SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation

JOHANNES GROHMANN, University of Würzburg, Germany
SIMON EISMANN, University of Würzburg, Germany
ANDRÉ BAUER, University of Würzburg, Germany
SIMON SPINNER, IBM, Germany
JOHANNES BLUM, University of Konstanz, Germany
NIKOLAS HERBST, University of Würzburg, Germany
SAMUEL KOUNEV, University of Würzburg, Germany

Resource demands are crucial parameters for modeling and predicting the performance of software systems. Currently, resource demand estimators are usually executed once for system analysis. However, the monitored system, as well as the resource demand itself, are subject to constant change in run-time environments. These changes additionally impact the applicability, the required parametrization as well as the resulting accuracy of individual estimation approaches. Over time, this leads to invalid or outdated estimates, which in turn negatively influence the decision-making of adaptive systems.

In this paper, we present *SARDE*, a framework for self-adaptive resource demand estimation in continuous environments. *SARDE* dynamically and continuously tunes, selects, and executes an ensemble of resource demand estimation approaches to adapt to changes in the environment. This creates an autonomous and unsupervised ensemble estimation technique, providing reliable resource demand estimations in dynamic environments. We evaluate *SARDE* using two realistic data sets. One set of different micro-benchmarks reflecting different possible system states and one data set consisting of a continuously running application in a changing environment. Our results show that by continuously applying online optimization, selection and estimation, *SARDE* is able to efficiently adapt to the online trace and reduce the model error using the resulting ensemble technique.

CCS Concepts: • **Computing methodologies** → *Learning paradigms*; **Model development and analysis**; • **Software and its engineering** → *Software performance*.

Additional Key Words and Phrases: self-adaptive systems; resource demand estimation; machine learning; optimization; self-tuning algorithms

ACM Reference Format:
Johannes Grohmann, Simon Eismann, André Bauer, Simon Spinner, Johannes Blum, Nikolas Herbst, and Samuel Kounev. 2021. SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation. *ACM Trans. Autonom. Adapt. Syst.* 1, 1, Article 1 (January 2021), 32 pages. https://doi.org/10.1145/3463369

Authors' addresses: Johannes Grohmann, johannes.grohmann@uni-wuerzburg.de, University of Würzburg, Würzburg, Germany; Simon Eismann, simon.eismann@uni-wuerzburg.de, University of Würzburg, Würzburg, Germany; André Bauer, andre.bauer@uni-wuerzburg.de, University of Würzburg, Würzburg, Germany; Simon Spinner, sspinner@de.ibm.com, IBM, Boeblingen, Germany; Johannes Blum, johannes.blum@uni-konstanz.de, University of Konstanz, Konstanz, Germany; Nikolas Herbst, nikolas.herbst@uni-wuerzburg.de, University of Würzburg, Würzburg, Germany; Samuel Kounev, samuel.kounev@uni-wuerzburg.de, University of Würzburg, Würzburg, Germany.

1556-4665/2021/1-ART1 $15.00
https://doi.org/10.1145/3463369

1 INTRODUCTION

Timely and precise resource demand estimates are a crucial input to auto-scaling mechanisms [2] or performance modeling techniques [36, 69] used for elastic resource provisioning. Therefore, it has been shown that statistical estimation of resource demands is a valid and useful tool to realize precise elastic cloud resource management [2, 92]. A *resource demand* (or service demand [79]) is the average time a unit of work (e.g., request or transaction) spends obtaining service from a resource (e.g., CPU or hard disk) in a system over all visits, excluding any waiting times [48, 59]. Unfortunately, measuring resource demands during system operation is not feasible in most realistic systems [79] due to instrumentation overheads and possible measurement interference. Therefore, a number of approaches for resource demand estimation have been proposed over the years, using different statistical estimation techniques (e.g., linear regression [8, 72] or Kalman filters [88, 98]) and based on different modeling approaches from queueing theory.

When selecting an appropriate approach for a given scenario, a user has to consider different characteristics of the estimation approach, such as the expected input parameters, configuration settings, its accuracy and its robustness to measurement anomalies. The accuracy of the different approaches is heavily dependent on factors like, including but not limited to, system load, workload type, deployment structure, internal state, and monitoring granularity [79]. Additionally, Spinner et al. [79] show that no single approach is optimal in all scenarios. This is in accordance with the no-free-lunch theorems for machine learning [93] and optimization [94], stating that any two algorithms are equivalent when their performance is averaged across all possible problems.

First steps towards solving the above issues focus on combining different estimation approaches into a single usable tool [80], optimizing configuration parameters based on measurement data [27, 29], and recommending the most promising approach using machine learning [30]. However, existing work focuses on one-time estimation and optimization, ignoring the impacts of system change. As modern software paradigms like DevOps and elastic cloud operations become increasingly popular, timely and precise resource demand estimations get increasingly complex as more and more variables are continuously subject to change and estimates have to be continuously updated. For example, any auto-scaler is constantly changing the deployment structure of the considered software system. In addition, the applied workload is never truly constant in any online application. In consequence, the considered environment is both unknown at design time, and constantly evolving during operation time [10]. As the system and measurement data are changing, the best-suited estimation approach is also subject to change. It is therefore impossible for any human user to continuously select, parameterize and supervise resource demand estimators during system operation.

Therefore, in this paper, we introduce *SARDE*, a framework for continuous, *S*elf-*A*daptive *R*esource *D*emand *E*stimation. *SARDE* is able to operate, parameterize and select multiple different resource demand estimations in a continuous manner and adapts autonomously to changes in its environment in form of the system under study. This work focuses on combining and interlacing the different building blocks in order to create an adaptable and robust framework that can be applied in any continuous environment without requiring expert knowledge. To that end, *SARDE*

(i) continuously estimates resource demands,
(ii) continuously selects the best-suited estimation approach,
(iii) continuously learns and adapts the selection strategy in order to adapt to changing environments, and
(iv) continuously tunes the parameters of individual approaches based on online observations.

To summarize, *SARDE* works as a fully autonomous, situation-aware, and self-adaptive ensemble resource demand estimation approach. *SARDE* utilizes the above techniques to improve the performance of current state-of-the-art approaches without the need for human supervision or expert knowledge. We already presented one application for *SARDE* in previous papers [55, 56], where we integrate adaptive monitoring probes into continuous integration and deployment pipelines. Our approach can be used to constantly update or improve a performance model of a running application. Therefore, *SARDE* represents a significant step forward towards our vision of self-aware performance models [28], but also towards the vision of autonomic and self-aware computing [41, 45, 82] in general, as the techniques we introduce — although focused on the area of resource demand estimation — can also be transferred to other areas of research. The source code of the proposed approach is available online[1]. In addition, we published the code for constructing and analyzing the experimentation data set[2], and also published a replication package as a CodeOcean capsule[3].

The remainder of this work is structured as follows. We discuss the progress of the area of resource demand estimation in Section 2 and then motivate the idea behind *SARDE* in Section 3. Following, we introduce the general overview of *SARDE* in Section 4 and explain the concepts in more detail in Section 5. Section 6 presents our methodology for evaluating the framework, while Section 7 presents the obtained results. We discuss these insights in Section 8, and analyze the threats to validity in Section 9 and the limitations of our approach in Section 10. Finally, we conclude the paper in Section 11.

2 RELATED WORK

In this section, we will discuss the related work on the topics of resource demand estimation, algorithm optimization, and algorithm selection in self-adaptive systems.

2.1 Resource Demand Estimation

As resource demands are a crucial parameter for many modeling approaches, the topic of estimating resource demands received a lot of attention in recent years and many different authors proposed respective approaches. Spinner et al. [79] present a literature survey covering the most prominent approaches. However, concerning the evaluation of the different approaches, most works unfortunately only cover a selected set of one or two approaches.

The first experiments are presented by Rolia and Vetland [72, 73] using linear regression techniques. Pacifini et al. [64], Casale et al. [12, 13], and Stewart et al. [83] extend these works by investigating limitations of linear regression in resource demand estimation and the impact of different factors. The performance of Kalman Filters for resource demand estimation is researched by Zheng et al. [98, 99] and Kumar et al. [47]. Kraft et al. [46] and Sharma et al. [74] both compare least-squares regression with their maximum likelihood estimation and independent component analysis approach, respectively.

The only works aiming at combining a set of different approaches are the Filling-the-Gap tool by Wang et al. [90] and the LibReDE tool by Spinner et al. [81]. Filling-the-Gap [90] provides and compares implementations of the complete information method [65], Gibbs sampling with queue lengths [88], a maximum likelihood estimator based on a Markov chain representations [65], a maximum likelihood estimator using a fluid approximation [65], a regression-based approach [65], utilization-based regression [96], and utilization-based optimization [53].

[1] Available at https://github.com/jo102tz/LibReDE-SARDE
[2] Available at https://github.com/jo102tz/LibReDE-SARDE-data
[3] Available at https://doi.org/10.24433/CO.8429465.v2

Similarly, the publicly available tool LibReDE (**Lib**rary for **Re**source **D**emand **E**stimation) [81] offers open source implementations of currently eight different estimators:

- Service Demand Law (SD) [8]
- Approximation with response times (RT) [8]
- Least-squares regression using queue-lengths and response times (RR) [46]
- Least-squares regression using utilization law (UR) [72]
- Kalman Filtering using utilization law (WF) [88, 89]
- Kalman Filtering using response times and utilization (KF) [47, 98]
- Recursive optimization using response times (MO) [57]
- Recursive optimization using response times and utilization (LO) [53]

The results of Spinner are furthermore published in a respective study [79]. However, apart from our previous works [27, 29, 30] incorporated into *SARDE*, there exists no work on automatic and systematic evaluation targeting at performance optimization of resource demand estimation approaches for a given target scenario, since previous approaches only do manual testing and develop rules of thumb for a chosen small set of parameters (see the above-mentioned articles [12, 13, 47, 79, 98, 99]). Similarly, we are not aware of any techniques that use the acquired information to develop automatic selection algorithms as we propose in this work.

As LibReDE[4] is a publicly available ready-to-use implementation of different resource demand estimation approaches, our implementation of *SARDE* builds upon the LibReDE tool and uses the listed approaches as base estimators.

2.2 Algorithm Optimization in Self-adaptive Systems

Although no works with a focus on resource demand estimation have been proposed, the idea of continuously adapting and optimizing a system in a changing environment is not new. For example, the communities of self-aware, self-adaptive, self-organizing, or self-* systems tackle challenges of monitoring, managing, and optimizing complex intelligent systems in continuously changing environments [45].

As such, the ideas presented in this paper and incorporated into *SARDE* have been successfully applied to other domains. For example, Porter et al. [67] present Rex, a development platform that is also able to apply online learning and optimization based on a linear bandit model. Others define self-organization or self-assembly to achieve a similar goal [22, 44, 71]. Fredericks et al. [23, 24] present an overview of different optimization techniques in self-adaptive systems. They divide works into techniques using probabilistic, combinatorial, evolutionary, stochastic, or mathematical optimization. Additionally, D'Angelo et al. [17, 18] present a survey and a taxonomy for online learning of collective self-adaptive systems. If we interpret our single estimators as individual agents, *SARDE*'s estimators could classify as fully altruistic, non-autonomous agents with full knowledge access. While the task of choosing the best estimator can be seen as a combinatorial optimization problem [61, 66], the presented techniques for parameter optimization fall in the category of mathematical optimization [11, 20, 51, 75]. The proposed hyper-parameter tuning is also a common topic in machine learning. Therefore, a set of algorithm configuration approaches, like Sequential Model-based Algorithm Configuration (SMAC) [37], or Stepwise Sampling Search (S3) [62, 63] have been proposed, as well as analysis and visualization tools [3]. A sub-field is also Neural architecture search (NAS) [21, 38], where the goal is to automatically find neural network architectures; these techniques could also be applied in future work.

However, while all of the presented approaches demonstrate the feasibility of applying the proposed techniques in practice, none of these works focuses on the area of resource demand

[4]LibReDE: Available for download at http://descartes.tools/librede.

estimation. Therefore, our contribution in respect to this field is to demonstrate and verify the applicability of continuous algorithm optimization in the specific domain of continuous resource demand estimation.

2.3 Algorithm Selection in Self-adaptive Systems

An orthogonal field in the context of continuous optimization is algorithm selection [4, 42]. Algorithm selection [70] (closely related to the field of hyper-heuristic selection [9, 77] or meta-learning [78, 84]) is defined as choosing from a set of algorithms the best for a specific problem instance and has found many application areas in prior research [4, 26, 35, 52, 54, 68, 95].

However, the creation and selection of features for selection is a critical task influencing the performance [4, 42]. Hence, by tailoring our features to the specific task at hand, we can provide better results than generic optimization and selection frameworks. The application in *SARDE* is different from most of the proposed techniques as it offers the possibility to perform selection on continuously incoming data streams, which currently only a few works consider [42, 84, 85]. In addition, *SARDE* provides an application for online algorithm selection [1, 19, 25]. Both areas have been identified as specific research challenges by prior works [42].

Again, as no works concentrate on resource demand estimation, the focus of this work is to demonstrate the feasibility of continuous algorithm selection in our specific domain. However, similar to the previous section, many of the proposed techniques can be applied to our task as well in order to further improve the results presented in this work.

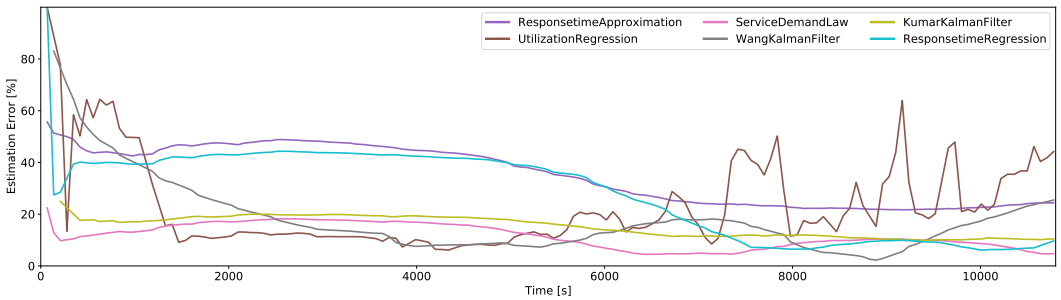

Fig. 1. Motivating example showing the estimation error of different estimators over time.

3 MOTIVATING EXAMPLE

In order to illustrate and motivate the idea behind *SARDE*, Figure 1 shows the error (calculated as described in Section 6.2) of the continuously updated estimation using all available approaches over time. Details on the used system and workload are included in Section 6.1.2.

Envision that during estimation, continuous monitoring streams of throughputs, response times, and resource utilizations are collected. For illustration purposes, imagine that during the first interval, a CPU utilization of 80% is measured, while 20, 40, and 5 requests of the respective workload classes are measured. In the second interval, the utilization drops to 60%, as 30, 20, and 10 requests were processed. The task of the resource demand estimators is now to calculate the resource demand of each workload class, based on this set of coarse-grained measurements.

We observe that over the course of 3 hours, the performance of each estimator is massively influenced by the type and amount of monitoring data available, as well as the underlying characteristics of the system. As a result, service demand law (pink) starts as the best estimator, followed by

utilization regression (brown). However, the accuracy of utilization regression starts to decline after a while, and in fact, continues to have the worst estimation performance of all available approaches.

In total, four of six available estimators exhibit to be the best estimator at least once during our three hour experiment. Additionally, it is not clear in advance which estimator will perform how well, especially as some estimators also have the tendency to be very unstable. Hence, *SARDE* acts as an ensemble estimator able to combine the best from all estimators and compensate for the weaknesses of the other approaches. In other words, the aim of *SARDE* is therefore to successfully learn and adapt to the changing performance of the estimators in order to be able to always select the best approach for each scenario. In addition to that, we observe that some approaches are very susceptible to changes in their parameter settings [29]. Therefore, by adapting these parameters to the applied scenario, *SARDE* could even improve the performance beyond the current best method without the need for human supervision or expert knowledge.

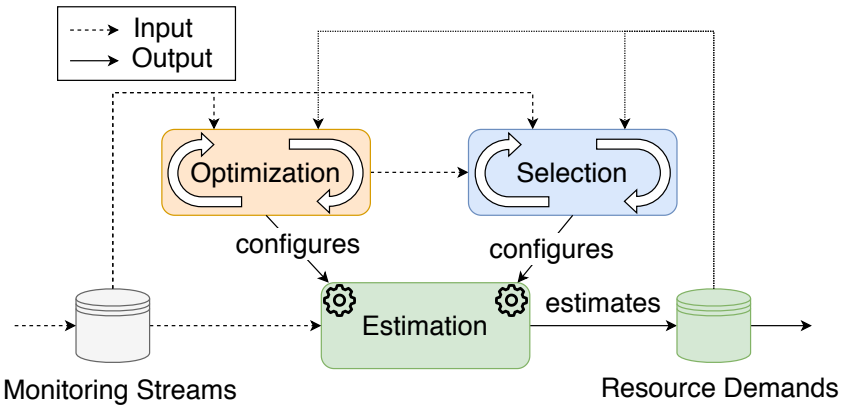

Fig. 2. High-level overview of the *SARDE* approach.

4 OVERVIEW

This section gives a high-level overview of *SARDE* as illustrated in Figure 2. More details on the implementations and communication of the components can be found in Section 5.

First, *SARDE* comprises two running databases: One containing monitoring streams from the system under study, another storing the sequence of resource demand estimations made over time. Next to the databases, *SARDE* continuously runs the estimation engine, performing periodic resource demand estimations based on the continuously updated monitoring streams. The estimation engine offers different configuration interfaces, like the specific approach to use or the parameter settings of the individual approaches. The resulting estimations are then stored in the resource demand database. From there, external processes (e.g., an auto-scaler [2] or a performance model extractor [87]) can retrieve the latest resource demand estimations. On top of that, *SARDE* consists of two interacting feedback loops: *Optimization* and *Selection*.

The optimization process deals with parameter tuning (e.g., the aggregation interval or the monitoring window) of the individual approaches. To that end, monitoring data from the system as well as the corresponding resulting estimations are utilized. The optimization then specifically tailors the parameters of each available estimation approach to the specific system under study in order to minimize the resource demand estimation error.

The selection process utilizes the same data as the optimization process. Instead of optimizing the parameters for all approaches, however, the selection process fits a machine learning model

predicting which approach to select for a given situation. This is done based on specific features of the monitoring data, like e.g., the average CPU utilization, or based on properties of the system, like e.g., the number of servers or workload classes. Based on these features, the selection process can then select the best-suited estimation approach for the given situation.

As the optimized parameter settings influence the performance of the individual approaches, these settings have to be considered while training the machine learning model and are therefore directly fed into the selection process. The selection itself interacts only indirectly with the optimization, as the process has an impact on the resulting resource demand estimations in the resource demand database, which is in turn an input to the optimization loop. In addition to utilizing the historical data, both processes perform additional computations and resource demand estimations in order to explore the space of all possible configurations.

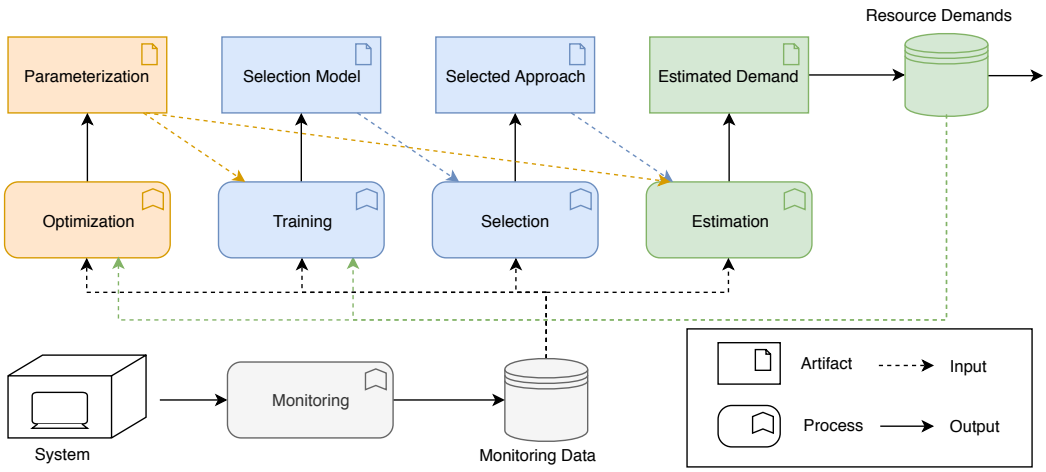

Fig. 3. Conceptual flowchart of the different *SARDE* processes.

5 APPROACH

In this section, we describe the two feedback loops presented in Section 4 and how communication between them is organized in more detail. As both the optimization process and the selection process interact with the estimation engine as shown in Figure 2, synchronization and communication is required. In order to keep all sub-systems of *SARDE* up-to-date, we introduce a set of semaphore artifacts. These artifacts can only be written by one respective process but may be read by all other processes. This way, it can be ensured that the different feedback loops do not block each other during execution while using the most recent version.

Figure 3 depicts the five different activities running in parallel: (1) monitoring, (2) parameter optimization, (3) selection model training, (4) approach selection, and finally (5) resource demand estimation. In the following, we will discuss each of the individual processes in more detail.

5.1 Monitoring

As the required resource demand estimation approaches require both system- and application-level monitoring, the monitoring engine has to monitor application-level metrics (like throughput and response time per workload class) and system-level metrics (e.g., average CPU-utilization per instance) live from the running system. These monitoring streams are then stored in a database

and each entry is assigned a corresponding time-stamp. The gathered data can then be fed into the remaining four processes, each of which requires the information as input.

5.2 Optimization

As explained in Section 2, different resource demand estimation approaches offer several parameters to be tuned. Additionally, some parameters like, e.g., the aggregation interval of the monitoring data (step size) or the measurement window to consider (window size) can be tuned for all approaches. This is done by analyzing the estimation error of individual estimation approaches via cross-validation on the monitoring data gathered on the system. A configurable search algorithm then applies different parameter settings and searches for a (near-)optimal configuration of those parameters for each of the available approaches. Although simple, the optimization still bears many challenges, as the number of different possible configurations rises exponentially with the number of parameters, and as the time available for optimization is limited. The challenge is therefore to utilize an algorithm that is able to find a good parameter configuration using a small number of exploration runs.

The applied self-tuning algorithm is generally abstract and works for any generic parameter providing a minimum and a maximum value. The Stepwise Sampling Search (S3) (also referred to as Iterative Parameter Optimization [63]) was developed by Noorshams et al. [62] in the context of regression model optimization. Here, we utilize this algorithm in order to optimize the parameters of our resource demand estimation techniques. This adaptation was already presented in our prior work [29].

The S3 algorithm can be configured by three hyper-parameters: The number of splits per parameter k, the number of exploration points considered per iteration n, and the maximum number of iterations j_{max}. Noorshams et al. [63] show that the total complexity of the algorithm is given by $O(j_{max} \cdot n \cdot (k+2)^l)$, where l is the number of parameters that are optimized simultaneously. Therefore, S3 offers good control over the trade-off between run-time and solution quality by tuning its hyper-parameters. Additionally, it is possible to optimize an arbitrary number of parameters simultaneously. This is important as inter-parameter influences, i.e., one parameter value influencing the optimal value of the other can be taken into account. However, it has to be noted that the number of parameters to be simultaneously optimized heavily influences the computational complexity. Note that S3 is just one possible search algorithm. Technically, all algorithms focusing on modeling or optimizing configurable software systems [31–33, 76, 97] are applicable as well.

Although this step can be executed offline using a large trace database, the optimization is usually more effective when optimizing for a specific kind and type of system. Additionally, as the system under study evolves and/or the amount of available monitoring data increases, the parameters need to be adapted continuously. Therefore, the process is periodically re-triggered. However, depending on the chosen algorithm, this process can be very time-consuming, running for multiple hours or even days for huge systems. Therefore, the execution is triggered rather seldom.

5.3 Training

The third step is the process of training the estimation approach selector. The selection process in Figure 2 is split into two activities as the selection itself is executed far more frequently than the training of the selection model. During the training phase, a model is learned which is able to predict the best suitable approach for the given estimation problem. This model is then stored as the Selection Model, which is used by the actual selection process.

5.3.1 Problem Formalization. The problem of selecting the best algorithm for a specific problem instance was also formulated by Rice [70] as the *algorithm selection problem*. Based on this work,

Smith-Miles [78] formalized the following four components for modeling a selection problem: (i) the problem space, (ii) the feature space, (iii) the algorithm space, and the (iv) performance space. In this work, we can translate this to the task of selecting the best-suited resource demand estimation approach as follows:

- The problem space P represents the measurement traces available for estimation,
- the feature space F contains the characteristics of each trace, as described in Section 5.3.3,
- the algorithm space A is the set of available resource demand estimators, and
- the performance space Y represents the mapping of each algorithm to the estimation error.

For a given measurement trace $p \in P$ with characteristics $f(p) \in F$, the objective is to find a selection mapping $S(f(p))$ into the algorithm space A, such that the selected algorithm $\alpha \in A$ minimizes the performance mapping $y(\alpha(p)) \in Y$. The task of the model learning is to find the function S, mapping each possible trace characteristic to the selected algorithm, while the actual selection process (see Section 5.4) is executing $S(f(p))$.

5.3.2 Data set. Note that the training procedure itself can be done either online or offline. This decision mainly influences what data is available during the training phase to extract knowledge from.

Offline training. We refer to offline training as training that is performed once, using a variety of systems and configurations. Based on this set, one can apply all available approaches to the different training sets and use the feedback from those runs to determine which approach is best suited for the specific problem instance. This information, together with a set of descriptive features is then given to a machine learning algorithm, which learns a model from all training sets, extrapolating the relationship between the different features and the best-suited approach. We call this resulting model the selection model. This approach was proposed and partially evaluated in our prior works [30]. Naturally, the accuracy of this approach highly benefits from an increasing amount of training data and a high similarity of the training systems to the current problem instance.

Online training. Offline training has the disadvantage of being trained before being applied to the system under study. Therefore, in online training, we continuously monitor the current system and the performance of the different approaches, as these can also serve as training samples for our selection model [42]. Furthermore, the performance of the individual approaches changes if the optimization process described in Section 5.1 adapts the parameter settings of the respective approaches. If so, the training must be repeated for the newly found parameterization, which can be cost-intensive for the offline data set. However, online learning has the disadvantage that the trained model is prone to over-fitting to a specific system and cannot adapt very well to changes in the configuration or the structure of the system under study. This is due to the drastic reduction of training data in comparison to the larger data set used in offline training.

Hybrid training. As a consequence, we introduce hybrid training, a combination of both offline and online training in this work. The idea of hybrid training is to utilize the training data sets as applied in offline training, but iteratively adding online data from the system under study to the data set and periodically re-triggering the training process. Therefore, the training process is able to adapt to the feedback of the running system, while also maintaining robustness towards major changes of the respective system.

5.3.3 Features. Another central aspect of all machine-learning-based approaches is the feature set used for training. This section contains the list of features we extract from each monitoring trace. These features capture certain characteristics of the input traces that we deem useful for judging which algorithm would be most suitable for estimating that respective trace.

The machine learning algorithms are heavily dependent on those features and a careful selection, as well as the right amount, is crucial for a satisfactory outcome. Since machine learning algorithms try to distinguish between different classes of traces, too many features can actually be harmful. A *trace* refers to one training example of our data set. A trace usually consists of a set of a time series, e.g., of the CPU utilization of each resource, the response time, and the arrival rate of each request of the respective workload classes. The CPU-utilization measures the average utilization of the CPU for a certain interval, the response time contains the response time of each request and the arrival rate holds the number of incoming requests for a certain interval. These traces are then given to the estimation approaches for their estimations. For each trace, we want to create a feature representation y that captures the characteristics of this trace.

Next to the time series itself, we have some general meta-information about the traces, including the number of resources (e.g., number of CPUs and/or CPU cores) and the number of different workload classes. For example, Spinner et al. [79] showed that the number of workload classes has a direct impact on the performance of the estimators. This meta-information is therefore also added to the feature set.

Another big impact on the performance of estimators is the utilization of the system [79]. It is therefore useful to include information about the average utilization of the available resources as well as the minimum and the maximum utilization. Therefore, it seems reasonable also to extract statistical information about the time series of each trace.

However, it does not seem useful to average this information over *all* resources. Especially, since different workload classes are known for stressing each resource differently. We, therefore, define a set of statistical features to extract utilization information for each individual resource, together with information about the arrival rate and response times of each workload class, and concatenate them to one feature vector y.

The extracted statistical features for a time series $T = (d_1, \ldots, d_n)$ consisting of an ordered set of data points are as follows:

- The number of data points: $n = |T|$
- The arithmetic average: $\overline{T} = \frac{1}{n} \sum_{i=1}^{n} d_i$.
- The geometric average: $\hat{T} = \left(\prod_{i=1}^{n} d_i\right)^{\frac{1}{n}}$.
- The standard deviation: $\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (d_i - \overline{T})^2}$.
- The quadratic average or root mean square: $x_{\text{rms}} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} d_i^2}$.
- The minimum value: $T_{min} = \min T$
- The maximum value : $T_{max} = \max T$
- The kurtosis, a measure for the tailedness of the graph of T (see [91]): $k = \frac{\frac{1}{n} \sum_{i=1}^{n} (d_i - \overline{T})^4}{\left(\frac{1}{n} \sum_{i=1}^{n} (d_i - \overline{T})^2\right)^2} - 3$.
- The skewness, a measure for asymmetry (see [40]): $s = \frac{\frac{1}{n} \sum_{i=1}^{n} (d_i - \overline{T})^3}{\left[\frac{1}{n-1} \sum_{i=1}^{n} (d_i - \overline{T})^2\right]^{3/2}}$.
- The 10^{th} percentile: $l = P_{10}(T)$
- The 90^{th} percentile: $u = P_{90}(T)$

This results in a total of eleven statistical measures. Given that these are calculated for each resource and twice for each workload class (for arrival rates and response times), and add in the meta-information about the number of resources and workload classes available, the total number of features amounts to $|y| = 2 + 11 \cdot r + 22 \cdot w$, with r being the number of resources and w being the number of workload classes in the training set.

One advantage of the selected features is that they are fairly easy and fast to compute. In addition, most of the features are standard statistical measures that are easy to comprehend as a user. Exceptions might be the kurtosis and the skewness metrics; however, those are common metrics in time series analysis [40, 91] and are therefore included, because all traces are time series. In previous works, we also experimented with other and more features [30], including the correlations and the co-variances between the traces, the variance inflation factor, and information about the statistical distributions. While it might seem useful to include further features into the training, these features are costly to calculate and therefore greatly increased the required selection time [30]. As the respective features did not significantly impact the prediction accuracy, we decided to settle on the final feature list presented above. We also excluded any feature probing techniques [39, 43] as we consider the performance impact too high. Additionally, removing any more features from the above list negatively influenced the selection results, while offering only an insignificant run-time advantage.

5.3.4 Labels. After acquiring the feature vector per trace, one can execute all resource demand estimators on the given trace and then use the resulting estimation error as labels in order to train a machine learning algorithm. A selection engine can then be built by training different regression models, each predicting the error of individual estimators and then choose the one with the best expected error [4]. However, in the following, we work with a classifier-based approach. In order to do so, we compare the error values of each estimator in order to label each feature set with the value of the best algorithm. During the selection, the predicted label of the classifier can be viewed as the approach expected to perform best. This way, only one classifier model needs to be trained and executed, which saves computation time during online execution.

What remains is the determination of the estimation error of each approach during training. If available, the real estimation error can be used, if the training set contains a set of artificial or specifically monitored traces. However, this will not be feasible for many traces, for example, during online training. As the real resource demand is per definition unknown to *SARDE*, we have to rely on the internal error calculation based on cross-validation. The validation error used in this work is explained in more detail in Section 6.3.

5.4 Selection

After the training process produced an accurate selection model, the selection process analyses the type and structure of the monitoring streams and uses the provided selection model to make an informed decision about which approach to use for estimation. Simply put, the acquired machine learning model is utilized and its prediction for the best-suited estimator is applied. This process was deliberately split from the training process, as this process can use the same selection model multiple times in order to update the selected approach based on changes in the system or the monitoring streams.

Figure 4 illustrates an exemplified timeline, visualizing the five processes running in parallel. While monitoring is a continuous process, the estimation is executed quite frequently, with the more computationally expensive procedures running slower and fewer iterations. Note that this is just an exemplary configuration, the actual intervals of *SARDE* can be tuned by the user. Furthermore, the arrows of the respective colors show, how the results of the particular process influence the other running processes. We observe that for example, a finished training process updates the selection model used for the next selection process that has not started yet. This model is then used until it gets updated by a subsequent training iteration. Similarly, the output of the selection process, the selected approach to use for estimation, is applied for all subsequent estimation runs as long as the selection is not updated. It is furthermore shown, how the optimization results influence the next

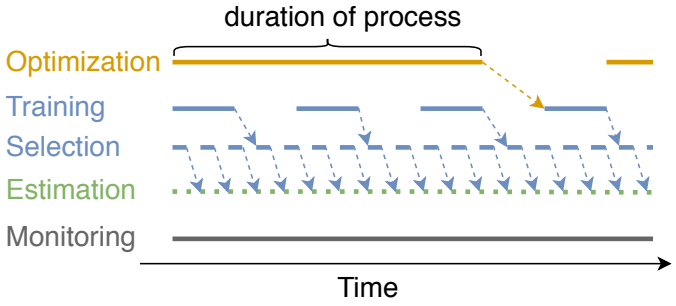

Fig. 4. Exemplified timeline visualization.

training process. After a successful optimization, the optimization results take a while to come into effect at the actual estimation, as the estimation uses the old parameterization until the training with the new parameterization is finished and the newly parameterized approaches are selected for estimation. This has the advantage of protecting the continuous estimation from negative effects by a disadvantageous optimization run, as the training process is able to double-check and filter the respective approaches if necessary. However, the cost of this approach is the delay between a finished optimization and its parameterization coming into effect.

5.5 Estimation

The most frequent process is the actual estimation process. Its frequency mainly depends on the variability of the system and the monitored traces, as well as the quality of the estimated resource demands itself. Upon execution, the estimation process loads the approach selected by the selection process and updates it with the optimized parametrization by the optimization process, if available. Then, the estimation is executed on the newest monitoring data. Note that, as depicted in Figure 4, multiple subsequent estimation executions might be performed using the same approach. This is on purpose, as the monitoring data is updated between those executions, which impacts the estimation result. To that end, all process executions always utilize the most recent monitoring data available at the start of each process.

6 EVALUATION

In this section, we evaluate and analyze the performance of *SARDE* concerning various aspects. To this end, we pose ourselves the following research questions:

RQ1 *What is the gain of continuously repeating the estimation?*
RQ2 *What is the impact of applying optimization, selection, and both combined to the repeated estimation?*
RQ3 *What is the overhead of applying these techniques?*

In the following, we will describe and analyze the experiment series we conducted in order to answer these questions.

6.1 Experiment Setup

We designed two different experiments to validate the accuracy of our approach. First, we applied a common data set in Section 6.1.1 consisting of a set of micro-benchmarks executed on a system and already applied in a variety of previous studies [27, 29, 30, 79]. Second, we extend this analysis

by adding a long-term measurement trace from a realistic application, described in more detail in Section 6.1.2.

6.1.1 Micro-benchmark data sets. This data set consists of a set of measurements obtained by executing micro-benchmarks on a real system. A set of 210 traces, each with approximately one hour run-time, was collected. The micro-benchmarks generate a closed workload with exponentially distributed think times and resource demands. The think times themselves were set to fit the targeted load level of each specific experiment. As mean values for the resource demands, we selected 14 different subsets of the base set [0.02s; 0.25s; 0.5s; 0.125s; 0.13s] with a varying number of workload classes $C = \{1; 2; 3\}$ and target load levels $U = \{20\%; 50\%; 80\%\}$. The subsets were arbitrarily chosen from the base set. This way, we can ensure that the resource demands are not linearly growing across workload classes. Additionally, the subsets intentionally contained cases where two or three workload classes had the same mean resource demand.

6.1.2 Realistic Application. In addition to the micro-benchmark data sets, we conducted a long-term study of a realistic, containerized application measured on a real system. However, in order to evaluate the accuracy of the approach, it is necessary that we know the exact resource demands to be estimated. Therefore, we developed a synthetic application that offers three different services via a REST API that perform a prior defined load for each service call. For the following of this section, the first workload class (WC1) performs an exponentially distributed load with a mean of 0.01s, the second workload class (WC2) performs an exponentially distributed load with a mean of 0.03s, and the third workload class (WC3) performs a normally distributed load with a mean of 0.005s and a standard deviation of 0.001.

In order to evaluate the adaptability of the individual approaches in comparison to *SARDE* with respect to different influence factors, we varied both the load intensity and the distributions of the individual workload classes. Figure 5 depicts the load intensity, i.e., the number of requests per second of each workload class as a stacked line chart. The load is intentionally noisy and strongly varies over time. Additionally, the relative share of the different workload classes changes. As the different workload classes each have different resource demands, the resulting utilization curve is non-obvious.

In order to reflect a realistic cloud setup, we deployed the application inside an Ubuntu 18.04 Virtual Machine (VM) associated with 1 pinned CPU core and 4 GB RAM running on an HPE ProLiant DL160 Gen9 server equipped with an Intel® Xeon® CPU E5-2640 v3 @ 2.60GHz and 32 GB RAM total RAM, using a KVM hypervisor. The load driver generating the REST requests was situated on another host in the same cloud in order to isolate the performance behavior and also include the network overhead per request.

6.2 Evaluation Metrics

In this section, we describe the metrics we use during our evaluation of *SARDE*. We focus mainly on execution time and estimation accuracy. All execution times were measured using the publicly available Java implementation of *SARDE*[1] and version 1.1 of the underlying LibReDE engine[5] by relying on the internal time measurement. All reported experiment times were conducted on a Windows 10 machine using an Intel® Core® i7-6600U CPU @ 2.60 GHz and 16 GB RAM.

For accuracy, we evaluate the estimation error ϵ_E per approach by averaging the relative estimation error of each workload class:

[5]This is also the version endorsed by SPEC research. Available at https://research.spec.org/tools/overview/librede.html

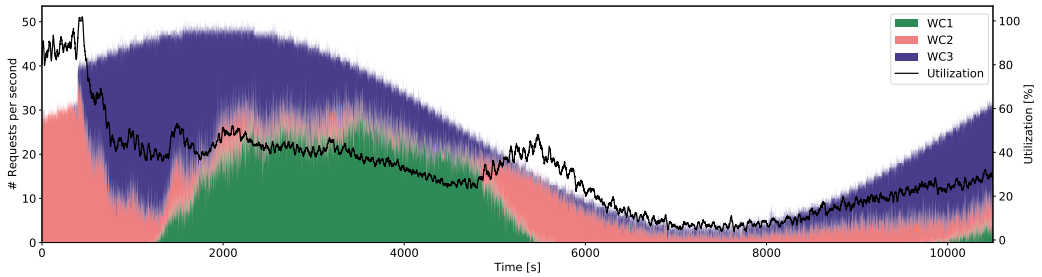

Fig. 5. Server utilization and throughput of the different workload classes of our monitored application over time.

$$\epsilon_E = \frac{1}{C} \sum_{c=1}^{C} \left| \frac{\tilde{D}_c - D_c}{D_c} \right|, \tag{1}$$

where C is the number of workload classes, $\tilde{D}_c$ is the resource demand estimate for workload class c, and D_c is the real resource demand of class c.

6.3 Configuration

There are several generic and configurable parts of the *SARDE* approach described in Section 5. In this section, we describe the specific configurations that we applied for the presented evaluation.

First, we concentrate on the estimation of the resource demand error. As all evaluations and optimizations performed by *SARDE* rely on the internal estimated error, it is crucial that the applied error validation closely resembles the actual resource demand error. Recall, that *SARDE* does not have the real resource demands available for validation as they are naturally unknown to *SARDE* during operation. Therefore, *SARDE* calculates the estimated validation error ϵ_V using the estimated relative response time error ϵ_R and the estimated absolute utilization error ϵ_U. This error is then used for all internal validation processes. The two error functions are defined as follows:

$$\begin{aligned} \epsilon_R &= \frac{1}{C} \sum_{c=1}^{C} \left| \frac{\tilde{R}_c - R_c}{R_c} \right|, \\ \epsilon_U &= \left| \sum_{c=1}^{C} (X_c \cdot \tilde{D}_c) - U \right|, \end{aligned} \tag{2}$$

with C being the number of workload classes, R_c the average measured response time of workload class c over all resources, $\tilde{R}_c$ the predicted average response time using Mean Value Analysis [5] based on the estimated resource demands, X_c the measured throughput of workload class c, $\tilde{D}_c$ the estimated resource demand of workload class c, and U the average measured utilization over all resources.

Using both errors, we can compute the compound validation error ϵ_V as a weighted sum of ϵ_R and ϵ_U:

$$\epsilon_V = \frac{1}{2} \min(1, \epsilon_U) + \frac{1}{2} \min(3, \epsilon_R). \tag{3}$$

Note that we bound the utilization error at 1 and the response time error at 3. This is necessary, since both errors are effectively unbounded, and therefore might dominate the other error during the validation. The values are chosen, as during capacity planning response time errors are usually

acceptable to be higher than utilization errors [58, 60]. Apart from that, both ϵ_U and ϵ_R are currently weighted 1:1. However, this configuration could be adapted if a user is more interested in minimizing the respective error value.

For the online analysis of the realistic application, we use an estimation interval of 70 seconds, a selection interval of 170 seconds, a training interval of 700 seconds, and an optimization interval of 1000 seconds in order to keep a reasonable amount of repetitions for each activity during the experiment. Based on our results in Section 7.1.1 we applied a random forest classifier as the selection algorithm. Concerning the S3 optimization algorithm, we use 5 splits, 4 exploration points, and 5 iterations for single parameter optimizations. For multi-parameter optimizations, we need to rely on 1 split, with 2 exploration points, and 2 iterations in order to reduce the algorithmic complexity.

7 RESULTS

In this section, we present the results obtained from the experiments outlined in the previous section. First, Section 7.1 focuses on the analysis of the selection process, while Section 7.2 analyses the performance of the optimization algorithm. Finally, we put both aspects together and analyze the performance in Section 7.3.

7.1 Selection

This section presents results concerning the selection of the best-suited estimation approach. The first section compares different selection algorithms with each other using our set of micro-benchmark experiments. Then, we analyze the performance of continuous training and selection over time in our realistic application.

7.1.1 Micro-benchmarks. To compare the different selection algorithms with each other, we utilize the set of micro-benchmarks as they represent a wide variety of different scenarios in their characteristics. Therefore, we can get a holistic analysis of the performance of each selection algorithm.

We include a Decision Tree (DT) [7], AdaBoost [34], Random Forest (RF) [6], Logistic Regression (LogReg) [15], Support Vector Machine (SVM) [14], and Neural Network (NN) algorithm. The neural network is a sigmoid perceptron consisting of two fully connected inner layers, an input layer, as well as an output layer for the selection. We used 100 neurons in total and applied the back-propagation algorithm based on the least-squares error for learning. For all algorithms, we relied on the implementations provided by the SMILE [50] library. For a fair comparison, all algorithms were used in their default parameterization. Furthermore, we add a random classifier always choosing a random approach as a baseline. We split the 210 available scenarios into 168 training and 42 validation traces. The machine learning algorithms were trained with the 168 training sets and Table 1 shows their performance on the 42 remaining validation sets.

Table 1. Comparison of different selection approaches using the micro-benchmark set.

Algorithm	Random	DT	AdaBoost	RF	LogReg	SVM	NN
Avg. estimation error	43.5%	22.5%	19.8%	17.9%	25.0%	18.0%	18.0%
Hit-rate	16.7%	52.4%	66.7%	71.4%	42.9%	59.5%	59.5%
Train time	–	211.1s	241.1s	533.0s	305.6s	262.3s	243.2s
Avg. estimation time	1.4s	1.1s	2.0s	2.1s	1.5s	1.5s	13.4s

The first line of Table 1 shows the average resource demand estimation error on the 42 remaining traces when applying the respective selected approach. We observe that—as expected—the random classifier has the worst performance; the decision tree and logistic regression algorithm also fall behind. However, AdaBoost, Random Forest, SVM, and NN all perform comparatively. Random Forest has the best accuracy, with an average estimation error of 17.9%. This is impressive if you consider that the average minimum error of all approaches (and therefore the de-facto perfect result) is 17.6%. Therefore, the performance of the approaches chosen by random forest is just 0.3% worse than the theoretical optimum. These results are in line with the hit rate, i.e., the relative share of scenarios in which the algorithm selects the best approach. Again, Random Forest outperforms all other approaches with a hit rate of almost 72%, while a random classifier baseline achieves only 16.7%.

When analyzing the training time, we observe that all approaches take between 4 and 10 minutes for completing the training with a training corpus of 168 traces. Here, random forest takes the longest time for training (almost 10 minutes), while all other approaches terminate within 4 - 5 minutes. However, considering the large amount of the training set (168 measurement hours), we find a training time of 10 minutes more than acceptable for online use. Similarly, the average estimation time (including feature extraction, selection, and the estimation process itself) is sufficiently fast. Most approaches finish between 1 and 2.5 seconds, only the NN approach requires up to 15 seconds of estimation time. As typical estimation windows are usually in the range of several minutes, these time scales are more than sufficient. One interesting observation is that the random baseline, despite the lack of an actual selection, is not the fastest of the approaches. This undermines our observation that the most dominant time factor for the average estimation time is in fact not the selection algorithm itself (excluding NN), but the estimation time of the selected approach.

Based on our results, for the remainder of this paper, we concentrate on the Random Forest algorithm with a parameterization of five trees (`ntrees`), two features per node decision (`mtry`), a maximum leaf node size of one (`nodeSize`), applying the Gini splitting criterion (`rule`) and using feature sampling with replacement (`subsample`).

7.1.2 Realistic application. Following the broad analysis of multiple validation scenarios, we now analyze the performance of the random forest selection for our realistic application. For this, we look at the continuous training and selection of the algorithm over time. Figure 6 shows the estimation error for every approach over time. The activities are depicted in the time diagram in the top of Figure 6. The red bars indicate time and duration of training phases, the orange bars indicate selections accompanied by an abbreviation of the chosen approach and the blue bars indicate the regularly repeated estimations of all approaches.

In each training phase, the chosen selector algorithm (Random Forest in this case), was trained on all available offline traces from the previous section, plus the additional experience from the currently running trace (hybrid training). Therefore, the first trained model only has the micro-benchmark data set available as training data set. The second one has the micro-benchmark set, plus the first 700 seconds of experiment time, and so on. As we had a maximum of three different workload classes ($r = 3$) and one resource ($w = 1$) in the training set, the feature vector y had a length $|y|$ of 57 for training (compare Section 5.3.3.

We observe that the estimates, as well as the corresponding accuracy of each individual approach, are massively changing during the experiment. There is therefore a good rationale for continuously repeating the resource demand estimations, and simultaneously for changing the applied approach (see Section 3). This also answers our first research question (**RQ 1**).

Additionally, we observe that the *SARDE* approach (blue) jumps between different respective approaches. While *SARDE* needs a while to learn and adapt to the current trace (before 2000), it

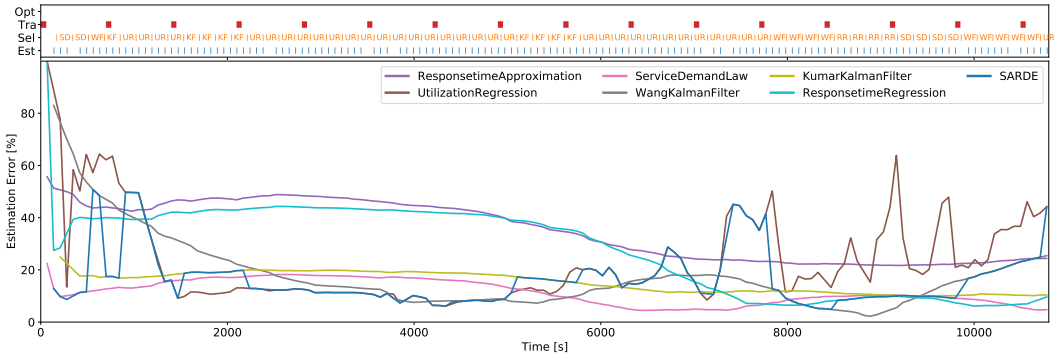

Fig. 6. Results showing the real error of running the selection over time.

then is able to predict and select among the best performing approaches until the environment changes and the approach decreases in accuracy (starting at 6000). In reaction to this development, another approach is chosen at around 8000 until its performance decreases as well.

Table 2. Overview on the quality of selected approaches using the realistic application.

Approach	Average Rank	Accuracy (%)	Accuracy Loss (%)
ServiceDemandLaw	2.02	11.52	3.11
ResponseTimeApproximation	5.47	35.04	26.63
ResponseTimeRegression	3.69	27.94	19.53
WangKalmanFilter	2.94	18.74	10.33
UtilizationRegression	3.64	23.84	15.43
KumarKalmanFilter	3.21	15.17	6.91
SARDE	2.82	16.88	8.64
Random	3.08	18.49	10.15

In the following, we will analyze Table 2 for more detail on the selection results. Table 2 shows the average rank of each selection approach, together with its average total accuracy loss, i.e., the average difference of the relative estimation error of the given approach in comparison with the current best approach. We observe that Kumar Kalman Filter and Service Demand Law both have relatively low ranks and a small accuracy loss in comparison to other approaches. The response time approximation has a particularly high accuracy loss, as its performance is consistently worse than any of the other approaches.

SARDE is able to achieve an average rank of 2.82 with only 8.6% of accuracy loss towards the theoretical optimum. Compare this with a baseline approach of the random classifier, which achieves an average rank of 3.08 together with an accuracy loss of 10.2%. Note that it is not possible to simply choose service demand law as the best approach for example, as the knowledge about the performance of the individual approaches is not known prior to execution. Instead, the self-adaptive features of the selection approach of *SARDE* enable it to constantly monitor the performance of the individual approaches and switch between the most promising approaches. Therefore, *SARDE* is able to learn from and adapt to a scenario without any prior knowledge or training for that environment.

7.2 Optimization

After analyzing the selection process in detail, this section now focuses on the optimization. Similar to the previous section, we first analyze the set of different micro-benchmarks representing a wide variety of test applications and then concentrate on a more in-depth analysis of our realistic application.

7.2.1 Micro-benchmarks. The focus on this section is to show the potential benefit of parameter optimization on our trace data set. Naturally, not all estimation approaches have the same set of parameters available. For example, the two Kalman-Filter-based approaches Kumar Kalman Filter (KF) and Wang Kalman Filter (WF) have five approach-specific parameters that can be tuned. On the other side, other approaches, like Service Demand Law (SD) or Response Time Approximation (RT) do not have any parameters to fine-tune the respective approach. Table 3 shows the available optimization parameters for *SARDE* as well as the respective lower and upper bounds.

Table 3. Overview over available optimization parameters.

Parameter name	Lower bound	Upper bound	Supported approaches
Step size	10	360	SD, RT, UR, RR, WF, KF
Window size	1	60	SD, RT, UR, RR, WF, KF
Initial bounds distance	0.0	0.1	WF, KF
Bounds factor	0.0	1.0	WF, KF
State noise covariance	0.0	2.0	WF, KF
Observe noise covariance	0.0	0.1	WF, KF
State noise coupling	0.0	2.0	WF, KF

The only two parameters that are common to all approaches are concerned with the input processing of monitoring data. The *step size* describes the aggregation interval, i.e., the interval for which all monitoring measurements are aggregated, and serves as the minimal time unit for each estimation approach. Additionally, the *window size* defines the memory of each approach, i.e., the number of steps that are considered for each estimation approach. For example, if the step size is 60 seconds, and the window size is 60, then only the last $60s \cdot 60 = 3600s$ of measurements are considered for the estimation. Hence, the specific tuning of both parameters is more dependent on the individual trace than to the specific approaches, as it is more a configuration parameter (i.e., a parameter that needs to be set based on external requirements), than an optimization parameter (i.e., a parameter that can be freely chosen to optimize performance). We observe this effect also in Figure 7.

Therefore, Table 4 focuses on the parameters of the two Kalman-Filter-based approaches Kumar Filter (KF) and Wang Filter (WF). Table 4 shows the performance of our optimization tuning the five tunable parameters *initial bounds distance*, *bounds factor*, *state noise covariance*, *observe noise covariance*, and *state noise coupling* using the bounds defined in Table 3. In order to evaluate the results on the micro-benchmarking training sets, we split the 210 traces into 168 training traces and 42 validation traces. The training algorithm optimized the parameter of the training traces, while Table 4 shows the performance of the remaining 42 validation traces.

We observe that the default parameterizations (as proposed by the default configuration of the implementations) are sub-optimal for both Kalman filter scenarios. Both estimators could significantly improve the estimated error on the validation set. However, it is interesting that the KF, which performs already significantly better than WF in its default configuration, also profits

Table 4. Estimation error and chosen configuration parameters of our validation benchmarks before and after optimization.

Algorithm	KF-Default	KF-Optimized	WF-Default	WF-Optimized
Optimization time (s)	–	6456	–	8878
Average estimated error	0.273	0.227	0.823	0.752
Relative improvement	–	16.7 %	–	8.6 %
Parameter values:				
Initial bounds distance	0.0001	0.0	0.0001	0.1
Bounds factor	0.9	0.75	0.9	1.0
State noise covariance	1.0	0.0	1.0	1.0
Observe noise covariance	0.0001	0.1	0.0001	0.0
State noise coupling	1.0	2.0	1.0	1.0

more from the optimization. Although the absolute error reduction is greater for the WF, the relative improvement for the KF (16%) is almost double the relative improvement for the WF. In addition, we note that, although KF is slightly faster than WF, both optimizations take comparatively long to optimize as they need to take all 168 training traces into account. To summarize, we can say that the optimization finds effective parameter optimizations, even if the validation traces are unknown to the algorithm.

7.2.2 Realistic application. After analyzing the performance of our optimization procedure on the different micro-service benchmarks we now continue on our realistic application data set. As already discussed in the previous section, most approaches are limited to only two configurable parameters: the step size and the window size. Therefore, we configure the optimization used in the previous section to optimize the Kalman filter parameters for the two Kalman filter approaches, while focusing on step size and window size for all other approaches. (See Table 3.) As these two parameters heavily influence each other, the optimization combines both into one parameter that only changes the window size relative to the respective step size.

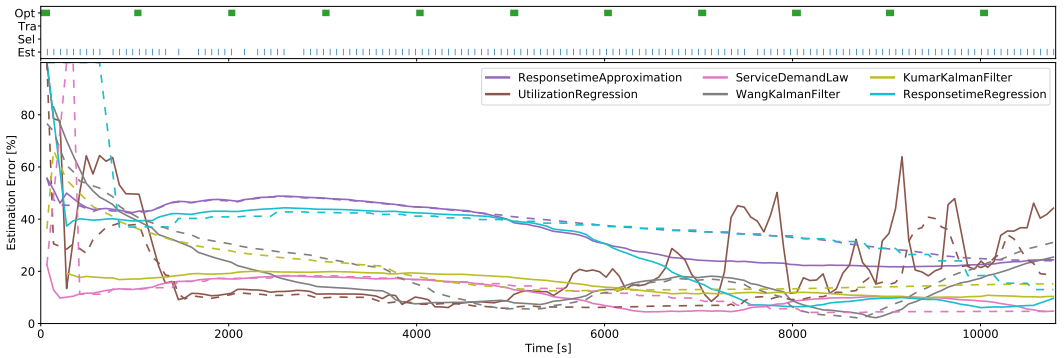

Fig. 7. Results showing the real error of running the optimization. Drawn lines represent the original error, dotted lines are the optimized versions.

Figure 7 depicts the estimation accuracy of the different approaches over time. In addition, the dashed lines of each color represent the accuracy of the optimized approach. A new parameterization

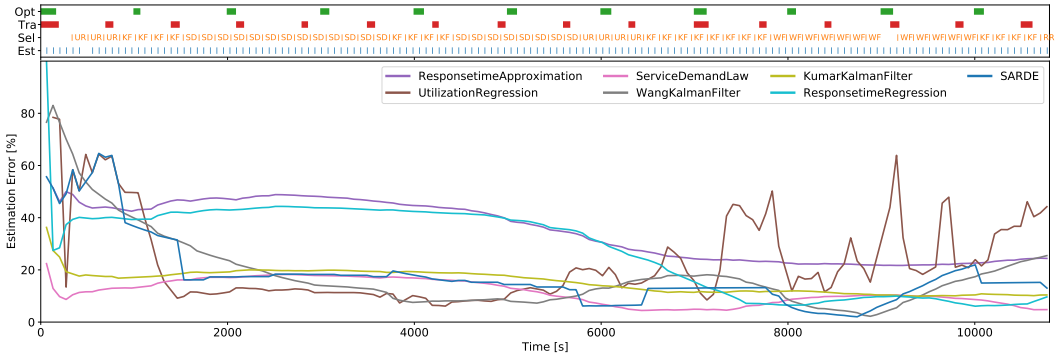

Fig. 8. Results showing the real error of running *SARDE* over time.

comes into effect at the first estimation interval (blue) after the end of each optimization interval (green). Every optimization run is able to utilize more data, as all collected data from the previous trace is used.

First, we observe that not all approaches (purple, turquoise) are able to profit from the parameter optimization. This is due to the limitations of the optimizable parameter set as discussed above. On the other hand, there are other approaches (green, pink) that can profit greatly from changing the parameters. However, in summary, Figure 7 does unfortunately not conclusively prove or disprove the applicability and the effect of the optimization process. It can certainly affect the performance of the algorithms in both ways; it is therefore important to analyze the interplay between the optimization and the selection component. If the correct approaches are chosen, the optimization can help to improve the current approaches, while its negative effects are mitigated by the selection process. We therefore analyze the interplay of both processes in the following section.

7.3 Combination

Finally, we now combine the two processes of optimization and selection in order to evaluate their interplay as intended by the *SARDE* approach. For this, we focus solely on the realistic application data set, as the optimization procedure and the selection interplay can only be analyzed over time which is infeasible for the 210 available micro-benchmark traces.

Analogously to the previous sections, Figure 8 depicts the estimation errors of the individual approaches over time. The individual approaches remain unchanged in comparison to the previous experiments. However, we include the blue estimation line that represents the *SARDE* estimation. We observe that *SARDE* is again efficiently able to choose between the different available selection approaches as already seen in the analysis of Section 7.1. In addition to that, however, the blue estimation line now deviates from the standard approach estimations as the parameter optimizations change the performance of the estimations.

In the first half, *SARDE* shows some degrees of instability observable from frequent changes in the selected approaches as well as sudden spikes in estimation error. However, as soon as a spike occurs, the self-adaptation mechanisms counteract that behavior by changing the chosen approach and/or the applied parameters. Therefore, towards the end of the trace, the stability gradually increases. Additionally, we observe that at different points in time, the blue estimation line exhibits a lower estimation error than any of the other approaches. This is possible, as the parameter optimization process gradually adapts to the specific properties of the trace and learns to fine-tune the estimation approaches towards that.

Table 5. Summary on selected approaches executing *SARDE*.

Approach	Number of selections
ServiceDemandLaw	23
UtilizationRegression	7
KumarKalmanFilter	20
WangKalmanFilter	12
ResponseTimeRegression	1

Table 5 summarizes the different selections also observable in the top of Figure 8. Similar to our analysis in Section 7.1, we can confirm that the selection algorithm still chooses from almost all estimation algorithms (except the poorly performing Response Time Approximation) in order to adapt to the respective situations.

Generally, it can be said that *SARDE* is able to effectively combine the accuracy gain achieved by optimization with a selection of the most suitable approach for a given situation on the evaluated data set. This enables us to answer **RQ 2**.

7.4 Workload Analysis

The analysis in Section 7.3 helps us to understand the performance of *SARDE* during a continuous estimation. However, another angle at analyzing the given workload is to section it into different intervals. This enables us not only to analyze the performance of *SARDE*, but also to relate it to the workload properties of the respective interval.

Therefore, Table 6 presents the main arrival rate properties of the three workload classes described in Section 6.1.2 together with the performance of *SARDE*, split into ten different intervals. Recall that workload class 1 (WC1) and workload class 2 (WC2) perform an exponentially distributed load with a mean of 0.01s and 0.03s, respectively. In contrast, the third workload class (WC3) performs a normally distributed load with a mean of 0.005s and a standard deviation of 0.001. Therefore, WC3 follows another intensity distribution and is comparatively light.

Table 6. Workload properties of different experiment intervals.

#	Mean			Standard Deviation			Index of Dispersion			*SARDE*
	WC1	WC2	WC3	WC1	WC2	WC3	WC1	WC2	WC3	
1	0.00	21.26	16.59	0.00	9.11	14.67	–	3.90	12.98	0.52
2	10.21	6.88	30.22	7.76	3.08	7.32	5.90	1.38	1.77	0.23
3	24.25	4.42	17.02	3.58	2.40	3.75	0.53	1.30	0.83	0.18
4	24.41	2.36	9.57	3.61	1.88	2.84	0.53	1.50	0.84	0.18
5	13.76	5.35	3.74	6.64	4.69	2.80	3.21	4.11	2.10	0.15
6	0.13	8.66	1.55	0.48	3.55	1.21	1.77	1.46	0.94	0.09
7	0.00	2.17	1.46	0.00	1.32	1.04	–	0.81	0.74	0.13
8	0.00	2.34	2.75	0.00	1.38	1.69	–	0.82	1.04	0.07
9	0.00	4.49	9.87	0.04	1.86	3.58	1.00	0.77	1.30	0.09
10	2.73	5.23	19.87	2.92	2.09	3.06	3.13	0.84	0.47	0.17

Table 6 shows the mean, the standard deviation, and the index of dispersion [16] of each workload class arrival rate in requests per second during the respective interval. The index of dispersion

is calculated by dividing the variance, i.e., the squared standard deviation, by the mean [16]. We observe that all ten intervals show vastly different workload characteristics. For WC1, the intervals vary between 0 and 25 requests per second, together with the standard deviation between 0 and over almost 8. The respective index of dispersion is not defined for mean values of 0, in other cases, the index rises up to almost 6 in interval 2. The other workload classes show similar behavior, with mean arrival rates varying by a factor of 10, and index of dispersion values ranging from as low as 0.8 up to a maximum of almost 13 in interval 1.

In addition, we note that the variations of the three workload classes are independent and spread along the different analyzed intervals. For example, in interval 1 WC3 has the highest Index of Dispersion of almost 13, while WC2 also has a significant amount of dispersion and WC1 is absent. In the following interval, the measured dispersion drops for WC2 and WC3, while in increased to the trace maximum of 5.9 for WC1. Hence, we conclude that all intervals contain vastly different workload patterns and intensity variations.

Therefore, we can now analyze the performance on *SARDE* on the different intervals, to see how the estimator performs. We observe relatively high errors in the first two intervals, while the performance stabilizes starting in interval 3. This can be either due to the massive dispersions shown by WC3 and WC1 in the first two intervals, or due to the fact that *SARDE* has not yet collected a sufficient amount of knowledge over the system. However, after these two critical intervals, we observe that *SARDE* delivers relatively stable estimations, which are not influenced by the distributions of the arrival data. One observation that we might draw is that the task at hand becomes significantly easier if one workload class is removed from the trace, as the accuracy improves for intervals 6–9, where WC1 is mostly absent. In summary, Table 6 shows that *SARDE* shows a reliable and stable performance in our test evaluation.

7.5 Overhead Analysis

Lastly, we evaluate the overhead introduced by applying the *SARDE* approach. Naturally, all self-adaptation and self-optimization processes we introduced in this paper increase the computation effort for estimating the resource demands. Therefore, the question arises whether or not the additional effort is worth spending and to weigh the achieved benefit with the required additional costs.

The additional computation effort can already be seen by analyzing the top part of Figure 8. However, for a more quantitative approach, we summarize the different execution times in Table 7.

Table 7. Overhead analysis of the individual activities.

Activity	Executions	Avg. execution time (s)	Std. dev. (s)	Total time spent (s)
Estimation	154	0.2	0.7	38.5
Optimization	11	113.1	23.4	1244.3
Selection	63	0.2	0.1	11.5
Training	16	96.8	33.4	1548.1

First, we notice that in total 154 resource demand estimations are conducted. On average, each estimation takes around 200 ms to compute, resulting in roughly 39 seconds of computation time spent for the continuous estimation. The second most executed process is the selection of an estimation approach based on an already trained machine learning model. This selection process is similarly cheap as the actual estimation process, resulting in additional 12 seconds of computation effort spent on recommending.

In contrast to executing the selection model, which is comparatively fast, each machine learning training run takes about 97 seconds to complete. Therefore, the training is executed much more sparsely, resulting in a total training time of just under 26 minutes. Finally, the optimization process is as expected the most expensive technique of all self-adaptation processes. However, due to the relatively low amount of 11 executions, just 21 minutes of computation power is spent, as each optimization procedure takes slightly less than over 2 minutes on average.

In total, *SARDE* consumes 2844 seconds or 48 minutes of computation time over the full duration of our three-hour experiment. Given that one is able to efficiently scale the required computation power (as standard in modern-day cloud computing environments), one is expected to utilize well under one CPU-core while running *SARDE* (27% in this experiment). Note that this number is strongly dependent on the used configurations, mainly on the two most expensive processes of optimization and training. Fewer executions or different parameterizations greatly influence the perceived overhead.

In order to translate those execution times into costs, we could move the continuous estimation process into a serverless cloud environment. For example, if we execute the four processes on AWS Lambda (assuming server location in central Europe), we would need to pay for 244 invocations consuming 2844 seconds of computation time. Even if we multiply the compute seconds with the number of cores available on the test machine (4 cores) and choose to run the largest function size allocation currently available (3 GB), this would currently cost us 0.57 $ for the whole experiment[6]. Therefore, we can conclude that the current configuration would result in maximum a cost of $ 0.19 per hour. Given that the monitored applications are usually much larger in size and therefore in operating cost, we assume the overhead costs of running *SARDE* are negligible. Hence, this answers **RQ 3**.

8 DISCUSSION

After we viewed and analyzed the results in the previous section, we discuss our findings in this section.

8.1 Continuous Updates

First, the question arises whether or not the continuously repeating activities, i.e., continuously repeating estimation, optimization, training, and selection activities is really necessary.

Is continuously estimating necessary? We argue that based on the continuous changes in the actual estimations, together with the respective error, and the comparatively low overhead of executing a single estimation, the continuous estimation of resource demands is useful and necessary. This question was already targeted by **RQ 1** and the results are in line with the discussion in Section 3 and Section 7.5.

Is continuously selecting necessary? Similarly, as the properties of the incoming data flows constantly change, the applicability of the different approaches changes as well. This is also observable in Figure 1 and from all our results in Section 7 as this is the main reason for the constantly changing error rates of each approach. Therefore, we strongly advocate the constant update of the selection. Furthermore, we observe that almost all approaches have their justification and that the selection process frequently makes use of the different available approaches. Especially, as the results of Section 7.5 suggest that the selection using an already trained machine learning model is unsurprisingly very fast. One could even consider increasing the frequency of the selection to select a new estimation approach for every estimation interval.

[6]Calculated by: https://aws.amazon.com/lambda/pricing

Is continuously training necessary? In contrast, the results of Section 7 do not suggest that continuously updating the selection strategy provides strong benefits. We observe from Sections 7.1 and 7.3 that the selection process indeed learns and adapts to the current trace and updating the selection strategy is useful. However, this is more due to an increase in available information and training data, than to the diligent repetition of the training process. Considering this and the fact that the training procedure is relatively expensive, a lower training frequency might be justified. We definitely see a benefit of repeating the training process; however, the costs of the training could be significantly lowered with little to no effect on the adaptation abilities by increasing the training interval. Two related interesting research questions towards that direction furthermore include *How much training data is enough?*, i.e., the minimal amount of training data that justifies training and using an estimator for selection and *How much offline data do we need?*, i.e., does it make sense to ignore all offline training data and utilize only online data for method selection. Alternatively, we could go ahead and simply replace the offline training data with the online data as it comes in.

Is continuously optimizing necessary? Similar results can be drawn for the optimization processes. The optimization is equally expensive as the training process, and takes even longer, depending on the parameterization of the used optimization algorithm. Although its positive influences can be seen in the analysis (see Section 7.2), its cost is significant in comparison to the standard estimation or selection procedure. As the accuracy gains take quite a while to come into effect, a lower optimization frequency would make sense if one wants to reduce the computational costs.

Summary. We conclude that all activities show effects and improvements to the overall estimation accuracy of *SARDE*. Therefore, continuous updates make sense for all of the proposed activities; the remaining questions are concerned with the optimal activity intervals.

8.2 Adapting Learning Intervals

Following our reflections of the previous chapter, we observe that the repetition intervals need to be updated as well. As we are currently tuning the adaptive processes of resource demand estimation, the dynamic adaptation of these adaptive processes can be seen as an additional layer of self-adaptation or *meta*-self-adaptation [49].

We can achieve these meta-adaptation capabilities by introducing an additional layer, tweaking the anticipated activity pause intervals based on their expected gain. This can be achieved via many possible functions. However, a straightforward solution is to utilize a function $f_{t_{max}}[0,1] \rightarrow [0, t_{max}]$, defining the length of a pause before the next activity cycle starts in dependence on a maximum pause time t_{max} and a normalized expected gain $g \in [0,1]$. This gain value g is based on the benefits of the last executed activity cycle, for example, by evaluating the relative improvement of a parameter optimization.

Using the calculated gain g, the optimal activity pause can be modeled using an exponential function:

$$f_{t_{max}}(g) = \frac{e \cdot t_{max}}{e-1} \left(\exp\left(-g^2\right) - \frac{1}{e} \right). \tag{4}$$

This version has the advantage of offering a smooth decay over the anticipated pause interval time with increasing gain but suggesting comparatively long intervals for small increases of gain. This makes sense, as a future gain is unlikely. However, with further increasing gain, the suggested interval time falls almost linearly and reaches its zero at exactly $g = 1$, the maximum available gain value. If we also want to take the costs of an activity into account, we can modify the length of the plateau, or the steepness of the interval decrease by modifying the exponent of g in the exponential term. Hence, we transform f into a two-dimensional function:

$$f_{t_{max}}(g, y) = \frac{e \cdot t_{max}}{e - 1} \left(\exp\left(-g^y\right) - \frac{1}{e} \right), \tag{5}$$

where g is the expected gain value, and $y \in [1, \infty[$ is the anticipated cost. While it is mathematically sound to have y unbounded, in practice, for most cases, we want to normalize y in order to limit its value (e.g., $y \in [1, 10]$) and therefore its influence. A caveat of this approach is that after *SARDE* has adapted reasonably well to a system, it consequentially chooses long adaptation times in order to save cost. Then, *SARDE* takes longer to react to sudden changes of the underlying system causes by, e.g., a deployment change or other large structural changes. The worst-case impact of this problem can be addressed by lowering t_{max}; however, this in turn also reduces the potential cost benefits for reduced intervals. As this opens another dimension of parameters to be evaluated, we exclude evaluations on this in the scope for this paper. However, this might be an interesting direction for future work.

8.3 Ensemble approaches

We observed that some classification algorithms responsible for selecting the best approach are able to assign a score to each of the estimation approaches. Currently, the aim is simply to select the approach assigned with the highest score as it has the highest probability of delivering the best results estimations according to that classifier. However, one could also go one step further in utilizing these scores as a weight function in order to produce a combined resource demand estimate. Given the vector of resource demand estimations of each of the n individual approaches $(\tilde{D}_{c,1}, \tilde{D}_{c,2}, \ldots, \tilde{D}_{c,n})$ for a set workload class c, and vector of assigned scores $(w_i, w_2, \ldots, w_n)$ calculated by a machine learning algorithm, we can compute the compound resource demand estimate for workload class $\tilde{D}_c$ as:

$$\tilde{D}_c = \frac{1}{n} \sum_{i=1}^{n} w_i \cdot \tilde{D}_{c,i}. \tag{6}$$

Note that in this example we assume the sum of all scores to sum up to 1. If they do not, we can normalize all scores in order to receive a valid weighting vector. If no classification algorithm seems suitable for this task, one could also utilize regression algorithms in order to learn the expected error and hence the resulting score of each estimator. However, similar to the previous chapter, this is out of scope for this paper as applying such a compound estimation technique would require further evaluations, parameter tuning, and deep analysis.

9 THREATS TO VALIDITY

Although we conducted the presented evaluations to the best of our knowledge, there might be some remaining threats to validity.

9.1 Internal validity

Our evaluation of the online application is based on a synthetic application, written especially for this analysis. This way, it is possible for us to exactly define and program the specific resource demands into the application, which is crucial in order to calculate the respective estimation errors. Therefore we are confident in the internal validity of the study. Unfortunately, the resource demands of any real-world application are not known in advance and would need to be estimated as well. Therefore, no meaningful evaluation about the accuracy of the used estimation techniques could be conducted, if no gold standard was available.

Finally, we note that all self-adaptation and optimization processes of *SARDE* are dependent on the internal validation error. The internal error estimates the error of the respective estimation based on the incoming measurements (as the gold standard is obviously unknown). Therefore, this internal error function is of paramount importance for the performance of all self-adaptation techniques of *SARDE*. Addressing these and other challenges discussed in the previous chapters might be possible topics of future work.

9.2 External validity

Concerning external validity, all presented error measures and especially the measured computation time of the realistic application reflect just the one repeatable estimation run. Different input data streams from different applications or measured on different systems could possibly lead to different results. Especially the overhead analysis must be viewed as an exemplary analysis, as its values are heavily dependent on the chosen parameterization as well as the respective machine learning algorithms or optimization techniques. As already discussed in the previous section, the repetition intervals can be arbitrarily changed as well, therefore the results of the overhead analysis can not be directly transferred to any arbitrary system.

In addition, our experiment results are limited to the evaluated workload patterns and resource demands presented in Section 6 and analyzed in Section 7.4. While we did our best to spread and diversify the analyzed scenarios, future work could aim at extending our analyses in order to verify whether the results transfer other scenarios as well.

10 LIMITATIONS

Next to the discussed design decisions discussed in Section 8, *SARDE* currently faces the following limitations.

The presented results only focus on six of the eight available approaches within LibReDE, as the two techniques based on recursive optimization [53, 57] are based on an incompatible optimization library and are therefore not usable for the presented study. However, the results using the presented six methods already show the benefits of *SARDE*. Note that this represents a strict technical limitation that does not affect the conceptual contribution of this work and could be therefore addressed in future work to further improve the presented results.

Similarly, LibReDE currently does not support the notion of uncertainty in the monitoring streams, being it due to missing values, low accuracy, or precision. Therefore, *SARDE* is also not able to support uncertain monitoring streams. However, future versions might enable confidence values or multiple measurement repetitions in order to remedy that problem.

While all activities are designed for continuous and online application, the current implementations are based on repeated batch learning. Therefore, while the data patterns offer the possibility for online learning and online algorithm selection capabilities, this is currently not implemented. However, it is expected that such techniques would mainly improve the computation times and therefore further simplify the use of the *SARDE*.

Finally, our experiment explicitly did not focus on extrinsic changes in resource demands. Such a resource demand change would for example occur if the running application is re-deployed or changed using Continuous-Integration and Continuous-Deployment pipelines following a new commit. This would of course invalidate all previous resource demand estimations and would require a reset of the monitoring traces of the affected parts of the system. We focus specifically on such incremental extraction approaches in another line of our research [56, 86]. However, as *SARDE* is designed for continuous changes in the environment, we are confident that the approach is able to work in such scenarios.

11 CONCLUSION

In this paper, we presented *SARDE*, a framework for continuous self-adaptive resource demand estimation. *SARDE* continuously (i) estimates resource demands, (ii) selects the most suitable estimation approach from a set of available alternatives, and (iii) optimizes the parameterization of the estimation approaches in order to minimize the estimation error. This is achieved by continuously evaluating the performance of each estimator in the current and constantly changing scenario. Based on the characteristics of the current situations, *SARDE* is able to adapt each estimator itself, but also to select the most suitable approach as well as improving and hardening the overall estimation error. This enables *SARDE* to serve as an ensemble resource demand estimator, capable of delivering reliable estimations in unknown and constantly changing environments without expert knowledge or human intervention.

We evaluate the selection of the estimator and the optimization using two different data sets: One collection of many different short-lived scenarios, and one realistic web application. Additionally, we analyze how the combination of both approaches inter-operates on the web application and also analyze the overhead of each individual activity performed by *SARDE*. We conclude that on our evaluated data sets the overhead is very limited in comparison to the achieved self-adaptive properties *SARDE* offers. The source code of *SARDE* is available as open-source[1], and a replication package of the results is published on CodeOcean[3].

ACKNOWLEDGMENTS

This work was co-funded by the German Research Foundation (DFG) under grant No. (KO 3445/11-1) and by Google Inc. (Faculty Research Award). We thank all anonymous reviewers as their suggestions significantly improved the quality of the paper throughout the review process.

REFERENCES

[1] Warren Armstrong, Peter Christen, Eric McCreath, and Alistair P Rendell. 2006. Dynamic algorithm selection using reinforcement learning. In *2006 International Workshop on Integrating AI and Data Mining*. IEEE, 18–25.

[2] André Bauer, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev. 2018. On the Value of Service Demand Estimation for Auto-Scaling. In *19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB 2018)* (Erlangen, Germany). Springer.

[3] A. Biedenkapp, J. Marben, M. Lindauer, and F. Hutter. 2018. CAVE: Configuration Assessment, Visualization and Evaluation. In *Proceedings of the International Conference on Learning and Intelligent Optimization (LION'18)*.

[4] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. 2016. ASlib: A benchmark library for algorithm selection. *Artificial Intelligence* 237 (2016), 41 – 58. https://doi.org/10.1016/j.artint.2016.04.003

[5] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. 1998. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley-Interscience, New York.

[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[7] L. Breiman, J. Friedman, R. Olshen, and C. Stone. 1984. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA.

[8] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. 2009. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *VALUETOOLS '09* (Pisa, Italy). Article 10, 10 pages.

[9] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society* 64, 12 (2013), 1695–1724. https://doi.org/10.1057/jors.2013.71

[10] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. 2012. Self-Adaptive Software Needs Quantitative Verification at Runtime. *Commun. ACM* 55, 9 (Sept. 2012), 69–77. https://doi.org/10.1145/2330667.2330686

[11] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. 2009. Qos-driven Runtime Adaptation of Service Oriented Architectures. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/1595696.1595718

[12] Giuliano Casale, Paolo Cremonesi, and Roberto Turrin. 2007. How to select significant workloads in performance models. In *CMG Conference Proceedings*. 58–108.

[13] Giuliano Casale, Paolo Cremonesi, and Roberto Turrin. 2008. Robust Workload Estimation in Queueing Network Performance Models. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008) (PDP '08)*. IEEE Computer Society, USA, 183–187. https://doi.org/10.1109/PDP.2008.80

[14] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[15] David R Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)* 20, 2 (1958), 215–232.

[16] David Roxbee Cox. 1966. The statistical analysis of series of events. *Monographs on Applied Probability and Statistics* (1966).

[17] M. D'Angelo, S. Gerasimou, S. Ghahremani, J. Grohmann, I. Nunes, E. Pournaras, and S. Tomforde. 2019. On Learning in Collective Self-Adaptive Systems: State of Practice and a 3D Framework. In *Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '19)*. IEEE Press, 13–24.

[18] Mirko D'Angelo, Sona Ghahremani, Simos Gerasimou, Johannes Grohmann, Ingrid Nunes, Sven Tomforde, and Evangelos Pournaras. 2020. Learning to Learn in Collective Adaptive Systems: Mining Design Pattern for Data-driven Reasoning. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 121–126.

[19] Hans Degroote, Bernd Bischl, Lars Kotthoff, and Patrick De Causmaecker. 2016. Reinforcement learning for automatic online algorithm selection-an empirical study. *ITAT 2016 Proceedings* 1649 (2016), 93–101.

[20] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. 2010. FUSION: A Framework for Engineering Self-tuning Self-adaptive Software Systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. ACM, New York, NY, USA, 7–16. https://doi.org/10.1145/1882291.1882296

[21] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *Journal of Machine Learning Research* 20, 55 (2019), 1–21. http://jmlr.org/papers/v20/18-598.html

[22] John M. Ewing and Daniel A. Menascé. 2014. A Meta-Controller Method for Improving Run-Time Self-Architecting in SOA Systems. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering* (Dublin, Ireland) *(ICPE '14)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/2568088.2568098

[23] E. M. Fredericks, I. Gerostathopoulos, C. Krupitzer, and T. Vogel. 2019. Planning as Optimization: Dynamically Discovering Optimal Configurations for Runtime Situations. In *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 1–10. https://doi.org/10.1109/SASO.2019.00010

[24] Erik M. Fredericks, Christian Krupitzer, Ilias Gerostathopoulos, and Thomas Vogel. 2019. *Planning as Optimization: Online Learning of Situations and Optimal Configurations - SASO 2019 - Accompanying material.* https://doi.org/10.5281/zenodo.2584266

[25] Matteo Gagliolo and Jürgen Schmidhuber. 2010. Algorithm selection as a bandit problem with unbounded losses. In *International Conference on Learning and Intelligent Optimization*. Springer, 82–96.

[26] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. 2011. A portfolio solver for answer set programming: Preliminary report. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 352–357.

[27] Johannes Grohmann, Simon Eismann, Andre Bauer, Marwin Zuefle, Nikolas Herbst, and Samuel Kounev. 2019. Utilizing Clustering to Optimize Resource Demand Estimation Approaches. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 134–139.

[28] Johannes Grohmann, Simon Eismann, and Samuel Kounev. 2018. The Vision of Self-Aware Performance Models. In *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)* (Seattle, USA). 60–63. https://doi.org/10.1109/ICSA-C.2018.00024

[29] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. 2017. Self-Tuning Resource Demand Estimation. In *Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017)* (Columbus, OH). https://doi.org/10.1109/ICAC.2017.19

[30] Johannes Grohmann, Nikolas Herbst, Simon Spinner, and Samuel Kounev. 2018. Using Machine Learning for Recommending Service Demand Estimation Approaches. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*. INSTICC, SciTePress, 473–480. https://doi.org/10.5220/0006761104730480

[31] Johannes Grohmann, Daniel Seybold, Simon Eismann, Mark Leznik, Samuel Kounev, and Jörg Domaschka. 2020. Baloo: Measuring and Modeling the Performance Configurations of Distributed DBMS. In *2020 IEEE 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) (MASCOTS '20)*.

[32] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-efficient performance learning for configurable systems. *Empirical Software Engineering* 23, 3 (2018), 1826–1867.

[33] Huong Ha and Hongyu Zhang. 2019. DeepPerf: performance prediction for configurable software with deep sparse neural network. In *IEEE/ACM 41st International Conference on Software Engineering*. 1095–1106.

[34] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. 2009. Multi-class adaboost. *Statistics and its Interface* 2, 3 (2009), 349–360.

[35] Malte Helmert, Gabriele Röger, and Erez Karpas. 2011. Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS 2011 Workshop on Planning and Learning*. Citeseer, 28–35.

[36] Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, and Manuel Bähr. 2017. Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language. *IEEE Transactions on Software Engineering* 43, 5 (2017), 432–452.

[37] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization*, Carlos A. Coello Coello (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 507–523.

[38] Frank Hutter, Manuel López-Ibáñez, Chris Fawcett, Marius Lindauer, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2014. AClib: A Benchmark Library for Algorithm Configuration. In *Learning and Intelligent Optimization - 8th International Conference, Lion 8, Gainesville, FL, USA, February 16-21, 2014. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8426)*, Panos M. Pardalos, Mauricio G. C. Resende, Chrysafis Vogiatzis, and Jose L. Walteros (Eds.). Springer, 36–40. https://doi.org/10.1007/978-3-319-09584-4_4

[39] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2014. Algorithm runtime prediction: Methods and evaluation. *Artificial Intelligence* 206 (2014), 79 – 111. https://doi.org/10.1016/j.artint.2013.10.003

[40] D. N. Joanes and C. A. Gill. 1998. Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society: Series D (The Statistician)* 47, 1 (1998), 183–189. https://doi.org/10.1111/1467-9884.00122

[41] Jeffrey O Kephart and David M Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[42] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. 2019. Automated algorithm selection: Survey and perspectives. *Evolutionary computation* 27, 1 (2019), 3–45.

[43] Lars Kotthoff, Pascal Kerschke, Holger Hoos, and Heike Trautmann. 2015. Improving the State of the Art in Inexact TSP Solving Using Per-Instance Algorithm Selection. In *Learning and Intelligent Optimization*, Clarisse Dhaenens, Laetitia Jourdan, and Marie-Eléonore Marmion (Eds.). Springer International Publishing, Cham, 202–217.

[44] Olga Kouchnarenko and Jean-François Weber. 2014. Adapting Component-Based Systems at Runtime via Policies with Temporal Patterns. In *Formal Aspects of Component Software*, José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue (Eds.). Springer International Publishing, Cham, 234–253.

[45] Samuel Kounev, Peter Lewis, Kirstie Bellman, Nelly Bencomo, Javier Camara, Ada Diaconescu, Lukas Esterle, Kurt Geihs, Holger Giese, Sebastian Götz, Paola Inverardi, Jeffrey Kephart, and Andrea Zisman. 2017. The Notion of Self-Aware Computing. In *Self-Aware Computing Systems*, Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu (Eds.). Springer Verlag, Berlin Heidelberg, Germany.

[46] Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. 2009. Estimating service resource consumption from response time measurements. In *VALUETOOLS '09* (Pisa, Italy). 1–10.

[47] Dinesh Kumar, Asser Tantawi, and Li Zhang. 2009. Real-time performance modeling for adaptive software systems. In *VALUETOOLS '09* (Pisa, Italy). 1–10.

[48] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. 1984. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[49] Peter Lewis, Kirstie L Bellman, Christopher Landauer, Lukas Esterle, Kyrre Glette, Ada Diaconescu, and Holger Giese. 2017. Towards a Framework for the Levels and Aspects of Self-aware Computing Systems. In *Self-Aware Computing Systems*. Springer, 51–85.

[50] Haifeng Li. 2014. Smile. https://haifengl.github.io.

[51] Jim (Zhanwen) Li, John Chinneck, Murray Woodside, and Marin Litoiu. 2009. Fast Scalable Optimization to Configure Service Systems Having Cost and Quality of Service Constraints. In *Proceedings of the 6th International Conference on Autonomic Computing* (Barcelona, Spain) *(ICAC '09)*. ACM, 159–168. https://doi.org/10.1145/1555228.1555268

[52] Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. 2015. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research* 53 (2015), 745–778.

[53] Zhen Liu, Laura Wynter, Cathy H. Xia, and Fan Zhang. 2006. Parameter inference of queueing models for IT systems using end-to-end measurements. *Perform. Evaluation* 63, 1 (2006), 36–60.

[54] Yuri Malitsky. 2014. Evolving instance-specific algorithm configuration. In *Instance-Specific Algorithm Configuration*. Springer, 93–105.

[55] Manar Mazkatli and Anne Koziolek. 2018. Continuous Integration of Performance Model. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar (Eds.). ACM, 153–158. https://doi.org/10.1145/3185768.3186285

[56] Manar Mazkatli, David Monschein, Johannes Grohmann, and Anne Koziolek. 2020. Incremental Calibration of Architectural Performance Models with Parametric Dependencies. In *2020 IEEE International Conference on Software Architecture (ICSA 2020)*. IEEE, 23–34.

[57] Daniel A. Menascé. 2008. Computing missing service demand parameters for performance models. In *CMG Conference Proceedings*. 241–248.

[58] Daniel A. Menasce and Virgilio Almeida. 2001. *Capacity Planning for Web Services: Metrics, Models, and Methods* (1st ed.). Prentice Hall PTR, USA.

[59] Daniel A. Menascé, Lawrence W. Dowdy, and Virgilio A. F. Almeida. 2004. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[60] Daniel A. Menasce and A. F. Almeida Virgilio. 2000. *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning* (1st ed.). PTR.

[61] G. A. Moreno, O. Strichman, S. Chaki, and R. Vaisman. 2017. Decision-Making with Cross-Entropy for Self-Adaptation. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 90–101. https://doi.org/10.1109/SEAMS.2017.7

[62] Qais Noorshams. 2015. *Modeling and Prediction of I/O Performance in Virtualized Environments*. Ph.D. Dissertation. Karlsruhe Institute of Technology (KIT). http://digbib.ubka.uni-karlsruhe.de/volltexte/1000046750

[63] Qais Noorshams, Dominik Bruhn, Samuel Kounev, and Ralf Reussner. 2013. Predictive Performance Modeling of Virtualized Storage Systems Using Optimized Statistical Regression Techniques. In *ACM/SPEC ICPE 2013* (Prague, Czech Republic) *(ICPE '13)*. ACM, New York, NY, USA, 283–294.

[64] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser Tantawi. 2008. CPU demand for web serving: Measurement analysis and dynamic estimation. *Perform. Evaluation* 65, 6-7 (2008), 531–553.

[65] Juan F. Pérez, Giuliano Casale, and Sergio Pacheco-Sanchez. 2015. Estimating Computational Requirements in Multi-Threaded Applications. *IEEE Trans. Software Eng.* 41, 3 (2015), 264–278. https://doi.org/10.1109/TSE.2014.2363472

[66] P. Pilgerstorfer and E. Pournaras. 2017. Self-Adaptive Learning in Decentralized Combinatorial Optimization - A Design Paradigm for Sharing Economies. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 54–64. https://doi.org/10.1109/SEAMS.2017.8

[67] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. 2016. {REX}: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 333–348.

[68] Luca Pulina and Armando Tacchella. 2009. A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints* 14, 1 (2009), 80–116.

[69] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. 2016. *Modeling and Simulating Software Architectures: The Palladio Approach*. MIT Press.

[70] John R. Rice. 1976. The Algorithm Selection Problem**This work was partially supported by the National Science Foundation through Grant GP-32940X. This chapter was presented as the George E. Forsythe Memorial Lecture at the Computer Science Conference, February 19, 1975, Washington, D. C. Advances in Computers, Vol. 15. Elsevier, 65 – 118. https://doi.org/10.1016/S0065-2458(08)60520-3

[71] Roberto Rodrigues Filho and Barry Francis Porter. 2017. Defining emergent software using continuous self-assembly, perception and learning. *ACM Transactions on Autonomous and Adaptive Systems* 12, 3 (Sept. 2017). https://doi.org/10.1145/3092691

[72] Jerome Rolia and Vidar Vetland. 1995. Parameter estimation for performance models of distributed application systems. In *CASCON '95* (Toronto, Ontario, Canada). IBM Press, 54.

[73] Jerome Rolia and Vidar Vetland. 1998. Correlating resource demand information with ARM data for application services. In *Proceedings of the 1st international workshop on Software and performance* (Santa Fe, New Mexico, United States). ACM, 219–230.

[74] Abhishek B. Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M. Voelker. 2008. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev.* 36 (Aug. 2008), 16–25. Issue 2.

[75] Stepan Shevtsov and Danny Weyns. 2016. Keep It SIMPLEX: Satisfying Multiple Goals with Guarantees in Control-based Self-adaptive Systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 229–241.

[76] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*.

284–294.
[77] Kevin Sim, Emma Hart, and Ben Paechter. 2015. A Lifelong Learning Hyper-Heuristic Method for Bin Packing. *Evol. Comput.* 23, 1 (March 2015), 37–67. https://doi.org/10.1162/EVCO_a_00121
[78] Kate A Smith-Miles. 2009. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)* 41, 1 (2009), 1–25.
[79] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. 2015. Evaluating Approaches to Resource Demand Estimation. *Perform. Evaluation* 92 (October 2015), 51 – 71. https://doi.org/10.1016/j.peva.2015.07.005
[80] Simon Spinner, Giuliano Casale, Xiaoyun Zhu, and Samuel Kounev. 2014. LibReDE: A Library for Resource Demand Estimation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)* (Dublin, Ireland). ACM Press, New York, NY, USA, 227–228. https://doi.org/10.1145/2568088.2576093
[81] Simon Spinner, Giuliano Casale, Xiaoyun Zhu, and Samuel Kounev. 2014. LibReDE: A Library for Resource Demand Estimation. In *ACM/SPEC ICPE 2014* (Dublin, Ireland) *(ICPE '14)*. ACM, New York, NY, USA, 227–228.
[82] Simon Spinner, Johannes Grohmann, Simon Eismann, and Samuel Kounev. 2019. Online model learning for self-aware computing infrastructures. *Journal of Systems and Software* 147 (2019), 1 – 16.
[83] Christopher Stewart, Terence Kelly, and Alex Zhang. 2007. Exploiting nonstationarity for performance prediction. *SIGOPS Oper. Syst. Rev.* 41 (March 2007), 31–44. Issue 3.
[84] Jan N. van Rijn, Geoffrey Holmes, Bernhard Pfahringer, and Joaquin Vanschoren. 2014. Algorithm Selection on Data Streams. In *Discovery Science*, Sašo Džeroski, Panče Panov, Dragi Kocev, and Ljupčo Todorovski (Eds.). Springer International Publishing, Cham, 325–336.
[85] Jan N van Rijn, Geoffrey Holmes, Bernhard Pfahringer, and Joaquin Vanschoren. 2018. The online performance estimation framework: heterogeneous ensemble learning for data streams. *Machine Learning* 107, 1 (2018), 149–176.
[86] Sonya Voneva, Manar Mazkatli, Johannes Grohmann, and Anne Koziolek. 2020. Optimizing Parametric Dependencies for Incremental Performance Model Extraction. In *Companion of the 14th European Conference Software Architecture (ECSA 2020) (Communications in Computer and Information Science, Vol. 1269)*, Henry Muccini, Paris Avgeriou, Barbora Buhnova, Javier Cámara, Mauro Caporuscio, Mirco Franzago, Anne Koziolek, Patrizia Scandurra, Catia Trubiani, Danny Weyns, and Uwe Zdun (Eds.). Springer, 228–240.
[87] Jürgen Walter, Christian Stier, Heiko Koziolek, and Samuel Kounev. 2017. An Expandable Extraction Framework for Architectural Performance Models. In *Proceedings of the 3rd International Workshop on Quality-Aware DevOps (QUDOS'17)* (l'Aquila, Italy). ACM, 6.
[88] Wei Wang, Xiang Huang, Xiulei Qin, Wenbo Zhang, Jun Wei, and Hua Zhong. 2012. Application-Level CPU Consumption Estimation: Towards Performance Isolation of Multi-tenancy Web Applications. In *IEEE CLOUD 2012*. 439 –446.
[89] Wei Wang, Xiang Huang, Yunkui Song, Wenbo Zhang, Jun Wei, Hua Zhong, and Tao Huang. 2011. A statistical approach for estimating CPU consumption in shared Java middleware server. In *IEEE COMPSAC, 2011*. IEEE, 541–546.
[90] Weikun Wang, Juan F. Pérez, and Giuliano Casale. 2015. Filling the Gap: A Tool to Automate Parameter Estimation for Software Performance Models. In *Proceedings of the 1st International Workshop on Quality-Aware DevOps* (Bergamo, Italy) *(QUDOS 2015)*. Association for Computing Machinery, New York, NY, USA, 31–32. https://doi.org/10.1145/2804371.2804379
[91] Peter H Westfall. 2014. Kurtosis as peakedness, 1905–2014. RIP. *The American Statistician* 68, 3 (2014), 191–195.
[92] Felix Willnecker, Markus Dlugi, Andreas Brunnert, Simon Spinner, Samuel Kounev, and Helmut Krcmar. 2015. Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques. In *EPEW 2015* (Madrid, Spain) *(Lecture Notes in Computer Science, Vol. 9272)*, Marta Beltrán, William Knottenbelt, and Jeremy Bradley (Eds.). Springer, 115–129.
[93] David H Wolpert. 1996. The lack of a priori distinctions between learning algorithms. *Neural computation* 8, 7 (1996), 1341–1390.
[94] David H Wolpert and William G Macready. 1997. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation* 1, 1 (1997), 67–82.
[95] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research* 32 (2008), 565–606.
[96] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. 2007. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Fourth International Conference on Autonomic Computing (ICAC'07)*. 27–27.
[97] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance prediction of configurable software systems by fourier learning (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 365–373.
[98] Tao Zheng, C.M. Woodside, and M. Litoiu. 2008. Performance Model Estimation and Tracking Using Optimal Filters. *IEEE TSE* 34, 3 (May 2008), 391–406.

[99] Tao Zheng, Jinmei Yang, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. 2005. Tracking time-varying parameters in software systems with extended Kalman filters. In *CASCON '05* (Toranto, Ontario, Canada). IBM Press, 334–345.