Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques

Johannes Grohmann, Simon Eismann,
Sven Elflein, Jóakim v. Kistowski, Samuel Kounev
University of Würzburg
Würzburg, Germany
{first_name}.{last_name}@uni-wuerzburg.de

Manar Mazkatli
Karlsruhe Institute of Technology
Karlsruhe, Germany
manar.mazkatli@kit.edu

***Abstract*—Architectural performance models are a common approach to predict the performance properties of a software system. Parametric dependencies, which describe the relation between the input parameters of a component and its performance properties, significantly increase the prediction accuracy of architectural performance models. However, manually modeling parametric dependencies is time-intensive and requires expert knowledge. Existing automated extraction approaches require dedicated performance tests, which are often infeasible.**

In this paper, we introduce an approach to automatically identify parametric dependencies from monitoring data using feature selection techniques from the area of machine learning. We evaluate the applicability of three techniques selected from each of the three groups of feature selection methods: a filter method, an embedded method, and a wrapper method. Our evaluation shows that the filter technique outperforms the other approaches. Based on these results, we apply this technique to a distributed micro-service web-shop, where it correctly identifies 11 performance-relevant dependencies, achieving a precision of 91.7% based on a manually labeled gold-standard.

***Index Terms*—Performance Engineering, Performance Modeling, Machine Learning, Feature Selection, Parametric Dependencies**

I. Introduction

Architectural performance models are a common approach to predict the performance properties of a software system during system design [1] as well as the impact of reconfigurations at run-time [2]. A significant factor for the prediction accuracy of a performance model is its parameterization, i.e., the values for model parameters such as loop frequencies, branching probabilities or resource demands [3]. However, these model parameters often depend on the input parameters of a component (e.g., the size of a list impacts the time required to sort it). Therefore, many architectural performance models allow to explicitly model input parameters and their influence on model parameters in the form of so-called *parametric dependencies* [4]–[8]. These dependencies describe, e.g., the resource demand of a function call for a given set of input parameters. The importance of including such influences has been discussed by a variety of authors [9]–[12].

Manually modeling parametric dependencies requires expert knowledge; it is quite error-prone and causes significant manual overhead. For example, in a case study by Krogmann et al. [13] more than 24 hours were required to manually model the parametric dependencies in a small system, which shows that manually modeling parametric dependencies for large systems is infeasible. The required effort furthermore hinders the adoption of performance modeling techniques in practice [14].

Courtois et al. [15] propose to use regression splines combined with automated performance testing to derive functions that describe resource demands based on input parameters. Krogmann et al. [13] use genetic search to find dependencies between a component's input parameters and the number of executed bytecode instructions. However, both approaches require dedicated performance tests to extract the parametric dependencies.

We propose to derive parametric dependencies solely from monitoring data available at run-time. This task can be split into two sub-tasks: (1) detecting the dependencies, that is, identifying which parameters influence a model variable, and (2) characterizing the dependencies, that is, describing how the value of a parameter can be derived from the influencing parameters. In this paper, we focus on sub-task (1), the detection of parametric dependencies. As to sub-task (2), multiple approaches have been proposed in the literature showing how known parametric dependencies can be modeled using standard regression techniques [15]–[17]. Hence, this paper focuses on the identification of parameters from monitoring data, i.e., finding if and which model parameters influence other parameters.

Sub-task (1) can be framed as a classic application of feature selection: We define one model parameter as a target parameter and consider all other model parameters as potential features. The challenges when applying feature selection to this domain are to obtain suitable measurement streams, to filter and select the most promising dependencies, and to discard a detected dependency if there is no modeling gain.

The contributions of this paper are:

- We propose a generic algorithm for the automated identification of parametric dependencies on monitoring streams. This includes the pre-processing of monitoring records, the creation of feature selection tasks, an interface to integrate a chosen feature selection technique, and three different heuristics to filter the identified dependencies. These heuristics utilize domain knowledge to dras-

tically decrease the number of identified dependencies reducing them to only performance relevant ones.
- We apply and evaluate three different feature selection approaches: a filter method based on correlation analysis, a wrapper method based on M5 [18], and an embedded method based on Random Forest Regression [19].
- Based on our experimentation, we uncover additional insights about the applicability of our approach to monitoring data from different load levels and conclude that the load level has a surprisingly small impact on the solution quality.

Our approach significantly reduces the required effort of modeling parametric dependencies, enables automated extraction of more detailed models and therefore makes the modeling of dependencies feasible for large systems. The approach can also be used as assistance for a performance modeling expert, who can use our system's suggestions as candidate dependencies. Furthermore, the approach requires only runtime monitoring data of the managed application and otherwise treats the application as a black box. All required structural information is obtained from the monitoring data. Therefore, this work fits well into the vision of self-learning and self-improving performance models [20].

II. Related Work

In this section, we discuss the current state of the art for the automated extraction of performance models in Section II-A. Subsequently, Section II-B details how the approach presented in this paper can be integrated into existing model extraction pipelines to enable fully automated extraction of parametric dependencies.

A. State of the art

Multiple approaches have been proposed to extract performance models from monitoring data collected during system operation [16], [21]–[23], [23]–[25]. However, none of these approaches is capable of extracting parametric dependencies. Therefore, the extracted performance models cannot predict the impact of input parameters on the performance of software components. However, the importance of including such influences has been discussed by a variety of authors, for example, by Woodside et al. [9], Pozetti et al. [10], Menasce [11], and Koziolek [12].

Therefore, a variety of works explicitly considered parametric dependencies in their work. Krogmann et al. [13] perform dedicated performance experiments after instrumenting the application to monitor method call parameters and the number of executed bytecode instructions. The number of executed bytecode instructions are later mapped to resource demands for a specific system using bytecode benchmarks. Therefore, the proposed approach cannot be applied at run-time, as the used bytecode instrumentation causes monitoring overheads of up to 250% [13]. Courtois et al. [15] propose to use regression splines to extract parametric dependencies. They perform dedicated performance tests to obtain the data on which they fit the regression splines. This approach is not applicable to monitoring data from a running system, as there is no way to influence what monitoring data will be collected next. Brosig et al. [16] propose an approach to extract Palladio Component Model (PCM) instances based on monitoring data collected by Oracle WebLogic Server. Their approach requires as input information about parameter tuples for which a dependency exists. Our approach presented in this paper can be used to automate this step and provide the required input for this approach. Ackermann et al. [17] introduce a framework for modeling dependencies based on monitoring data, i.e., deriving a function that calculates a value for a dependent parameter based on its influencing parameters. This approach again requires as input information about parameter tuples for which a dependency is assumed to exist, which can be provided by our approach.

To summarize, approaches from literature either require to run preliminary experiments in a testing environment [13], [15] or require the detected parametric dependencies as input [16], [17]. The latter is in fact what we aim to achieve in this work, which is why our work integrates well with existing literature. To the best of our knowledge, there is currently no approach that can automatically identify parametric dependencies based on monitoring data from production environments.

B. Integration with existing work

The approach presented in this paper can be used to enhance existing model extraction pipelines, as shown in Figure 1. Currently, a performance model without parametric dependencies can be extracted from monitoring data using existing model extraction approaches [16], [21]–[24]. The approach presented in this paper can be applied in parallel on the same monitoring data to automatically identify dependencies between model parameters. Next, existing approaches for the characterization of dependencies, such as [15]–[17], can be applied. These find a concrete function that describes the dependency between the target model parameter and the influencing parameters.

The resulting functions can then be integrated into the previously extracted performance model to improve its expressiveness. The exact nature of this integration step depends on the modeling features offered by the specific modeling formalism. However, the presented approach is designed to be integrated into the existing performance modeling formalisms, performance model extraction techniques, and dependency characterization approaches.

III. Approach

A high-level summary of our approach is shown in Algorithm 1. As the monitoring data is usually unstructured, we define the input data $in = \{r_1, \ldots, r_n\}$ as a set of n unordered monitoring records $r_1, \ldots, r_k$. We describe a dependency $d = (p, p_e)$ as a tuple consisting of a dependent parameter p and an explanatory or independent parameter p_e used to describe p. In the following, we assume a dependency to involve just one independent parameter p_e. Note that the approach is still capable of identifying two separate dependencies for the

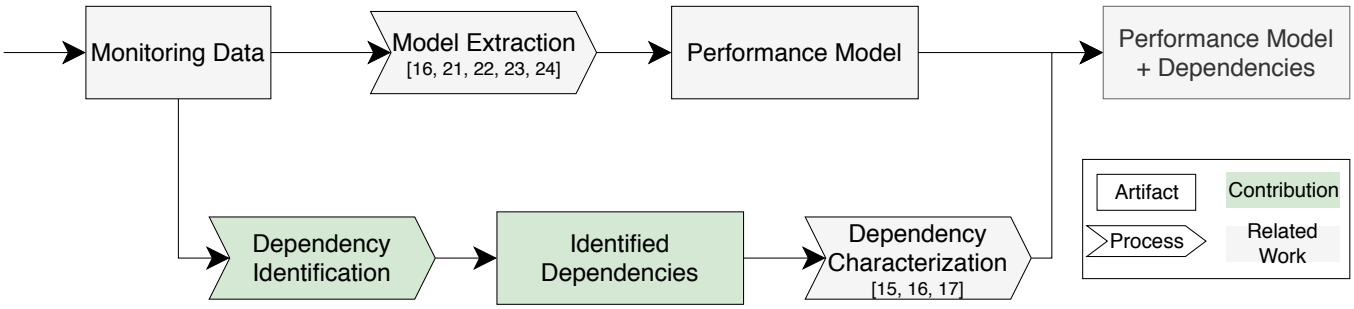

Fig. 1. Model extraction workflow.

same dependent parameter. Therefore, we can create multi-parameter dependencies by combining two or more separate dependencies.

Algorithm 1: High-level summary of proposed approach.

Input: Monitoring data $in = \{r_1, \ldots, r_k\}$.
Output: Set $D = \{(p_1, p_{l_1}), \ldots, (p_n, p_{l_n})\}$ containing all found dependencies.

1 $D = \emptyset$
2 S = extractDataStreams(in)
3 **foreach** s_i **in** S **do**
4 T_i = createFSTasks(s_i)
5 **foreach** $t_{i,j}$ **in** T_i **do**
6 $scores_{i,j}$ = applyFSAlgorithm($t_{i,j}$)
7 $D = D \cup$ createDependencies$_\theta$($t_{i,j}$, $scores_{i,j}$)
8 $D =$ filterResult(D)
9 **return** D

First, Algorithm 1 initializes an empty set D containing all found dependencies. Then, we transform the unstructured monitoring entries into data streams in line 2. The resulting set of data streams S is a collection of different sub-streams. Different sub-streams are necessary as loop and recursion structures make it impossible to aggregate all data streams on the same call-path. We elaborate on that problem in Section III-B.

We then iterate through all found data streams S to create a list of individual feature selection tasks T_i for each sub-stream s_i in line 4. Each task consists of one defined dependent parameter, together with all possible independent variables. Each task $t_{i,j}$ is then fed into a black-box feature selection algorithm in line 6. The algorithms are required to return a vector $scores_{i,j}$, assigning a weight to each independent parameter for the specific task. These scores are then used to create the set of dependencies in line 7. Currently, the method `createDependencies` includes dependencies if its score is higher than a given threshold θ. However, this threshold depends on the used feature selection algorithm and the corresponding scoring technique. Therefore, `createDependencies` has to be parameterized with a certain threshold θ. After all sub-problems and the resulting tasks have been investigated, the algorithm filters all found dependencies in line 8 and finally returns the set of found dependencies D.

In the following, we elaborate on the individual steps taken by Algorithm 1. Section III-A1 defines the monitoring requirements and some necessary pre-processing steps to receive a valid input for the described algorithm. We elaborate on the process of extracting data streams from the monitoring data (`extractDataStreams`) in Section III-B. Section III-C gives some details on how to create the individual feature selection tasks (`createFSTasks`). The application of the different feature selection techniques (`applyFSAlgorithm`) is explained in Section III-D. Finally, creation of the actual dependencies (`createDependencies`) and the filter procedures (`filterResult`) are described in Section III-E and Section III-F, respectively.

A. Monitoring

In this section, we describe the monitoring streams used by our approach to identify parametric dependencies.

1) Monitoring Requirements: Our approach completely relies on monitoring data without the additional consideration of system architecture or source code. The following data has to be contained in the monitoring stream of each relevant method invocation in order to ensure that the presented approach can extract parametric dependencies:

1) Method identifier (e.g., method signature),
2) Call-path trace information to reconstruct method invocations using recursion or loops,
3) Method parameter identifiers (e.g., the parameter name, type, as well as the concrete parameter values),
4) Method return identifiers, type, as well as the concrete parameter values,
5) Resource demand of the specific method invocation.

The call-path trace is required to extract structural information from the monitoring data, for example, which method called which other methods. This is required for detecting dependencies between different method scopes. For example, if one parameter in method A influences the behavior of method B.

A monitoring framework that satisfies these requirements is Kieker [26]. As of now, Kieker does not support logging parameter and return values. Instead, it offers the option to insert custom probes enabling customization of the collected

metrics. Therefore, we created a custom monitoring probe collecting the required information. Additionally, for complex parameters likes collections and arrays, we log the size of the collection or the length of the array. Binary objects can be captured using their id or their size in bytes.

2) Resource Demand Estimation: A resource demand 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 at the resource excluding any waiting times [27], [28]. In most realistic systems, the direct measurement of service demands is not feasible during operation due to instrumentation overheads and possibly measurement interference [3]. Therefore, our monitoring data only provides response time measurements, which is the sum of waiting time and resource demand. However, one main interest of this work are dependencies describing the resource demand of a function call for a given set of input parameters. We therefore require accurate resource demand estimates.

If the load is sufficiently low, we can assume the measured response time and resource demand to be similar, as there would be no queueing delays in that case [29]. As a result, we suggest circumventing the issue by selectively utilizing low load scenarios for the input data. If this approach is infeasible, other approaches exist that can be used to estimate the resource demands as it is an active field of research [3], [30]–[33]. Willneker et al. [34] show that statistical estimation approaches can provide comparable accuracy to direct measurements. However, we require individual resource demands per method invocation, which is not supported by most statistical approaches. We investigate the impact of higher utilization levels in Section IV-C.

3) Anomaly Detection: As our approach is based on measurement data and we aim to find relations and correlations in the measurements, the proposed technique is susceptible to outliers. Therefore, an additional step of anomaly detection and filtering is necessary while pre-processing the monitoring data. The concrete amount depends heavily on the used algorithms for feature selection. In our case, it was sufficient to apply a 99.9-percentile filter to any measured vector. Hence, after collecting the data, we take all values that are below the 99.9-percentile of the measurements to remove any outliers. This has proven to be sufficient in our experiments, however, it may, of course, depend on the used platform, measurement infrastructure, and application stack. Hence, we cannot propose a general solution and acknowledge that other experiment setups might require more sophisticated solutions [35].

B. Creation of Data Streams

Monitoring streams generated by a system with different components are generally not structured in such a way that they can be directly inputted into state-of-the-art machine learning algorithms. Consequently, pre-processing steps are necessary to adjust the data to fit into the required format. For this, structural information has to be extracted from the monitoring data, loops and recursions have to be resolved, and resource demands have to be extracted from the measured response times. These so-called data streams can then be fed into machine learning algorithms.

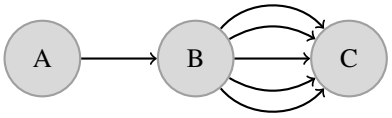

Fig. 2. Example graph representation of call-path. A invokes B once, B invokes C in a loop (5 times).

In order to create the data streams from the monitoring streams, the initial data is transformed into a directed multi-graph representing the call-path, where the vertices represent a method signature and edges the invocation of a method A calling method B. This graph is constructed by evaluating the execution order index and execution stack size of each method invocation of the current trace. This can be done using the data listed under (1) and (2) in Section III-A1. Each edge can now be associated with the respective measurements (parameter logs, together with runtime measurements). Based on the constructed graph, the data can be transformed into data streams. This process is subdivided into two steps.

First, every call-path of the graph is extracted. Each call-path resembles a succession of method invocations. We define a call-path as a sub-graph, containing all direct and indirect successive calls of the specified root-vertex.

Second, loops and recursions are resolved as they are problematic for our parameter correlation. Consider, for example, the case of function A calling function B, which then calls function C in a loop. This is illustrated in Figure 2. In this case, for every call of A, we get a set of parameter measurements of A and one measurement set for B, but multiple sets of measurements for C. Even more problematic is the fact that the number of measurements is variable and probably different for every invocation of A and B. This, however, poses a challenge for our detection algorithms as most machine learning algorithms cannot deal with a varying number of measurements per sample. Hence, the collected data cannot be directly written into the data stream, since each entry in one data stream has to correspond to the entry of another data stream with the same index. Therefore, we have to aggregate all calls from B to C into one call to process them in the algorithm later on. However, if we ignore C and collapse all measurements into one aggregation, we lose a lot of information as C might itself consist of multiple vertices and call multiple other components (including further loops). Therefore, a solution has to be found that collapses calls of loops into one entry, while preserving the ability to extract dependencies in the loop.

a) Resolving loops: A loop can be identified in the graph data structure by analyzing the number of edges from one vertex to another. The number of calls from B to C can be directly assessed by counting the number of directed edges from B to C (see Figure 2). Since this is done for each call-path trace, we can analyze the number of invocations

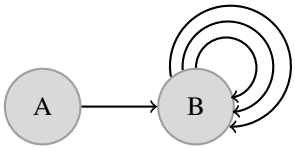

Fig. 3. Example graph representation of a direct recursion of B calling itself.

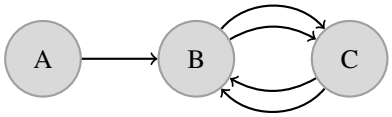

Fig. 4. Example graph representation of an indirect recursion of C and B calling each other.

from B to C. If the number of invocations varies on different call-paths, we assume that we have a dynamic loop structure depending on the internal state or the input parameters.

This is solved by dividing the original data stream into two sub-problems, (i.e., two sub-streams). That is, on the one hand, the scope of the function calling upon a looped function (A and B), and on the other hand, the scope of the loop itself (C). In order to do this, we create one data stream instance of our original problem with the loop collapsed to one entry including the number of collapsed entries as a parameter. We call the number of entries invocation count or loop count, as it describes the number of calls from B to C. This addition is necessary to be able to extract dependencies containing the iteration count of the loop since this information is lost when collapsing the entries and splitting into sub-problems. The second sub-problem instance contains the parameters inside the loop. This is necessary to avoid losing information about dependencies inside vertex C (as C could consist of several vertices and edges itself). Our identification algorithms are then applied to each of the sub-problems individually.

Given the graph in Figure 2, two sub-problems are created: one with the data streams of methods A, B and C with all execution entries of C collapsed to one entry; the other sub-problem is the loop itself with data streams of methods B and C, while the entries of B are replicated for each invocation of C.

b) Resolving recursions: Another common programming concept in algorithms is recursion. Recursion can be identified by searching the constructed graph for cycles. We can distinguish between direct and indirect recursion as seen in Figure 3 and Figure 4, respectively. However, in this work, we treat both types of recursions as loops. We represent each recursion as a black-box and only save the initial call parameters, the aggregated call parameters, the result, and the response time. Additionally, the recursion depth is saved as this information could be meaningful concerning the performance of the method (similar to the loop invocation count). All vertices concerned in indirect recursions are contracted to a single vertex for this analysis.

In the example shown in Figure 3, only the parameters of the initial invocation of the recursion of B would be considered, while the other parameter values are aggregated to one set. However, the recursion depth parameter is added with a value of 3. Applying the approach on the graph in Figure 4 would lead to the vertices B and C being merged into a single node BC and the data of the invocations between both nodes being aggregated, while adding the recursion depth parameter with a value of 2.

C. Creating Feature Selection Tasks

This section describes the creation of single feature selection tasks from existing data streams. Recall that a data stream $s_i = \{v_1, \ldots, v_k\}$ is a collection of k vectors v_i with $i \in 1, \ldots, k$. The feature selection techniques introduced in Section III-D require a distinction between a *dependent* variable (a.k.a. target variable) and a set of *independent* variables (a.k.a. features). The dependent variable is predicted based on the values of the independent variables. In order to find all dependencies within a data stream, we construct a feature selection task for each vector v_i, which analyses the impact of the remaining vectors on this vector.

Our approach for creating feature selection tasks from data streams is presented in Algorithm 2. It requires a strict total order of all vectors as well as an additional label l_i for each vector v_i. The label $l \in \{MP, AVG, RET, NONE\}$ describes the data type of each vector v_i:

1) Model parameters (MP): The vector contains performance relevant model variables, like resource demand measurements or loop invocation counts.
2) Averaged value (AVG): This vector contains averaged values as they are created when loops or recursions occur in the data stream (see Section III-B).
3) Return value (RET): These are the logged return values of a specific function invocation.
4) Normal ($NONE$): Values of these vectors are non-averaged and usually describe function parameter values.

Note that any vector is assigned exactly one label depending on the first rule that applies. Hence, averaged return values are always classified as AVG, instead of RET, as rule (2) applies first.

Together with the labels, Algorithm 2 requires a strict total order of all vectors. This order describes the chronological order of the parameter occurrences. The vector v_i is monitored after v_j, if $v_i > v_j$ for $i, j \in 1, \ldots, k$ with $i \neq j$. For simplicity, we assume that we have no concurrent records and therefore $\forall i, j \in 1, \ldots, k : v_i = v_j \Leftrightarrow i = j$. Both the strict total ordering and the set of labels can be obtained directly during the creation of the data streams in Section III-B without additional overhead.

Algorithm 2 describes how the four different classes influence the creation of feature tasks. First, we consider the performance relevant model parameters V_{MP} as the most critical parameters, as we are aiming to find dependencies describing them. Hence, we use all available parameters to find such dependencies in lines 5 and 6 of Algorithm 2. However, we do not relate MPs with other MPs, as such dependencies

Algorithm 2: Creation of Feature Selection Tasks.

Input: Data stream $S = \{(v_1, l_1), \ldots, (v_k, l_k)\}$
Output: Set $T = \{t_1, \ldots, t_l\}$ containing l created tasks.

1 $T = \emptyset$
2 Let V_{MP} be the subset of S, with $\forall v_i \in V_{MP} : l_i = MP$
3 Define $V_{AVG}, V_{RET}, V_{NONE}$ analogously
4 Let $V_{IND} = S \setminus V_{MP}$ `// indep. variables`
5 **foreach** v_i **in** V_{MP} **do**
6 $\quad T = T \cup (v_i, V_{IND})$
7 $V_{IND} = V_{IND} \setminus V_{RET}$
8 $V'_{RET} = \emptyset$
9 **foreach** v_i **in** descending ordered $V_{IND} \cup V_{RET}$ **do**
10 $\quad V_{IND} = V_{IND} \setminus v_i$
11 $\quad$ **if** $v_i \notin V_{AVG}$ **then**
12 $\quad\quad T = T \cup (v_i, V_{IND} \cup V'_{RET})$
13 $\quad$ **if** $v_i \in V_{RET}$ **then**
14 $\quad\quad V'_{RET} = V'_{RET} \cup v_i$
15 **return** T

are irrelevant (they are usually both known or unknown at the same time).

Second, we merge both return values V_{RET} and all remaining vectors V_{IND} and sort them in descending order in line 12 of Algorithm 2. Here, we implicitly use the defined ordering. However, return values can only influence other values of *lower* order, while standard parameters only influence parameters of *higher* order. Therefore, we first delete all V_{RET} from the set of independent variables in line 7 and then successively add them to the set of returned parameters V'_{RET} in line 14. These returned parameters V'_{RET} can then influence other parameters. On the contrary, all standard vectors V_{NONE} are left in the set of independent variables, but they get successively removed from V_{IND} in line 10, as they can no longer serve as independent variables. We never use the averaged parameters V_{AVG} as dependent variables. Since they contain aggregated values over multiple invocations, they are considered in a different data stream in more detail. However, we use them as independent variables for all other tasks, but successively remove them from the set of independent variables V_{IND} in line 10.

D. Applying Feature Selection Techniques

After creating concrete sub-tasks for each data stream, we can use standard feature selection techniques from machine learning to select the most promising variables as dependencies. Recall from Algorithm 1 that one task $t = (v_D, V_{IND})$ contains a dependent vector v_D and a set of independent vectors V_{IND} with the same length. The goal of the feature selection is to receive a $scores$ vector, that is, a ranking for each $v_l \in V_{IND}$ for the given v_D. This ranking expresses how useful v_l is to describe v_D, that is, how well v_D can be described using v_l. This task is a classic feature selection problem, as machine learning engineers often face the problem of selecting the most promising features (independent variables) to predict a given target (dependent variable). Therefore, all presented algorithms can return such a ranking.

However, usually, this ranking is not normalized. In classic feature engineering, this is not an issue as only the relative score of the different independent variables is of interest. Our problem statement is different in the sense that we not only need a relative rating, but also a normalized – and hence comparable – score. This score is required during the filtering step in Section III-F, where this score is the decision parameter, whether or not a feature selection task is modeled into a dependency. Therefore, we have to adapt some state-of-the-art techniques to make them applicable in our scenario.

In the following, we discuss the three main classes of feature selection techniques [36]: filter, wrapper and embedded methods.

1) Filter: Filter techniques assign a value to the possible independent variables without any interaction with the target algorithm. Features are then ranked based on the score and either selected or removed based on a cutoff value for the scoring. The methods are often uni-variate and solely consider the independent variables; therefore they are computationally cheap and commonly used as a pre-processing method.

In this work, we use Pearson's r as a measure for correlation as it was shown to be effective in related work [37]. Hence, for each independent variable, we compute the correlation with the dependent variable and use it as score. One advantage of Pearson's r is that it is already normalized between -1 and 1.

2) Wrapper: Unlike filter approaches, wrapper methods evaluate subsets of independent variables proposed by different search algorithms (forward selection, backward elimination, etc.), and thus enable the detection of interactions between different independent variables. Scores are assigned based on the accuracy of the predictive model using the particular subset. Since multiple subsets have to be evaluated and a model has to be created for each evaluation step, the main disadvantages are the significant computation time and the increased risk of over-fitting [36]. Additionally, the result highly depends on the chosen regression algorithm for evaluation.

For evaluating subsets of the variable space, M5 trees [18] are used. This enables the ability to identify non-linear and complex dependencies. In order to find the optimal score, we evaluate the performance of all subsets and store the Root Mean Squared Error (RMSE) [38]. Then, each independent variable gets a score based on the weighted average of all scores of all subsets it was involved in. Each subset is weighted with the inverse of the number of parameters involved. Hence, the more parameters involved, the less influence on the score of the involved parameters it has.

However, as this score is based on the RMSE of the resulting regression, it is not normalized and therefore not comparable between different feature selection tasks. Therefore, we divide the score of each independent variable by the inverse of the RMSE of a baseline classifier. The baseline classifier

always returns the mean value of all training samples. Hence, we weight the performance of each independent variable in dependence on the performance of this simple baseline. If the baseline performs well, the dependent variable does not show enough variance. This, therefore, justifies a low score, as the modeled dependency does not express a lot of information gain.

3) Embedded: Embedded techniques are a result of trying to combine the advantages of both filter and wrapper techniques. The selection of independent variables is accomplished during the execution of a specific learning algorithm, thus reducing computation time while still considering the interactions of variables. However, not all approaches support this technique. We chose the random forest algorithm [19] for this task, a specific type of bootstrap aggregated decision trees [39].

The core idea of bagging is to use bootstrap samples using a standard training set for fitting the k ensemble models. A bootstrap sample of size n is generated by uniformly sampling n instances from the training set with replacement. Random Forest uses a modified tree learning algorithm selecting a random subset of independent variables at each candidate split in the learning process instead of considering the entire variable space. The random forest algorithm assigns a rating based on the performance of the individual decision tree to each tree. Therefore, we use the selection of the independent variable in the respective decision tree together with the rating to obtain a ranking of the relevance of the independent variable for predicting the dependent variable.

There are different measures for evaluating the importance of a variable in tree-based models [40], [41]. We are considering the Gini importance, which is also called mean decrease in impurity (MDI). Additionally, we include an additional variable consisting of noise. We use it to compare the performance of independent variables to the noise as a criterion for normalization similar to the baseline regressor of the wrapper approach. We divide the score of each independent variable by the score of the noise variable. Therefore, we receive a ratio of how much more information a particular variable contains in comparison to a baseline variable.

E. Creating Dependencies

After obtaining the score vector computed in Section III-D, we use this score to create dependencies for the discovered relations. Each of our adapted algorithms returns a vector $scores$ for each variable selection task. This vector assigns each independent variable an individual weight, reflecting its importance for describing the specific dependent variable. Furthermore, we ensure that the given score is normalized. Therefore, we can compare the scores of the different variable selection tasks with each other. Based on this score, we either accept a task and model the found relation as a dependency, or we reject the task as we do not see any valuable dependency in the considered relation.

We achieve this by applying a threshold θ for each selection task. As the methods introduced in Section III-D all support different scores, we define three different thresholds: θ_{Filter} for the filter approach, $\theta_{Wrapper}$ for the wrapper, and $\theta_{Embedded}$ for the embedded approach. Therefore, the method is parameterized with θ as already discussed in Algorithm 1. For any given θ, if the score of a variable is higher than θ we create a dependency from that particular variable to the dependent variable of the current feature selection task v. If none of the scores exceeds the defined threshold value θ, we do not create any dependencies and return an empty set. We analyze the impact of these thresholds in Section IV-A. We describe the procedure for dependency creation more formally in Algorithm 3.

Algorithm 3: Creating the resulting dependencies.

Input: Selection task $t = (v, V_{IND})$,
vector $scores = (s_1, \ldots, s_{|V_{IND}|})$
Output: Set of found dependencies D or $\emptyset$

```
1 Let p, p_i be the corresponding parameters of v, v_i for
    i ∈ 1, ..., |V_IND|.
2 D = ∅
3 foreach v_i in V_IND do
4     if s_i ≥ θ then
5         D = D ∪ (p, p_i)
6 return D
```

Note that in line 1 of Algorithm 3, we assume to have the corresponding parameter names of each vector v_i available. This is necessary as Algorithm 3 models the dependencies between the parameters, not their corresponding measurement vectors v_i. As the parameter names are usually given during implementation, we do not include it as a formal input, but rather retrieve it during execution.

F. Result Filtering

Finally, in the last step of Algorithm 1, we filter all modeled dependencies from Section III-E to reduce the number of resulting dependencies. Since we are extracting dependencies for performance models, some parameters are more critical than others. Hence, although some dependencies might be correct in terms of existing correlations in the monitoring data, the relation between the parameters might not be useful in our performance model. In the following, we present three post-processing steps filtering the dependencies that are important to us.

1) Filtering identical parameters: After observing initial results, we discovered that many identified dependencies are actually due to identical parameters. This is a common practice in software engineering. For example, one parameter p (e.g., a list) is passed to one method m_1, which then forwards this parameter to the next method m_2, which processes it. Therefore, method m_1 and m_2 share the same parameter p, which is correctly modeled and identified by our approach. However, now the parameter values p_{m_1} and p_{m_2} of m_1 and m_2 both exhibit a dependency to the resource demand p_d of method m_2, as m_2 is concerned with the processing of p.

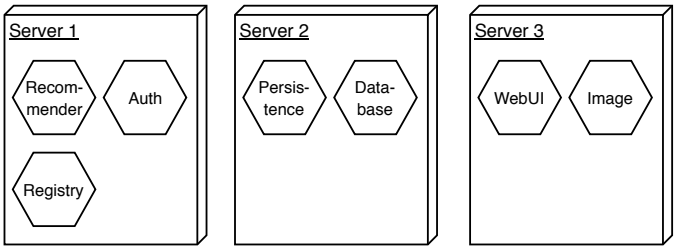

Fig. 5. Deployment of the TeaStore services.

Therefore, we delete the dependency from p_{m_1} to the resource demand of m_2. The relation between p_{m_1} and m_2 is captured by the dependency between p_{m_1} and p_{m_2} and the successive dependency of p_{m_2} to the resource demand of m_2. Note that this step only applies if p_{m_1} and p_{m_2} are exactly identical.

2) Filtering correlating dependencies: It is explicitly the desired behavior of our algorithm to extract two different dependencies d_1 and d_2 to a dependent parameter p_d. Both independent parameters p_1 and p_2 influence p_d, but are not identical (if they are, the dependency is filtered in the previous step.). Now, we analyze the correlation between p_1 and p_2. If both p_1 and p_2 correlate, we have a redundant dependency and therefore want to select the stronger relation. Hence, we analyze the scores of the dependencies of p_1 and p_2 assigned by the feature selection technique in Section III-D. We delete the dependency with the lower score and therefore keep the stronger relation. This reduces over-fitting as we select fewer, but more significant variables. Note that both relations are kept, if there is no strong correlation between p_1 and p_2. In our experiments, a Pearson's r correlation coefficient of at least 0.8 has shown to be sufficient.

3) Graph-based filtering: Our last step is to eliminate all dependencies that do not influence any performance-relevant model parameters. For example, two input parameters might be influencing each other. However, if none of them influences any resource demand, they do not have any (modeled) performance impact. Therefore, capturing the relation between them is useless as it does not give us any performance-relevant information.

Hence, we construct a graph consisting of all model parameters as nodes and all dependencies as directed edges between them. The edges are the opposite direction of the dependency (i.e., if a dependency has a model parameter as target, the model parameter will be the source of the directed edge). Now, we iterate through all performance relevant parameters and perform a breadth-first search. All dependencies that were not discovered by any of the breadth-first searches are deleted. As they are not discovered, they have no path and therefore no (transitive) relation to any of the performance relevant parameters. We consider resource demands as well as loop counts as performance relevant parameters.

IV. Evaluation

We evaluate our approach in a case study using TeaStore, a distributed micro-service benchmark application that represents a webshop for tea [42]. Users can browse the available products by category and look at individual products. After logging in, the user can add items to the shopping cart, modify the content of the shopping cart, and checkout by entering shipping and payment information. Previous orders are tracked on the user's profile page. TeaStore displays advertisements for other products based on the user's previous orders, his current shopping cart, and the item/category he is currently looking at.

TeaStore consists of five services, a database, and a service registry. The `WebUI` service delivers static web pages and fills them with dynamic information by querying the other four services. The `ImageProvider` delivers and caches the product images. Password hashing and session validation are managed by the `Auth` service. The `Persistence` service encapsulates access to the database and provides a caching mechanism. Finally, the `Recommender` service tailors the displayed advertisement to the current user by training a machine learning model on the previous orders. TeaStore uses a modern technology stack and is representative of current distributed micro-service applications [42].

Using this application, we aim to answer the following research questions:

- **RQ1:** *Which feature selection technique is best suited for the identification of parametric dependencies?*
- **RQ2:** *What is the effect of the proposed result filtering approach on the quality of the identified dependencies?*
- **RQ3:** *What is the impact of increased system utilization on the quality of the identified dependencies?*

Based on these research questions, we perform the following three experiments. Section IV-A presents the first experiment, designed to answer **RQ1**. **RQ2** will be analyzed in Section IV-B. Our third experiment – which answers **RQ3** – is presented in Section IV-C.

A. Comparison of Feature Selection Techniques

Our approach is modular and can include any existing feature selection technique. As described in Section III-D, we integrate one technique from each of the three categories of feature selection methods: a filter method, an embedded method, and a wrapper method. We evaluate which of these techniques produces the best results and what the advantages and disadvantages of each technique are.

1) Experiment setup: For this experiment, we distribute TeaStore across three servers as shown in Figure 5: The `Recommender`, the `Auth` and the `Registry` are deployed on Server 1 (Intel Xeon E5-2650 v3, 10 cores). The `Persistence` and the `Database` are co-located on Server 2 (Intel Xeon E5-2650 v4 with, 12 cores). Lastly, the `Webui` and the `Imageprovider` are deployed on Server 3 (Intel Xeon E5-2640 v3, 8 cores). Apart from the CPU, all servers are identical HPE ProLiant DL360 Gen9 servers, equipped with 32 GB of RAM, running Debian 9 and Docker 17.03.2-ce. All services run inside Docker containers, each assigned one core and 4 GB of memory limits to ensure resource isolation. We apply a closed workload of one user with a think time

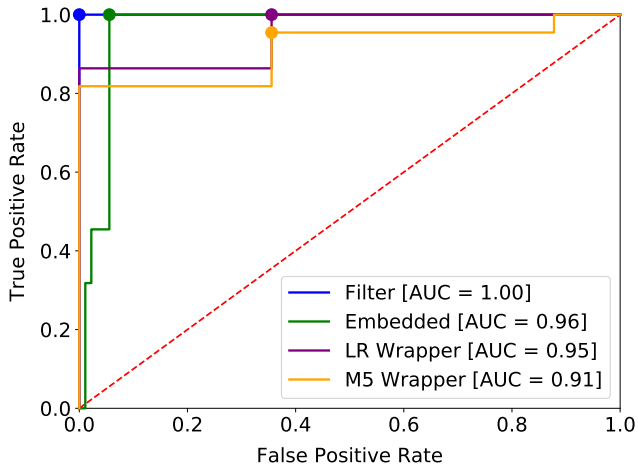

Fig. 6. ROC-curves for the four analyzed features selection techniques, circles indicate the selected threshold.

of zero. The workload profile is similar to TeaStore's browse profile, which covers the full functionality of TeaStore. We use the closed workload so we can ensure that the monitored response time is equal to the resource demand. We run the experiment for two hours, resulting in a total of 2.4 million monitoring records. We implemented our framework using Java and the WEKA library [43] for the machine learning techniques.

In order to evaluate the feature selection techniques in isolation, we apply our approach without the result filtering presented in Section III-F. This enables us to compare the unprocessed results of the different selection techniques. As a gold standard, an expert creates a set of expected dependencies. For this experiment, the expert labels any dependencies that exist and not only the dependencies he would include in a performance model, as we are looking to evaluate the ability of the feature selection techniques to detect any existing dependencies from the monitoring data. As the process of manual labeling is time-intensive and requires in-depth analysis of the source code, we limit the scope of this evaluation to the `Recommender` service.

2) Threshold analysis: All feature selection approaches can be parameterized with a threshold parameter θ as introduced in Section III-E. We study the impact of the threshold parameter θ by analyzing the receiver operating characteristic (ROC) curve of each proposed feature selection technique. Figure 6 depicts the true-positive rate against the false-positive rate. In the area of machine learning, the true-positive rate (aka recall) is defined as the amount of true positives (i.e., the amount of correctly identified dependencies) divided by the sum of the true positives and the false negatives (i.e., the amount of missed dependencies) for any given threshold setting θ. Analogously, the false-positive rate (aka fall-out) is the number of false positives (i.e., the amount of falsely identified dependencies) divided by the sum of false positives and true positives. Simply put, the ROC curve displays the impact of decreasing the threshold θ, as it shows how for a given share of falsely classified dependencies, how many correct dependencies will be found. The area under the ROC curve (AUC) score is the integral over this function and is therefore maximized at 1. Higher values are preferable, as they express the probability of ranking a positive example over a negative one.

TABLE I
ACCURACY OF DIFFERENT FEATURE SELECTION TECHNIQUES.

Approach	TP	FN	FP	TN	Precision	Recall	F1
Filter	22	0	0	90	1.00	1.00	1.00
Embedded	22	0	5	85	0.81	1.00	0.90
LR Wrapper	22	0	32	58	0.41	1.00	0.58
M5 Wrapper	21	1	32	58	0.40	0.95	0.56

Figure 6 depicts the ROC curves for the four approaches, together with the corresponding AUC score. The dashed red line shows the theoretical performance of a random classifier. All approaches clearly outperform a random classifier, each offering a certain trade-off. We note that all approaches achieve an AUC score > 0.9, with the LR Wrapper and the embedded approach slightly outperforming the M5 wrapper. However, the filter approach even exceeds their performance by offering a perfect ranking and an AUC score of 1.0. This means that there exists a threshold resulting in a perfect classification of all possible dependencies.

As our focus is on the detection of dependencies, a high number of true positives is desirable. Therefore, based on the threshold analysis, we choose the thresholds depicted in Figure 6, with a threshold θ_{Filter} of 0.8 for the filter approach, a threshold $\theta_{Wrapper}$ of 200 for the M5 wrapper and LR wrapper, and a threshold $\theta_{Embedded}$ of 30 for the embedded approach.

3) Results: The first thing to note is that the wrapper algorithm using M5 (M5 Wrapper) takes over 2 hours to complete. Therefore, we also include a wrapper variant using linear regression (LR Wrapper), which is expected to be faster than M5. This variant terminates after 10 minutes. In comparison, the embedded approach terminates after 14 minutes, while filter needs only 5 minutes. All but the measured run time for M5 wrapper lie in an acceptable range, as monitoring data from a two-hour period is analyzed. In the following, we will discuss the threshold settings as well as the performance of the individual approaches in terms of accuracy.

The results of the respective approaches using the defined thresholds are presented in Table I. It shows the absolute number of true positives (TP), false negatives (FN), and false positives (FP) of each approach, as well as the gold standard based on human-labelling. We furthermore add the true negatives (TN), that is, the number of correctly not-identified dependencies as well as the precision, the recall, and the F1 score. The precision, calculated as TP / (TP + TN), expresses the share of correct dependencies from all identified ones. Analogously, the recall expresses the share of correct

TABLE II
IMPACT OF EACH RESULT FILTERING STEP.

Result filtering	Identified dependencies			
	Relevant	Irrelevant	Invalid	Total
None	11	94	5	110
Identical (1)	11	45	5	61
1+Correlating (2)	11	35	1	47
1+2+Graph-based (3)	11	8	1	20

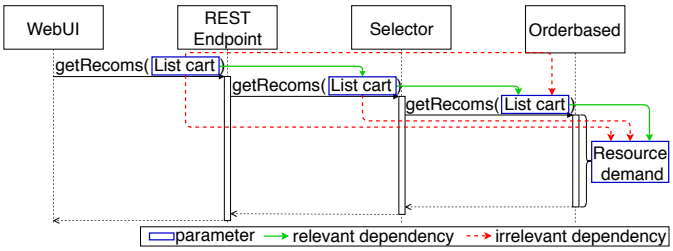

Fig. 7. Call path for recommendations annotated with identified dependencies; class and call names abbreviated, list parameter refers to list size.

dependencies that were found. It is defined as TP / (TP+FN). The F1 score is the harmonic mean of precision and recall.

We observe that the filter approach seems best suited for the automated identification of parametric dependencies (RQ1). It manages to identify all 22 dependencies, without identifying any incorrect dependencies and therefore achieves an F1 score of 1.0. The embedded approach performs reasonably well, as it also identifies all 22 correct dependencies with only five false positives, achieving an overall F1 score of 0.9. In contrast, both wrapper approaches identify 32 additional incorrect dependencies. However, they also identify most of the 22 dependencies. The M5 Wrapper misses one of the correct dependencies, resulting in the lowest F1 score of 0.56.

Summarizing, all approaches manage to identify almost all or all of the required dependencies and achieve a high recall. However, the filter approach outperforms the comparable approaches by achieving a perfect F1 score of 1.0 not having any false positives. We attribute the large number of false positives of the wrapper and the embedded approach to the fact that their respective underlying machine learning techniques are prone to over-fitting. Additionally, the tuning of the hyper-parameters of the underlying machine learning algorithms for both the embedded and wrapper approaches has a significant influence on the classification performance. Overall, we conclude that the wrapper approach is not well suited for the identification of parametric dependencies, as it results in a large number of false positives. Furthermore, the long run time of the M5 wrapper approach (over 2 hours for a single service) makes it inapplicable for larger systems. The embedded approach can be used in scenarios where a human validates the proposed dependencies, as in such cases one may prefer a slight tendency towards false positives in order not to miss any relevant dependencies.

B. Benefits of Result Filtering

In Section III-F, we introduced three steps of filtering the identified dependencies in order to retain only performance relevant dependencies that should be considered for inclusion in a performance model. We now evaluate if our approach accidentally removes any performance relevant dependencies or if it fails to remove any dependencies that are not performance relevant; further, we evaluate the overall quality of the identified dependencies.

1) Experiment setup: We apply the same test setup as before but with traces from the entire TeaStore application as input for our dependency detection approach. This makes it infeasible to have a pre-defined gold-standard by a human expert, as the evaluated system is too complex for detailed manual inspection. Instead, we go through all identified dependencies and label them with respect to our knowledge of the system.

This setup is appropriate for this evaluation step, as here we are only interested in analyzing the impact of the result filtering (RQ2). The results of Section IV-A suggest that the filter approach is the most effective technique to identify dependencies. Hence, in the following experiments, we focus our evaluation on the filter approach.

2) Results: We distinguish between three types of dependencies: (1) relevant dependencies, (2) irrelevant dependencies, and (3) invalid dependencies. Relevant dependencies are the ones we are looking for and that we want to include in our performance model. Irrelevant dependencies are dependencies that are semantically correct, but do not influence the accuracy of the performance model (neither positively, nor negatively). Invalid dependencies are dependencies that are either semantically incorrect or negatively influence the accuracy of the performance model. Hence, we want to focus on relevant dependencies to keep the performance model reasonably sized and understandable for a human. The goal of the filtering step is not to filter invalid dependencies, as it is not possible for an automated approach to decide whether or not a modeled dependency is semantically correct based on the monitoring data. Instead, we focus on reducing the number of irrelevant dependencies as already explained in Section III-F.

Table II presents the impact of the individual steps of the result filtering. Initially, our approach identifies 113 dependencies on the data set, of which 95 are irrelevant and five invalid. After filtering dependencies that involve identical parameters, we are left with 61 dependencies. The first step filters 49 dependencies, all of which are irrelevant. Figure 7 shows some example dependencies that are deleted during Step 1 of the filtering process. The `WebUI` issues a query, requesting a list of product recommendations based on the list of items in the cart of a user. This query is received by the REST endpoint of the `Recommender` component, which then forwards the request to a strategy-selector. The strategy-selector chooses the appropriate algorithm (in this case the `Orderbased` recommender), which then processes

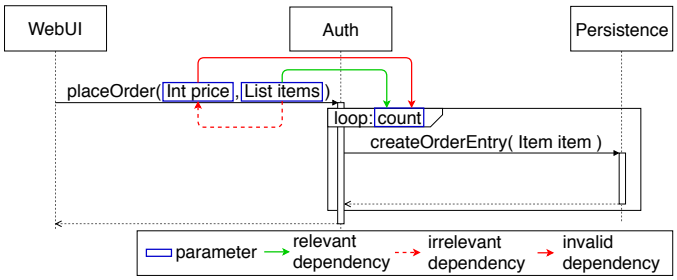

Fig. 8. Call path for purchases annotated with identified dependencies; class and call names abbreviated, list parameter refers to list size.

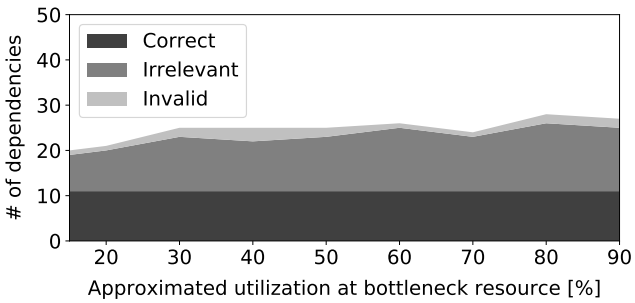

Fig. 9. Classes of identified dependencies from monitoring data of an increasingly utilized system.

the actual request. The resource demand of the `Orderbased` recommender is dependent on the number of items in the cart (i.e., the length of the list `cart`), which is correctly identified. However, as shown in Figure 7, the approach actually detects a dependency from all list-sizes to the respective resource demand. While this is technically correct, we drastically improve readability by removing all red dependencies in Figure 7. We end up with the description of the list being forwarded through several components and its impact on the resource demand of the `Orderbased` recommender, where it finally gets processed. As these types of indirection are a common practice in software engineering, 49 similar cases of parameter pass-throughs can be filtered in Step 1.

The second filtering step deals with correlating dependencies. An example is shown in Figure 8. Here, the `WebUI` places an order which is sent to the `Auth` service to verify that the user is logged in. If so, for each item in the cart, a new `OrderEntry` is created at the `Persistence` service. This represents a loop that is called once for each element in the `items` list. However, we can see that two dependencies influencing the loop count are identified: (1) the size of the list, which does make sense, and (2) the price of the order, which seems strange. Indeed, there is a correlation between the size of the list and the price of the order. The probability of a higher price of the order increases, if more elements are in the cart. Therefore, both the cart size and the total price of the order correlate with the loop count. However, the price of the order directly influencing the loop count is an invalid dependency since it would imply that an increase in item prices would lead to a higher loop count – which is invalid. Now, as there exists a dependency and therefore a correlation between the size of the cart and the total cart price, Step 2 deletes the dependency of `price` to the loop count, since the relation involving the size of the cart is stronger. In addition to ten irrelevant dependencies, the second step filters four invalid dependencies, resulting in a total of 14 filtered dependencies.

Note that the irrelevant dependency mapping the size of the cart to its price, as depicted in Figure 8, is not deleted in Step 2. However, the graph-based filtering from Section III-F filters this dependency. The graph-based filtering will not mark this dependency during the breadth-first step, as it is not related to any performance relevant parameter. The irrelevant dependency depicted in Figure 8 will therefore be deleted by the third filter step, together with 27 other irrelevant dependencies.

Summarizing, we can say that all three filtering steps combined delete 86 irrelevant and four invalid dependencies. Therefore, over 90% of "unwanted" dependencies are filtered. After analyzing the remaining one invalid dependency, we conclude that its existence is rooted in the applied workload. Our approach finds a dependency between the number of requested images and their requested size at the `ImageProvider` component. This is due to the implementation of the `WebUI`, requesting either a list of small product review pictures or one single high-resolution in-detail product image. Therefore, this relation can be observed from the monitoring data. However, we classify it as invalid as this relationship is not based on the software code, but rather on the workload profile of the specific component. Without in-depth knowledge about the application, it is not possible for our approach to filter these rare edge-cases as – based on the monitoring data – this observation is actually correct.

Additionally, seven of the remaining eight irrelevant dependencies are added due to the addition of the discussed invalid dependency. These are technically correct (e.g., modeling the parameter passing of the requested image size), but finally relate to the discussed invalid dependency and are therefore marked as irrelevant as they do not model a performance-relevant property. If we were to remove the discussed invalid dependency by hand, these seven irrelevant dependencies would subsequently be filtered by the described filtering mechanisms, as they describe a relation to this invalid dependency. After doing so, we are left with one single irrelevant dependency.

Summarizing, we conclude that our approach was able to enrich the base performance model of TeaStore with 11 relevant dependencies learned from monitoring data. We can furthermore calculate the precision using relevant dependencies as true positives and invalid ones as false positives (given that irrelevant dependencies can neither be counted as true positives nor false positives). With 11 relevant and a single invalid dependency, our approach achieves a precision of 91.7% after all filtering steps.

C. Impact of System Utilization

Our approach approximates the resource demands based on the observed response time, as detailed in Section III-A2. This approximation works well under low resource utilization, but becomes inaccurate with increasing load [3]. We now evaluate the impact of different utilization levels on the accuracy of our approach. To this end, we conduct a third measurement series.

1) Experiment setup: We start with the identical setup as in Section IV-B. However, we now increase the number of users concurrently using TeaStore to cause higher system utilization. We first determine that the `WebUI` is the bottleneck service by analyzing the utilization of each container. Next, we empirically determine the workload intensity of a closed workload that leads to a certain CPU-utilization of the bottleneck service. For each utilization level, we collect monitoring data for two hours. We feed this data into the filter-based dependency detection algorithm already used in Section IV-B and label the resulting dependencies according to the same scheme. The experiment setup used in Section IV-B created $\sim 15\%$ utilization at the `WebUI`. In this experiment, we added measurements for utilization levels of approximately 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90% CPU utilization at the bottleneck resource.

2) Results: The results of our analysis are shown in Figure 9. We can observe that the number of irrelevant and invalid dependencies increases with higher system utilization. This is expected, as the accuracy of the response time estimation for resource demands decreases with increasing utilization [3]. However, our approach still identifies all correct dependencies even with increasing resource utilization at the bottleneck service. Furthermore, we observe that there exist stronger factors influencing the accuracy of the approach than the load level alone, as the number of irrelevant and invalid dependencies is not increasing monotonously.

This is contrary to our initial expectation as inaccuracies in the estimated resource demands should lead to inaccuracies in the dependency detection. We conclude that the inaccuracies of resource demand estimates do not affect the approach as strongly as expected since the relative differences between the resource demand estimates remain the same. Even though the increased load has a negative effect on the accuracy of the estimated resource demands, all demands get affected to a similar degree. Therefore, the relative proportions between the estimates stay intact and are therefore still observable by our approach. We conclude that the approach can still be used on monitoring data from higher utilization levels, however, manual review of the identified dependencies becomes increasingly necessary as the precision slightly deteriorates.

V. Limitations

From our experiments, we can not conclusively show that the thresholds determined in Section IV-A are transferable to other systems or if the threshold tuning is an elemental step of the algorithm calibration for each system. Exploring this would require a case study spanning a large number of representative systems and workloads, including a variety of approaches for hyper-parameter tuning.

Parameters describing the current state of a system, for example, the number of entries in a database, can also influence the performance of a software component [44]. Our approach could be extended to support such dependencies by adding state variables into every single feature selection task. However, this would drastically increase the run-time of the approach and manual identification of state variables by an expert.

We evaluate the capability of our approach to accurately identify the existence of parametric dependencies. However, we do not investigate how much the prediction accuracy of a performance model improves after including the identified parametric dependencies since this depends on the applied modeling formalism, model solver, dependency characterization approach, the system under consideration, and the granularity of the model. As part of our future work, we aim to integrate our approach into an existing model extraction pipeline and evaluate the impact of our approach in an end-to-end scenario.

VI. Conclusion

In this paper, we introduced an approach to identify parametric dependencies for performance models. The presented approach uses monitoring data from a running system without any further knowledge about the application, the deployment, or the component structure. This monitoring data is then analyzed and correlations between different parameters are identified with the use of different feature selection approaches from the area of machine learning. In our evaluation, we analyze three different approaches for feature selection and show that a filter-based approach outperforms the competing techniques in terms of solution quality and run-time. Furthermore, we analyze different post-processing steps intended to reduce the number of irrelevant dependencies on a microservice reference application. The post-processing steps eliminate over 90% of the unwanted dependencies and increase the precision for 11 correctly identified dependencies to 91.7%. The limitations of the proposed approach are analyzed by conducting additional experiments with different load levels. We observe that although the precision of the approach suffers with increasing load levels, it still correctly identifies all dependencies under high utilization.

This work represents a significant step towards the vision of self-aware performance models [20]. We aim to augment our technique with automated model extraction and characterization techniques, and integrate the identified dependencies into machine learning based performance models [45]. This will enable a performance model to autonomously learn and improve itself during system operation in a production environment.

Acknowledgements

This work was funded by the German Research Foundation (DFG) under grant No. (KO 3445/11-1).

References

[1] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, *Modeling and Simulating Software Architectures: The Palladio Approach*. MIT Press, 2016.

[2] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bähr, "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 432–452, 2017.

[3] S. Spinner, G. Casale, F. Brosig, and S. Kounev, "Evaluating Approaches to Resource Demand Estimation," *Perform. Evaluation*, vol. 92, pp. 51 – 71, October 2015.

[4] H. Koziolek, "Parameter dependencies for reusable performance specifications of software components," Ph.D. dissertation, Universität Oldenburg, 2008.

[5] D. Hamlet, "Tools and experiments supporting a testing-based theory of component composition," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 3, p. 12, 2009.

[6] E. Bondarev, P. de With, M. Chaudron, and J. Muskens, "Modelling of input-parameter dependency for performance predictions of component-based embedded systems," in *31st Conference on Software Engineering and Advanced Applications*, 2005, pp. 36–43.

[7] S. Eismann, J. Walter, J. von Kistowski, and S. Kounev, "Modeling of Parametric Dependencies for Performance Prediction of Component-based Software Systems at Run-time," in *IEEE International Conference on Software Architecture*, 2018.

[8] M. Sitaraman, G. Kulczycki, J. Krone, W. Ogden, and N. Reddy, "Performance specification of software components," *SIGSOFT Software Engineering Notes*, vol. 26, no. 3, pp. 3–10, 2001.

[9] M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, "The stochastic rendezvous network model for performance of synchronous client-server-like distributed software," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 20–34, 1995.

[10] E. Pozzetti, G. Serazzi, V. Vetland, and J. A. Rolia, "Characterizing the resource demands of tcp/ip," in *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, ser. HPCN Europe '95. London, UK: Springer-Verlag, 1995, pp. 79–85.

[11] D. A. Menascé, "A framework for software performance engineering of client/server systems," in *Int. CMG Conference*, 1997, pp. 460–469.

[12] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, Aug. 2010.

[13] K. Krogmann, M. Kuperberg, and R. Reussner, "Using genetic search for reverse engineering of parametric behavior models for performance prediction," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 865–877, 2010.

[14] C. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker, "How is Performance Addressed in DevOps?" in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019.*, 2019, pp. 45–50.

[15] M. Courtois and M. Woodside, "Using regression splines for software performance analysis," in *Proceedings of the 2nd International Workshop on Software and Performance*, 2000, pp. 105–114.

[16] F. Brosig, N. Huber, and S. Kounev, "Automated extraction of architecture-level performance models of distributed component-based systems," in *26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*, Oread, Lawrence, Kansas, November 2011.

[17] V. Ackermann, J. Grohmann, S. Eismann, and S. Kounev, "Black-box learning of parametric dependencies for performance models," in *13th International Workshop on Models@run.time (MRT), co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*, ser. CEUR Workshop Proceedings, October 2018.

[18] J. R. Quinlan *et al.*, "Learning with continuous classes," in *5th Australian joint conference on artificial intelligence*, vol. 92. Singapore, 1992, pp. 343–348.

[19] T. K. Ho, "Random decision forests," in *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ser. ICDAR '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 278–282.

[20] J. Grohmann, S. Eismann, and S. Kounev, "The Vision of Self-Aware Performance Models," in *Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA 2018)*, 2018.

[21] J. Walter, C. Stier, H. Koziolek, and S. Kounev, "An Expandable Extraction Framework for Architectural Performance Models," in *Proceedings of the 3rd International Workshop on Quality-Aware DevOps (QUDOS'17)*. ACM, April 2017.

[22] C. E. Hrischuk, C. M. Woodside, J. A. Rolia, and R. Iversen, "Trace-based load characterization for generating performance software models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 1, pp. 122–135, Jan. 1999.

[23] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proceedings of the 5th International Workshop on Software and Performance*, ser. WOSP '05. New York, USA: ACM, 2005, pp. 147–158.

[24] A. Mizan and G. Franks, "An automatic trace based performance evaluation model building for parallel distributed systems," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 61–72, Sep. 2011.

[25] S. Spinnner, J. Grohmann, S. Eismann, and S. Kounev, "Online model learning for self-aware computing infrastructures," *Journal of Systems and Software*, vol. 147, pp. 1 – 16, 2019.

[26] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering*, 2012, pp. 247–248.

[27] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[28] D. A. Menascé, L. W. Dowdy, and V. A. F. Almeida, *Performance by Design: Computer Capacity Planning By Example*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[29] F. Brosig, S. Kounev, and K. Krogmann, "Automated extraction of palladio component models from running enterprise java applications," in *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS '09. Brussels, Belgium: ICST, 2009, pp. 10:1–10:10.

[30] J. Grohmann, N. Herbst, S. Spinner, and S. Kounev, "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, March 2018, pp. 473–480.

[31] A. Bauer, J. Grohmann, N. Herbst, and S. Kounev, "On the Value of Service Demand Estimation for Auto-Scaling," in *Proceedings of 19th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems (MMB 2018)*. Springer, February 2018.

[32] J. Grohmann, N. Herbst, S. Spinner, and S. Kounev, "Self-Tuning Resource Demand Estimation," in *Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017)*, July 2017.

[33] J. Grohmann, S. Eismann, A. Bauer, M. Züfle, N. Herbst, and S. Kounev, "Utilizing Clustering to Optimize Resource Demand Estimation Approaches," in *Proceedings of Workshop on Self-Aware Computing (SeAC 2019) as part of FAS*(IEEE ICAC/SASO) conferences companion*, June 2019.

[34] F. Willnecker, M. Dlugi, A. Brunnert, S. Spinner, S. Kounev, W. Gottesheim, and H. Krcmar, "Comparing the accuracy of resource demand measurement and estimation techniques," in *Computer Performance Engineering*, M. Beltrán, W. Knottenbelt, and J. Bradley, Eds. Cham: Springer International Publishing, 2015, pp. 115–129.

[35] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, Oct 2004.

[36] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003.

[37] M. A. Hall, "Correlation-based feature subset selection for machine learning," Ph.D. dissertation, University of Waikato, Hamilton, New Zealand, 1998.

[38] R. J. Hyndman and A. B. Koehler, "Another look at measures of forecast accuracy," *International Journal of Forecasting*, vol. 22, no. 4, pp. 679 – 688, 2006.

[39] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[40] M. Kursa and W. Rudnicki, "Feature selection with the boruta package," *Journal of Statistical Software, Articles*, vol. 36, no. 11, pp. 1–13, 2010.

[41] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[42] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '18, September 2018.

[43] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[44] L. Happe, B. Buhnova, and R. Reussner, "Stateful component-based performance models," *Software and Systems Modeling*, vol. 13, no. 4, pp. 1319–1343, 2014.

[45] S. Eismann, J. Grohmann, J. Walter, J. von Kistowski, and S. Kounev, "Integrating Statistical Response Time Models in Architectural Performance Models," in *Proceedings of the 2019 IEEE International Conference on Software Architecture (ICSA)*, March 2019, pp. 71–80.