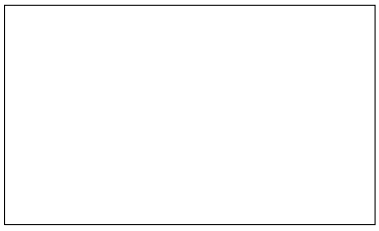
Graphical Abstract

Continuous Integration of Architectural Performance Models with Parametric Dependencies – The CIPM Approach

Manar Mazkatli,David Monschein,Martin Armbruster,Robert Heinrich,Anne Koziolek



Highlights

Continuous Integration of Architectural Performance Models with Parametric Dependencies – The CIPM Approach

Manar Mazkatli,David Monschein,Martin Armbruster,Robert Heinrich,Anne Koziolek

- Commit-based updates of architectural performance models at Dev-time.
- Automated adaptive instrumentation of source code parts changed by the recent commit.
- Incremental calibration of performance models parameters with parametric dependencies
- Measurements-based updates of architectural performance models at Ops-time.
- Self-validation and continuous improvement of architectural performance models.

Continuous Integration of Architectural Performance Models with Parametric Dependencies – The CIPM Approach

Manar Mazkatli*, David Monschein, Martin Armbruster, Robert Heinrich and Anne Koziolek

KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany

ARTICLE INFO

Keywords:
Software Architecture, Architecture-based Performance Prediction, Models' Consistency, Models Parametrization with Parametric Dependencies, Self-Validation, DevOps Pipeline

ABSTRACT

Explicitly considering the software architecture supports efficient assessments of quality attributes. In particular, Architecture-based Performance Prediction (AbPP) supports performance assessment for future scenarios (e.g., alternative workload, design, deployment, etc.) without expensive measurements for all such alternatives.

However, accurate AbPP requires an up-to-date architectural Performance Model (aPM) that is parameterized over factors impacting performance like input data characteristics. Especially in agile development, keeping such a parametric aPM consistent with software artifacts is challenging due to frequent evolutionary, adaptive and usage-related changes. The shortcoming of existing approaches is the scope of consistency maintenance since they do not address the impact of all aforementioned changes. Besides, extracting aPM by static and/or dynamic analysis after each impacting change would cause unnecessary monitoring overhead and may overwrite previous manual adjustments.

In this article, we present our Continuous Integration of architectural Performance Model (CIPM) approach, which automatically updates the parametric aPM after each evolutionary, adaptive or usage change. To reduce the monitoring overhead, CIPM calibrates just the affected performance parameters (e.g., resource demand), using adaptive monitoring. Moreover, CIPM proposes a self-validation process that validates the accuracy, manages the monitoring and recalibrates the inaccurate parts. As a result, CIPM will automatically keep the aPM up-to-date throughout the development time and operation time, which enables AbPP for a proactive identification of upcoming performance problems and evaluating alternatives at low costs.

CIPM is evaluated using three case studies, considering (1) the accuracy of the updated aPMs and associated AbPP and (2) the applicability of CIPM in terms of the scalability and the required monitoring overhead.

1. Introduction

Software systems can fail or not be used as expected if they do not meet the performance objectives. Nowadays, most developers follow agile practices (e.g., DevOps software development [10]) and apply application performance management to measure the current performance of the system. Here, architectural design decisions are made throughout the development [67] since there usually is no dedicated architecture design phase nor architecture model. However, assessing the impact of such design decisions on performance is expensive because it requires setting up test environments and measurements for all design alternatives (**P1**).

Instead of relying on measurements in real environments, software performance engineering [90, 67] uses models to predict the software performance, identify potential issues and respond to them earlier [5]. In particular, architectural performance modeling approaches [63] model the system at an architecture level, without implementation details. In general, architecture models help in increasing the human understandability of the system and consequently the productivity of software development [59]. Moreover, architecture models can be a good basis for predicting the performance of architectural design decisions [63].

A problem of applying Architecture-based Performance Prediction (AbPP) in agile development is that modeling is a time-consuming process (**P2**). Moreover, agile developers do not trust the models since they are just approximations [91] and there is no validation of the accuracy of AbPP (**P3**).

Moreover, accurate AbPP is challenging for various reasons. First, aPMs can be outdated due to frequent software changes (**P4**). For example, continuous integration of source code at the Development time (Dev-time) can affect the accuracy of aPMs (**P4.1**). Similarly, the adaptive changes at Operation time (Ops-time), like changes in system composition and deployment, also affect the aPMs (**P4.2**). Second, the accuracy of AbPP depends mainly on the estimated Performance Model Parameters (PMPs), such as resource demand. These parameters can depend on influencing factors that may vary over Ops-time, e.g., usage profile and execution environment. Considering these factors during parameterizing PMPs allows AbPP for unseen states, e.g., AbPP for unseen workloads. Ignoring the so-called *parametric dependencies* [7]) can lead to inaccurate AbPP for design alternatives (**P5**). The parameterized PMPs can also be falsified over time by the aforementioned software changes. Re-estimating all PMPs frequently after each impacting change causes monitoring overhead (**P6**), because PMPs are mostly calibrated by dy-

mazkatli@kit.edu (M. Mazkatli); david.monschein@alumni.kit.edu (D. Monschein); martin.armbruster@kit.edu (M. Armbruster); robert.heinrich@kit.edu (R. Heinrich); koziolek@kit.edu (A. Koziolek)

https://mcse.kastel.kit.edu/staff_Mazkatli_Manar.php (M. Mazkatli); https://dsis.kastel.kit.edu/staff_robert_heinrich.php (R. Heinrich); https://mcse.kastel.kit.edu/staff_Koziolek_Anne.php (A. Koziolek)

ORCID(s): 0000-0003-4261-8477 (M. Mazkatli); 0000-0003-4303-0712 (D. Monschein); 0000-0002-2554-4501 (M. Armbruster); 0000-0003-0779-9444 (R. Heinrich); 0000-0002-1593-3394 (A. Koziolek)

namic analysis of the whole system [68].

Keeping the aPMs as well as their parameterized PMPs consistent with the running system, which is iteratively and incrementally evolving over time, requires repeated manual effort (**P7**). Hence, the efficient maintenance of the consistency between the *parameterized* architectural Performance Model (aPM) and the software system is an important aspect for more comprehensive of the system's architecture and proactive performance management with an accurate AbPP.

1.1. State of the Arts

Some approaches suggested partial automation of the consistency maintenance between software artifacts. These approaches can be divided into two main categories (C1 and C2): The first Category (C1) includes approaches that reverse-engineer the current architecture based on static analysis of source code [2, 48, 6], dynamic analysis [8, 84, 85] or both [40, 44, 46, p. 137]. These approaches suffer from three shortcomings: First, the accuracy of an extracted aPM is uncertain (**P3**) since not all impacting changes at Dev-time or Ops-time are observed and addressed (**P4**). The second shortcoming is that extracting and calibrating aPMs frequently would cause high monitoring overhead (**P6**). The third shortcoming is that the possible manual modifications of the extracted aPMs would be discarded by the next extraction (**P7**). The second category includes approaches that maintain the consistency incrementally (C2) either at Dev-time based on consistency rules [47, 18, 81, 79, 38, 13] or at Ops-time [28, 70, 77] based on dynamic analysis. Like C1, none of the C2 approaches succeed in updating aPMs according to both evolution and adaption nor provide a statement on the accuracy. Thus, the accuracy of the resulting architecture model and its AbPP is still uncertain and cannot be trusted (**P3**). This applies as well to approaches that estimate parametric dependencies to increase the accuracy of AbPP, e.g., [45, 44, 23].

1.2. Approach and contribution

In this article, we propose the Continuous Integration of Performance Models (CIPM) approach [51, 52]. This approach maintains the consistency between the aPM and software artifacts (source code and measurements). As shown in Figure 1, CIPM automatically updates aPMs according to observed Dev-time and Ops-time changes to enable AbPP (**P4**, **P1**, **P2** and **P7**). CIPM calibrates also aPMs with parametric dependencies (**P5**) and reduces the required overhead for updating and calibrating aPMs through adaptive instrumentation and monitoring (**P6**) [52]. Moreover, CIPM provides a statement on the accuracy of the AbPP and automatically recalibrates inaccurate parts [58] (**P3**). In this way, the resulting aPMs allows an accurate AbPP (**P1**) and more comprehension of the system architecture. This supports adopting proactive actions for upcoming performance issues and answering what-if questions about design alternatives. The contributions of our approach can be summarized as follows:

- *Automated consistency maintenance at Dev-time:* The proposed commit-based strategy automatically (section 5) updates the models (e.g., aPMs) that are affected by the Continuous Integration (CI) of the source code (**P4.1**). Contrary to other approaches, it uses regular Git commits as input and does not require a specialized editor, which eases its integration with CI.
- *Automated adaptive instrumentation:* We propose model-based instrumentation of changed parts in source code (**P6**), cf. section 6. Comparing to existing approaches, our instrumentation detects automatically where and how to instrument the source code.
- *Incremental calibration:* We propose a novel incremental calibration of the PMPs based on adaptive monitoring [52]. Our calibration uses statistical analysis to learn parametric dependencies. It also optimizes them based on a genetic algorithm if necessary [82]. In comparison to existing approaches, CIPM can be performed at Ops-time and addresses **P7**, **P6** and **P5**.
- *Automated consistency maintenance at Ops-time:* The Ops-time calibration observes Ops-time changes based on dynamic analysis and updates the aPMs accordingly [58] (cf. section 9). In comparison to existing approaches, CIPM automatically updates the aPMs including PMPs, system composition and resource environment using adaptive monitoring (**P4.2**).
- *Self-validation of updated aPMs:* The self-validation (cf. section 8) estimates the accuracy of AbPP compared with real measurements (**P3**). It manages the adaptive monitoring by activating and deactivating monitoring probes based on the validation results. This reduces the required overhead to a minimum (**P6**). According to our knowledge, our approach is the first approach that enables self-validation of aPMs and dynamic management of monitoring overhead.
- *Model-based DevOps pipeline:* The proposed pipeline [52] integrates and automates the CIPM activities to enable continuous AbPP during DevOps. To implement it, we designed and implemented a transformation pipeline [58] based on [28] using the tee and join pipeline architecture [14]. In contrast to existing pipelines, our pipeline maintains the consistency during the whole DevOps life cycle and enables AbPP.

This paper is an extension of our previous work [52, 58, 28, 82]. New contributions are the automated consistency maintenance at Dev-time with its novel commit-based strategy (section 5) as well as an evaluation of the adaptive instrumentation (subsection 10.4) and the scalability of the approach (subsection 10.6)[1]

[1] To make the review easier, here we provide a detailed list of which sections we have reused from our previous work and which sections are new or extended: The abstract and introduction (Sec. 1) have been newly written with a detailed discussion of the problems tackled by our approach. The foundations (Sec. 2) are reused and only the description of genetic algorithms has been added. The running example (Sec. 3) is reused and has been adapted to this paper. The overview of our approach (Sec. 4) is reformulated from previous publications and has been extended to include the new contributions. Section 5 on the CI-based update of software models is new contribution in this paper except for Section 5.4. However, Section 5.4 is newly written. Most of Section 6 on adaptive instrumentation is new written. Although the idea of the adaptive instrumentation is introduced

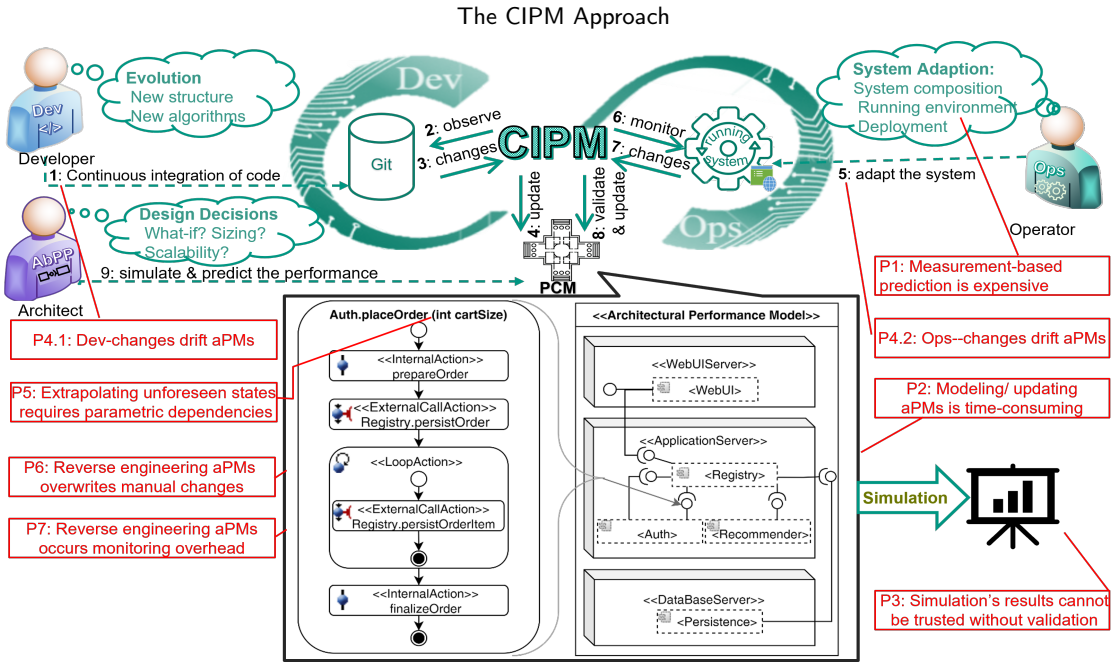


Figure 1: Overview on CIPM approach and the main actors. Red boxes shows the problems that CIPM addresses. The lower part of the figure abstracts some aspects of aPM (architecture and deployment on the right side and the behavior on the left side) using the TeaStore example [83] (c.f. section 3) and the annotations of Palladio Component Model (PCM) [63] (c.f. subsection 2.1)

2. Foundations

In the following, we introduce the foundations on the approaches that we build upon throughout this article.

2.1. Palladio Component Model

The Palladio Component Model (PCM) is a model-based approach for the analysis of software architectures [63]. The PCM focuses on the evaluation of performance aspects, such as the detection of bottlenecks and scalability problems. The PCM supports predicting the impact of design decisions to avoid high costs resulting from wrong decisions. 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 abstract behavior of services provided by these components using 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., central processing unit (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 (the way they interact with the system).

The Palladio SEFF [7] describes the behavior of a component service on an abstract level using different control flow elements (see the left lower part of Figure 1): *internal actions* (a combination of internal computations that do not include calls to required services), *external call actions* (calls to required services), *loops* and *branch actions*. SEFF loops and branch actions include at least one external call. Other loops and branches in the service implementation are combined into internal actions to increase the level of abstraction.

To predict the performance measures (response times, CPU utilization and throughput) the architects have to enrich the SEFFs with PMPs. Examples of PMPs are resource demands (processing amount that internal action requests from a certain active resource, such as a CPU or hard disk), the probability of selecting a branch, the number of loop's iterations and the arguments of external calls.

Palladio uses the stochastic expression (StoEx) [42] to define PMPs as expressions that contain random variables or empirical distributions. StoEx expresses calculations and comparisons using parameter characterization (e.g., value, number of lists elements or size of file).

2.2. Co-evolution approach with VITRUVIUS

The co-evolution approach [47, 46] keeps the architecture model and the source code consistent during the software system evolution. It defines change-driven Consistency Preservation Rules (CPRs) to propagate changes in source code to the architecture model and vice versa using model-based transformations. These rules are defined based on VITRUVIUS [36], a view-based framework that encapsulates the heterogeneous models of a system and the semantic relationships between them in a Virtual Single Underlying Model (VSUM) in order to keep them consistent. VITRUVIUS defines *mappings* and *reactions* languages. The first one declares the bidirectional specifications of consistency rules on the metamodel-level,

previously in [52], in particular we have re-implemented the approach for the now used EMF-based Java model and now also evaluated it. Section 7 on the incremental calibration of the models is partially reused and extended, but sections 7.2.4 and 7.3 are new. Sections 8 on self-validation and section 9 on Ops-time Calibration are reformulated from previous works. In the evaluation section (Sec. 10), evaluation questions EQ 1.1, 1.2, 1.2.1, 1.5 and 2.1 are new and we newly conducted experiment 1 for them. Additionally, we newly conducted experiment 5 to validate EQ 3 in detail. Section 10.4., 10.5, and 10.6 describe these new results. We rewrote Section 11 on related work and provided a table for a systematic comparison. To reflect all changes, we extended the conclusions (Sec. 12). We will delete this footnote in a potential later camera-ready version.

which are transformed into imperative unidirectional specifications using the *reactions* language. The *reactions* language describes then the CPRs that describe the consistency repair logic for each kind of changes, i.e., which and how the artifacts of a metamodel must be changed to restore the consistency after a change in a related metamodel has occurred. Thus, VITRUVIUS stores the mapping between corresponding model elements, in a correspondence model to reuse them by the consistency preservation process.

Using VITRUVIUS, the co-evolution approach keeps Java source code (using an intermediate model [25] for Java 6 [33]) and PCM consistent. The CPRs update the `Repository Model` of a PCM model and its behavior (SEFFs *without* PMPs) as a reaction to changes in the source code. Similarly, changes in PCM model are propagated to the Java source code.

2.3. iObserve

iObserve considers the adaptation and evolution of cloud-based systems as two interwoven processes [27]. 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 [78, 27]. 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 [28, 61].

2.4. Genetic Algorithm

The genetic algorithm [43] is a heuristic algorithm to resolve optimization problems. It is based on the evolutionary Principe. It starts from initial solutions as candidates for optimum solutions and evolves them. The algorithm starts with a set of solutions representing the first population, which can be modeled by experts. After that, the main evolutionary loop of the genetic algorithm generates new offspring based on two major operators: 'crossover' and 'mutation'. These operators generate new candidates for the new offspring population in two different ways. The crossover generates the candidate solutions by combining genetic materials from the parents, whereas mutation generates them by applying random changes. After crossover and mutation, the new candidates must be evaluated based on their ability to solve the optimization problem. This evaluation is done by 'phenotype mapping' of candidates (genotype) to the actual solution in order to avoid introducing a bias. This step can be escaped if the genotype represents the solution itself. After that, the new offspring population is evaluated by the 'fitness function'. The goal of the fitness function is to evaluate the quality of the solutions in order to select the best offspring to be parents in the new parental population. This achieves the progress towards finding the optimal solution. However, the main evolutionary loop should have a termination condition that determines when the search for optimal solution must be stopped. The condition can be predefined number of generations or acceptable quality degree of the found solutions. Usually the termination condition is related to the time and cost of the designed fitness function.

3. Running Example

The TeaStore is a web-based application, which implements a shop for tea. The application is based on a distributed microservice architecture and designed to be suitable for the evaluation of approaches for performance modeling [83].

The TeaStore consists of six microservices: *Registry*, *Image Provider*, *Auth*, *Persistence*, *Recommender* and *WebUI*. All microservices 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 [21].

TeaStore includes four different Recommender implementations that suggest related products to the users. The developers implemented these versions along different development iterations. These implementations have different performance characteristics. Performance tests or monitoring can be used to discover these characteristics for the current state, i.e., for the current implementation, current deployment, current environment, current system composition and the current workload. However, predicting the performance for another state (e.g., different deployment, workload, environment ...) is expensive and challenging because it requires setting up and performing several tests for each implementation alternative. In our example, answering the following questions is challenging based on application performance management [24]: "Which implementation would perform better if the load or the deployment is changed?" or "How well does the *Recommender* perform during yet unseen workload scenarios?" An example for the latter question would be an upcoming offer of discounts, where architects expect an increased number of customers and also a changed behavior of customers in that each customer is expected to order more items. Another question can be: "How does the current system composition look like? What the performance would be if the system composition is changed?". Changes in the system landscape of TeaStore at Ops-time are common due to the load balancing, which allows replications and de-replications without great effort. Therefore, it is inevitable to constantly update the associated architecture model to remain consistent with the system. An up-to-date architecture model can answer questions regarding performance, scalability and other quality aspects. Therefore, the goal of our approach is to provide an accurate architecture model at any point time and to keep the required manual effort and monitoring overhead as low as possible.

4. Overview

CIPM can be considered as an extension of the iterative software development process (e.g., agile or DevOps). Currently, the developers rely on an automated build, a test

automation, a Continuous Integration (CI) and a Continuous Delivery (CD). CIPM aims to increase this level of automation in the development process by providing fast feedback on the performance by enabling AbPP for design alternatives.

The next section (subsection 4.1) describes the models that CIPM updates in order to execute the AbPP using Palladio simulator, whereas subsection 4.2 explains how to integrate the CIPM processes into DevOps pipeline.

4.1. Models to Keep Consistent

Executing AbPP using the Palladio simulator requires having an up-to-date PCM containing the following sub-models: (A) `Repository`, (B) `System`, (C) `Resource Environment`, (D) `Allocation` and (E) `Usage Model` (cf. subsection 2.1).

To update the `Repository Model` (A), CIPM extends the Co-evolution approach [47, 49, 46] to incrementally update the software structure using commit-based consistency preservation rules, cf. section 5. For estimating the PMPs of the `Repository Model`, CIPM uses an adaptive instrumentation and monitoring to collect the required data while the application is running cf. section 6. It is called adaptive instrumentation, because only selected parts of source code are fine-granular instrumented. Besides, it is adaptive monitoring, because the fine-grained monitoring can be deactivated after the calibration to reduce the monitoring overhead. The `Repository Model` is calibrated at Dev-time using test-data (cf. section 7) and refined at Ops-time using the monitoring data from production environment (cf. section 9). The incremental calibration (cf. section 7) estimates the PMPs, such as the resource demand (cf. subsection 7.1) considering the parametric dependencies (cf. subsection 7.2). If necessary, adaptive optimization of PMPs can be activated to estimate the possible complex dependencies (cf. subsection 7.3). The processes mentioned in this paragraph keeps the `Repository Model` (A) of aPM up-to-date at both Dev-time and Ops-time.

Regarding `System Model` (B), we provide semi-automatic extraction at Dev-time based on static analysis of source code and automatic updates at Ops-time based on dynamic analysis of monitoring data. More detail is found in subsection 5.4 and subsection 9.3. CIPM updates the `Resource Environment Model` (C) based on the dynamic analysis too (cf. subsection 9.2). To update the `Allocation Model` (D) and `Usage Model` (E), CIPM integrates the dynamic analyses of iObserve approach [28] (cf. subsection 9.4 and subsection 9.5).

As a result, CIPM processes update the aPM parts (A-E) continuously to keep it consistent with the running system. To ensure the accuracy of the AbPP based on the updated aPM, CIPM provides a self-validation process (cf. section 8). It compares the simulation results with the monitoring data to recognize the deviations and to eliminate them.

The following section (subsection 4.2) describes how we integrate CIPM processes within the DevOps pipeline.

4.2. MbDevOps Pipeline

DevOps practices aim to close the gap between development and operations through integrating them into one reliable process [10]. We extend these practices to integrate and automate the CIPM approach in a Model-based DevOps (MbDevOps) pipeline. This enables AbPP during DevOps oriented development as the next paragraphs explain.

The MbDevOps pipeline (shown in Figure 2) starts on the "development" side with the Continuous Integration (CI) process [55] that merges the source code changes of the developers. CI triggers the first process, *CI-based update of software models (1)* (cf. section 5). This process updates the source code model in the VSUM of VITRUVIUS (1.1) (cf. subsection 5.1). Then, the predefined CPRs in VITRUVIUS respond to the changes in source code with updating the `Repository Model` (1.2) (cf. subsection 5.2). Similarly, CPRs update the Instrumentation Model (IM) (1.3) with new probes corresponding to the recently updated parts of `Repository Model` to calibrate them later (subsection 5.3). Besides, the first process extracts the `System Model` semi-automatically (1.4) (cf. subsection 5.4). The second process, the *adaptive instrumentation (2)*, instruments the changed parts of source code using the instrumentation points from IM (section 6). The following process is the *performance testing (3)* using the instrumented source code. It generates the necessary measurements for calibration. The pipeline divides the measurements into 80% training and 20% validation set [92]. Using the training set, the *incremental calibration (4)* estimate the PMPs considering the parametric dependencies and enriches the `Repository Model` with them (cf. section 7). After the calibration, the pipeline starts the *self-validation (5)* with the test data, which uses the validation set to evaluate the calibration accuracy. If the model is deemed accurate, developers can use the resulting aPM to answer the performance questions using AbPP (6). If not, they can either change the test configuration to recalibrate aPM again or wait for the Ops-time calibration. Answering the what-if performance questions using AbPP instead of the test-based performance prediction reduces both the effort and cost to perform this prediction before the operation.

The "operation" side of Figure 2 starts on the *continuous deployment (7)* in the production environment. The *Monitoring (8)* in the production environment generates the required run-time measurements. The measurements in a customizable time interval are grouped and sent to the subsequent processes: *self-validation (9)* and *Ops-time calibration (10)*. The *self-validation (9)* is an essential process to improve the accuracy of the aPM. It compares between the monitoring data and monitored simulation results to validate the estimated aPM. The result of self-validation is used as an input to the Ops-time calibration and used to manage the monitoring overhead. If the aPM is not accurate enough, the Ops-time calibration process (section 7) recalibrates the inaccurate parts based on the feedback of the self-validation, e.g., updating PMPs of `Repository Model`, `Resource Environment Model`, `System Model`, `Allocation Model` or `Usage Model` (cf. section 9). Moreover, the self-validation deactivates the fine-grained monitoring of the accurate parts. To keep in mind, self-validation (9) and calibration at Ops-time (10) are triggered frequently according to customizable trigger time to react to the new monitoring data, improve the accuracy of aPM and to respond to the possible Ops-time changes.

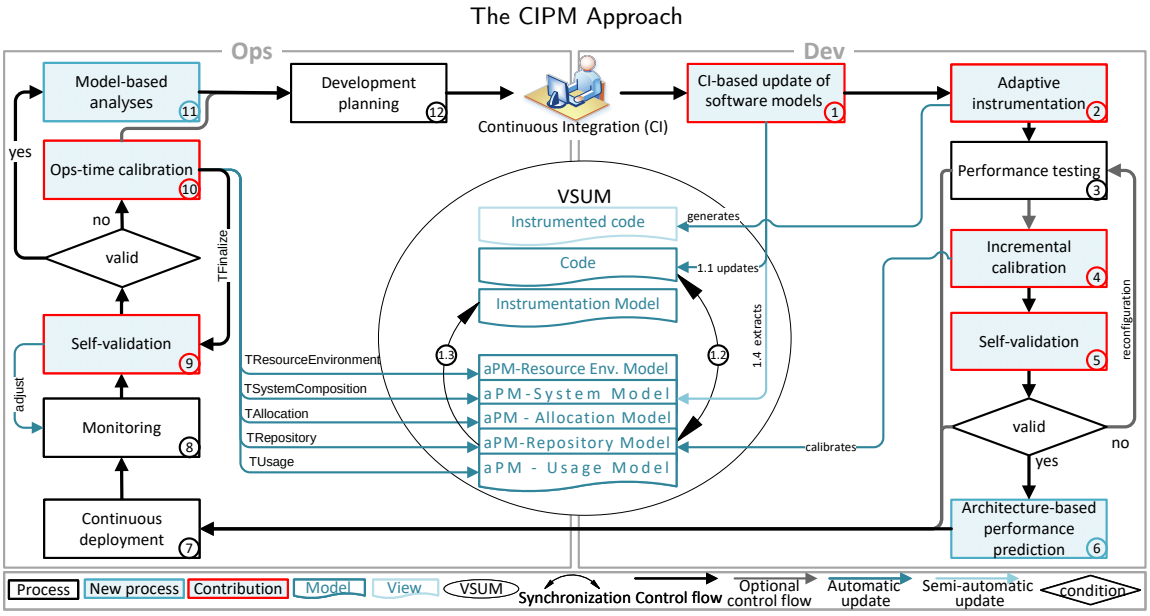


Figure 2: Model-based DevOps pipeline

Finally, the developers can perform more *model-based analyses (11)* on the resulting model, e.g., model-based auto scaling. Additionally, having an up-to-date descriptive aPM supports the development planning (12) . This is due to the advantages of models: increasing the understandability of the current version, modeling and evaluating design alternatives and answering what-if questions.

The following Sections (5-9) describe the new processes that are marked as contributions in the Figure 2.

5. CI-based Update of Software Models

This section describes updating the models that are affected by the changes in source code. The process is based on (A) the static analyses of the source code and (B) the predefined Consistency Preservation Rules (CPRs) within VITRUVIUS platform. This process updates four models:

- The `source code model` within the VSUM of VITRUVIUS based on (A) as subsection 5.1 explains.
- The structure of `Repository Model` based on CPRs of VITRUVIUS as subsection 5.2 describes.
- The IM based on CPRs of VITRUVIUS as explained in subsection 5.3.
- The `System Model` based on (A) as explained in more detail in subsection 5.4.

The following subsections describe how updating the source code model in the VSUM (subsection 5.1) triggers the CPRs of VITRUVIUS. CPRs update in their turn both of `Repository Model` (subsection 5.2) and IM (subsection 5.3). Additionally, subsection 5.4 explains the extraction of `System Model` at Dev-time after the commit.

5.1. CI-based Update of the Source Code Model

The goal of this update process is to extract all source code changes from a commit and to apply them on the source code model in the VSUM of the VITRUVIUS platform. As a result, it is not required to develop the source code within the VITRUVIUS platform, i.e., developers can use their own external tools for developing the source code and managing the different versions. CIPM updates the aPM solely based on the commits of the continuous integration of source code.

To achieve the update goal, all files in the repository with the changes of the recent commit are parsed (cf. Figure 3). The result of the parsing must be an EMF model to be comparable with the corresponding model in the VSUM. This allows the extraction of EMF changes which can be applied on the source code model in the VSUM.

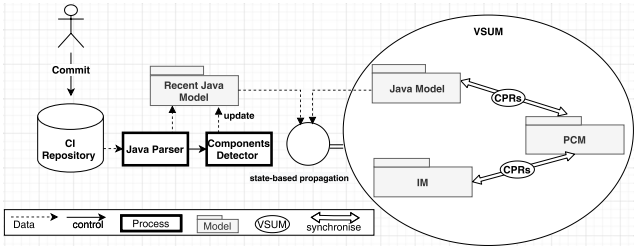


Figure 3: Commit-based Update of the aPM

Our implementation considers the Java programming language versioned in Git repositories. For the Java models, we build upon the existing Java Model Parser and Printer (JaMoPP) [25]. We extended the metamodel of JaMoPP to enable the support of the Java versions 7-15. Our extensions expand the metamodel with new features, for instance, the diamond operator, lambda expressions, or modules. Moreover, we implemented our own printer and parser. The parser implementation is based on the Eclipse Java Development Tools (JDT) [80] from which Abstract Syntax Trees (ASTs) and binding information are retrieved. The ASTs are converted to EMF model instances while the bindings support the resolution of references between model instances introduced by, e.g., imports. In contrast, the printer outputs model instances into valid Java code. More detail on our extension is documented in a technical report [4] [2].

[2]The source code is available on `https://github.com/PalladioSimulator/Palladio-Supporting-EclipseJavaDevelopmentTools`

As a result, we use the extended JaMoPP to parse the files after a commit. To include the source code's dependencies in order to retrieve complete code models, a build of the source code can be performed before parsing it to obtain and include libraries the source code depends on. Additionally, we implemented a trivial recovery which creates model elements for missing dependencies and which can also be used when no build is performed to gain complete models. After parsing the source code, EMF Compare [88] extracts the EMF changes by comparing the parsed model with the source code model in the VSUM. The extracted changes are sorted and applied on the Java model within the VSUM. This triggers the CPRs that update the related models as described in the following subsections.

5.2. CI-based Update of the Repository Model

The goal of this process is to update the `Repository Model` according to the changes that are transformed into the source code model in the last step (see subsection 5.1). It is mainly based on the CPRs defined by the Co-evolution approach [46] that we adapted for our applications.The adjustments in the CPRs are mainly related to technology-specific component and interface detection.

In the case of microservice-based applications, the component detection aims to find the microservices to generate a component for each microservice. Currently, our VSUM includes the source code model without any models of the configuration files, which rises the difficulty of model-based detection of the microservices within the VSUM. Therefore, we added a pre-processing step after the parsing of the last commit to provide the source code model with additional information on the components based on the file structures of code and the configuration files. For instance, in our implementation, a microservice is identified if a set of classes is complemented by a Docker and build file (e.g., a `pom.xml` file for Maven). If there is only a build file without a Docker file, the set of classes is considered as a component candidate, and the developer is asked to decide whether the candidate is a component or not. While the CPRs are focused on microservices, we allow regular components. When the developer decides that a component candidate is a component, they also determine what the component represents: a microservice or a regular component. After all components have been identified, a Java module model is created for each component. As a result, our CPRs map the Java module to a corresponding PCM component and consequentially generate a component for each created module.

Moreover, we defined the CPRs that detect the interfaces of microservices. Typically, microservices define a Representational State Transfer (REST) API as their interface. Thus, we implemented CPRs for the Jakarta RESTful Web Services (JAX-RS, formerly Java API for RESTful Web Services) and Jakarta Servlet (formerly Java Servlet) specification [31], because these technologies are used in our case study. In the context of JAX-RS, classes that are annotated with `@Path` or `@ApplicationPath` constitute a REST API [15], so that an interface is created for such annotated classes. `HttpServlets` as a specialization of `Servlet` provides another possibility to implement a REST API by overriding HTTP-specific methods in subclasses of `HttpServlet` [32]. Consequently, classes that inherit from `HttpServlet` are also modeled as interfaces. The interfaces of regular components are identified by public classes and interfaces in a particular regular component.

During the execution of the CPRs, a mapping between the source code and the aPM is established. This is necessary for the consistency preservation process in VITRUVIUS. An important extension of CPRs is the mapping between SEFF actions and their correlated source code statements. If a SEFF action, e.g., an internal action, is updated, then, the mapping between this internal action and the related statements will be stored in the correspondence model of VITRUVIUS.

In addition to the consistency preservation process, we use the mapping between source code and aPM for two processes in the context of CIPM: (A) the adaptive instrumentation (cf. subsection 5.3 and section 6) and (B) the Dev-time `System Model` extraction, which is explained in subsection 5.4.

5.3. CI-based Update of the Instrumentation Model

For this step, we defined CPRs that update the IM according to the changes in a commit (cf. Figure 3). The defined CPRs generate probes referring to SEFF actions whose source code statements have changed in the most recent commit.

The adaptive instrumentation uses collected probes in the IM and the mappings in the correspondence model to generate the instrumented source code as section 6 explains.

5.4. Dev-time Extraction of System Model

This section presents a semi-automatic process for extracting the `System Model`. This process can reduce the required time and effort of creating the `System Model` manually. This helps in applying AbPP at the Dev-time and supporting the design decision at this stage (e.g., proactive model-based assessment of the deployment plan). The process is composed of two main steps: extracting the so-called Service-Call-Graph (SCG) that represents the calls between the services and extracting the system composition based on SCG.

In the following, we introduce the structure of the SCG. After that, subsubsection 5.4.2 describes the SCG extraction at Dev-time. The extraction of `System Model` from the SCG, follows in subsubsection 5.4.3

5.4.1. Service-Call-Graph (SCG)

Conceptually, a SCG describes "calls-to" relationships between services. We also consider the resource container (computer) on which the respective services are executed. Graphically, this can be displayed as a directed graph where the pair of a service and resource container is a node. The edge indicates that a service on a particular container calls a service on a certain container. Figure 4 shows a truncated version of the SCG from the TeaStore case study, which was introduced in Sec. 3 and visualized in Figure 1.

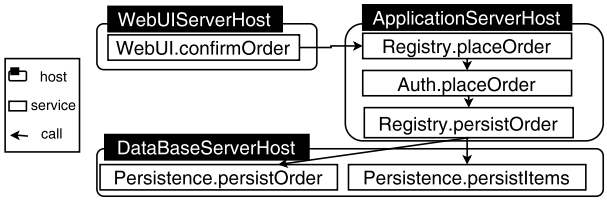


Figure 4: SCG extracted from an excerpt of TeaStore example

5.4.2. SCG Extraction at Dev-time

The extraction of SCG begins with a bytecode-level analysis that indicates the invocation dependencies between Java methods [75]. Based on the mapping between the source code and the `Repository Model` that has been stored in VITRUVIUS (cf. subsection 5.2), the method call graph of the services and the related components are transformed into an SCG. In some cases, e.g., inheritance or conditions, it is uncertain which call paths will be chosen at Ops-time. Therefore, the extraction of SCG at Dev-time considers all possible execution semantics. Moreover, at Dev-time it is unknown where the components of the services will be hosted. Therefore, we leave this information leer, what causes conflicts by extracting `System Model`, cf. subsubsection 5.4.3. These conflicts are resolved by the user.

5.4.3. System Model Extraction from SCG

The extraction of `System Model` starts from modeling the boundary interfaces that the system provides (provided role), which the user determines (architect or developer). In our example, the provided role is `CartActions` interface that provides services for purchasing products and managing orders.

For each provided role, the `Repository Model` is searched for the components that provided it. If more than one component provide the same interface, the user will be asked to select the correct one. Then assembly contexts for the selected components are created and linked to the provided roles using delegations. In our running example, just the `WebUI` component provides the `CartActions` interface. Therefore, an instance of the `WebUI` component is created and added to `System Model`, cf. the lower part side of Figure 1.

To complete the system model, the required roles of the added assembly contexts must be satisfied. For that, the SCG is traversed to detect all services called by the services of provided roles (as shown in Figure 4, the service `confirmOrder` calls `PlaceOrder`). Like the previous step, the components that provide the called services are detected based on `Repository Model`. If more than one component provides the same required role, the user should resolve the so-called connection conflict and select the right component (This is not the case for `Registry` component that provides `PlaceOrder`). In the next step, an assembly context for each selected component is created and added if no one is available in `System Model`. Otherwise, the user must resolve the so-called "assembly context conflict" by deciding whether to use the available assembly context or to add a new one (in our example, an instance of `Registry` component is added and no conflict at this stage occurs). Afterwards, the required roles are connected with the related provided roles. Recursively, each required role of the recently added assembly contexts is satisfied by adding the required component instance or by connecting to a previously added one, until all required roles are satisfied. In our example, the required roles of `Registry` are satisfied by adding instances of components that provide them (e.g., `Auth` and `Persistence`) . Subsequently, the required role of `Auth` is satisfied, since "Auth.placeOrder" callls "Registry.persistOrder" as shown in Figure 4. In this case, an "assembly context conflict" occurs because an instance of the `Registry` component is already available. As shown in Figure 1, the user can resolve this conflict by using the available instance of `Registry` instead of creating a new one.

6. Adaptive Instrumentation

The goal is to instrument the parts of source code, which have been changed and their changes may affect the validity of related PMPs. In our running example, if `Auth.PlaceOrder` service shown in (cf. section 3 is newly added then the adaptive instrumentation injects monitoring points (probes) to provide measurements for calibrating its PMPs: the resource demands of `prepareOrder` and `finalizeOrder`, the loop execution number of loopAction and the parameters of `Rigistry.persistOrder` and `Rigistry.persistOrderItem` external calls. If the source code of the remaining services have not been changed, the old estimations of their PMPs remain valid and they are not instrumented fine-granular.

To automate the adaptive instrumentation, we, first, extend the CPRs between the source code model and PCM [46], which respond to changes in the method body (e.g., adding/ updating statements) by reconstructing the SEFF using a reverse engineering tool [44]. Our extension extracts the mapping between code statements and their related SEFF actions and saves it in VITRUVIUS correspondence model.

Second, we define the CPRs that react to changes in SEFF by providing IM with the corresponding probes (e.g., service probe, internal action probe, loop probe or branch probe). Consequently, IM includes probes related to SEFF elements that code statements have been changed. However, both architect and the self-validation process can also add additional deactivated probes to IM, which can be activated by the self-validation, if their related PMPs are not accurate enough.

Third, we define the measurements metamodel (cf. Figure 5), which consists of monitoring record types corresponding to the probe types that are mentioned in the second step:

- `ServiceContextRecord` monitors the input parameters properties (e.g., type, value, number of list elements, etc.) to be considered as candidates for parametric dependency investigation (`parameters`), the caller of this service execution (`callerExecutionID`) to build SCG and calibrate the SEFF external call (`externalCallID`), the allocation context that captures where the component offering this service is deployed (`hostID`, `hostName`) and the service execution time to be used as a reference by the self-validation (`entryTime, exitTime`).
- `InternalActionRecord` monitors the execution time of internal actions (`entryTime, exitTime`) to estimate their resource demands.

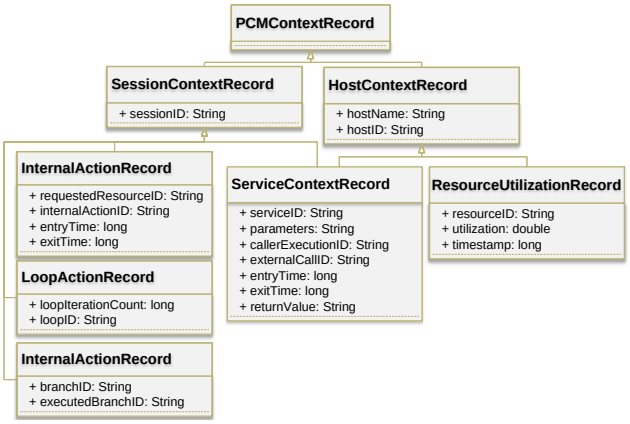


Figure 5: The Measurements Metamodel

- `ResourceUtilizationRecord` monitors the `utilization` of a resource (`resourceID`) in a time stamp (`timestamp`).
- `LoopActionRecord` monitors the number of loop iterations (`loopIterationCount`).
- `BranchActionRecord` monitors the selected branch (`executedBranchID`).

For implementing our specific monitoring records, we used the instrumentation record language [34] of Kieker [78].

Finally, we implemented a model-based instrumentation to generate the instrumented source code as a VITRUVIUS view. This view combines the information from two models: the source code model and the IM. The instrumentation starts with generating the instrumentation code for each probe in the IM according to the probe type. Then it injects the instrumentation code into a copy of the source code model. To detect the correct places for the instrumentation codes, the instrumentation process uses the mapping stored in VITRUVIUS correspondence model (cf. subsection 5.2), i.e., the relation between probes and SEFF elements as well as the relation between SEFF elements and their source code statements. After the injection of the instrumentation code finishes, the instrumented source code model is ready to be printed and deployed in test/ production environment. This provides the measurements for the calibration described in section 7.

7. Incremental Calibration of PMPs

The goal of incremental calibration of PMPs is to enable AbPP. Applying this process at the Dev-time before the deployment in the production environment enables AbPP for design alternatives instead of the expensive test-based prediction. However, the calibration of `Repository Model` at Dev-time is similar to the calibration at the Ops-time. The only difference is that at Dev-time we use measurements from the test environment instead of the production environment. Therefore, the Dev-time calibration step can be skipped. In this case, the PMPs will be calibrated for the first time at the Ops-time using measurements from production environment– section 9 explains the Ops-time calibration in more detail.

We mean with the incremental calibration that we just calibrate the new or updated SEFF actions. Moreover, the incremental calibration recalibrates also the inaccurate PMPs that are detected with the self-validation.

As explained in subsection 2.1, the PMPs are the parameters of SEFF actions. They can be (1) the resource demand of an `internal action`, (2) the iterations' number of a `SEFF loop`, (3) the branch transitions of `SEFF branch` and finally (4) the arguments and return value of an `SEFF external call`.

Contrary to the PMPs (2-4), we cannot measure the resource demand directly by monitoring. Therefore, we envision an incremental estimation of resource demands based on the adaptive monitoring, cf., subsection 7.1. Then, we identify the parametric dependencies to estimate all PMPs (1-4) in relation to the impacting factors, cf. subsection 7.2. Finally, we describe the adaptive optimization of the found parametric dependencies using the genetic algorithm, cf. subsection 7.3.

7.1. Incremental Estimation of Resource Demands

The incremental calibration of the internal actions with Resource Demand (RD) is challenging because we aim to estimate the RD of internal actions incrementally without high monitoring overhead. The existing Resource Demand Estimation (RDE) approaches either estimate the RDs at the service level [68] or require expensive fine-grained monitoring [9, 45]. Therefore, we propose in the following paragraph a light-weight RDE process that is based on adaptive instrumentation and monitoring to allow for an incremental RDE.

Our incremental RDE extends the approach of Brosig et al. [9] to estimate the RDs in the case of adaptive monitoring, i.e., monitoring the changed parts of source code.

Basis: Non-incremental resource demand estimation: Brosig et al. approximate the RDs with measured response times in the case of low resource utilization, typically 20%. Otherwise, they estimate the RD of internal action i (a part of SEFF, see subsection 2.1) for resource r ($D_{i,r}$) based on service demand law [54] shown in equation (1). Here, $U_{i,r}$ the average utilization of resource r due to executing internal action i and C_i is the total number of times that internal action i is executed during the observation period of fixed length T:

$$D_{i,r} = \frac{U_{i,r}}{C_i/T} = \frac{U_{i,r} \cdot T}{C_i} \tag{1}$$

Brosig et al. measure the C_i and estimate $U_{i,r}$ by using the weighted response time ratios of the total resource utilization, which is not applicable in our adaptive case where not all internal actions are monitored. Therefore, we extend their approach to estimate $U_{i,r}$ and as a result $D_{i,r}$ based on the available measurements and the old RDs estimations.

Incremental estimation of resource demands: Our approach distinguishes internal actions into two categories based on whether they have been modified in the source code commit preceding the incremental calibration. We denote internal actions whose corresponding code regions have been modified in the preceding source code commit as Monitored Internal Actions (MIAs), – these code regions are instrumented (cf. section 6) and monitored to produce measurements for RDE. We denote internal actions whose corresponding code regions have not been changed in the

preceding source code commit as Not Monitored Internal Actions (NMIAs), – monitoring data for these code regions has already been observed in a previous iteration and, consequently, we have already an estimation of their RDs.

Based on the fact that the total utilization U_r is measurable and the utilization due to executing NMIAs can be estimated based on the old estimations of RDs, we can estimate $U_{r,MIAs}$ and estimate the RD of each internal action $i \in MIAs$ accordingly as it will be explained in the following paragraphs.

To estimate ($U_{r,NMIAs}$), we estimate which internal actions $nmi \in NMIAs$ are processed in this interval and how many times nmi are called (C_{nmi}). For that, we analyze the service call records (see section 6) to determine which services are called in an observation period T and which parameters are passed. Then we traverse the service's control flows (i.e. their SEFFs) to get NMIAs and predict their RD using the input parameters. This requires evaluating branches and loops of the control flow to decide which branch transition we have to follow and how many times we have to handle the inner control flow of loops. Our calibration, adjusts the new or outdated branches and loops using the monitoring data (as will be described in Sections 7.2.2 and 7.2.3) before starting this incremental RDE. Thus, we make sure that we can traverse the SEFFs control flow. Consequently, we can sum up the predicted RDs for all calls of the NMIAs and divide the result by T to estimate the $U_{r,NMIAs}$ based on the utilization law as shown in the equation 2:

$$U_{r,NMIAs} = \frac{\sum_{nmi \in NMIAs} \sum_{k \leq C_{nmi}} D_{nmi_k,r}}{T} \tag{2}$$

Accordingly, we estimate the utilization due to executing the MIAs ($U_{r,MIAs}$) using the measured U_r and the estimated $U_{r,NMIAs}$ as shown in equation (3):

$$U_{r,MIAs} = U_r - U_{r,NMIAs} \tag{3}$$

Hence, we can estimate the utilization $U_{i,r}$ due to executing each internal action $i \in MIAs$ using the weighted response time ratios as shown in equation (4), where R_i and C_i are the average response time of i and its throughput. R_j is the average response time of the internal action $j \in MIAs$ and C_j is its throughput in T.

$$U_{i,r} = U_{r,MIAs} \cdot \frac{R_i \cdot C_i}{\sum_{j \in MIAs} R_j \cdot C_j} \tag{4}$$

Using $U_{i,r}$ we can estimate the resource demand for i ($D_{i,r}$) based on the service demand law (equation (1)).

In the case that the host has multiple processors, our approach uses the average of the utilizations as U_r.

Note that we assume that each internal action is dominated by a single resource. If this is not the case, we follow the solution of Brosig et al. [9] to measure processing times of individual execution fragments, so that the measured times of these fragments are dominated by a single resource. To differ between the CPU demands and disk demands, we suggest detecting the disk-based services in the first activity of CIPM using specific notation or based on the used libraries.

7.2. Identification of Parametric Dependencies

This process estimates the PMPs in relation to the impacting input data and their properties, e.g., the number of elements in a list or the size of a file. We begin by estimating the dependencies of loops, branches, arguments of external calls because the incremental RDE requires traversing the SEFF control flow to estimate the utilization of NMIAs (subsections: 7.2.2, 7.2.3 and 7.2.4). Then, we learn relations between PMPs and input data using decision trees for branches and regression analysis for remaining PMPs. Based on the cross-validation results of PMPs, an adaptive optimization by the genetic algorithm can be started (cf. subsection 7.3).

7.2.1. Resource demands

To learn the parametric dependency between the resource demand of an internal action i and input parameters P, we first estimate the resource demand on resource r for each combination of the input parameters ($D_{i,r}(P)$) using the proposed incremental RDE as described in section 7.1.

Second, we adjust the estimated RDs of an internal action using the processing rate of the resource, where it is executed, to extract the resource demand independently of the resource' processing rate $D_i(p)$.

Third, if the input parameters include enumerations, we perform additional analysis to test the relation between RDs and enumeration values using the decision tree. If a relation is found, we build a data set for each enumeration value. Otherwise, we create one data set for each internal action that includes the estimated RDs and their related numeric parameters. The goal is to find potential significant relations by the regression analysis of equation (5):

$$\begin{aligned} D_i(P) = (&a * p_0 + b * p_1 + \cdots + z * p_n + \\ &a_1 * p_0^2 + b_1 * p_1^2 + \cdots + z_1 * p_n^2 + \\ &a_2 * p_0^3 + b_2 * p_1^3 + \cdots + z_2 * p_n^3 + \\ &a_3 * \sqrt{P_0} + b_3 * \sqrt{p_1} + \cdots + z_3 * \sqrt{p_n} + \mathbf{C}) \end{aligned} \tag{5}$$

p_0, p_1.. p_n are the numeric input parameters and the numeric attributes of objects that are input parameters.

a... z, a_1... z_1, a_2... z_2 and a_3... z_3 are the weights of the input parameters and their transformations using quadratic, cubic and square root functions. C is a constant value.

Fourth, we perform the regression analysis to find the weights of the significant relations and the constant C.

Fifth, we replace the constant value C with a stochastic expression that describes the empirical distribution of C value instead of the mean value delivered by the regression analysis. This step is particularly important when no relations to the input parameters are found. In that case, the distribution function will represent the RD of internal action better than a constant value. To achieve that, we iterate on the resulting equation that includes the significant parameters and their weights to recalculate the value of C for each RD value and their relevant parameters. Then, we build a distribution that represents all measured values and their frequency.

Finally, we build the stochastic expression of RD that may include the input parameters and the distribution of C.

7.2.2. Loop iterations count

To estimate how the number of loop iterations depends on input parameters, we need both the loop iterations' count and the input parameters for each service call. To achieve that, we use our loop records that log the loop iterations' count, every time a loop finishes (see section 6). These records refer to the service call record that contains the input parameters.

The reason of using additional records instead of counting the total amount of enclosing service calls, is that the loop may have a nested branch or loop, which does not allow one to infer the correct count of loop iterations.

To estimate the dependency of the loop iteration count on input parameters, we combine the monitored loop iterations with the integer input parameter into one data set. To do so, we filter out all non-integer parameters and take into account their integer properties like the number of list elements or size of files. Then, we add transformations (quadratic and cubic) of parameters to the data set to test more relations. Finally, we use regression analysis to estimate the weights of the influencing parameters. Due to the restriction that the loop iterations count is an integer number, we have to ensure that the output value is always an integer value, which is not always the case. Therefore, we have to approximate the non-integer weights or express them as a distribution of integer values. For instance, we can express the value 1.6 using a Palladio distribution function of an integer variable which takes the value 1 in 40% of all cases and the value 2 in 60% of cases. Similar to the fifth step of parameterized RDE in subsubsection 7.2.1, we replace the constant value of the resulting stochastic expression with an integer distribution function. This will be especially useful when the cross-validation (e.g., based on correlation coefficient) finds no relation to the input parameters.

7.2.3. Branch transitions

To estimate the parameterized branch transitions, we use the predefined branch monitoring records that log which branch transitions are chosen in addition to a reference to the enclosing service call.

We monitor each branch instead of predicting the selected transition according to the external call execution enclosed in the branch due to potential nested control flows (e.g., nested branches), where we cannot infer the selected transition.

The used monitoring records allow us to build a data set for each branch, which includes the branch transitions and the input parameters. To estimate the potential relations, we use the J48 decision tree of the Weka library, an implementation of the C4.5 decision tree [60]. We filter out the non-significant parameters based on cross-validation. Finally, we transform the resulting tree into a boolean stochastic expressions for each branch transition. If no relation is found, the resulting stochastic expression will be a boolean distribution representing the probability of selecting a branch transition.

7.2.4. External call arguments

This step predicts the parameters of an external call in relation to the input parameters of the calling service.

For each parameter of an external call, we check whether it is constant, identical to one of input parameters, or depend on some of them. Moreover, we check the dependency to the data flow, i.e., the return value of the previous internal/ external actions.

To identify the dependencies, we apply linear regression in the case of numeric parameters and build a decision tree in the case of boolean/ enumeration one. For the remaining types of parameters, we build a discrete distribution.

7.3. Adaptive Optimization of Parametric Dependencies

As the previous subsection described, we estimate the PMPs with the parametric dependencies as a StoEx.

The optimization process is triggered just for the PMPs that have high cross-validation errors. For the cross-validation we used the Mean Squared Error (MSE) as a metric. The MSE threshold was set empirically. The goal of the optimization process is to improve the accuracy of the inaccurate PMPs and to reduce the complexity of the resulting StoEx.

The input of the optimization is the estimated StoEx and the output is an StoEx with cross-validation error less or equal to the error before the optimization.

For the optimization, we used the Genetic Programming (GP). We define the genes as a mathematical function represented as an Abstract Syntax Tree (AST) similar to [44].

Then we set up GP with a data set of the possible dependencies. We limited ourselves to the numeric candidates, i.e., the numeric characterization of the input data. Then we transform the initial StoEx that should be optimized to an AST and pass it as a starting individual. Then, the main evolutionary loop of the GP generates a new population based on 'crossover' and 'mutation'. We defined the 'crossover' that randomly swaps two subtrees belonging to two trees. The 'mutation' alters some genes to ensure the diversity in the new population, e.g., altering subtree or some values.

In the next step, the fitness function evaluates the individuals based on two criteria: the prediction accuracy using MSE and the complexity of the mathematical expression using the depth of the AST. These two metrics must be reduced by the GP. Based on the defined fitness function, the selector chooses the individuals for the next generation.

The evolution is terminated, if an optimal solution is found, i.e., the best fitness is lower than a given minimum threshold, the maximum execution time of GP is exceeded or fixed number of generations have evolved.

Finally, the resulting AST is converted to stochastic expression and transferred to `Repository Model`. More detail on the adaptive optimization is found in [82].

After the incremental calibration finishes, the self-validation process with the test data is triggered to inform the user about the accuracy of the calibrated aPM (cf. section 8).

8. Self-Validation

The goal of Self-Validation is to continuously evaluate the accuracy of the performance predictions related to the updated aPM. 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. At Dev-time we use the monitoring data from test environment and at Ops-time we use monitoring data from the production environment.

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. 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. 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. We use the following metrics to compare the two distributions: Wasserstein distance [65], Kolmogorov–Smirnov test (KS test) [19] and the difference of conventional statistical measures (e.g., average and quartiles). More details on the metrics are in subsubsection 10.2.2. These metrics give the user feedback on the accuracy of aPM. If the model is considered accurate, developers can trust it and use it to answer What-if performance questions. Otherwise, the self-validation determines the inaccurate parts to be recalibrated. The re-calibration can access the calculated metrics and use them to reduce the deviation and improve the accuracy of the resulting aPM. In addition, the metrics are used to adjust the granularity of the monitoring.

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. This feature is important to reduce the monitoring overhead especially if the self-validation is executed at Ops-time.

If the self-validation is executed in test environment and the re-calibration fails by increasing the accuracy, then the tester may change the test configuration to get more representative measurements for the re-calibration aPM. Another option is to wait for the Ops-time calibration. At Ops-time, measurements based on the real usage of system can be continuously monitored until the accuracy degree is accepted and the monitoring is deactivated. Besides, the self-validation can activate the fine-granular monitoring for some parts if their accuracy degree is not enough. In the case that the inaccurate parts are not instrumented, new probes for them are added to IM to recalibrate these parts after the next deployment.

9. Ops-time Calibration

The goal of Ops-time Calibration is to update the aPM according to monitoring data of the production environment. For that, we use a transformations pipeline similar to iObserve [27]. To realize the transformation pipeline we used a tee and join pipeline architecture [14], based on parts of iObserve.

The transformation pipeline (shown in Figure 2) consists of the following transformations: $T_{Preprocess}$ for pre-processing monitoring data, $T_{ResourceEnvironment}$ for updating `Resource Environment Model`, $T_{SystemComposition}$ for updating`System Model`, $T_{Allocation}$ for updating `Allocation Model`, $T_{Repository}$ for updating `Repository Model`, T_{Usage} for updating `Usage Model`, and $T_{Finalize}$ for validating aPM and managing the monitoring data. The following subsections explain the processes within these transformations in more detail.

9.1. Pre-processing of Ops-time Calibration

The $T_{Preprocess}$ (the first step in the transformation pipeline) filters the monitoring data and converts them 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. 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.

9.2. Ops-time Updating of Resource Environment

After the monitoring data has been pre-processed, the current Ops-time environment is analyzed by $T_{ResourceEnvironment}$. 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 CPR based on the VITRUVIUS platform [36] to keep our REM consistent with the resource environment in the corresponding aPM (PCM). The advantages and the idea behind the REM are twofold: REM 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.

9.3. Ops-time Updating of System Model

Similar to Dev-time, extracting a `System Model` at Ops-time requires a SCG that is used to update the `System Model`. The next two paragraphs explain how we extract SCG from monitoring data and how we use it to update the `System Model`.

SCG Extraction at Ops-time To build trees of service calls, T_{Trace} analyses the monitoring data that refer directly to the related services in `Repository Model`. The mapping to `Repository Model` is woven into the source code by the instrumentation (cf. section 6). As a result, The service call traces

can easily reflect which services call each other, which is the information required for the construction of the SCG.

In this context, it is crucial that we know explicitly which methods have been called, so the information quality is much higher compared to SCG extracted at Dev-time from the static code analysis (cf. subsubsection 5.4.2). Moreover, we can attach the information about the host to the SCG nodes, because this information is included in the monitoring data.

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. For our approach, we have implemented this exemplary for HTTP requests. A header is added to each request, which indicates by which service call trace it was triggered. The monitoring data is finally bundled and sent to a backend, which can reassemble the traces based on this information [56].

Updating the System Model The $T_{SystemComposition}$ updates the `System Model` based on the extracted SCG and the methodology that subsubsection 5.4.3 introduced. The only difference is that no conflict happens, because the SCG at Ops-time is more accurate and includes the host information.

9.4. Ops-time Updating of Allocation Model

After the extraction of the SCG (cf. subsection 9.3), the system deployment is investigated in parallel to updating the `System Model`. The $T_{Allocation}$ recognizes deployments and undeployment events because the SCG also contains information about the hosts (cf. Figure 5 and section 6). As a result, $T_{Allocation}$ can add/ delete allocation context to/ from `Allocation Model`. This supports $T_{SystemComposition}$ with the required information about the current allocation of the system components. More detail is found in [56].

9.5. Ops-time Updating of Repository Model

The $T_{Repository}$ calibrates the inaccurate PMPs that are revealed by the self-validation using the monitoring data. The calibration process is similar to the process at Dev-time (cf. section 7) and includes the validation results to optimize the PMPs. The simple optimization is based on the applied regression analysis. It adjusts the regressions' parameters based on the resulting validation. i.e., the error between the measured response time and monitored one. According to the resulting validation, an optimization of PMPs based on the genetic algorithm can be triggered, cf. subsection 7.3. In parallel, T_{Usage} analyses the user behavior and updates the `Usage Model` based on iObserve [28].

9.6. Finalization of Ops-time Calibration

The final step of Ops-time Calibration, $T_{Finalize}$, executes the self-validation (cf. section 8). Based on the self-validation results and configurable criteria, the granularity of the monitoring is adjusted, i.e., fine-granular monitoring can be activated/ deactivated. In our example, the measured response times of `confirmOrder` service 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 this service is deactivated. Finally, the validation results are entered as input into the next execution of the Ops-time calibration.

10. Evaluation

In our evaluation, we follow the Goal-Question-Metrics (GQM) [73]. Accordingly, we define, first, the goals of the evaluation in subsection 10.1. Second, we drive the Evaluation Questions EQs that can check whether the described goals are reached or not. Third, we define metrics that can answer the evaluation question in subsection 10.2. Finally, we performed goal-oriented experiments to calculate the metrics and answer the EQs, cf. subsection 10.3.

The results of our evaluation are described in three subsections that are related to the evaluation's goals: the accuracy of AbPP using the updated aPMs in subsection 10.4, the required monitoring overhead in subsection 10.5 and the scalability of the approach in subsection 10.6. Finally, we discuss the threats of validity in subsection 10.7.

10.1. Evaluation Goals and Questions

The main goal of the evaluation is to evaluate the applicability of the approach in the mean of accuracy (Goal 1 (G1)), monitoring overhead (G2) and scalability (G3): CIPM should provide accurate models (G1.1) and allows accurate AbPP (G1.2) without high monitoring overhead (G2). Moreover, CIPM have to avoid performance issues at Ops-time by quickly identifying and resolving inconsistencies (G3).

In the following, we drive the EQs that check the aforementioned goals:

- **G1.1: Accuracy of the incrementally updated models**:
  - **EQ-1.1:** How accurately does CIPM reflect changes from a Git commit to a source code model?
  - **EQ-1.2:** How accurately do CIPM update the `Repository Model` and the IM after a new commit?
  - **EQ-1.2.1:** Is there a difference between the propagation of multiple commits and the propagation of these commits as a single commit?
  - **EQ-1.3:** How accurately is the extraction of the `System Model` at Dev-time?
  - **EQ-1.4:** How accurately does CIPM update the `Resource Environment Model`, the `Allocation Model` and the `System Model` at Ops-time when applying software adaption scenarios?
  - **EQ-1.5:** How accurately does the adaptive instrumentation instrument the source code based on the collected instrumentation points in IM?
- **G1.2: Accuracy of AbPP using the updated aPMs:**
  - **EQ-1.6:** How accurate is the AbPP using the incrementally updated aPM?
  - **EQ-1.7:** How correctly can CIPM identify the parametric dependencies and to what extent can the estimation of them improve the accuracy of the AbPP?

  - **EQ-1.8:** What impact do Ops-time changes have on the accuracy of the AbPP?

- **G2: Monitoring Overhead**
  - **EQ-2.1:** How much can the adaptive instrumentation reduce the monitoring overhead?
  - **EQ-2.2:** To what extent does the adaptive monitoring at Ops-time help to reduce the monitoring overhead?

- **G3: Scalability of the transformation pipeline:**
  - **EQ-3:** How does the transformation pipeline of CIPM scale with an increasing amount of monitoring records?

10.2. Evaluation Metrics

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

10.2.1. Model Conformity

The Jaccard similarity coefficient (JC) [20] is used to quantify the equality of two sets (A and B) as follows:

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{6}$$

The range of resulting JC is from 0 to 1. The higher the value, the more similar the two sets. JC is 1 if the sets are identical.

Since the models are sets of elements, we can also apply this concept to models. For that we implement a matching algorithm that determines the identical elements (intersection) in two models A and B based on different factors: the types of elements, some of properties like the name of named elements, some references like the implementing interface of a SEFF and the model structure like the position of SEFF actions. The matched elements will be considered as the intersection of the models $A \cap B$.

We implement JC to evaluate the equality of `System Models`, `Allocation Models` and `Resource Environment Models` [56, p. 69]. In the case of `JaMoPP Models` and `Repository Models`, we implement the structural matching algorithm based on EMF Compare [88], see [3, p. 37]. Regarding JaMoPP, we used a custom language-specific matching algorithm [39]: We combine the default matching algorithm of EMF Compare with a hierarchical Java-specific matching algorithm that we extended from SPLevo [37] to be compatible with Java 7-15. Thus, Java-specific properties and structure is considered to provide accurate matching despite of the high required implementation effort.

10.2.2. Distribution Comparison

To compare distributions, we use three types of metrics: conventional statistical measures [74], non parametric tests (Kolmogorov–Smirnov test) [66] and distance functions (Wasserstein) [53]. 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) [19]. It calculates the maximum distance between the Cumulative Distribution Functions (CDFs). The minimum is 0 (if both distributions are perfectly identical) and the closer 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. Additionally, KS test is sensitive to the shifts and shapes of distributions, which may produce undesirable false positive alerts (high values) [30]. For example, KS test may result in high value, if two distributions with the same mean have different shapes. Therefore, we use the KS test in combination with other metrics during our evaluation.

The Wasserstein metric is a distance measure for distributions [53]. In simple terms, it describes how much a distribution must be changed 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.

Classic statistics metrics (e.g., mean or quartiles) are calculated for both distributions.

Using these three commonly known metrics, it is possible to get an overview of the two distributions and their dissimilarities in a simple and quick way.

10.3. Experiment Setup

The evaluation of our approach is based on three case studies: TeaStore [83] that is introduced in section 3, Common Component Modeling Example (CoCoME) [26], and the real-world case study "TEAMMATES" [72].

CoCoME is a trading system for supermarkets [26]. It supports several processes such as scanning products at a cash desk or processing sales using a credit card. We used a cloud-based implementation of CoCoME, where the enterprise server and the database are running in the cloud.

TEAMMATES is a cloud-based tool to manage feedback students' [72]. It consists of a Web-based frontend and a Java-based backend, on which we concentrated in the evaluation.

In this article, we present five experiments that focus on evaluating the following points: the accuracy of aPMs after changes at Dev-time (E1, E2) and after changes at Ops-time (E4), the accuracy of AbPP that is achieved by self-validation (E3, E4) and after adaptions at Ops-time (E4), the required overhead (E4) and the scalability (E5). We exclude from this paper the experiments that evaluate the accuracy of AbPP (A) after incremented calibrations of aPM based on the adaptive instrumentation [52] and (B) after optimization of the PMPs [82]. The main reason of excluding these experiments is to avoid producing too long paper, since the incremental calibration based on adaptive monitoring (A) is also evaluated partly in our experiments (E3, E4). Regarding (B), the optimization using genetic algorithm will not be triggered in our case studies because they include no complex dependencies. Anyway, we sum up the results of the excluded experiments when we present the results of the following experiments [3].

[3] More information on source Code, replication packages and experiments is on `https://sdqweb.ipd.kit.edu/wiki/CIPM`

Experiment 1 (E1) The goal of E1 is to evaluate the commit-based update of the aPM and to answer **EQ-1.1**, **EQ-1.2** and **EQ-1.2.1**. To achieve this goal, we used two case studies: TeaStore and TEAMMATES.

E1 on TeaStore: we propagate the changes between version 1.2 and 1.3.1 of the TeaStore. The commits from version 1.2 to version 1.3.1 can be split into 3 intervals (I) [1.2, 1.2.1], (II) [1.2.1, 1.3], (III) and [1.3, 1.3.1]. The first interval consists of 20 commits of which 12 commits affect 5 Java files with overall 141 added lines and one removed line. Three Java files (123 lines in total) were added. In interval (II), seven of 11 commits affect four Java files with overall 121 added and 134 removed lines while nine Java files with overall 215 added and 227 removed lines are affected by 12 of 100 commits in interval (III).

The initial commit includes a lot of architectural-relevant changes. Propagating this commit is like an incremental reverse engineering of the version 1.2 of code (IRE). The successive commits in interval (I) include three architectural-relevant changes: (A) in the Auth service (A1) and WebUI service (A2), a new REST endpoint for obtaining the readiness has been added whereby both implementations are identical, (B) a method corresponding to a SEFF was extended by one statement, and (C), in a supporting service, a servlet has been added which provides functions to control and access log files. The remaining changes are not architectural-relevant. Besides, there are no changes in the dependencies. Both interval II and interval III contain no architectural-relevant changes and no changes in the dependencies. Table 1 displays the commits in which the architectural-relevant changes occurred. The commits are continuously numbered beginning with version 1.2 as commit 0.

Before the changes are propagated, we use the changes between an empty repository and version 1.2 as an initial commit to integrate it into VITRUVIUS. Then, we perform the commit-based update of the models by using the commits that transform the version 1.2 into the version 1.3.1. In addition, we execute the adaptive instrumentation after the update process.

In the next step, we evaluate whether the models of the VSUM are correctly updated based on the changes in a commit. This applies to the (1) source code model, (2) the `Repository Model`, and (3) the IM. Afterwards, we evaluate whether the instrumented source code is correctly generated (4). Finally, we evaluate the reduction of monitoring overhead resulting by the adaptive instrumentation (5).

Regarding (1), an updated source code model in the VSUM shall be in the same state as if the complete code model of a commit would have been integrated into the VSUM. Therefore, the source code of the last commit is parsed to be used as a reference for evaluating the updated on in the VSUM. We compare the updated source code model with the generated reference by calculating the JC metric.

For the evaluation of the automatically updated `Repository Model` (2), we use a manually updated `Repository Model` as a reference to compare with. For example, in the manual update for (A1) and (A2), a new interface with one method is added for each new REST endpoint. Furthermore, the WebUI and Auth components provide their interface and contain a new SEFF for the method. The added statement by change (B) is included in an internal action and requires no adjustment of the corresponding SEFF. As a consequence of (C), the servlet is added as a new interface provided by the supporting service. For more evaluation of CPRs updating `Repository Model`, we propagate also the version 1.3.1 to compare the updated model with a manual available one [57].

Table 1
Overview of architectural-relevant changes over commits

Commit	0	10	11	13	17	18
Architectural-relevant change	IRE	(B)	Reverts (B)	(B), (C)	(A1)	(A2)

The expected changes in the SEFFs cause the generation of new probes in the IM (3). Thus, for every SEFF, we check that the IM contains a matching coarse-grained probe. The IM also must include fine-granular probes for all actions of the SEFFs that have recently been changed or created.

Regarding (4), we first check that no compilation errors occur because of the instrumentation. Then, we check whether the instrumentation statements that are related to the probes in the IM are correctly injected in source code.

Regarding (5), we investigate how much the adaptive instrumentation can reduce the monitoring overhead (**EQ-2.1**) by calculating the ratio of adaptively instrumented probes to all probes that required to calibrate the whole aPM. In addition, the ratio of the fine-grained adaptively instrumented probes to all possible fine-grained probes is determined.

To answer **EQ-1.2.1**, for each interval, we propagate all commits individually and then the commits between two versions as a single commits. For example, the 20 commits between version 1.2 and 1.2.1 is propagated as a single commit. Then, we compare the resulting `Repository Model` to the result of the propagation of multiple commits and a manually updated `Repository Model` by calculating the JC. The resulting source code model and IM are also checked as in (1) and (3).

E1 on TEAMMATES Similar to TeaStore, we repeated E1 with the real git history of TEAMMATES. The evaluation covers 17.859 commits that impacts 1.428 files. The considered commits are propagated in five steps, i.e., the commits are integrated and propagated as five commits to show the results in a simple way, since **EQ-1.2.1**. evaluates the accuracy of integrating multiple commits. Therefore, we propagate the commit 64842 (TM-0) as the initial commit, and 48b67 (TM-1), 83f51 (TM-2), f33d0 (TM-3), and ce446 (TM-4) as the following commits. While TM-0 spans 17832 commits and adds 114468 code lines in 709 Java files, TM-1 spans 3 commits with 154 added and 129 removed lines in 122 Java files. Between TM-0 and TM-1, the maintainer role was introduced. From TM-1 to TM-2, public fields were made private including the addition of corresponding get and / or set methods and an adaptation of the direct field accesses to the new methods. TM-2 spans 2 commits with 3249 added and 2978 removed lines in 227 Java files. With TM-3, 2 commits affected 65 Java files adding 502 lines and removing 340 lines. Static variables were made non-static while some

classes were converted to singletons. In the last commit TM-4, JavaDoc was updated and more classes were converted to singletons. It spans 20 commits adding 3457 and removing 1293 lines in 147 Java files [4]. Similar to TeaStore, we repeat the E1 on TEAMMATES to answer the following questions: **EQ-1.1**, **EQ-1.5**, **EQ-1.2** and **EQ-2.1**.

Experiment 2 (E2) In this experiment, we evaluate the extraction of a `System Model` at Dev-time that is explained in Sec. 5.4. The input is the source code of TeaStore and CoCoME. Then, the resulting models are compared to reference models that represent the actual system compositions. Based on the results of this experiment, **EQ-1.3** can be answered.

Experiment 3 (E3) The goal of this experiment is to evaluate the accuracy of Ops-time calibration (G1) and answer **EQ-1.6**. The Experiment starts with monitoring the application under examination and execution of a load test. Meanwhile, metrics about the monitoring are collected and the monitoring data is used as input for the transformation pipeline. The monitoring data is then compared with the simulation results of the derived models. Finally, metrics are calculated to quantify the deviation between the monitoring data and the simulation results. Here, monitoring data from a parallel and independent run were used for comparison, and the experiment ran 10 times for 180 minutes to eliminate possible outliers. This experiment was performed for CoCoME and the results allow initial answers to **EQ-1.6**.

Experiment 4 (E4) This experiment extends E3. It evaluates the accuracy of aPM (G1.1), the accuracy of AbPP (G1.2) and the required monitoring overhead (G2) after simulated adaptions. The foundation is a number of predefined change scenarios, such as replications, allocations, workload changes and system composition changes. The *Scenario Generator* selects several 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 JC. We focus on the `System Model`, the `Resource Environment Model` and the `Allocation Model` (**EQ-1.4**). The `Usage Model` is excluded from this paper, as it is extensively addressed in [28].

For evaluating the accuracy of AbPP, the monitoring data is compared to simulation results of the derived models to estimate how well the models are parameterized at Ops-time and how well they represent the performance characteristics of the actual system. Here, we used the metrics that were introduced in Sec. 10.2.2 to answer **EQ-1.6**, **EQ-1.7**and **EQ-1.8**. 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 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 eliminate the warm-up phase and only consider the time period between the 30 and the 150 minute, to avoid side effects that falsify the measurement, e.g., just in time compiling.

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 to call service of Recommender. 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 (**EQ-1.7**) and the system composition (**EQ-1.4**).

We also consider another dimension while performing E4 is the measurement of the emerging monitoring overhead and the amount of generated monitoring data in the worst case, where the whole source code is fine-granular instrumented. This information is used to evaluate whether the monitoring overhead can be reduced by the self-validation, which answers **EQ-2.2**.

Experiment 5 (E5) 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,

[4] see `https://sdqweb.ipd.kit.edu/wiki/CIPM_Evaluation_Details`

Table 2
Overview over the propagated commits and their evaluation results for the updated models

Commit	4	5	6	7	8	10	11	13	14	15	16	18
JC Java Model	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
JC `Repository Model`	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
SEFFs / SEFF Actions without probes	0	0	0	0	0	0	0	0	0	0	0	0

that it produces worst-case execution times. By means of this experiment, scalability questions can be answered (**EQ-3**).

10.4. Accuracy

In this section, we present the results of evaluating the accuracy of updated aPM in subsubsection 10.4.1 and the related AbPP in subsubsection 10.4.2.

10.4.1. Model Accuracy

In this section, we present the evaluation of models' accuracy after changes at Dev-time (E1,E2) and changes at Ops-time E4. Table 2 shows the evaluation results for the updated models in interval I of experiment E1. It reveals that the calculated JC for the Java models is one, i.e., the source code models in the VSUM are correctly updated answering **EQ-1.1**. The comparison between the manually and automatically updated `Repository Model` results in JC values of one. This means the `Repository Model` is also correctly updated. Considering the IM, the evaluation shows that the right probes are generated, i.e., no SEFF or SEFF action that shall be instrumented is without a probe in IM, see the last line of Table 2. As a consequence for answering **EQ-1.2**, these results indicate that the `Repository Model` and IM were correctly updated. The results of interval II, interval III, and TEAMMATES are identical. We obtained JC values of one for all Java models and repository models comparisons.

By comparing the integrated version 1.2 and 1.3.1 with the manually created `Repository Model`, we discovered that both models contain components for the microservices and the interactions between them. However, the resulting `Repository Models` include more technical details which are not present in the manually created one.

Additionally, we check the instrumented source code that is generated during experiment E1 to ensure that it includes all probes that were represented in IM. We find that the source code is correctly instrumented, which answers **EQ-1.5**.

The evaluation results for the propagation of the changes as one commit for answering **EQ-1.2.1** are visualized in table 3. It shows that all models in the VSUM are correctly updated. This indicates there is no difference for the resulting `Repository Model` if multiple commits are propagated or if the commits are propagated as one commit. As a result, developers can choose to propagate, for example, every or specific commits. Considering the IM, there is a difference because the IM after the single propagation contains all newly generated probes at once while the IM is continuously updated during the propagation of multiple commits.

Next, based on in experiment E2, the accuracy of the `System Model` extraction at Dev-time is investigated. The key findings of the experiment for the selected case studies are shown in Table 4. It can be seen that in both cases an identical model to the reference model is 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, **EQ-1.3** can be answered, as it became clear that the system compositions were reflected correctly in the extracted `System Models`.

Table 3
Evaluation results for the propagation of commits between two versions of the TeaStore as a single commit

Versions	1.2.1	1.3	1.3.1
Java Model	1.0	1.0	1.0
Updated `Repository Model` and references one	1.0	1.0	1.0
`Repository Model` after single and multiple propagations	1.0	1.0	1.0
SEFFs/ SEFF Actions without probes	0	0	0

Table 4
Results for deriving the System Model at Dev-time

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

Table 5
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

Finally, Table 5 shows the results of experiment E4 and lists the minimal JC for all considered change scenarios at Ops-time. The results show that three model types are correctly inferred in all cases. Consequently, it can be concluded for **EQ-1.4** that the change scenarios were recognized and correctly propagated to the models.

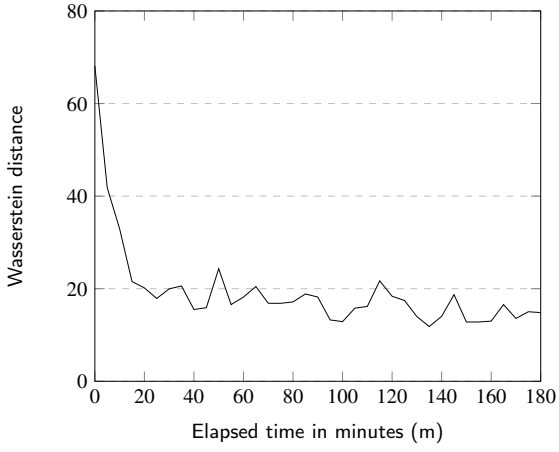
Summary: We conclude that CIPM can update the software models automatically and accurately. This applies to the following models: source code (`JaMoPP`), instrumented source code, `Repository Model`, `System Model`, `Allocation Model` and `Resource Environment Model`. Exceptional case was the update of `System Model` and `Repository Model` at the Dev-time, where the update process is not fully-automatically: the user can be asked to confirm the detected components of the `Repository Model` or to decide whether to create or reuse available component's instances for the `System Model`.

10.4.2. Prediction Accuracy

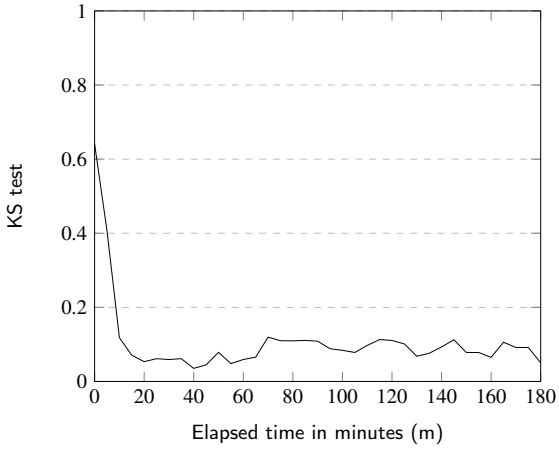
In this section, we present the results regarding the accuracy of the AbPP, broken down according to the two case studies, CoCoME and TeaStore.

CoCoME (E3): Figure 6 (a) shows the Wasserstein distance between the simulations of the derived models and the monitoring data for the response times of the "bookSale"

service, which is triggered when a purchase is initiated.



(a) Wasserstein distance over time



(b) KS test over time

Figure 6: Overview of the metrics over time for the CoCoME case study (comparing the distributions which result from the analysis and the monitoring)

These results provide initial answers to **EQ-1.6**. It can be seen that the Wasserstein distance decreases very rapidly at the beginning and then settles below a value of 20 with minor fluctuations. This means that the accuracy of the simulations increases over time and consequently the accuracy of the derived PCM models increases too. This observation conforms to the chart on the KS test (Figure 6 (b)). In both graphs the metrics decrease over time and then stabilize at a low level. The fluctuations in the graphs are caused by the fact that the simulations are stochastic processes. In other words, it is very unlikely that two simulations produce identical results, even if the used models are equal. Consequently, the accuracy of the models increases over time and then remains at a constant level. The improvement in accuracy over time can be explained by two factors: first, the pipeline receives more and more data over time and thus information about the performance characteristics of the application. Second, the repository transformation learns over time from the monitoring data (see Section 9).

TeaStore (E4): Figure 7 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 6 summarizes the accuracy metrics over time.

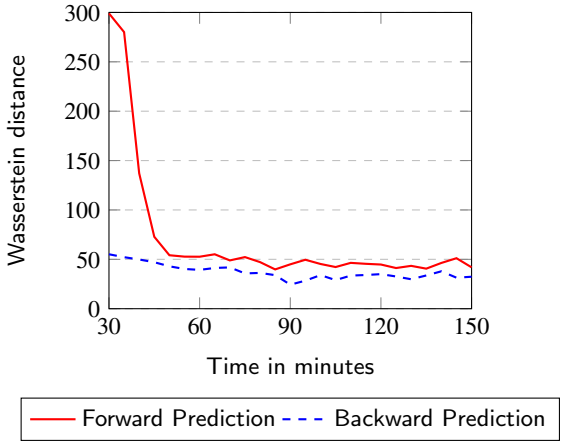


Figure 7: Median Wasserstein distance of the forward and backward prediction for a model derived at a given point in time - regarding the response times of TeaStores *confirmOrder* service

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.

The results also imply that the parametric dependency of the response time on the number of orders in the database is detected. The advantage of parameterized models has been studied in detail in our previous work [52]. The experiments compared the accuracy of AbPP using an aPM that CIPM parameterized with dependencies to AbPP using a non-parameterized one. The results show that AbPP using parameterized aPM is obviously more accurate in case of predicting the performance for unseen state [52] (**EQ-1.7**). Additionally, the PMPs in E4 are well calibrated without an optimization since TeaStore includes no non-linear parametric dependencies. Therefore, the optimization of the genetic algorithm is not triggered in this scenario. However, the incremental optimization of PMPs is evaluated in a previous work in more detail [82]. The results show that the optimization can improve the accuracy of AbPP for unseen state till five times if the PMPs have non-linear dependencies.

The observations can be confirmed using additional metrics (see Table 6). The interpretation of the mean distance depends on the average response time of the "confirmOrder" service, which amounted to approximately 1000 ms in the experiment. In addition, it is important to note that the average and standard deviation of the forward prediction metrics

Table 6
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

are strongly affected by the high values at the beginning of the experiment. With the help of these findings, **EQ-1.6** and **EQ-1.8** 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.

Summary: The Experiment E3 confirmed that the accuracy of the aPMs at Ops-time is increasing over the time by learning from more monitoring data. Then, the accuracy stabilizes at a good level: In case of CoCoME the maximum KS test was 0.075947 and the maximum Wasserstein distance amounted to 12.384184. Similar results are obtained by applying E3 to TeaStore [56, page 86]. Applying E4 on TeaStore confirmed also the accuracy of aPMs despite of the simulated Ops-time changes.

Additional experiments on TeaStore and CoCoME in [52] studied the incremental calibration based on adaptive instrumentation in more detail and confirmed also the accuracy of AbPP (**EQ-1.6**). The experiments result in KS test values that do not exceed in average 0.16 and Wasserstein distances that are in average lower than 39.6.

10.5. Monitoring Overhead

First, we calculated the reduced monitoring overhead that the adaptive instrumentation is able to achieve according to the last evaluation step of experiment E1 (5). According to E1's results shown in Table 7, the adaptive instrumentation can reduce the monitoring overhead for all probes between 46.0% and 68.1% (**EQ-2.1**). On the level of the fine-granular probes, the reduction of the monitoring overhead is between 72.8% and 99.5%. Even if the developer decides to instrument the whole source code, the monitoring overhead will be reduced by the adaptive instrumentation because it only activates the probes that changed after the last commit.

Besides, the overall monitoring overhead is analyzed and observed over time in the worst case, where the source code is fully instrumented. Every 5 minutes, the sum of the monitoring overhead from the last 5 minutes is 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. Figure 8 shows the results which are obtained by forming the median from multiple experiment executions.

The dashed line in the graph highlights the point in time

Table 7
Reduction of the monitoring overhead caused by the adaptive instrumentation according to experiment E1

Case study	Commit	The reduction ratio of probes	The reduction ratio of fine-granular probes
Teastore	10	67.8%	96.3%
	11	67.8%	96.3%
	13	67.8%	96.3%
	18	68.1%	97.6%
Teammate	1	55.4%	88.9%
	2	46.0%	72.8%
	3	62.8%	99.5%
	4	60.7%	96.4%

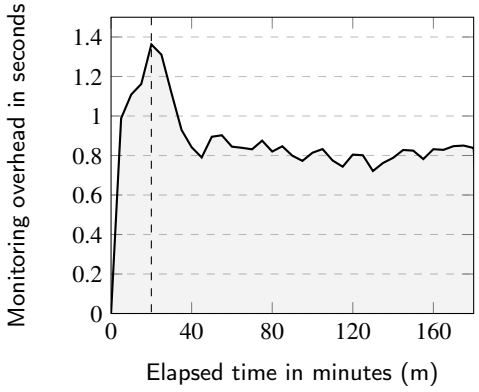


Figure 8: Median of the arising monitoring overhead over time when considering five minute intervals

when the first switch from fine-granular monitoring to coarse-granular monitoring happened. Based on this graph, an answer to **EQ-2.2** can be given. After the self-validation process begins to find individual services that are well calibrated, the granularity of monitoring is reduced since some fine-grained probes will be deactivated. This happens after about 20 minutes. At the peak, a monitoring overhead of approx. 1.362s (i.e. 0,454% of 5 minutes) 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. In a previous work [52], we obtained similar results in evaluating the monitoring overhead in case of adaptive-instrumentation.

10.6. Scalability

We address **EQ-3** by examining the scalability properties of all sub-transformations in the pipeline separately in the upcoming sections.

10.6.1. Repository Model Transformation

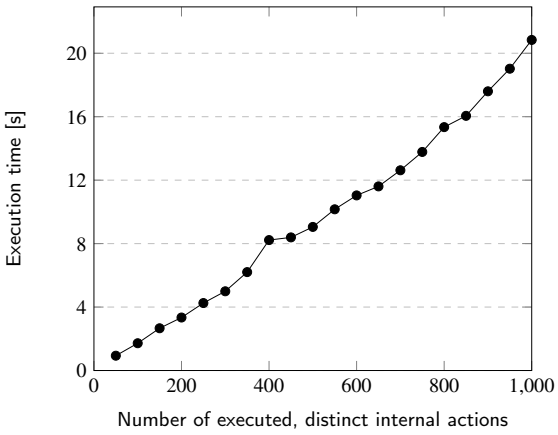
First of all, the scalability of the repository model transformation has been analyzed. The transformation can roughly be

divided into two parts. In the first part the validation results are analyzed and the results of this analysis are used as input for the second step which executes the optimizations. The analysis of the validation results is irrelevant regarding the execution times. The results are iterated only once and even if the simulations are configured with excessive simulation times and measuring points, the execution time is negligible within the overall context. Therefore, we will only consider the second part in the following scalability analysis.

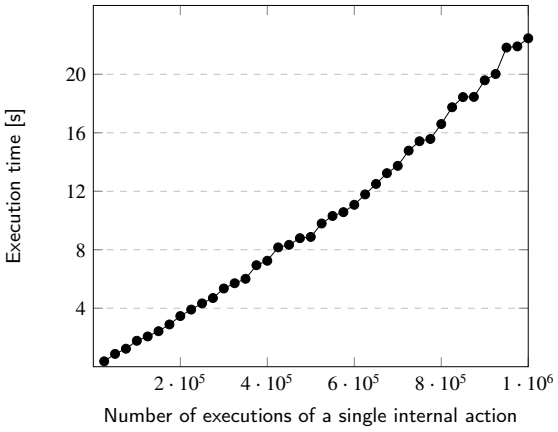
In our case study the optimization is performed based on the regression. However, if the genetic algorithm would be applied, it can be configured so that its execution time does not exceed a specific threshold. As a result, there is a tradeoff between the execution time and the accuracy of the resulting `Repository Model`, requiring a more detailed scalability analysis that also considers possible side effects. This point will be evaluated in a future work. For these reasons, we focused here on the scalability of the transformation when using the regression.

The regression is performed for each stochastic expression that needs to be calibrated. The number of data points within a regression is variable and depends on the monitoring data. This can be well illustrated using the example of internal actions whose resource demands need to be calibrated. There are two factors that influence the execution time of the transformation: the number of internal actions that are observed and the number of data points that are recorded for each internal action. The number of observed internal actions corresponds to the number of triggered regressions and the number of data points directly affects the duration of the regressions. Based on this, we built the scenarios that are considered in the scalability analysis. First, we examined the execution times of the transformation for an increasing number of internal actions. Subsequently, we observed the runtimes for an increasing number of data points for a single internal action. The results are summarized in Figure 9.

In both cases it is visible that the duration of the transformation scales linearly with increasing parameters. Even for a high number of internal actions (a), the growth of the execution time remains linear. Exactly the same can be observed when increasing the data points per internal action. In summary, it can be concluded that the transformation also scales linearly in worst-case scenarios and therefore no unexpected side-effects emerge.



(a) Scalability of the transformation with an increasing number of executed distinct internal actions



(b) Scalability of the transformation with an increasing number of executions of a single internal action

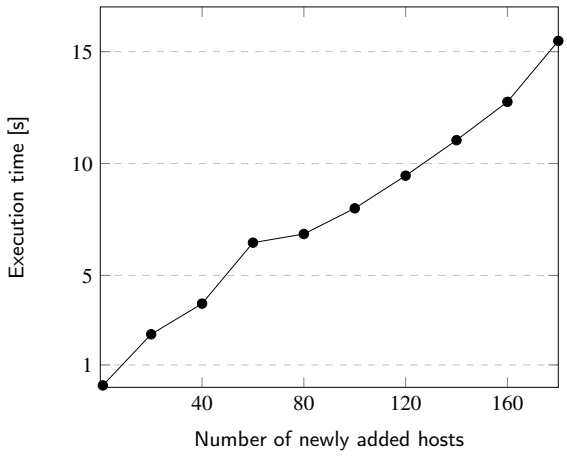
Figure 9: Exploration of the scalability of the repository transformation under various circumstances

10.6.2. Resource Environment Transformation

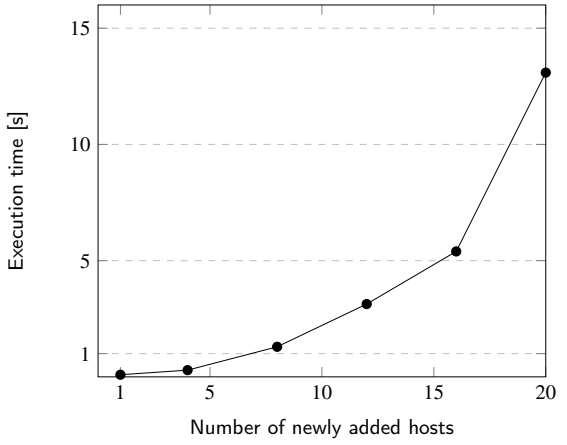
The resource environment transformation identifies changes to the hosts and the network connections within the Ops-time environment. The detected hosts and connections are inserted into the Runtime Environment Model (REM). Using the consistency rules based on VITRUVIUS, the corresponding resource containers and linking resources are created in the `Resource Environment Model`. The execution time of the transformation is dominated by the change propagation via VITRUVIUS. In the following, we will examine the execution times of the transformation with an increasing number of new hosts and connections. Therefore, we consider two scenarios:

1. Increasing number of new hosts; sparse meshed - indicating that each of the new hosts has only one network connection
2. Increasing number of new hosts; fully meshed - indicating that each of the new hosts has a connection all other hosts

The chart (a) in Figure 10 shows the scalability of sparse meshed Ops-time environments. The execution time scales almost perfectly linear with up to 180 new hosts. For realistic values of about 20 new hosts or less (within a single execution of the transformation), an execution time of 2 seconds is not exceeded in our test setup. In contrast, the execution time rises exponentially when adding fully meshed hosts as chart (b) in Figure 10 shows. This is simply because the connections between hosts need to be synchronized one by one. The number of connections increases exponentially with the number of hosts when considering a fully meshed network. However, it can still be concluded that the transformation provides adequate execution times for most use cases, as an



(a) Scalability of the transformation with an increasing number of newly added hosts; each new host has **only one** connection to another host



(b) Scalability of the transformation with an increasing number of newly added hosts; each new host has a connection **to all** other hosts (fully meshed)

Figure 10: Scalability analysis of the transformation of `Resource Environment Model` in different scenarios

appearance of more than 10 new fully meshed hosts between two executions of the pipeline will rarely occur in practice.

In summary, the scalability analysis for the `Resource Environment Model` transformation has shown that the execution times scale appropriate for realistic use cases.

10.6.3. System Model and Allocation Model Transformation

The transformations that are responsible for updating the `Allocation Model` and the `System Model` are reviewed together within the scalability analyses. For the derivation of updates in the `System Model`, the number of changes in the system composition is crucial. On the other hand, for the derivation of updates in the `Allocation Model`, the number of changes in the deployments is crucial. Consequently, these two parameters determine the design of the scalability analysis. First, the number of changes is determined, one half is populated with deployment changes and the other half with changes to the system composition. These change scenarios are generated with different sizes and used as input for the combination of both transformations ($T_{SystemComposition}$ and $T_{Allocation}$). Figure 11 shows the cumulated execution time of both transformations for an increasing number of changes. The chart

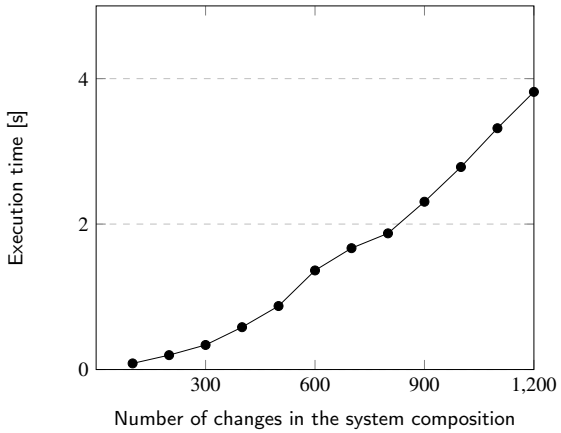


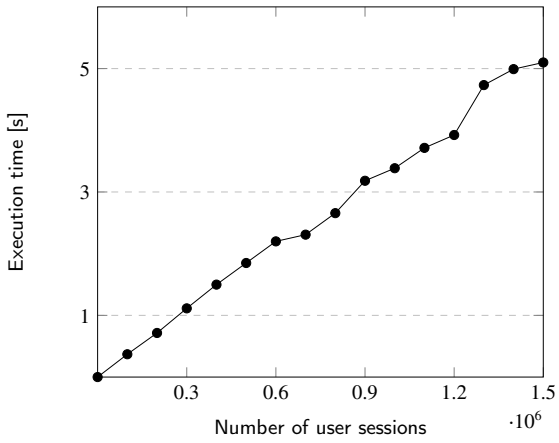
Figure 11: Execution times of System Model transformation with an increasing number of changes in the system composition

shows that the execution times scale approximately linearly, with a slightly lower slope at the beginning compared to a higher but stable slope from about 500 changes onwards. Even for a total number of 1200 changes the execution time is lower than 4 seconds. Because such a number of changes between two pipeline executions probably never occurs in practice, it can be concluded that both transformations scale well.

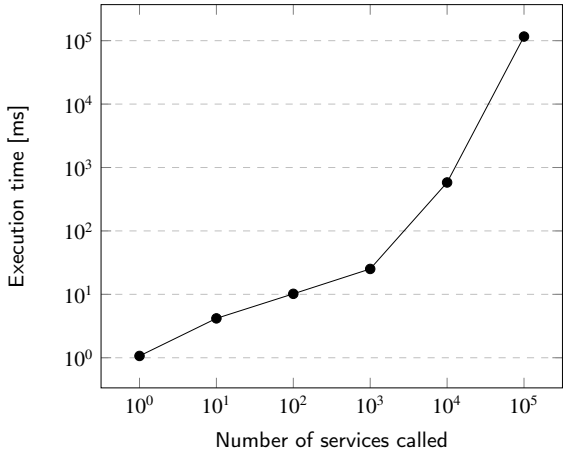
10.6.4. Usage Model Transformation

The `Usage Model` transformation is adopted from iObserve and ported to our monitoring data structure. Hence, both approaches are conceptually identical. A detailed scalability analysis has already been done for iObserve [28]. The goal of the scalability analysis in the context of CIPM is to show that the results are consistent with those of iObserve. We consider two different cases. First, an increasing number of users, all of them triggering exactly one service call and second, an increasing number of service calls triggered by a single user. Figure 12 shows the scalability analysis for both scenarios. Here, (a) shows the increase in execution times for a rising number of users and (b) shows the growth for an increasing number of service calls initiated by a single user. When looking at the sub figure (b) it should be noted that the axes are scaled logarithmically. In this way, we wanted to ensure that the results can be compared to those obtained from the iObserve scalability analysis.

The first experiment shows that the execution time scales almost perfectly linear with an increasing number of users. The same conclusion can be drawn from the results of iObserve's scalability analysis [28]. We also obtained consistent results when analyzing the execution time for an increasing number of service calls initiated by a single user. For 100 or less initiated service calls, the execution time increases sublinearly and thereafter superlinearly with an explosive growth in execution time above 10000 initiated service calls.



(a) Scalability of the transformation when the number of users increases; each user triggers a single service call



(b) Scalability of the transformation with an increase in the number of triggered service calls for a single user (axes are scaled logarithmically)

Figure 12: Scalability analysis of the `Usage Model` transformation

In the superlinear segment, the execution time is dominated by the loop detection [28]. The extreme increase of the execution time with a high number of triggered service calls is not critical, because such a user behavior is unlikely in practice.

In summary, it can be stated that the results of our scalability analysis are in line with those of iObserve. Therefore, it can also be concluded that the execution times of the `Usage Model` transformation are appropriate.

10.7. Threats of Validity

The identification of the threats to validity is based on guidelines for case study research [64]. 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 [29, 50, 82] 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 [28]. Another threat concerns the execution of experiment E2 (see Sec. 10.3). 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. A threat to validity by applying E1 on Teammates, is that we do not have a reference PCM to evaluate the first PCM, which we obtained by propagating the initial commit. However, we evaluated the resulting PCM by the comparison with the available well-documented architecture[5].

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, TeaStore and Teammates, which are widely used in research and address common business use cases [62, 83]. By combining the several 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 [78, 50].

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.

11. Related Work

There are a lot of approaches that aim to achieve the consistency between the software artifacts automatically. As we explained in the introduction, these approaches belong to two main categories: the first one (C1) generates the up-to-date artifacts as a batch process (marked as $\checkmark_B$ in Table 8), e.g., reverse engineering approaches. The second one checks the consistency and try to eliminate it (called as incremental approach and marked as $\checkmark_{inc}$ in Table 8). Both C1 and C2 can be also classified in three subcategories based on the phase, in which the consistency is maintained: at Dev-time, at Ops-time or at both. The Table 8 summarizes the related works and just assigns them to these subcategories, since the main category is labeled as $\checkmark_B$ for C1 and $\checkmark_{inc}$ for C2.

Consistency management at Dev-time As shown in Table 8, there are a lot of approaches that focus on consistency maintenance at Dev-time. For that, they either extract an architecture model or maintain an existing one (C2). The reverse engineering approaches at Dev-time (C1) is based mainly on static analysis of source code. For example, SoMoX [6] and Extract [48] extract parts of PCM. Similarly,

[5] see https://teammates.github.io/teammates/design.html

the ROMANTIC-RCA [2] extracts component-based architecture from an object-oriented system based on relational concept analysis. A shortcoming of C1 is that they ignore the possible manual optimization of the extracted model by the next extraction. The incremental consistency maintenance at Dev-time (C2) includes the approaches that minimize, prevent, or repair architecture erosions [17].A good summary of these approaches is presented in [17]. Moreover, Knodel and Popescu compare the approaches that minimize the architecture erosion by detecting architectural violations at Dev-time [38]. JITTAC tool [13], for instance, detects the inconsistencies between architecture models and source code, but does not eliminate them automatically. Archimetrix [81] detects also the most relevant deficiencies through continuous architecture reconstruction based on reverse engineering. Examples of C2 approaches that prevent the inconsistency between source code and architecture model at Dev-time are the co-evolution approaches, e.g., the mbeddr approach [79] and Langhammer's approach [46]. The mbeddr approach uses a single underlying model for implementing, testing and verifying system artifacts like component-based architecture. Similarly, the approach of Langhammer [46] uses virtual single underlying model to allow the co-evolution of PCM and source code. The Focus approach [18] avoids the inconsistencies by recovering the architecture and using it as a basis of evolution of object-oriented application. The main limitation of the consistency management at Dev-time is that the provided models are mostly considered as system documentation and should be enriched with PMPs if the AbPP is to be support. Therefore, some of these approaches are extended to allow AbPP. For example, Langhammer [46] calibrated the co-evaluated approach with an approximation of resource demand (response times) to show that it can be used for AbPP. Similarly, Krogmann extended SoMoX with a calibration of PMPs with the parametric dependencies [45] based on the dynamic analysis (Beagle approach [44]). However, the approach of Krogmann requires high monitoring overhead which restricting collecting the monitoring data from the production environment rather from test environment. Therefore, we assign Krogmann's approach to the Dev-time approaches rather the hybrid approaches. In general, the calibration of the whole project after each adjustment in the models causes monitoring overhead and ignores possible manual adjustments of PMPs, which our approach overcomes by the incremental calibration. Moreover, all the consistency management approaches at Dev-time ignore the effect of adaption at Ops-time on the accuracy of aPMs.

Consistency management at Ops-time The approaches that maintain the consistency at Ops-time are based mainly on the dynamic analysis of monitoring events. For example, the approach of Brosig et al. [8], PMX [85] and PMW [12] are reverse engineering approaches (c1) that extract parts of the aPM based on dynamic analysis to allow AbPP. Furthermore, the SLAstic approach [77, 76, 84] extracts aPM and can detect some changes at Ops-time such as migrations and reflect them in the model (C2). A previous work of us, iObserve [28], can also respond to changes in deployment and usage by updating the related parts in PCM (C2), what we also integrated in this work (CIPM). Other approaches that extract/ update performance models at Ops-time are summarized in [71]. The Drawbacks of the Ops-time approaches are that continuously high monitoring effort required to extract/ update models. Moreover, they cannot model system's parts that have not been called and consequentially not covered by the monitoring. Besides, the source code changes are ignored and the accuracy of the models is not validated by these approaches.

Hybrid approaches The scope of hybrid approaches spans Dev-time and Ops-time. For example, Langhammer introduces also a reverse engineering tool (EjbMox) [46, P. 140] that extracts the behavior of the underlying Enterprise Java Bean source code, by analyzing it at Dev-time and calibrating it based on the dynamic analysis at Ops-time. The Konersmann's approach integrates information about the architecture model into the code via annotations [40]. This enables the dynamic generation of an architecture model from the source code via transformations [40]. Moreover, the approach of Konersmann synchronizes allocation models with running software [41]. Spinner et al. [70] propose an agent-based approach to update architectural performance models (C2). As an input of the Spinner et al. approach, a static analysis of the source code is performed to detect the components and apply the instrumentation. However, the above-mentioned approaches show a much smaller scope of consistency preservation (e.g., limited recognition of evolution and adaption scenarios).

Instrumentation Similar to CIPM, Kiciman et al. propose a platform (AjaxScope) for the instrumentation of JavaScript code [35] to allow performance analysis and usability evaluation. Based on coarse-grained monitoring, AjaxScope identifies where the source code runs slowly and instruments it to find the cause of the slowness. AIM [87] provides adaptable instrumentation of the services of application under test to get more accurate measurements for estimating the resource demands of an architectural performance model. Contrary, our approach detects what parts should be instrumented fine-granular. The measurements are collected then from test or production environments.

Resource Demands The related approaches estimate the resource demands either based on coarse-grained monitoring data [68, 69] or fine-grained data [9, 89]. The latter approaches give a higher accuracy by estimating PMPs but have a downside effect because of the overhead of instrumentation and the monitoring. Our approach reduces the overhead by the automatic adaptive instrumentation and monitoring. Similar to CIPM, Grohmann et al. [22] update the resource demand continuously at Ops-time. For that, they tune, select, and execute an ensemble of resource demand estimation approaches to adapt to changes at Ops-time. The resulting estimation is a constant value. In contrast, CIPM considers the parametric dependencies and optimizes the

Table 8
An Overview of the Related work

scope	Consistency Management Approaches	aPM					PMPs	AbPP	Parametric dependencies	Self-Validation
		Repository	System	Allocation	Res. Env.	Usage				
Dev-time	Co-evolution [46, p. 35]	✓ inc					✓ B	✓		
	Mbeddr [79], Focus [18]	✓ inc								
	Consistency checker [38], [13], [81]	✓ inc								
	Extract [48]	✓ B				✓ B		✓		
	ROMANTIC-RCA [2]	✓ B								
	SoMoX+Beagle [45, 44]	✓ B					✓ B	✓	✓	
Ops-time	Brosig et al.[8]	✓ B	✓ B	✓ B		✓ B	✓ B	✓		
	PMX[85]	✓ B	✓ B	✓ B			✓ B	✓		
	Brunnert et al. [12], PMW [11]	✓ B	✓ B	✓ B	✓ B		✓ B	✓		
	SLAstic [77, 76], SLAstic.SIM [84]	✓ inc	✓ inc	✓ inc			✓	✓		
	iObserve [28]	✓ inc		✓ inc		✓ inc		✓		
Hybrid	EjbMoX [46, P. 140]	✓ B						✓		
	Konersmann´s approach [40, 41]	✓ B		✓						
	PRISMA [70]	✓ inc	✓ inc	✓ inc	✓ inc	✓ inc	✓ B	✓	✓	
	CIPM	✓ inc	✓ inc	✓ inc	✓ inc	✓ inc	✓ inc	✓	✓	✓

estimated stochastic expression at Ops-time.

Parametric dependencies In addition to the approach of Krogmann (SoMoX+Beagle) [44], the works of Ackermann et al. [1] and Curtois et al. [16] characterize the parametric dependencies. Additionally, Grohmann et al. [23] considers the characterization of parametric dependencies in performance models at Ops-time. Similarly, CIPM allows die characterization of parametric dependencies during an incremental update and calibration of the aPM at Ops-time.

12. Conclusion

Considering the software architecture model increases the understandability as well as the productivity of software development [59]. Moreover, Applying AbPP promises proactive detection of performance problems by simulation instead of the expensive measurement-based performance prediction.

In this article, we presented the continuous integration of the architectural performance model that keeps the aPM continuously up-to-date. Our approach maintains the consistency between software artifacts at Dev-time and Ops-time. It updates aPM after the evolution and adaptation of the system.

At Dev-time, the commit-based strategy extracts source code changes from commits and updates the structure of aPM including abstract behavior and the system composition.

To allow the simulation of aPM, our calibration estimates the parameterized PMPs incrementally and uses a novel incremental resource demand estimation based on adaptive monitoring. The calibration identifies the parametric dependencies and optimize it based on the genetic algorithm.

In addition to PMPs, the Ops-time calibration observes the adaptive changes and updates the affected parts of aPM accordingly. This applies to changes in deployment, resource environment, usage and even system composition. The proposed self-validation continuously analyses the accuracy of the AbPP. The results of self-validation is used to manage the monitoring and calibrate aPM at Ops-time.

For the evaluation we performed various experiments based on two case studies: CoCoME [26] and TeaStore [83]. We were able to update the structure of the aPM based on the commit. The accuracy of the updated 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.

In future works, we will expand the scope of the optimization of the accuracy using genetic algorithm to cover the whole abstract behavior of the services (SEFFs) instead of just the performance parameters. Moreover, we plan to evaluate the scalability of our approach in the case of the optimization using genetic algorithm, since there are trade between the impact of the algorithm's configuration on PMPs (their accuracy) and the overhead that this configuration requires. Besides, we are aware that our implementation of CPRs are currently just suitable for the domain of microservice applications that are based on Java language and specific technologies mentioned in subsection 5.2. However, our approach is based on the VITRUVIUS platform, that allows defining domain-specific metamodels and consistency preservation rules [36]. The required overhead to adjust metamodels and CPRs for other domains in future works should be acceptable comparing to the gained advantages by applying CIPM approach. We also plan to consider technology-based calls explicitly, for example calls to REST interfaces or calls to messaging queues [86]. Then, CIPM can consider such specific calls and calibrate them at Ops-time based on the dynamic analysis of our extendable incremental calibration. This will result in more accurate AbPP and avoid asking the developers during the CI-based update of aPM about the

real target of such technology-based calls. Currently, we are adapting the CIPM approach so it is applicable to Lua-based Industrial Internet of Things (IIoT) applications. Our adaption covers parsing and printing LUA source code and adjusting the CPRs to detect the IIoT components. The goal of this extension is to evaluate the applicability of CIPM with a Real-world IIoT applications from the industrial sector.

13. Acknowledgment

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.01) as well as by the DFG (German Research Foundation) – project number 432576552, HE8596/1-1 (FluidTrust).

References

[1] V. Ackermann, J. Grohmann, S. Eismann, and S. Kounev. Black-box learning of parametric dependencies for performance models. In *Proceedings of 13th Workshop on Models@run.time (MRT), co-located with MODELS 2018* (Oct. 14, 2018), CEUR Workshop Proceedings, Copenhagen, Denmark, Oct. 2018.

[2] Alae-Eddine El Hamdouni, Abdelhak-Djamel Seriai, and Marianne Huchard. Component-based architecture recovery from object oriented systems via relational concept analysis. In pages 259–270. University of Sevilla, 2010.

[3] M. Armbruster. *Commit-Based Continuous Integration of Performance Models*. Master Thesis, Karlsruher Institut für Technologie, Sept. 14, 2021.

[4] M. Armbruster. Parsing and Printing Java 7-15 by Extending an Existing Metamodel. Technical report, 2022. Also available as https://publikationen.bibliothek.kit.edu/1000149186.

[5] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.

[6] S. Becker, M. Hauck, M. Trifu, K. Krogmann, and J. Kofron. Reverse Engineering Component Models for Quality Predictions. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track*, pages 199–202. IEEE, 2010.

[7] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. 82:3–22, 2009.

[8] F. Brosig, N. Huber, and S. 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, pages 183–192, USA. IEEE Computer Society, 2011.

[9] F. Brosig, S. Kounev, and K. Krogmann. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *Proceedings of the 1st International Workshop on Run-time mOdels for Self-managing Systems and Applications (ROSSA 2009). In conjunction with the Fourth International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2009)*, 10:1–10:10, Pisa, Italy. ACM, New York, NY, USA, Oct. 2009.

[10] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. R. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert. Performance-oriented DevOps: A Research Agenda. Technical report SPEC-RG-2015-01, SPEC Research Group - DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), Aug. 2015.

[11] A. Brunnert, C. Vögele, A. Danciu, M. Pfaff, M. Mayer, and H. Krcmar. Performance management work. *Business & Information Systems Engineering*, 6(3):177–179, 2014.

[12] A. Brunnert, C. Vögele, and H. Krcmar. Automatic performance model generation for java enterprise edition (ee) applications. In *Computer Performance Engineering*, pages 74–88. Springer, 2013.

[13] J. Buckley, S. Mooney, J. Rosik, and N. Ali. Jittac: a just-in-time tool for architectural consistency. *2013 35th International Conference on Software Engineering (ICSE)*:1291–1294, 2013.

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

[15] Contributors to Jakarta RESTful Web Services. Jakarta RESTful Web Services, version 3.0, Sept. 23, 2020.

[16] M. Courtois and M. Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2nd Int. Workshop on Software and Performance*, 2000.

[17] L. de Silva and D. Balasubramaniam. Controlling software architecture erosion: a survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[18] L. Ding and N. Medvidovic. Focus: a light-weight, incremental approach to software architecture recovery and evolution. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 191–200, 2001.

[19] Y. Dodge. *Kolmogorov–smirnov test*. In *The Concise Encyclopedia of Statistics*. Springer New York, New York, NY, 2008, pages 283–287.

[20] H.-F. Eckey, R. Kosfeld, and M. Rengers. *Multivariate Statistik*. Jan. 2002.

[21] O. F. F. Filho and M. A. G. V. Ferreira. Semantic Web Services: A RESTful Approach. In *IADIS International Conference WWWInternet 2009*, pages 169–180. IADIS, 2009.

[22] J. Grohmann, S. Eismann, A. Bauer, S. Spinner, J. Blum, N. Herbst, and S. Kounev. Sarde. *ACM Transactions on Autonomous and Adaptive Systems*, 15(2):1–31, 2021.

[23] J. Grohmann, S. Eismann, S. Elflein, M. Mazkatli, J. von Kistowski, and S. Kounev. Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques. In *Proceedings of the 27th IEEE Int. Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '19, Rennes, France, Oct. 2019.

[24] C. Heger, A. van Hoorn, M. Mann, and D. Okanović. Application performance management: state of the art and challenges for the future. In *Proceedings of the 8th ACM/SPEC on Intl. Conference on Performance Engineering*, ICPE '17, pages 429–432, L'Aquila, Italy. ACM, 2017.

[25] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the gap between modelling and java. In M. van den Brand, D. Gašević, and J. Gray, editors, *Software Language Engineering*. Volume 5969, Lecture Notes in Computer Science, pages 374–383. Springer Berlin Heidelberg, 2010.

[26] R. Heinrich, S. Gärtner, T. Hesse, T. Ruhroth, R. Reussner, K. Schneider, B. Paech, and J. Jürjens. The CoCoME platform: a research note on empirical studies in information system evolution. *Int. Journal of Software Engineering and Knowledge Engineering*, 25(09&10):1715–1720, 2015.

[27] R. Heinrich. Architectural run-time models for performance and privacy analysis in dynamic cloud applications. *ACM SIGMETRICS Performance Evaluation Review*, 43(4):13–22, 2016.

[28] R. Heinrich. Architectural runtime models for integrating runtime observations and component-based models. *Journal of Systems and Software*, 169, 2020.

[29] R. Heinrich, P. Merkle, J. Henss, and B. Paech. Integrating business process simulation and information system simulation for performance prediction. *Software & Systems Modeling*, 16(1):257–277, 2017.

[30] C. Huyen. *DESIGNING MACHINE LEARNING SYSTEMS: An iterative process for production-ready applications*. O'REILLY MEDIA, INC, USA, [S.l.], first edition edition, 2022.

[31] Jakarta EE Platform Team. Jakarta EE Platform, version 8, Aug. 26, 2019.

[32] Jakarta Servlet Team. Jakarta Servlet Specification, version 5.0, Sept. 7, 2020.

[33] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Reading, 2000.

[34] R. Jung, R. Heinrich, and E. Schmieders. Model-driven instrumentation with kieker and palladio to forecast dynamic applications. In *Symposium on Software Performance*, volume 1083, pages 99–108. CEUR, 2013.

[35] E. Kiciman and B. Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. *ACM Trans. Web*, 4(4), 2010.

[36] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner. Enabling consistency in view-based system development – The Vitruvius approach. *Journal of Systems and Software*, 171, 2021.

[37] B. Klatt. *Consolidation of Customized Product Copies into Software Product Lines*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, Oct. 2014.

[38] Knodel Jens and Popescu Daniel. A comparison of static architecture compliance checking approaches. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, page 12, 2007.

[39] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: an analysis of approaches to support model differencing. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, 2009.

[40] M. Konersmann. *Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code*. PhD thesis, May 2018.

[41] M. Konersmann and J. Holschbach. Automatic synchronization of allocation models with running software. *Softwaretechnik-Trends*, 36(4), 2016.

[42] H. Koziolek. Modeling Quality. In *Modeling and simulating software architectures: the Palladio approach*. MIT Press, Cambridge, Massachusetts, 2016.

[43] M. E. Kramer. *Specification Languages for Preserving Consistency between Models of Different Languages*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2017. 278 pages.

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

[45] K. Krogmann, M. Kuperberg, and R. Reussner. Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. *IEEE Transactions on Software Engineering*, 36(6):865–877, 2010. M. Harman and A. Mansouri, editors.

[46] M. Langhammer. *Automated Coevolution of Source Code and Software Architecture Models*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2017. 259 pages.

[47] M. Langhammer and K. Krogmann. A co-evolution approach for source code and component-based architecture models. In *17. Workshop Software-Reengineering und-Evolution*, volume 4, 2015.

[48] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. Automated extraction of rich software models from limited system information. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 2016.

[49] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. Automated extraction of rich software models from limited system information. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 99–108, Apr. 2016.

[50] M. Mazkatli, D. Monschein, J. Grohmann, and A. Koziolek. 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), pages 23–34. IEEE Computer Society, Los Alamitos, 2020.

[51] M. Mazkatli and A. Koziolek. Continuous integration of performance model. In *Companion of the 2018 ACM/SPEC Int. Conference on Performance Engineering*, ICPE '18, Berlin, Germany. ACM, 2018.

[52] M. Mazkatli, D. Monschein, J. Grohmann, and A. Koziolek. Incremental calibration of architectural performance models with parametric dependencies. In *IEEE International Conference on Software Architecture (ICSA 2020)*, pages 23–34, Salvador, Brazil, 2020.

[53] F. Mémoli. Gromov-wasserstein distances and the metric approach to object matching. *Foundations of Computational Mathematics*, 11(4):417–487, 2011.

[54] D. A. Menasce, V. A. Almeida, L. W. Dowdy, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[55] M. Meyer. Continuous integration and its tools. *IEEE software*, 31(3):14–16, 2014.

[56] D. Monschein. *Enabling Consistency between Software Artefacts for Software Adaption and Evolution*. Master Thesis, Karlsruher Institut für Technologie, 2020.

[57] D. Monschein. *Enabling Consistency between Software Artefacts for Software Adaption and Evolution*. Master Thesis, Karlsruher Institut für Technologie, 2020.

[58] D. Monschein, M. Mazkatli, R. Heinrich, and A. Koziolek. Enabling consistency between software artefacts for software adaption and evolution. In *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, pages 1–12, 2021.

[59] T. Olsson, M. Ericsson, and A. Wingkvist. Motivation and impact of modeling erosion using static architecture conformance checking. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 204–209. IEEE, 4/5/2017 - 4/7/2017.

[60] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[61] 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. *9th Int'l Workshop on Models@run.time*:41–46, 2014.

[62] R. Reussner, M. Goedicke, W. Hasselbring, B. Vogel-Heuser, J. Keim, and L. Märtin. *Managed Software Evolution*. Springer, Cham, June 2019.

[63] 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, Cambridge, MA, Oct. 2016. 408 pages.

[64] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st edition, 2012.

[65] 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.

[66] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall/CRC, 4th edition, 2007.

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

[68] S. Spinner, G. Casale, F. Brosig, and S. Kounev. Evaluating approaches to resource demand estimation. *Performance Evaluation*, 92, 2015.

[69] S. Spinner, G. Casale, X. Zhu, and S. Kounev. Librede: a library for resource demand estimation. In *Proceedings of the 5th ACM/SPEC Int. Conference on Performance Engineering*, ICPE '14. ACM, 2014.

[70] S. Spinner, J. Grohmann, S. Eismann, and S. Kounev. Online model learning for self-aware computing infrastructures. *Journal of Systems and Software*, 147:1–16, 2019.

[71] M. Szvetits and U. Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Softw. Syst. Model.*, 15(1):31–69, Feb. 2016.

[72] Teammates developer web site, June 29, 2022.

[73] *The goal question metric approach*. 1994.

[74] G. Upton and I. Cook. *A Dictionary of Statistics*. Oxford University Press, 2008.

[75] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: a java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224, Toronto, Ontario, Canada. IBM Corp., 2010.

[76] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In N. Medvidovic and T. Tamai, editors, *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010 on - WUP '09*, page 41, New York, New York, USA. ACM Press, 2009.

[77] A. van Hoorn. *Model-Driven Online Capacity Management for Component-Based Software Systems*, number 2014/6 in Kiel Computer Science Series. Department of Computer Science, Kiel University, Kiel, Germany, 2014. Dissertation, Faculty of Engineering, Kiel University.

[78] A. van Hoorn, J. Waller, and W. 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)*, pages 247–248, Boston, Massachusetts, USA, April 22–25, 2012. ACM, Apr. 2012.

[79] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. 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, pages 121–140, Tucson, Arizona, USA. Association for Computing Machinery, 2012.

[80] L. Vogel, S. Scholz, and F. Pfaff. Eclipse jdt - abstract syntax tree (ast) and the java model. *vogella GmbH*, 2009-2020.

[81] M. von Detten. Archimetrix: a tool for deficiency-aware software architecture reconstruction. In *WCRE 2012*, [Piscataway, N.J.] IEEE, 2012.

[82] S. Voneva, M. Mazkatli, J. Grohmann, and A. Koziolek. Optimizing parametric dependencies for incremental performance model extraction. In H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Caporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, and U. Zdun, editors, *Software Architecture*, pages 228–240, Cham. Springer International Publishing, 2020.

[83] 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 *2018 IEEE 26th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236. IEEE, 2018.

[84] R. von Massow, A. van Hoorn, and W. Hasselbring. Performance simulation of runtime reconfigurable component-based software architectures. In I. Crnkovic, V. Gruhn, and M. Book, editors, *Software Architecture*, pages 43–58, Berlin, Heidelberg. Springer Berlin Heidelberg, 2011.

[85] 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, Apr. 2017.

[86] D. Werle, S. Seifermann, and A. Koziolek. Data stream operations as first-class entities in component-based performance models. In A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann, editors, *Proceedings of the 14th European Conference on Software Architecture, ECSA 2020*, volume 12292 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 2020.

[87] A. Wert, H. Schulz, and C. Heger. Aim: adaptable instrumentation and monitoring for automated software performance analysis. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pages 38–42. IEEE, 5/23/2015 - 5/24/2015.

[88] What is emf compare?, 2019.

[89] 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 *European Workshop on Performance Engineering*. Springer, 2015.

[90] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, USA. IEEE Computer Society, 2007.

[91] M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. In *Proceedings of ICSE 2007, Future of SE*, pages 171–187. IEEE Computer Society, Washington, DC, USA, 2007.

[92] Y. Xu and R. Goodacre. On splitting training and validation set: a comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning. *Journal of analysis and testing*, 2(3):249–262, 2018.