.

8 CONCLUSION

In this paper we proposed a way of learning action's impact models for self-adapting systems, and represent them in a readable form, which will allow the user to be kept in the adaptation loop. What differentiates our solution from existing approaches is the fact that we consider and explicitly represent probabilistic impacts, while also considering a continuous space. To learn impact functions from a dataset, knowing that a single input can have multiple output values, we used K-Plane algorithm. We tested this solution using a well-known auction application, RUBiS. Specifically, we tested its accuracy, its ability to handle uncertainty by capturing multiple outcomes of an actions, and its speed in revising an existing impact model. Furthermore, we tested the way it deals with noisy measurements and showed how that affected parametric choices.

Acknowledgments: This work has been partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects with references PTDC/EEI-SCR/1741/2014 (Abyss) and UID/CEC/50021/2013.

REFERENCES

[1] Paul S Bradley and Olvi L Mangasarian. 2000. k-Plane clustering. *Journal of Global Optimization* 16, 1 (2000), 23–32.
[2] Yuriy Brun, Cristina Serugendo, Giovannaand Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. 2009. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*. Springer, 48–70.
[3] Javier Cámara, Antónia Lopes, David Garlan, and Bradley R. Schmerl. 2016. Adaptation impact and environment models for architecture-based self-adaptive systems. *Science of Computer Programming* 127 (2016), 50–75.
[4] Javier Cámara, Gabriel A. Moreno, and David Garlan. 2015. Reasoning about Human Participation in Self-Adaptive Systems. In *SEAMS 2015*. Florence, Italy, 146–156.
[5] Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software* 85, 12 (2012), 2860–2875.
[6] Stephen L Chiu. 1994. Fuzzy model identification based on cluster estimation. *Journal of Intelligent and Fuzzy Systems* 2, 3 (1994), 267–278.
[7] Maria Couceiro, Paolo Romano, and Luis Rodrigues. 2010. A Machine Learning Approach to Performance Prediction of Total Order Broadcast Protocols. In *SASO 2010*. 184–193.
[8] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ASPLOS 2013*. Houston, USA, 77–88.
[9] Diego Didona and Paolo Romano. 2015. Using Analytical Models to Bootstrap Machine Learning Performance Predictors. In *ICPADS 2015*. Melbourne, Australia, 405–413.
[10] Christoph Dorn and Richard N. Taylor. 2013. Coupling software architecture and human architecture for collaboration-aware system adaptation. In *ICSE 2013*. San Francisco, USA, 53–62.
[11] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *PVLDB* 2, 1 (2009), 1246–1257.
[12] Ahmed M. Elkhodary, Naeem Esfahani, and Sam Malek. 2010. FUSION: a framework for engineering self-tuning self-adaptive software systems. In *SIGSOFT 2010*. Santa Fe, NM, USA, 7–16.
[13] Antonio Filieri, Lars Grunske, and Alberto Leva. 2015. Lightweight Adaptive Filtering for Efficient Learning and Updating of Probabilistic Models. In *ICSE 2015*, Vol. 1. Florence, Italy, 200–211.
[14] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir Molzam Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. 2017. Control Strategies for Self-Adaptive Software Systems. *TAAS* 11, 4 (2017), 24:1–24:31.
[15] Alexander Frömmgen, Robert Rehner, Max Lehn, and Alejandro P. Buchmann. 2015. Fossa: Learning ECA Rules for Adaptive Distributed Systems. In *ICAC 2015*. Grenoble, France, 207–210.
[16] Paul Horn. 2001. Autonomic computing: IBM's Perspective on the State of Information Technology. (2001).
[17] Markus Huebscher and Julie McCann. 2008. A survey of autonomic computing-degrees, models, and applications. *Comput. Surveys* 40, 3 (2008), 7.
[18] Pooyan Jamshidi, Claus Pahl, and Nabor C. Mendonça. 2016. Managing Uncertainty in Autonomic Cloud Elasticity Controllers. *IEEE Cloud Computing* 3, 3 (2016), 50–60.
[19] Jeffrey Kephart and David Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.
[20] Jeff Kramer and Jeff Magee. 2007. Self-Managed Systems: an Architectural Challenge. In *ISCE/FOSE 2017*. Minneapolis, MN, USA, 259–268.
[21] Max Kuhn, Steve Weston, Chris Keefer, and Nathan Coulter. 2012. Cubist Models For Regression. *R package Vignette R package version 0.0* 18 (2012).
[22] Palden Lama and Xiaobo Zhou. 2010. Autonomic Provisioning with Self-Adaptive Neural Fuzzy Control for End-to-end Delay Guarantee. In *MASCOTS 2010*. Miami, FL, USA, 151–160.
[23] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos - A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology* 16, 3 (2016), 15:1–15:23.
[24] Tom M. Mitchell. 1997. *Machine Learning, International Edition*. McGraw-Hill.
[25] David Mosberger and Tai Jin. 1998. httperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review* 26, 3 (1998), 31–37.
[26] João Paiva, Pedro Ruivo, Paolo Romano, and Luís E. T. Rodrigues. 2014. AutoPlacer: Scalable Self-Tuning Data Placement in Distributed Key-Value Stores. *TAAS 2014* 9, 4 (2014), 19:1–19:30.
[27] J Ross Quinlan. 2014. *C4. 5: programs for machine learning*. Elsevier.
[28] Carl Edward Rasmussen and Zoubin Ghahramani. 2001. Infinite Mixtures of Gaussian Process Experts. In *NIPS 2001*. 881–888.
[29] Timothy J. Ross. 2010. *Fuzzy logic with engineering applications*. Chichester, U.K. John Wiley.
[30] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *ESEC/FSE 2015*. Bergamo, Italy, 284–294.
[31] Daniel Sykes, Domenico Corapi, Jeff Magee, Jeff Kramer, Alessandra Russo, and Katsumi Inoue. 2013. Learning revised models for planning in adaptive systems. In *ICSE 2013*. 63–71.
[32] Gerald Tesauro, Nicholas Jong, Rajarshi Das, and Mohamed Bennani. 2006. A hybrid reinforcement learning approach to autonomic resource allocation. In *ICAC 2006*. 65–73.
[33] Jing Xu, Ming Zhao, José Fortes, Robert Carpenter, and Mazin Yousif. 2008. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing* 11, 3 (2008), 213–227.