Enabling Consistency between Software Artefacts for Software Adaption and Evolution

David Monschein
Karlsruhe Institute of Technology, Germany
david.monschein@student.kit.edu

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

Robert Heinrich
Karlsruhe Institute of Technology, Germany
robert.heinrich@kit.edu

Anne Koziolek
Karlsruhe Institute of Technology, Germany
koziolek@kit.edu

***Abstract*—Short development times of software became crucial to stay competitive. However, the quality should not suffer from the faster development processes, which is why increasingly more automation is gaining ground in this context. If models are involved in the development process and used for performance prediction, there are delays due to emerging inconsistencies between different software artifacts. The elimination of these inconsistencies is a time consuming, complex and error prone activity. Currently, there are already approaches for automated consistency preservation of software artifacts. Nevertheless, the limited scope in terms of supported change scenarios is a significant disadvantage.**

Therefore, we present a comprehensive approach for the maintenance of consistency between the system design and adaptive as well as evolutionary changes. In comparison to existing approaches, the consistency preservation has been significantly extended in our approach to cover a multitude of changes resulting from adaptation and evolution. Ultimately, several validation steps were integrated into the approach, enabling continuous assessment regarding the quality of the consistency preservation. In a case study based evaluation, we measured the accuracy of the updated models and associated performance predictions.

***Index Terms*—Software Architecture, Architecture-based Performance Prediction, Model Consistency, Run-time Model, Self-Validation, DevOps**

I. Introduction

While software systems are becoming increasingly more complex, development cycles get shorter and shorter due to the growing popularity of practices such as DevOps [4]. This is problematic, because due to high dynamics in terms of changes to the requirements and the infrastructure, there is the risk that the implemented application (source code) continuously drifts away from the software design (architecture). In particular, this drift is driven by software evolution and adaptation. For example, software adaption includes scenarios like reconfigurations of the deployment, which are common and occur frequently. For this reason, it is necessary to ensure consistency between the system design and adaptive as well as evolutionary changes. Otherwise, it is challenging to assess the impact of design decisions. This is important, because quality characteristics like performance heavily depend on architectural design decisions.

There are several approaches that can serve as a basis for architectural decisions. One of them is to measure the effects of the possible decisions. However, in many cases the effort would be too high, which is a significant problem (P1). Other approaches such as the software performance engineering [2, 37] use quantified methods to assess performance aspects already in early development phases. Commonly, models are used to represent the architecture of the software and can provide reliable predictions about quality properties. The improved planning possibilities are a major advantage of the models, but a significant drawback is that the modeling process is time-consuming and challenging (P2). Furthermore, it is important that accurate models are available at any time, i.e. during evolution, but also throughout adaptations while the system is in operation (P3). In order to keep the models up to date during operation, it is necessary to observe the application, for example with the help of monitoring techniques. A key disadvantage of monitoring is the introduced overhead by the measurements, which can negatively affect the performance of the observed software (P4). A general disadvantage of using models is the fact that their quality is uncertain. In other words, there is no trust in the predictions made on the basis of the model. One must rely on the model to represent the real application well (P5).

There are various approaches that address the introduced problems. These can roughly be divided into three categories further described in the remainder of the paper. The first category includes approaches that focus on consistency preservation at Development time (Dev-time) (time frame until the deployment of an application). The second category covers approaches that target consistency preservation at Operation time (Ops-time) (time frame beginning with the deployment). Procedures that target Dev-time, mainly build upon rules and associated actions, which define relationships between the architectural model and the source code [22, 5, 42]. In contrast, for the techniques that target Ops-time, consistency preservation relies primarily on the collection of monitoring data and a subsequent analysis to build an up-to-date architectural model [3, 24, 44, 10]. Finally, there is a third category, which consists of approaches that consider the maintenance of consistency relationships at Dev-time and at Ops-time [26, 18, 20]. However, all three categories of approaches suffer from gaps in the extent of the consistency preservation; so far, the system composition has not yet been considered. Furthermore, automated analyses of the quality of the derived models have

978-1-7281-6261-4/21/$31.00 ©2021 IEEE.
DOI: 10.1109/ICSA51549.2021.00009

not been addressed at all or only in prototypical fashion.

In this paper, we present an approach that embeds architecture models in software development and keeps them consistent throughout evolution and adaption. Detailed investigations regarding the accuracy of the performance predictions have been integrated, so called validations, which can be used to reveal inaccuracies of the models. In addition, these validations make it possible to adjust the monitoring in order to observe only the parts of the software that are poorly represented in the models. Thus, the monitoring overhead can be kept to a minimum. This approach is an extension of Continuous Integration of Performance Models (CIPM) [26, 25] and iObserve [9, 10], which significantly expands the maintenance of consistency relationships and the self-validation concept. The objective of the iObserve approach is to enrich an architecture model created at Dev-time with the help of monitoring data collected at Ops-time [10]. The CIPM approach considers Dev-time changes like source code modifications and some Ops-time changes, with the goal of providing an up-to-date architectural model for performance prediction [25].

The main contributions of our approach are:

- **(i)** In-depth automated consistency preservation between system design and adaptive as well as evolutionary changes. By covering both Dev-time and Ops-time, an architecture model is available at any point in time and can be used for performance predictions (P3). In comparison to existing approaches, the system composition is included in the process. It is analyzed at Dev-time based on the source code and at Ops-time based on monitoring events. With the help of a newly introduced data structure, the so-called Service-Call-Graph (SCG), the system composition in the architecture model can now be updated in an automated way. All in all, the modeling effort can be reduced (P2) and with the help of accurate models, architecture decisions can be evaluated in a profound way (P1). Sec. V describes this enhancement in more detail.
- **(ii)** Extension of the self-validations, based on the CIPM approach. The scope of self-validations is increased, recognized inaccuracies are passed into the consistency preservation process, which allows the process to react to them (P5). As a result, the maintenance of consistency relationships becomes more dynamic and can learn from previous shortcomings. The component of our approach that realizes this strategy is called Validation Feedback Loop (VFL) and is introduced in Sec. VI.
- **(iii)** Design and implementation of a transformation pipeline that uses monitoring events as input to update the architecture model at Ops-time. Based on the validation results, the monitoring is adjusted at Ops-time. In case of large inaccuracies, the level of detail of the monitoring, denoted as granularity, can be increased. On the other hand, the monitoring granularity can be reduced if the model is accurate enough. As a result, the arising overhead can be reduced (P4). Sec. VII provides an overview of the pipeline structure.

The evaluation was carried out based on the case studies CoCoME [33, 11] and TeaStore [14]. These were selected because they are established in the field of performance modeling and have also been used for the evaluation of related approaches [30, 14]. The results show a high quality of the derived models and adequate scalability characteristics of the implemented approach. Furthermore, it was shown that by controlling the granularity of the monitoring, the arising overhead could be reduced.

II. Foundations

The paper relies on several concepts and approaches, the most important of them are introduced within this section.

A. Palladio Component Model

The Palladio Component Model (PCM) is a model-based approach for the analysis of software architectures [31]. The PCM focuses on the evaluation of performance aspects, such as the detection of bottlenecks and scalability problems. The PCM makes it possible to make statements about the impact of design decisions at an early stage of development. As a result, high costs for wrong decisions in development can be avoided and scalable software can be designed.

The PCM divides the specification of the software architecture into five different meta-models. The *Repository Model* contains a repository with components and interfaces. In addition, it includes the descriptions of the behavior of services that are provided by these components. These are called Service Effect Specifications (SEFFs). The *System Model* describes the composition of the software architecture, based on the components and interfaces specified in the repository. The *Resource Environment Model* reflects the actual hardware environment which is composed of containers with processing resources (e.g., CPU) and links between them. The mapping from the system composition (System Model) to the resources (Resource Environment Model) is described by the *Allocation Model*. Finally, the *Usage Model* defines the behavior of users, i.e. the way in which they interact with the system.

B. iObserve

iObserve considers the adaptation and evolution of cloud-based systems as two interwoven processes [9]. The main idea is to use Ops-time observations to detect changes during the operation and to reflect them by updating an architecture model which is then applied for quality predictions. The PCM is used as the basis for the quality predictions and Kieker is used for monitoring the system during operation [13, 9]. Briefly summarized, iObserve collects monitoring data at Ops-time with the help of Kieker and applies necessary changes to the architecture model (PCM instance) which originated at Dev-time.

Adaptation and evolution are interwoven and shared models are used throughout the application life-cycle to close the gap between Ops-time and Dev-time. The mapping between elements in the architecture model and corresponding elements in the source code is based on the runtime architecture correspondence model [10, 29].

C. Continuous Integration of Performance Models

CIPM (Continuous Integration of Performance Models) [25, 26] aims to update the architectural model after each source code commit at Dev-time and after changes at Ops-time. To accomplish this, CIPM extends the rule-based incremental extraction of the architecture models from Langhammer [22] with further functionalities like the incremental calibration. Furthermore, it integrates parts of iObserve to update the PMs according to Ops-time changes, i.e., changes in usage or deployment. The incremental calibration of CIPM estimates resource demands and recognizes parametric dependencies. The findings are used to update the Repository Model.

For the calibration, CIPM uses an adaptive monitoring to collect the required data while the application is running. Thereby, not only information about service calls is collected (so called *coarse-grained monitoring*), but also about the behavior within the service calls (*fine-grained monitoring*) is tracked [26]. It is called adaptive, because just the parts that have been changed are instrumented fine granular. Besides, the fine-grained monitoring can be deactivated to reduce the monitoring overhead. To control which services have to be observed fine granular, CIPM proposes the so-called *Instrumentation Model*, which stores instrumentation points and their status.

After the calibration of the architecture model, CIPM starts a self-validation process. It simulates the calibrated model and compares the simulation results to the monitoring data. Recognized deviations can then be used to adjust the monitoring and calibration process in order to eliminate them.

III. Running Example

In order to illustrate the concepts of our approach, we will introduce the TeaStore as a running example [14]. The TeaStore is a web-based application which, as the name already implies, implements a shop for all kinds of tea. The application is based on a distributed microservice architecture and designed to be suitable for the evaluation of approaches in the field of performance modeling [14].

The TeaStore consists of six components: *Registry*, *Image Provider*, *Auth*, *Persistence*, *Recommender* and *WebUI*. All components register themselves at the registry, which makes them available for the individual components. This enables client-side load balancing. Communication between the components is based on the widely used representational state transfer (REST) standard [8]. There are four different Recommender components (implementations) with identical functionality, which are used to suggest related products to the users. They offer two different services, "train" and "recommend". The "train" service trains the recommender with orders from the past, whereas the "recommend" service suggests products to the users that are related to their current shopping cart.

The TeaStore is a very dynamic application, especially due to the load balancing, which allows replications and de-replications without great effort. Therefore, changes of the system landscape at Ops-time are common. In such cases, it is inevitable that an associated architecture model must be constantly adapted in order to remain consistent.

An up-to-date architecture model can be used to answer various questions regarding performance, scalability and other quality aspects. Therefore, the goal of our approach is to provide an accurate architecture model at any point in time and the required manual effort should be kept as low as possible.

IV. Approach Overview

In this section, we provide a rough overview of the conceptual design of the approach and explain how it enables consistency between architectural models and source code as well as how Ops-time events are used to update the models.

The design and implementation is dedicated to the use of the PCM for describing software architectures. The main reasons for choosing the PCM included its precise realization of the principles of component-based software engineering and the rich set of analysis features [31]. It also enabled the seamless integration of existing functionalities from CIPM [26, 25] and iObserve [9, 10].

Fig. 1 shows a summary of the approach including the most important processes, artefacts and their interactions.

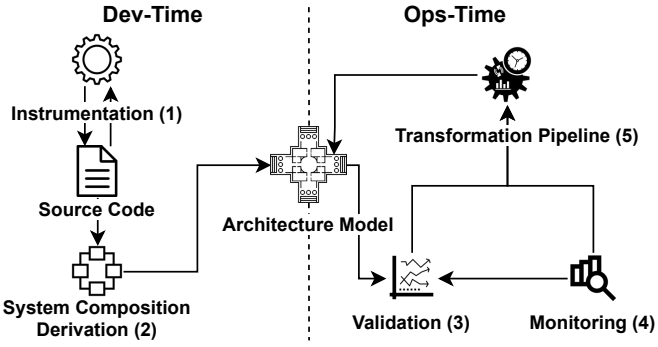

Fig. 1. Graphical overview of the most important processes and artefacts

Considering source code changes at Dev-time as starting point, we update the Repository Model and establish a mapping between entities in the source code and corresponding elements in the architecture model based on CIPM [25].

In order to obtain information about the running application, it is necessary to monitor it. Our monitoring is based on the idea that we weave information from the mapping into the source code during the instrumentation process [25]. This allows us to easily link the monitoring data to the elements in the architecture model. The instrumentation establishes a fine-grained monitoring for all services to be monitored (1). This fine-granular monitoring can be changed dynamically by defining an endpoint that specifies the monitoring granularity.

Using the mapping and the source code as input, our approach supports the derivation of a System Model at Dev-time (2). This process does not work fully automated, it is necessary that a developer manages the resolution of conflicts. At Ops-time, the same method is used within the transformation pipeline (5), but the conflicts are resolved without assistance. Details about the procedure are presented in Sec. V.

At Ops-time, the current architectural model is simulated and the results of the simulation are compared to the monitoring data (4). This process is called validation (3) and the resulting data is used as input for the transformation pipeline (5). In addition, the validation results are used to adjust the granularity of the monitoring. This concept is based on the CIPM approach, which calls this step "self-validation" [26]. Subsequently, the monitoring data (4), together with the results of the validation (3) are used as input for a transformation pipeline (5), which updates all parts of the architecture model accordingly. Consequently, the pipeline is able to adjust its behavior according to the validation results. Sec. VII discusses the transformation pipeline in more detail, while Sec. VI presents the methodology for validating the models.

V. System Composition Consistency

The procedure is composed of two phases. First, a graph is created that represents which services of the application lead to the execution of which other services. To represent this, a new data structure was introduced, the Service-Call-Graph (SCG). In the second step, the SCG is used to derive the model that reflects the system composition. In the following, we first discuss the structure of the SCG, then the strategies for extracting a SCG and finally we describe how a SCG can be used to build and update a System Model.

a) Service-Call-Graph (SCG): A SCG can be displayed as a directed graph where a service/resource container pair is a node and an edge indicates that a service on a particular container leads to the execution of another service on a certain container. Fig. 2 shows an exemplary SCG. It is a truncated version of the SCG from the TeaStore case study (cf. Sec. III). The sample graph shows that the "confirmOrder" service

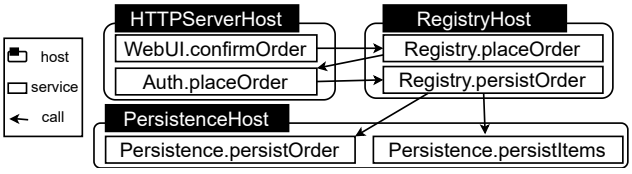

Fig. 2. Graphical representation of an exemplary Service-Call-Graph (SCG)

of the WebUI component calls the "placeOrder" service of the Registry component. The corresponding hosts, on which the services are executed, are displayed as black boxes. This information is not mandatory, i.e. an SCG can be created without specifying the corresponding hosts. This becomes clearer in the following, when the procedures for building the SCG are described.

b) SCG Extraction at Dev-time: For the automated generation of a SCG at Dev-time, three things are needed: source code, an existing Repository Model and a mapping between elements of the source code and the elements in the Repository Model. First, a code analysis is used to build a call graph that indicates the invocation dependencies between methods [41]. Using the mapping, these methods can be assigned to the corresponding services in the Repository Model. Thus, the method call graph can be transformed into a SCG.

An important characteristic of such analysis is that we have to make simplifying assumptions. Because of the limited ability of static source code to approximate the execution semantics, it is uncertain how decisions are actually made during execution [21]. This is the reason why we need the support of a developer to deduce the system composition at Dev-time. We also leave out information about the hosts in the generated SCG, since this information is usually not available at Dev-time.

c) SCG Extraction at Ops-time: At Ops-time, monitoring data serves as primary information source. With the help of the embedded information about the mapping, trees of service calls can be built (so-called service call traces). These in turn reflect which services call each other, which is the information needed for the construction of a SCG. Through observation, we obtain in-depth insight about the flow semantics and can record the actual execution paths, which is an important advantage compared to a static code analysis. In contrast to the SCG we are building at Dev-time, we also attach the information about the host to the nodes in the SCG.

Even service call traces that extend beyond system boundaries are recorded by the monitoring. When a call leaves a system, the necessary information is attached so that the traces can be merged subsequently in a central backend that receives the monitoring data [28].

d) System Model Generation from SCG: In the second stage, the extracted SCG is used to build a System Model. As mentioned before, in some cases a developer is needed which resolves conflicts that result from the lack of information at Dev-time. At Ops-time, this support is not needed, because it is ensured that conflicts are resolved automatically.

In the process there are several recurring tasks that are introduced in advance. By assembling these tasks, the whole process can be explained very compactly and clearly thereafter. In addition, the tasks are demonstrated by means of the running example.

- **Component Wiring**: Components provide various interfaces (so-called provided roles) and require certain interfaces (so-called required roles). In order for the component to be operational, the required roles must be connected to matching provided roles of other components. In the following, we refer to the instance of a component whose required roles need to be satisfied as component instance (CI). In the wiring step, we first group all components that can be reached in the SCG via services of the CI and map them to the matching required roles of the CI. If several components fit to the same required role, a *Component Selection* is performed. Once the appropriate component has been selected, an instance of this component is provided by the *Instance Selection* step. Finally, the required roles of the CI are linked to the provided roles of the determined component instances.
- **Component Selection**: This is a conflict that must be resolved. At Dev-time, a developer must select the component to be used. At Ops-time, the component that has actually been used is reflected in the service call traces.

- **Instance Selection**: The aim of this step is to provide a component instance for a given component type. If no instance of this type exists, it is obvious that a new one must be created. If one or more instances are present, one of them must be selected, or a new instance must be created. At Dev-time the selection must be done manually. At Ops-time there can be no such conflict, because we can distinguish the instances based on the service call traces.

In the following, we will walk through the *Component Wiring* step based on an instance of the WebUI component (see Fig. 2) and assuming that the procedure takes place at Dev-time. First, all outgoing edges of services that belong to the WebUI are examined and their destination component is discovered. In our example this is only one edge, the one from the service "WebUI.confirmOrder" to "Registry.placeOrder", meaning that the WebUI component has a required role which is fulfilled by a provided role of the Registry component. A *Component Selection* is not necessary here, as it is clear that the Registry component is used. If there would be a second component which offers the same interface as the Registry component, then a selection might be mandatory. For the instance selection, it matters whether an instance of the Registry component already exists. If this is not the case, we create one and otherwise the developer must determine which one to use (or whether a new one should be created). Finally, the WebUI component instance is attached to the Registry component instance and thus, the corresponding required role is fulfilled.

Based on the partial steps, the process can now be described with three activities:

1) A developer has to specify the interfaces that should be provided by the system. At Ops-time, we assume that these interfaces are already specified and do not change during execution. For each of these interfaces, the suitable components are collected and then a *Component Selection* with a subsequent *Instance Selection* are executed. The result is a list of component instances (Instance List) that provide and expose the interfaces which are offered by the system.
2) If the *Instance List* is empty, the procedure is finished. Otherwise the *Component Wiring* procedure is executed for each component instance in the list.
3) If new component instances were created during step two, all of them are added to the *Instance List* and the second step is executed again.

VI. Validation Feedback Loop

An important innovation of our approach is the so-called Validation Feedback Loop (VFL). The idea is to continuously evaluate the accuracy of the performance predictions related to the derived models.

In order to determine the prediction accuracy of a model, it is necessary to have a baseline. We use measurements from the real system as reference, which are available in the form of monitoring data. By comparing simulation data of the models with the monitoring data, it can be assessed how well the models represent the actually observed system in its current state. In case of high deviations, it is possible to intervene.

Fig. 3 presents the design of the VFL and visualizes the interaction with the transformation pipeline.

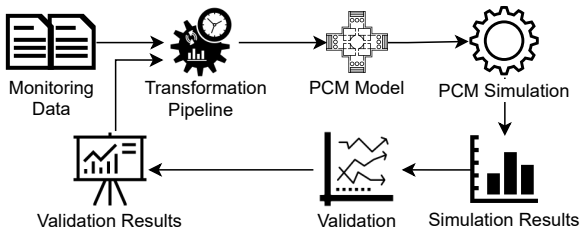

Fig. 3. Illustration of how the Validation Feedback Loop (VFL) works in combination with the transformation pipeline

The simulation results are grouped into so-called measuring points, i.e., the points at which measurements were taken. For example, a typical measuring point is the response time of a service. To be able to compare the simulation results with the monitoring data, we have to map the monitoring data to the corresponding measuring points in an upfront step. This assignment is based on the mapping between the Repository Model and the source code.

After the monitoring data has been mapped to the measuring points of the simulation results, we have two distributions for each measuring point. These are compared and different metrics are calculated to determine how close the simulation results are to the actual measured values. The reasons that were considered when selecting the metrics are outlined and summarized in Sec. VIII-B2. We have used the following ones to compare the two distributions: Wasserstein distance [34], Kolmogorov–Smirnov test (KS test) [16] and the difference of conventional statistical measures (e.g., average and quartiles).

All transformations within the transformation pipeline can access these metrics and use them to improve the resulting model. In addition, the metrics are used to adjust the granularity of the monitoring at Ops-time. With this extension, the monitoring for certain services can be deactivated if predefined criteria are met. As a result, it is possible to balance the trade-off between effort and granularity of the monitoring.

VII. Transformation Pipeline

The core at Ops-time is a transformation pipeline that processes the monitoring data with the goal of updating a PCM instance (similar to iObserve [9]). In order to realize the transformation pipeline we used a tee and join pipeline architecture [6], based on parts of iObserve [9] and the CIPM approach [25]. Fig. 4 shows the structure of the entire transformation pipeline. The following paragraphs explain the transformations in more detail.

a) $T_{Preprocess}$: In a pre-processing step, the monitoring data is filtered and converted into suitable data structures. One example is the construction of service call traces, which can be used to analyze the structure of the system composition. Furthermore, the monitoring data is divided into two sets.

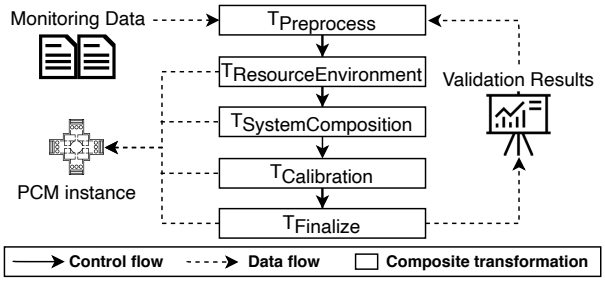

Fig. 4. Overview of all higher-level transformations within the pipeline

One set is used as input for the following transformations (training set) and the second set is used for the validations of the architecture model (validation set). The reason for this split is that the validation is much more meaningful when it is carried out on data that the transformations have never seen.

b) $T_{ResourceEnvironment}$*:* The Ops-time information (monitoring) is written to the so-called Runtime Environment Model (REM). The REM contains details about the hosts and the connections between them. We defined rules and actions based on the consistency preservation platform Vitruvius [15] to keep our REM consistent with the resource environment in the corresponding PCM. The advantages and the idea behind the REM are twofold: it ensures the separation of concerns principle (Dev-time vs. Ops-time concerns) and it allows to establish a mapping between the Ops-time environment and the elements in the architecture model via the correspondence mechanism of Vitruvius.

c) $T_{SystemComposition}$*:* Next, the system composition and the associated deployment are examined. First, a SCG is built and the system composition is updated with the methodology that was introduced in Sec. V. Afterwards, the SCG is used to recognize deployments and undeployment events. This is possible because the SCG also contains information about the hosts.

d) $T_{Calibration}$*:* The stochastic expressions in the Repository Model are calibrated and the user behavior is analyzed. The calibration of stochastic expressions within the Repository Model is largely based on the CIPM approach [25]. These were extended by an adaptive dimension, which dynamically varies the parameters of the regressions based on the validation data. The analysis of the user behavior and the corresponding extraction of user scenarios are based on iObserve [10, 9]. It uses clustering techniques to form user groups and derive usage scenarios which are integrated into the Usage Model.

e) $T_{Finalize}$*:* In the final step (*Finalize*), the derived architectural model is validated using the procedure introduced in Sec. VI. The validation results are investigated and depending on them, the granularity of the monitoring is adjusted. Based on configurable criteria, the fine-granular monitoring is activated or deactivated. This can be illustrated by means of the "confirmOrder" service of the running example: the measured response times are compared with those obtained by simulating the architecture model. If the deviation matches defined criteria (e.g., distance of the means is less than 5ms), the fine-grained monitoring for the service is deactivated.

Afterwards, the validation results are entered as input into the next execution of the pipeline.

VIII. Evaluation

This section presents the evaluation design and results.

A. Evaluation Goals

The evaluation focuses on three important characteristics of our approach. First, this concerns the accuracy of the derived models and the associated performance predictions, which are essential so that they can be used for the assessment of architectural design decisions (**Goal 1**). Second, the monitoring overhead was measured and investigated in detail. It is important that the overhead does not distort the performance characteristics of the application under consideration, otherwise our approach would not be applicable in practice (**Goal 2**). Third, the scalability of the transformation pipeline was analyzed. This is an important quality property, since it is required to react quickly to events at Ops-time in order to keep the models up-to-date (**Goal 3**). The analysis of these goals was divided into the following research questions (RQs):

- **Goal 1: Accuracy of the derived models:**
  - **RQ-1.1**: How accurate is the extraction of the System Models at Dev-time?
  - **RQ-1.2**: Are the Resource Environment Model, the Allocation Model and the System Model adjusted correctly at Ops-time when applying software adaption scenarios?
  - **RQ-1.3**: How accurate are the performance predictions of the derived models, when comparing them to the data which results from monitoring the application?
- **Goal 2: Monitoring overhead:**
  - **RQ-2.1**: How significant is the overhead that is caused by the monitoring?
  - **RQ-2.2**: To what extent does the dynamic adaption of the monitoring granularity help to reduce the monitoring overhead?
- **Goal 3: Scalability of the transformation pipeline:**
  - **RQ-3.1**: How do the transformations within the pipeline scale with increasing numbers of monitoring records as input?

The first two research questions, RQ-1.1 and RQ-1.2, address the important extensions of this approach regarding the scope of consistency maintenance (*contribution i*) and aim to demonstrate that they are working as intended. RQ-1.3 is then used to assess the overall quality of the derived architectural models, since all parts of the architectural model must be adequately adjusted for accurate simulation results. Consequently, this question is used to ensure that our extensions work as expected in conjunction with adopted functionalities from existing approaches.

RQ-2.1 addresses the overhead caused by the monitoring and aims to analyze the impact of the overhead on the observed application. Based on RQ-2.2, we will investigate

whether the combination of the transformation pipeline and the validation process successfully reduce the monitoring overhead by continuously adjusting the granularity of the monitoring (contributions ii and iii).

RQ-3.1 focuses on ensuring the scalability of the transformations within the pipeline to ensure general applicability. If all transformations possess adequate scalability properties, it can be assumed that the pipeline delivers suitable response times in relevant application scenarios.

B. Evaluation Metrics

The metrics used in the evaluation can be divided into the following two categories.

1) Model Conformity: For our evaluation, we need to be able to compare several model types. The applied metric is based on the Jaccard similarity coefficient (JC) [7]. The JC is well suited because the calculation is straightforward, its significance is high, and it is easy to interpret. The JC is defined as follows:

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In this context, A and B are sets. Normally, the JC is used to quantify the equality of sets. However, we can also apply this concept to models by considering the model as a set of model elements. If the resulting JC is equal to 1, the two models under investigation are considered to be fully identical.

2) Distribution Comparison: To compare distributions we use three types of metrics: conventional statistical measures [40], non parametric tests (KS test) [36] and distance functions (Wasserstein) [27]. These metrics can be used to compare the distributions of the monitored response times (reality) with the simulated response times of the models (prediction).

As non-parametric test we used the Kolmogorov–Smirnov test (KS test) [16]. It calculates the maximum distance between the Cumulative Distribution Functions (CDFs). The minimum is 0 (if both distributions are perfectly identical) and the closer it is to 1, the more different are the distributions under observation. Normally, this non-parametric test is used to check whether two random variables originate from the same underlying distribution. Therefore, this metric is not ideal for our use case. In particular, higher values are not meaningful because shifts of distributions are not considered. This fact was taken into account in the evaluation and is a reason why we use the KS test in combination with other metrics.

The Wasserstein metric is a distance measure for distributions [27]. In simple terms, it describes how much a distribution must be changed in order to be transformed into the other one. An advantage of this metric is that, unlike the KS test, it is not sensitive to shifts of the distributions. A drawback, however, is that the result is an absolute number that cannot be easily interpreted without having a baseline.

The classic statistics metrics (e.g. mean or quartiles) are calculated for both distributions and the distance is calculated. Using these commonly known metrics, it is possible to get an overview of the two distributions and their dissimilarities in a simple and quick way.

C. Experiment Setup

Within the evaluation, we followed a case study based approach and used **CoCoME** [33] and **TeaStore** [14]. Both are widely used in the field of performance modeling and represent a relevant business use case [30, 14]. First, we used CoCoME to get preliminary results and afterwards the TeaStore was investigated to provide detailed conclusions[1] [28]. In this paper, we focus on the results in the context of the TeaStore.

The evaluation procedure is based on the research questions and can be divided into the following three experiments:

a) Experiment 1 (E1): Extraction of a System Model at Dev-time using the procedure presented in Sec. V. As input, the source code of the case studies is used. Finally, the resulting model is compared to a reference model that represents the actual system composition. Based on the results of this experiment, RQ-1.1 can be answered.

b) Experiment 2 (E2): The foundation of this experiment is a number of predefined change scenarios, such as replications, allocations, workload changes and system composition changes. The *Scenario Generator* selects a number of change scenarios and for each of the selected scenarios, a reference model is generated. Since the *Scenario Generator* knows the executed change, it also knows how the reference model must look like. Besides the list of changes, another output is the *Change Orchestration* component, which applies the selected changes at Ops-time. The modified system is observed and the arising monitoring data is used as input for the transformation pipeline. Finally, the resulting models are compared to the reference models and deviations are detected by applying the Jaccard coefficient (JC). We concentrated on the System Model, the Resource Environment Model and the Allocation Model (RQ-1.2). The Usage Model was excluded from the scope of this paper, as it has already been extensively addressed in the context of the iObserve approach [10, 9].

Furthermore, the monitoring data is compared to simulation results of the derived models to estimate how well the models represent the performance characteristics of the actual system. Here, we used the metrics that were introduced in Sec. VIII-B2 to answer RQ-1.3. In order to quantify the accuracy of the performance predictions, the following procedure is used:

1) Execution of the experiment, storage of the derived models and the associated monitoring data.
2) Examination of the models at different points in time based on simulations.
3) Comparison of the simulation results with the monitoring data in two different ways:
   (a) Comparison with monitoring data collected *after* the construction of the model under consideration (forward prediction). This allows us to make statements about how well the derived model can be used to predict future scenarios.
   (b) Comparison with monitoring data that was collected chronologically *before* the construction of the model under consideration (backward prediction). In this way, it can

[1] https://github.com/CIPM-tools/CIPM-Pipeline

be determined how well the model is able to reproduce previously observed situations.

It must be taken into account that in future/previous points in time other system compositions, runtime environments or user behavior are present (due to the simulated changes). Therefore, the considered model must be adapted in such a way that it correctly reflects the system at the respective point in time.

The experiment was executed 10 times, each time with different changes. Every 5 minutes the next change is executed. In total, the experiment is carried out for 180 minutes. To ensure that the forward prediction and the backward prediction are meaningful, we only consider the time period between the 30 and the 150 minute mark.

Within this experiment, we only consider the TeaStore case study and focused on the service "confirmOrder". To increase the complexity of the service, it was slightly modified. After the order has been processed within the "confirmOrder" service, the Recommender component is re-trained in our experiment. As a result, the response time of the "confirmOrder" service increases with a growing number of orders in the database, since the execution time of the Recommender depends on the number of orders and the applied recommending strategy. Thus, we want to make sure, that our approach recognizes parametric dependencies and the system composition.

Another dimension that is considered while performing E2 is the measurement of the emerging monitoring overhead and the amount of generated monitoring data. This information can then be used to answer RQ-2.1 and RQ-2.2.

c) Experiment 3 (E3): In this experiment, synthetic monitoring data is generated and used as input for the individual transformations within the transformation pipeline. First, we identify the parameters that influence the execution times and subsequently we generate the monitoring data in such a way, that it produces worst-case execution times. By means of this experiment, scalability questions can be answered (RQ-3.1).

D. Model Accuracy

First, the accuracy of the System Model extraction at Dev-time was investigated based on experiment E1. The key findings of the experiment for the selected case studies are shown in Table I.

TABLE I
RESULTS FOR DERIVING THE SYSTEM MODEL AT DEV-TIME

Casestudy	JC	Model Elements	Conflicts
CoCoME	1.0	16	2
TeaStore	1.0	18	5

It can be seen that in both cases an identical model to the reference model was built, as the JC equals to one. Furthermore, the table shows the number of elements in the final model and the number of conflicts that had to be resolved manually during the process. According to these results, RQ-1.1 can be answered as it became clear that the system compositions were reflected correctly in the extracted System Models.

In experiment E2, the accuracy of three model types at Ops-time was investigated simultaneously: System Model, Resource Environment Model and Allocation Model.

TABLE II
MODEL ACCURACY WHEN SIMULATING ADAPTION SCENARIOS

Change Type	**Minimum Jaccard Index**		
	System	Allocation	Resource Environment
(De-)/Allocation	1.0	1.0	1.0
(De-)/Replication	1.0	1.0	1.0
Migration	1.0	1.0	1.0
System Composition	1.0	1.0	1.0
Workload	1.0	1.0	1.0

The obtained results (see Table II) show that all three model types are correctly inferred in all cases. Consequently, it can be concluded for RQ-1.2 that the change scenarios were recognized and correctly reflected to the models.

In the next step, the models were systematically simulated and compared to monitoring data as explained in Sec. VIII-C.

Figure 5 shows the median of the Wasserstein distance over time for the forward and backward prediction, regarding the response times of the "confirmOrder" service. In addition, Table III summarizes the Wasserstein distance, the KS test and the distance of the mean value over time.

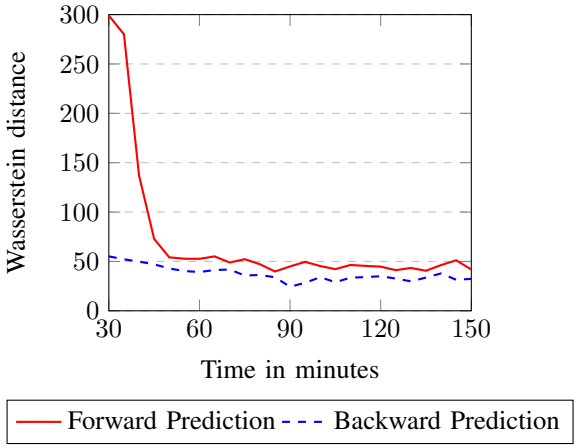

Fig. 5. Median Wasserstein distance of the forward and backward prediction for a model derived at a given point in time - regarding the response times of TeaStores *confirmOrder* service

The Wasserstein distance of the forward prediction is still very high at the beginning. The main reason for this is that the amount of data is not yet sufficient to estimate the behavior in general. After a short time, however, it decreases significantly and gets closer to the values of the backward prediction. Thereafter, the value settles at a level close to the backward prediction. Consequently, the derived models are very well suited to make predictions about the performance characteristics of the application.

Furthermore, this implies that the parametric dependency of the response time on the number of orders in the database was detected. The observations can be confirmed using additional metrics (see Table III).

TABLE III
AGGREGATED METRICS OF THE FORWARD PREDICTION AND BACKWARD PREDICTION OVER TIME

Metric	Q1	Q2	Q3	Mean	Std Dev
Forward Prediction					
Wasserstein	44.732	47.179	52.718	70.991	68.524
KS test	0.125	0.143	0.168	0.199	0.131
Mean distance	13.198	25.965	41.924	61.573	91.482
Backward Prediction					
Wasserstein	32.622	35.011	41.246	37.268	7.618
KS test	0.098	0.112	0.121	0.114	0.031
Mean distance	15.560	26.981	34.373	23.329	9.053

The interpretation of the mean distance depends on the average response time of the "confirmOrder" service, which amounted to approximately 1000ms in the experiment. In addition, it is important to note that the average and standard deviation of the forward prediction metrics are strongly affected by the high values at the beginning of the experiment. With the help of these findings, RQ-1.3 can be answered: all metrics show that the derived models represent the already observed behavior very well and on the other hand can also be used to predict the performance for scenarios that have not been observed so far.

E. Monitoring Overhead

In the first step, we measured the total monitoring overhead resulting from a single call to the "confirmOrder" service. We distinguish between the overhead caused by the fine-grained monitoring and the coarse-grained monitoring. The following listing shows the respective overheads in milliseconds and the corresponding standard deviations:

- **Fine-grained**: 1.755ms ($\sigma = 0.389ms$)
- **Coarse-grained**: 0.731ms ($\sigma = 0.144ms$)

Using these values, we can answer RQ-2.1. Considering the average response time of about 1000ms of the "confirmOrder" service within E2, the determined overhead is negligible and does not significantly influence the response time of the service.

In the second step, the overall monitoring overhead was analyzed and observed over time. Every 5 minutes, the sum of the monitoring overhead from the last 5 minutes was calculated. When considering the entire overhead, it is important to note that there are parts of the monitoring that are independent of the granularity of the monitoring, such as observing resource utilizations. Fig. 6 shows the results which were obtained by forming the median from multiple experiment executions.

The dashed line in the graph highlights point in time when the first switch from fine-granular monitoring to coarse-granular monitoring happened. Based on this graph, an answer to RQ-2.2 can be given. After the validation process begins to find individual services that are well represented in the model, the granularity of monitoring is reduced. This happens after about 20 minutes. At the peak, a monitoring overhead of approx. 1.362s arises and then decreases to an average of 0.822s as the experiment progresses. This corresponds to a reduction of **39.65%**. Together with the evaluation results about the model and the prediction accuracy, it can be concluded that the validation process successfully identified services that are well represented in the model and then reduced the granularity of the monitoring accordingly. Ultimately, this leads to a significant reduction of the monitoring overhead.

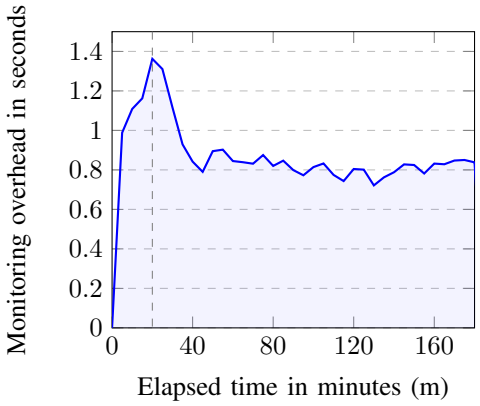

Fig. 6. Median of the arising monitoring overhead over time when considering five minute intervals

F. Transformation Pipeline Scalability

Using experiment E3 we examined the scalability of each transformation within the pipeline. In this paper, we will only summarize the results briefly. First, for all transformations that were taken over from iObserve (e.g., user behavior update), it was shown that the results of the scalability analysis are in line with the findings from the work in the context of iObserve [10]. For the newly introduced and implemented transformations ($T_{Preprocess}$, $T_{ResourceEnvironment}$, $T_{SystemComposition}$ and $T_{Finalize}$), a linear relationship between the number of monitoring records as input and the execution time of the respective transformation was found in all considered cases [28]. Thus, it can be concluded for RQ-3.1 that the overall scalability of the pipeline is adequate and does not lead to unexpectedly high execution times.

G. Threats to Validity

The identification of the threats to validity is based on guidelines for case study research [32]. Therefore, we distinguish between four dimensions of validity: *internal validity*, *external validity*, *construct validity* and *conclusion validity*.

Internal Validity: A threat to validity is the selection of metrics within the evaluation. For comparing distributions we used the Wasserstein distance, the KS-test and conventional statistical measures. These have been used in related studies [12, 25, 43] and by combining them we minimize the risk that a single metric distorts the evaluation results. The same applies for the JC, which has also been used in related work [10]. Another threat concerns the execution of experiment E1 (see Sec. VIII-C). The conflicts that occurred during the execution of the experiment were resolved manually, so the outcome depends on the person who performs the experiment. If this person does not know the system composition well enough and makes incorrect decision, the calculated JC would be lower.

External Validity: Another threat to validity is the selection of case studies. It may be possible that the results obtained from the case studies are not representative. To avoid this, we selected CoCoME and TeaStore, which both are widely used in research and address common business use cases [30, 14]. By combining the two selected case studies, the risk of non-representative results is further reduced.

Construct Validity: In the evaluation we rely on a combination of synthetically generated monitoring data and monitoring data generated directly by executing a case study. For the synthetically generated data, external factors such as the Ops-time environment or the type of load testing can be excluded. When observing the case study, however, the quality of the monitoring events must be ensured. Therefore, we decided to use the Kieker framework with extensions that have already been implemented in previous projects [13, 25].

Conclusion Validity: The subjectivity of a researcher must be avoided when interpreting the evaluation results. Many different, well known metrics have been used which are easy to interpret. All conclusions and arguments are well structured and based on metrics, making them easy to understand.

IX. Related Work

In the following, we will pick up the structure from the introduction and divide the related approaches into three groups according to the scope of their consistency management.

The first group consists of the approaches that focus on consistency at Dev-time. For example, Just-In-Time Tool for Architecture Consistency (JITTAC) [5] detects inconsistencies between architecture models and source code, but the elimination is not automated. The mbeddr approach [42] uses a single underlying model for implementing, testing and verifying system artefacts like component-based architecture. The goal is to avoid inconsistencies between artifacts, so that no explicit consistency maintenance processes are needed. Additionally, several reverse engineering tools exist (e.g., [1, 23]), which extract the architectural model from source code at Dev-time. In comparison to our approach, the related works of this category strictly target the maintenance of consistency at Dev-time.

The second category includes approaches that deal with the extraction and consistency preservation of architectural models based on observed Ops-time events. The approaches [3, 24, 44, 38] extract component-based architectural models from monitoring data. Furthermore, [24] also detects changes such as migrations and reflects them in the model. In addition, there are several other approaches for extracting and updating models at Ops-time, which are summarized in [39]. Drawbacks of these approaches include the continuously high monitoring effort required to extract or update models and that no information is collected about the accuracy of the models (no validation).

The third category covers approaches whose scope spans Dev-time and Ops-time. For example, Konersmann proposed the integration of information about the architecture model directly into the code via annotations [18, 17]. This enables the dynamic generation of an architecture model from the source code via transformations [18]. He also extended his work to synchronize allocation models with running software [19]. The approach of Krogmann [20] uses both source code and monitoring events to extract an architectural model. However, it applies a very fine-grained monitoring which causes a high monitoring overhead and makes it unsuitable for the use in production. Compared to our approach, both presented works show a much smaller scope of consistency preservation (e.g., limited recognition of adaption scenarios). Furthermore, they do not automatically analyze the quality of the models.

X. Conclusion and Future Work

In this paper, we presented an approach for maintaining the consistency between software artifacts, with a special focus on supporting evolution and adaptation scenarios. It provides up-to-date architecture models both at Dev-time and at Ops-time, along with a high degree of automation. We introduced a strategy that supports the consistency preservation of the system composition between the architectural model and the realization of the considered application. In addition, the Validation Feedback Loop was presented, which continuously analyzes the prediction accuracy of the models and passes the results to the transformation pipeline, which updates the architectural model at Ops-time. Based on the Validation Feedback Loop, the extent of the monitoring is dynamically refined to minimize the resulting overhead. These innovations close gaps of existing approaches, especially regarding the scope of consistency maintenance and the automated analysis of the model quality.

The evaluation is based on the case studies CoCoME [33] and TeaStore [14]. The accuracy of the derived models and the applicability of the consistency maintenance process were demonstrated. Besides, we measured the emerging monitoring overhead and revealed, that by continuously adjusting the monitoring based on the validation results, the arising overhead can be reduced. Finally, we analyzed the scalability characteristics of the transformation pipeline and discovered that all transformations within the pipeline scale adequately.

It turned out that there is still potential for extensions and improvements in some areas. Especially the validations that were integrated within the pipeline open up many opportunities for enhancements. Future work will focus on using the validation results to improve the accuracy of the updated model and associated performance predictions by incremental optimization of the parametric dependencies using genetic algorithms [43]. In addition, it is planned to consider additional scenarios for the evaluation to demonstrate the general applicability. Finally, for broad practical applicability, the consistency rules used for updating the architecture models at Dev-time have to be extended to update the models for systems using different architecture-inducing technologies [35].

Acknowledgements

This work is funded by the DAAD (German Academic Exchange Service), the DFG (German Research Foundation) – project number 432576552, HE8596/1-1 (FluidTrust), and the KASTEL institutional funding.

References

[1] Becker et al. “Reverse engineering component models for quality predictions”. In: *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE. 2010.

[2] C. Murray Woodside et al. “The Future of Software Performance Engineering”. In: *Future of Software Engineering (FOSE ’07)* (2007), pp. 171–187.

[3] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. “Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems”. In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’11. USA: IEEE Computer Society, 2011, pp. 183–192.

[4] Andreas Brunnert et al. *Performance-oriented DevOps: A Research Agenda*. Tech. rep. SPEC-RG-2015-01. SPEC Research Group - DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), Aug. 2015.

[5] Jim Buckley et al. “JITTAC: A Just-in-Time tool for architectural consistency”. In: *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 1291–1294.

[6] F. Buschmann. *Pattern-orientierte Software-Architektur: ein Pattern-System*. Professionelle Softwareentwicklung. Addison-Wesley, 1998.

[7] Hans-Friedrich Eckey, Reinhold Kosfeld, and Martina Rengers. *Multivariate Statistik*. Jan. 2002.

[8] Otávio Freitas Ferreira Filho and Maria Alice Grigas Varella Ferreira. “Semantic Web Services: A RESTful Approach”. In: *IADIS International Conference WWWInternet 2009*. IADIS, 2009, pp. 169–180.

[9] Robert Heinrich. “Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications”. In: *ACM SIGMETRICS Performance Evaluation Review* 43.4 (2016), pp. 13–22.

[10] Robert Heinrich. “Architectural runtime models for integrating runtime observations and component-based models”. In: *Journal of Systems and Software* 169 (2020).

[11] Robert Heinrich et al. “A Platform for Empirical Research on Information System Evolution”. In: *27th International Conference on Software Engineering and Knowledge Engineering*. 2015, pp. 415–420.

[12] Robert Heinrich et al. “Integrating business process simulation and information system simulation for performance prediction”. In: *Software & Systems Modeling* 16.1 (2017), pp. 257–277.

[13] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. “Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis”. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22–25, 2012: ACM, Apr. 2012, pp. 247–248.

[14] Jóakim von Kistowski et al. “TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research”. In: *2018 IEEE 26th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2018, pp. 223–236.

[15] Heiko Klare et al. “Enabling consistency in view-based system development – The Vitruvius approach”. In: *Journal of Systems and Software* 171 (2021).

[16] “Kolmogorov–Smirnov Test”. In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 283–287.

[17] M. Konersmann et al. “Towards architecture-centric evolution of long-living systems (the ADVERT approach)”. In: *9th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA’13), Vancouver, Canada, June 17-21, 2013*. Association for Computing Machinery (ACM), 2013, pp. 163–168.

[18] Marco Konersmann. “Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code”. PhD thesis. May 2018.

[19] Marco Konersmann and Jens Holschbach. “Automatic Synchronization of Allocation Models with Running Software”. In: *Softwaretechnik-Trends* 36.4 (2016).

[20] Klaus Krogmann. *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*. Vol. 4. The Karlsruhe Series on Software Design and Quality. KIT Scientific Publishing, 2012.

[21] William Landi. “Undecidability of Static Analysis”. In: *ACM Lett. Program. Lang. Syst.* 1.4 (Dec. 1992), pp. 323–337.

[22] Michael Langhammer. “Automated Coevolution of Source Code and Software Architecture Models”. PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2017. 259 pp.

[23] Michael Langhammer et al. “Automated extraction of rich software models from limited system information”. In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE. 2016.

[24] Robert von Massow, André van Hoorn, and Wilhelm Hasselbring. “Performance Simulation of Runtime Reconfigurable Component-Based Software Architectures”. In: *Software Architecture*. Ed. by Ivica Crnkovic, Volker Gruhn, and Matthias Book. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 43–58.

[25] M. Mazkatli et al. “Incremental calibration of architectural performance models with parametric dependencies”. In: *IEEE 17th International Conference on Software Architecture (ICSA 2020); Salvador, Brazil, November 2-6, 2020*. 17th International Conference on Software Architecture. ICSA 2020 (Salvador da Bahia, Brasilien,

Nov. 2–6, 2020). IEEE Computer Society, Los Alamitos, 2020, pp. 23–34.

[26] Manar Mazkatli and Anne Koziolek. “Continuous Integration of Performance Model”. In: *Proc. of the 4th International Workshop on Quality-Aware DevOps in Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE ’18. Berlin, Germany: ACM, 2018, pp. 153–158.

[27] Facundo Mémoli. “Gromov-Wasserstein Distances and the Metric Approach to Object Matching.” In: *Foundations of Computational Mathematics* 11.4 (2011), pp. 417–487.

[28] David Monschein. “Enabling Consistency between Software Artefacts for Software Adaption and Evolution”. Master Thesis. Karlsruher Institut für Technologie, 2020.

[29] R. Heinrich, E. Schmieders, R. Jung, K. Rostami, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl. “Integrating run-time observations and design component models for cloud system analysis”. In: *9th Int’l Workshop on Models@run.time* (2014), pp. 41–46.

[30] Ralf Reussner et al. *Managed Software Evolution*. Springer, Cham, June 2019.

[31] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, Oct. 2016. 408 pp.

[32] Per Runeson et al. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. Wiley Publishing, 2012.

[33] S. Herold et al. “CoCoME – the common component modeling example”. In: *The Common Component Modeling Example* (2008), pp. 16–53.

[34] F. Santambrogio. *Optimal Transport for Applied Mathematicians: Calculus of Variations, PDEs, and Modeling*. Progress in Nonlinear Differential Equations and Their Applications. Springer International Publishing, 2015.

[35] Yves R. Schneider and Anne Koziolek. “Towards Reverse Engineering for Component-Based Systemswith Domain Knowledge of the Technologies Used”. In: *Proceedings of the 10th Symposium on Software Performance (SSP)*. Softwaretechnik Trends. 2019, pp. 35–37.

[36] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. 4th ed. Chapman and Hall/CRC, 2007.

[37] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2003.

[38] Simon Spinner et al. “Online model learning for self-aware computing infrastructures”. In: *Journal of Systems and Software* 147 (2019), pp. 1–16.

[39] Michael Szvetits and Uwe Zdun. “Systematic Literature Review of the Objectives, Techniques, Kinds, and Architectures of Models at Runtime”. In: *Softw. Syst. Model.* 15.1 (Feb. 2016), pp. 31–69.

[40] Graham Upton and Ian Cook. *A Dictionary of Statistics*. Oxford University Press, 2008.

[41] Raja Vallée-Rai et al. “Soot: A Java Bytecode Optimization Framework”. In: *CASCON First Decade High Impact Papers*. CASCON ’10. Toronto, Ontario, Canada: IBM Corp., 2010, pp. 214–224.

[42] Markus Voelter et al. “Mbeddr: An Extensible C-Based Programming Language and IDE for Embedded Systems”. In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH ’12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 121–140.

[43] Sonya Voneva et al. “Optimizing Parametric Dependencies for Incremental Performance Model Extraction”. In: *ECSA ’20: Proceedings of the 14th European Conference on Software Architecture*. accepted, to appear. 2020.

[44] Jürgen Walter et al. “An Expandable Extraction Framework for Architectural Performance Models”. In: *Proceedings of the 3rd International Workshop on Quality-Aware DevOps (QUDOS’17)*. ACM, Apr. 2017.