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

Manar Mazkatli[1], David Monschein[1], Martin Armbruster[1], Robert Heinrich[1], Anne Koziolek[1]

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

Contributing authors: mazkatli@kit.edu; david.monschein@alumni.kit.edu; martin.armbruster@kit.edu; robert.heinrich@kit.edu; koziolek@kit.edu;

**Abstract**

Explicitly considering the software architecture supports system evolution and efficient assessments of quality attributes. In particular, Architecture-based Performance Prediction (AbPP) assesses the performance for future scenarios (e.g., alternative workload, design, deployment) 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. Existing approaches do not address the impact of all aforementioned changes. Besides, extracting the complete aPM after each impacting change causes 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, a self-validation process in CIPM validates the accuracy, manages the monitoring and recalibrates the inaccurate parts.

Consequently, CIPM will automatically keep the aPM up-to-date throughout the development and operation, which enables AbPP for a proactive identification of upcoming performance problems and for evaluating alternatives at low costs.

CIPM is evaluated using three open-source systems, considering (1) the accuracy of updated aPMs and associated AbPP and (2) the applicability of CIPM in terms of the scalability and monitoring overhead. The findings confirmed the accuracy of updated aPMs and the applicability for realistic cases.

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

# 1 Introduction

Most developers nowadays follow agile practices like DevOps (IEEE 2675, 2021). However, performance assurance during agile software development faces problems that we refer to by $P$. For example, the widely used application performance management (Heger et al, 2017) suffers from the required cost ($P_{Cost}$): Assessing the impact of design decisions on performance requires implementing them (Smith and Williams, 2003), setting up test environments and measurements for all design alternatives.

Instead of relying on measurements in real environments, software performance engineering (Woodside et al, 2007; Smith and Williams, 2003) uses models to predict the software performance and identify potential issues earlier (Balsamo et al, 2004). In particular, architectural performance modeling approaches, which model the system's architecture level without implementation details, can be a good basis for cost-effective performance prediction of architectural design decisions (Reussner et al, 2016). Besides, architectural Performance Models (aPMs) increase the human understandability of the system and consequently the productivity of software development (Olsson et al, 2017).

However, applying Architecture-based Performance Prediction (AbPP) in agile development is a demanding task: modeling is a time-consuming process ($P_{Cost}$) and the developers do not trust models since they are approximations and difficult to validate (Woodside et al, 2007)($P_{Uncertainty}$). Besides, accurate AbPP is challenging for various reasons. First, aPMs can be outdated due to frequent software changes ($P_{Inconsistency}$). Here, changes of source code at the Development time (Dev-time) can affect the accuracy of aPMs. Similarly, the adaptive changes at Operation time (Ops-time), like changes in system composition and deployment, also affect the aPMs. 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* (Becker et al, 2009) can lead to inaccurate AbPP for design alternatives ($P_{Inaccuracy}$). 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 ($P_{Monitoring-Overhead}$), because PMPs are mostly calibrated by dynamic analysis of the whole system (Spinner et al, 2015). Keeping the aPMs and their parameterized PMPs consistent with the running system, which is continuously evolving, requires repeated manual effort ($P_{Cost}$).

Hence, the efficient maintenance of the consistency between the *parameterized* aPMs and the corresponding software systems is an important aspect for more comprehensive of the system's architecture and proactive performance management with an AbPP.

## 1.1 State of the Art

Some approaches suggested partial automation of the consistency maintenance between software artifacts. These approaches can be divided into two categories (C1 and C2). C1 includes approaches that reverse-engineer the current architecture based on static analysis of source code (Hamdouni et al, 2010; Langhammer et al, 2016; Becker et al, 2010), dynamic analysis (Brosig et al, 2011; van Hoorn, 2014; Walter et al, 2017) or both (Konersmann, 2018; Krogmann, 2012) . These approaches suffer from shortcomings: First, Not all impacting changes at Dev-time or Ops-time are observed and addressed ($P_{Inconsistency}$). Second, frequent extraction and calibration of aPMs cause high monitoring overhead ($P_{Monitoring-Overhead}$). Third, the possible manual modifications of the extracted aPMs would be discarded and should be repeated by the next extraction ($P_{Cost}$). Finally, there is uncertainty of aPMs' accuracy since no automatic validation is available ($P_{Uncertainty}$).

C2 includes approaches that maintain the consistency incrementally either at Dev-time based on consistency rules (Langhammer, 2017; Ding and Medvidovic, 2001; von Detten, 2012; Voelter et al, 2012; Jens and Daniel, 2007; Buckley et al, 2013) or at Ops-time (Heinrich, 2020; Spinner et al, 2019; van Hoorn, 2014) based on dynamic analysis. None of the C2 approaches succeed in updating aPMs according to both evolution and adaption ($P_{Inconsistency}$). The accuracy of the resulting aPMs and its AbPP is uncertain and unreliable since no statement on the accuracy is provided ($P_{Uncertainty}$). This applies as well to approaches that estimate parametric dependencies to increase the accuracy of AbPP, e.g., (Krogmann, 2012; Grohmann et al, 2019).

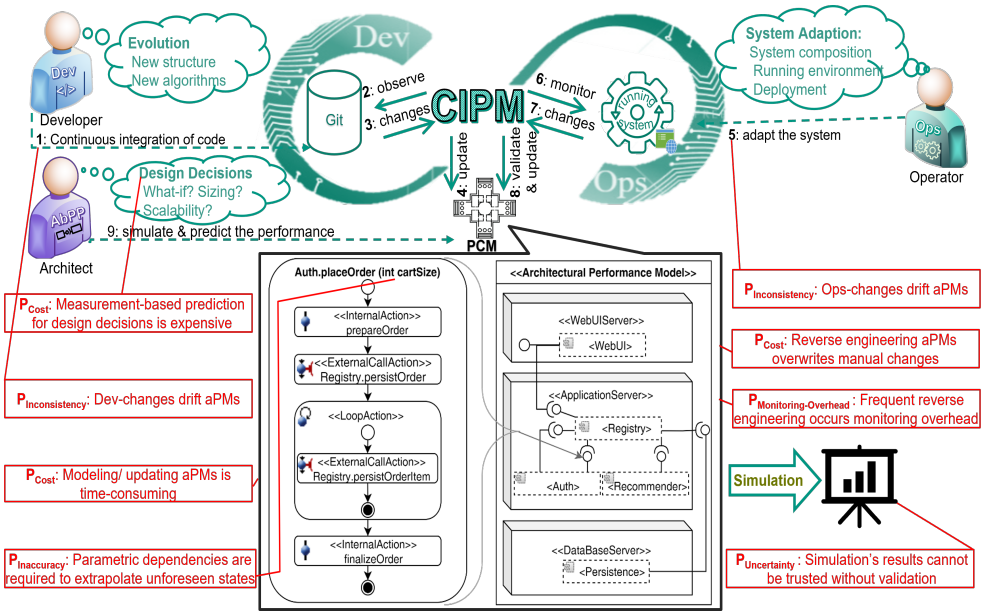


**Fig. 1**: Overview on CIPM approach and the main actors. Red boxes show the problems CIPM addresses. The lower part of the figure abstracts some aspects of aPM (architecture and deployment on the right side and the behavior on the left side) using TeaStore example (von Kistowski et al, 2018) and annotations of Palladio Component Model (PCM) (Reussner et al, 2016)

## 1.2 Approach and Contributions

In this article, we present the Continuous Integration of Performance Models (CIPM) approach. This approach maintains the consistency between the aPM and software artifacts (source code and measurements) (Mazkatli and Koziolek, 2018). As shown in Figure 1, CIPM automatically updates aPMs according to observed Dev-time and Ops-time changes ($P_{Inconsistency}$) to enable AbPP ($P_{Cost}$). CIPM also calibrates aPMs with parametric dependencies ($P_{Inaccuracy}$) and reduces the required overhead for updating and calibrating aPMs through adaptive instrumentation and monitoring ($P_{Monitoring-Overhead}$) (Mazkatli et al, 2020; Voneva et al, 2020). Moreover, CIPM provides a statement on the accuracy of the AbPP and automatically recalibrates inaccurate parts ($P_{Uncertainty}$) (Monschein et al, 2021). In this way, the resulting aPMs allow an accurate AbPP and more comprehension of the system architecture. This supports adopting proactive actions for upcoming performance issues and answering what-if questions about design alternatives ($P_{Cost}$). The novel contributions of this paper are

1. *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 ($P_{Inconsistency}$). Contrary to other approaches, it uses regular Git commits as input and does not require a specialized editor, which eases its integration with CI.
2. *Automated adaptive instrumentation:* We propose commit-based, model-based instrumentation of changed parts in source code (section 6). Compared to existing concepts, our instrumentation automatically detects where and how to instrument the source code to reduce excessive monitoring overhead ($P_{Monitoring-Overhead}$).

In addition, this paper presents the full CIPM approach, including contributions that have been previously published in workshop and conference papers (Mazkatli et al, 2020; Monschein et al, 2021; Mazkatli and Koziolek, 2018; Voneva et al, 2020):

3. *Incremental calibration:* We propose a novel incremental calibration of the PMPs based on adaptive monitoring (Mazkatli et al, 2020). Our calibration uses statistical analysis to learn parametric dependencies. It also optimizes them based on a genetic algorithm if necessary (Voneva et al, 2020). In comparison to existing approaches, CIPM can be performed at Ops-time and addresses $P_{Cost}$, $P_{Monitoring-Overhead}$ and $P_{Inaccuracy}$.
4. *Automated consistency maintenance at Ops-time:* The Ops-time calibration observes Ops-time changes based on dynamic analysis and updates the aPMs accordingly (Monschein et al, 2021) (section 9). In comparison to existing approaches, CIPM automatically updates the aPMs including PMPs, system composition and resource environment using adaptive monitoring ($P_{Inconsistency}$).
5. *Self-validation of updated aPMs:* The self-validation (section 8) estimates the accuracy of AbPP compared with real measurements ($P_{Uncertainty}$). It manages the adaptive monitoring by activating and deactivating monitoring probes based on the validation results. This reduces the required overhead ($P_{Monitoring-Overhead}$). According to our knowledge, our approach is the first approach that enables self-validation of aPMs and dynamic management of monitoring overhead.

6. *Model-based DevOps pipeline:* The proposed pipeline (Mazkatli et al, 2020) integrates and automates the CIPM activities to enable continuous AbPP during DevOps. To implement it, we designed and implemented a transformation pipeline (Monschein et al, 2021) based on (Heinrich, 2020) using the tee and join pipeline architecture (Buschmann, 1998). In contrast to existing pipelines, our pipeline maintains consistency during the whole DevOps life cycle enabling AbPP.

We constructed experiments to evaluate CIPM approach. We assessed the accuracy of the updated models and examined the applicability of CIPM in terms of monitoring overhead and scalability. In addition to the evaluation of novel contributions (1,2), we evaluated the scalability of the approach as a new contribution (subsection 10.6).

This paper is organized as follows. We provide a background on the foundational approaches used throughout the article in section 2. Then, we present a motivating example in section 3. An overview on CIPM approach is in section 4. We describe the novel contributions of the paper (the automatic consistency preservation at Dev-time and the adaptive instrumentation) in section 5 and section 6, respectively. The remaining contributions of CIPM, which calibrate, validate, and maintain the performance model's consistency to the measurements at Ops-time, are discussed in section 7, section 8, and section 9. We present the evaluation results in section 10 and discuss related researches in section 11. Finally, we summarize the article and discuss future work in section 12.

# 2 Background

This section presents background on the foundational approaches we use in the article.

## 2.1 Palladio Component Model

Palladio (Reussner et al, 2016) is a software architecture simulation approach that analyses the software at the model level for performance assessment, e.g., detecting bottlenecks and scalability problems. The Palladio AbPP supports proactive evaluation of design decisions to avoid high costs resulting from wrong decisions. The Palladio Component Model (PCM) consists of five sub-models: The `Repository Model` contains a repository with components, interfaces and roles describing which component provides or requires which interface. 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 users' behavior (how they interact with the system).

The SEFF (Becker et al, 2009) 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* (combinations 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. The remaining loops and branches are combined into internal actions for higher 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 employs the stochastic expression (StoEx) to define PMPs (Koziolek, 2016) using 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 (Langhammer, 2017) keeps the architecture model and the source code model consistent during the software system evolution consistent on Vitruvius platform (Klare et al, 2021). This view-based framework encapsulates the heterogeneous models of a system and the semantic relationships between them in a Virtual Single Underlying Model (VSUM) to keep them consistent. For that, the reactions language of Vitruvius describes the Consistency Preservation Rules (CPRs) at the metamodel level. CPRs 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. Here, Vitruvius stores the mapping between corresponding model elements in a correspondence model to reuse them during the consistency preservation process.

Using Vitruvius, the co-evolution approach keeps Java source code model (Heidenreich et al, 2010) for Java 6 (Joy et al, 2000) and PCM consistent. Langhammer define CPRs that update the `Repository Model` of a PCM model and its behavior (SEFFs *without* PMPs by completely reconstructing it) to respond to changes in the source code. Similarly, changes in the PCM are propagated to the Java source code model.

### 2.3 iObserve

iObserve considers the adaptation and evolution of cloud-based systems as two interwoven processes (Heinrich, 2020). 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 to quality predictions. The PCM is used as the basis for the quality predictions and Kieker (van Hoorn et al, 2012) is used for monitoring the system during operation. iObserve collects monitoring data at Ops-time using 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.

## 3 Running Example

The TeaStore is a web-based application implementing a shop for tea (von Kistowski et al, 2018). The application is based on a distributed microservice architecture and is designed to be suitable for the evaluation of performance modeling approaches.

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 (Filho and Ferreira, 2009).

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 (Heger et al, 2017): "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 would the performance 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 in time and to keep the required manual effort and monitoring overhead as low as possible.

# 4 Continuous Integration of architectural Performance Model

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.

subsection 4.1 describes the models CIPM updates for executing AbPP, while subsection 4.2 explains integrating CIPM processes into the 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 (Langhammer, 2017) 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 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 the source code are fine-grained 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 the 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 keep 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 the iObserve approach (Heinrich, 2020) (cf. subsection 9.4 and subsection 9.5).

Consequently, CIPM continuously updates the aPM parts (A-E) to keep them consistent with the running system. To ensure the accuracy of the AbPP using the updated aPM, CIPM provides a self-validation process (cf. section 8).The following subsection 4.2 describes how we integrate CIPM processes within the DevOps pipeline.

## 4.2 Model-based DevOps Pipeline

DevOps practices aim to close the gap between development and operations by integrating them into one reliable process (IEEE 2675, 2021). 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 we illustrate later.

The MbDevOps pipeline (shown in Figure 2) starts on the "development" side with the Continuous Integration (CI) process (Meyer, 2014) that merges the source code changes of the developers. CI triggers the first process, *CI-based update of software models (1)* (section 5). This process updates the source code model in the VSUM of Vitruvius (1.1) (subsection 5.1). Then, the predefined CPRs in Vitruvius respond to the changes in source code by updating the `Repository Model` (1.2) (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) (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.

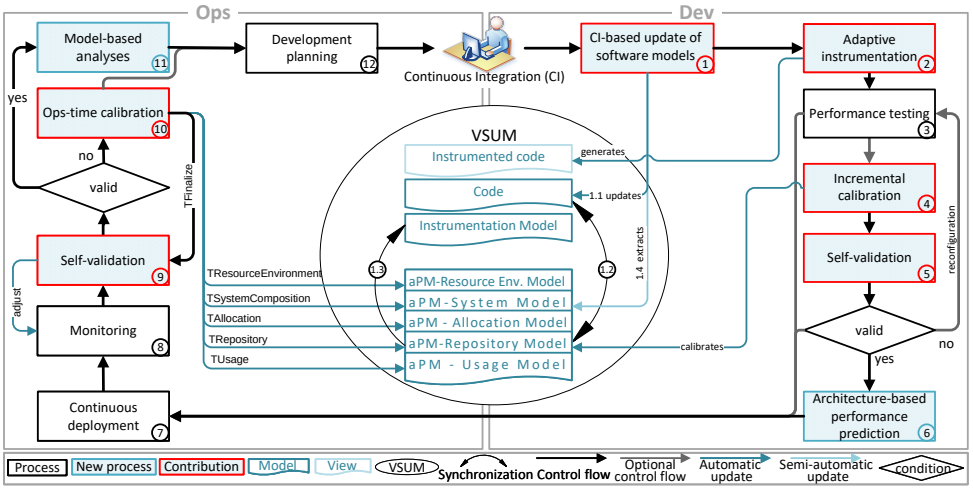


**Fig. 2**: Model-based DevOps pipeline

The pipeline divides the measures into 80% training and 20% validation set (Xu and Goodacre, 2018). Using the training set, the *incremental calibration (4)* estimate the PMPs with the parametric dependencies and enriches the `Repository Model` with them (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 of performing 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 the monitoring data and monitored simulation results to validate the estimated aPM. The result of self-validation is used as input to the Ops-time calibration and 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` (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 respond to the possible Ops-time changes and improve aPM accuracy by the new monitoring data.

The resulting model can be used to perform *model-based analyses (11)*, e.g., model-based auto-scaling. Besides, up-to-date descriptive aPM can support the *development planning (12)* by increasing the understandability of the current version, modeling and evaluating design alternatives and answering what-if questions.

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

# 5 CI-based Update of Software Models

This section describes updating the models affected by the changes in source code by (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 in the VSUM of VITRUVIUS based on (A) (subsection 5.1).
- The `Repository Model` in the VSUM of VITRUVIUS based on (B) ( subsection 5.2).
- The Instrumentation Model in the VSUM of VITRUVIUS by (B) (subsection 5.3).
- The `System Model` based on (A) (subsection 5.4).

The following subsections describe the update process of the models above in detail.

## 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, developing the source code within the VITRUVIUS platform is not required, 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 CI commits.

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.

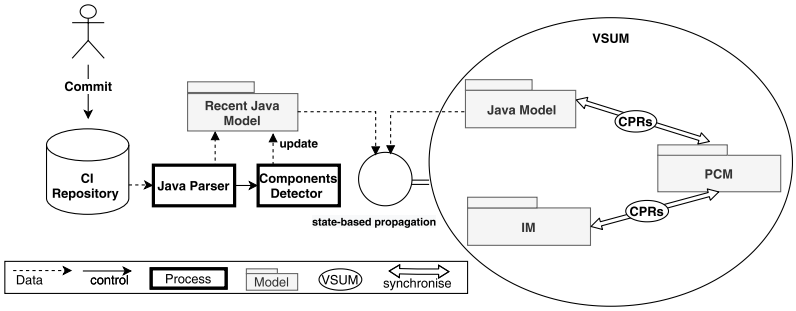


**Fig. 3**: Commit-based Update of the aPM

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) (Heidenreich et al, 2010). We extended the metamodel of JaMoPP to enable the support of Java versions 7-15. Our extensions expand the metamodel with new features, such as 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) (Vogel et al, 2009-2020) 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 on (Armbruster, 2022) [1].

[1]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. In order to retrieve complete code models, the source code can be built 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 (Langer, 2019) extracts the EMF changes by comparing the parsed model with the source code model in the VSUM state-based. In its first step, EMF Compare matches all elements which are considered equal. We use a custom language-specific matching algorithm (Kolovos et al, 2009): We combine the default matching algorithm of EMF Compare with a hierarchical Java-specific matching algorithm that we extended from SPLevo (Klatt, 2014) to be compatible with Java 7-15. Thus, the Java-specific properties and structures are considered to provide accurate matching. After the elements have been matched, the default algorithms of EMF Compare calculate the differences and resulting changes. These 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 previous step (see subsection 5.1). It is mainly based on the CPRs defined by the Co-evolution approach (Langhammer, 2017) that we adapted for our applications.The adjustments in the CPRs are mainly related to (1) technology-specific detection of components and interfaces, (2) update and deletion CPRs and (3) SEFF reconstruction.

Regarding (1), for 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 the model-based detection of the microservices within the VSUM or CPRs. Therefore, we added a pre-processing step after the parsing of the last commit to enhance the source code model with additional information on the components based on the file structures of the code and 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 for the discovery of regular components based on the packages in which a set of classes resides. When the developer decides that a component candidate is a component, they also determine what the component represents: a microservice or a regular component. This decision is also stored and reused in subsequent propagations to eliminate the need to ask the developer again. 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 (Jakarta EE Platform Team, 2019) because these technologies are used in our evaluation. In the context of JAX-RS, classes that are annotated with `@Path` or `@ApplicationPath` constitute a REST API (Contributors to Jakarta RESTful Web Services, 2020) so that an interface is created for such annotated classes. `HttpServlets` as a specialization of `Servlet` provide another possibility to implement a REST API by overriding HTTP-specific methods in subclasses of `HttpServlet` (Jakarta Servlet Team, 2020). Consequently, classes that inherit from `HttpServlet` are also modeled as interfaces. Regarding regular components, we use their public classes and interfaces for modeling their interfaces.

As a further step, the components and interfaces need to be connected. Therefore, if a PCM interface is created for a class, a connecting role between the interface and component containing the class is established by the component providing an implementation for the interface. To connect components with their required interfaces, the CPRs look at imported types and types of fields within a component. If such a type corresponds to an interface from outside the component, the component requires the interface, and a new connection is established.

Regarding (2), we also implemented CPRs for the deletion and change of elements, as the previously described CPRs are intended to detect new elements in the code and add them to the PCM. On one hand, if an element in the source code model is removed, corresponding PCM elements are also removed. For example, if a set of classes relating to a component is removed, the component, its provided and required interfaces, and all interfaces defined by classes contained in the set are deleted. On the other hand, CPRs for changes consider renames (and rename corresponding elements) or changes in methods with a corresponding SEFF so that the SEFF is reconstructed.

Regarding (3), we extend the CPRs that respond to method body changes (e.g., adding/updating statements) by reconstructing the SEFF using a reverse engineering tool (Krogmann, 2012). Our extension extracts the mapping between code statements and their related SEFF actions and stores it in Vitruvius correspondence model. For example, the mapping between internal actions and their associated statements is extracted and stored in Vitruvius's correspondence model.

CPRs continuously update the mapping between the source code and the aPM, which is necessary for the consistency preservation process in Vitruvius. Besides, we use the mapping for two processes in the context of CIPM: the adaptive instrumentation (cf. section 6) and the Dev-time `System Model` extraction, cf. subsection 5.4.

## 5.3 CI-based Update of the Instrumentation Model

For this step, we defined 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). (cf. Figure 3). The defined CPRs generate probes referring to SEFF actions whose corresponding source code statements have changed in the most recent commit. However, the architect and the self-validation process can also add deactivated probes to IM, which the self-validation can activate if their related PMPs are not accurate enough. The adaptive instrumentation uses collected probes in the IM and the mappings in the correspondence model to generate the instrumented source code, see section 6.

## 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 services and extracting the system composition using SCG.

In the following, we introduce the structure of the SCG in subsubsection 5.4.1. 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, an SCG describes "calls-to" relationships between services. We also consider the resource container (computer) on which the respective services are executed. 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, which was introduced in Sec. 3 and visualized in Figure 1.

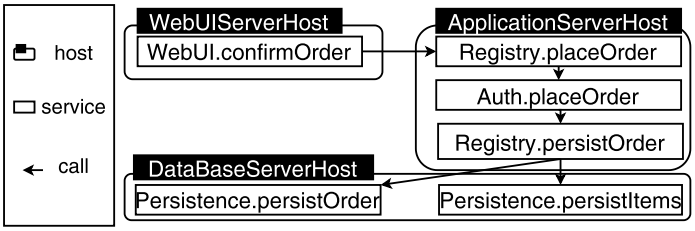


**Fig. 4**: SCG extracted from an excerpt of TeaStore example

### 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 (Vallée-Rai et al, 2010). Based on the mapping between the source code and the `Repository Model` that has been stored in Vitruvius (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, extracting SCG at Dev-time considers all possible execution semantics. At Dev-time, it is also unknown where the components of the services will be hosted. Therefore, we exclude this information, which causes so-called *conflicts* that the user needs to resolve, cf. subsubsection 5.4.3.

### 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 running example, the provided role is the `CartActions` interface, which exposes services for purchasing products and managing orders.

For each provided role, the `Repository Model` is searched for components provided it. If more than one component provide the same interface, the user is asked to select the correct one. Then, instances of the selected components called *assembly contexts* are created and linked to the provided roles using delegations. In our example, `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 (considering SCG 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` calls `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 public instance of `Registry` instead of creating a new one.

# 6 Adaptive Instrumentation

The goal is to automatically instrument parts of the source code that have been changed and that thus may affect the validity of related PMPs. In our running example, if `Auth.PlaceOrder` service (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 number of the loop execution and the parameters of `Rigistry.persistOrder` and `Rigistry.persistOrderItem` external calls. If the source code of the remaining services has not been changed, the old estimations of their PMPs remain valid and they are not instrumented fine-grained. To automate the adaptive instrumentation, we define the measurements metamodel (Figure 5), which consists of monitoring record types corresponding to the probe types defined in IM, cf. subsection 5.3:

- `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

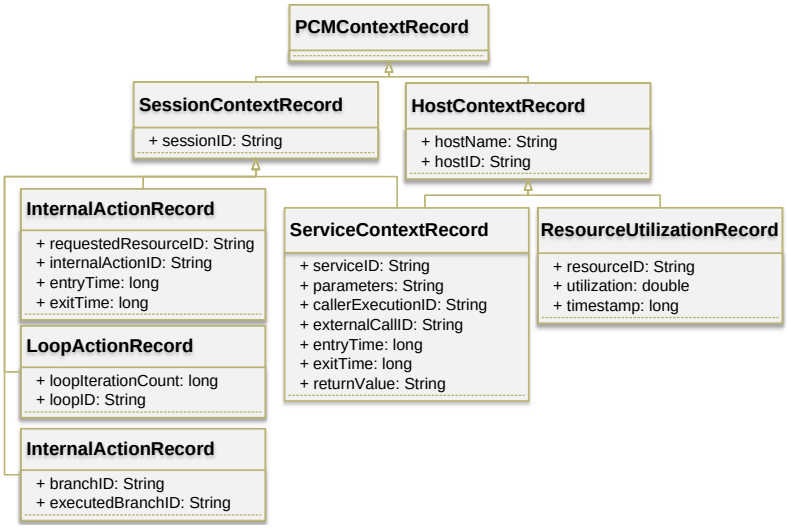


**Fig. 5**: The Measurements Metamodel

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.
- `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 (Jung et al, 2013) of Kieker (van Hoorn et al, 2012). Then, we implemented a model-based instrumentation to generate the instrumented source code as a Vitruvius view. Here, the instrumentation code is responsible for taking the measurements and creating the monitoring records. The Vitruvius 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 (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 the test/production environment to provide measurements for the calibration (section 7).

# 7 Incremental Calibration of PMPs

Based on the measurements resulting from the adaptive instrumentation described in the previous section, the PMPs potentially can be calibrated to enable AbPP.

Applying this process at the Dev-time before the deployment in the production environment allows AbPP for design alternatives instead of the expensive test-based prediction. The calibration of `Repository Model` at Dev-time is similar to that 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 the production environment as section 9 explains.

The incremental calibration involves calibrating new or updated SEFF actions as well as recalibrating inaccurate ones. As stated in subsection 2.1, PMPs are the parameters of SEFF actions, including resource demand of an internal action, the iterations' number of a SEFF loop, the branch transitions of a SEFF branch, and the arguments and return value of a SEFF external call. In subsection 7.1 we propose an incremental estimation of resource demands based on adaptive monitoring. We then identify the parametric dependencies to estimate all PMPs in relation to the impacting factors, see subsection 7.2. Finally, we describe the adaptive optimization of the found parametric dependencies using the genetic algorithm in 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 (Spinner et al, 2015) or require expensive fine-grained monitoring (Brosig et al, 2009; Krogmann et al, 2010). Therefore, we propose in the following paragraph a lightweight RDE process that is based on adaptive instrumentation and monitoring to allow for an incremental RDE.

Our incremental RDE extends the approach Brosig et al (2009) 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 $i$ for resource $r$ ($D_{i,r}$) based on service demand law shown in Equation 1.

$$D_{i,r} = \frac{U_{i,r}}{C_i/T} = \frac{U_{i,r} \cdot T}{C_i} \quad (Menasceet\ al,\ 2004) \tag{1}$$

Here, $U_{i,r}$ is 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 a fixed observation period of length $T$. 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 (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 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 Equation 1. If 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 (2009) 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. CIPM begins by estimating the dependencies of loops (subsubsection 7.2.2), branches (subsubsection 7.2.3), and arguments of external calls (subsubsection 7.2.4) because the incremental RDE requires traversing the SEFF control flow to estimate the utilization of NMIAs. Then CIPM estimates the parametric dependencies for the RDs (subsubsection 7.2.1). CIPM learns 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 (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:

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

$p_0$, $p_1$.. $p_n$ 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 the relationship between the number of loop iterations and input parameters, we use loop records that log the iterations' count and refer to the enclusing service calls. Then, we combine the monitored loop iterations with the integer input parameter the service call into one data set. Non-integer parameters are filtered out, and their integer properties are considered (e.g., number of list elements or file size). We also add quadratic and cubic transformations of parameters to test additional relationships. Regression analysis is used to estimate the weights of influencing parameters.

As the loop iteration count is an integer value, the output value must also be an integer. Approximations or distributions of integer values are used for the resulting non-integer weights, e.g., 1.6 is expressed as distribution that takes the value 1 in 40% of cases and the value 2 in 60% of cases.

If cross-validation finds no relationship to the input parameters, we replace the constant value of the resulting stochastic expression with an integer distribution function, similar to the fifth step of parameterized RDE in subsubsection 7.2.1.

### 7.2.3 Branch transitions

To estimate parameterized branch transitions, we utilize pre-defined branch monitoring records that log chosen branch transitions and the enclosing service call reference. We then build a dataset for each path, including the input parameters of the service call. We use a decision tree algorithm to identify potential relationships between the input parameters and the chosen path, and filter out any non-significant parameters based on cross-validation. If a relationship is found, we can transform the decision tree into a boolean stochastic expression for each branch transition, allowing us to calculate the probability of selecting a specific transition for a given set of input parameters. However, if no relationship is found, we can still estimate the probability by constructing a boolean distribution.

### 7.2.4 External call arguments

In this step, we predict the parameters of an external call by examining their relation to the input parameters of the calling service and the data flow, which refers to the return value of the previous internal or external actions. We first determine if each parameter is a constant value, identical to one of the candidates, or dependent on them. To identify the dependencies, we use linear regression for numeric parameters, decision tree for boolean/enumeration parameters, and discrete distribution for other parameter types.

## 7.3 Adaptive Optimization of Parametric Dependencies

The Genetic Algorithm (GA) is a heuristic algorithm used to optimize solutions for an optimization problem (Kramer, 2017). In the context of adaptive optimization of parametric dependencies, GA is used to improve the accuracy of inaccurate PMPs and reduce the complexity of the resulting StoEx. The optimization process is triggered only for the PMPs with high cross-validation errors. The GA evolves initial solutions represented as genes (previous StoEx estimates), aiming to output StoEx with cross-validation error no greater than the original. We represent the genes as mathematical

functions in an Abstract Syntax Tree (AST), similar to Krogmann (2012). Then, the evolutionary loop of GA generates new candidate solutions by evolving the genes using crossover and mutation operators. The crossover operator randomly swaps two subtrees belonging to two trees, while the mutation operator alters some genes to ensure diversity in the new population, e.g., altering a subtree or some values. The fitness function evaluates the resulting solutions based on two criteria: prediction accuracy using MSE and complexity of the mathematical expression using the depth of the AST. Best solutions are selected as parents for the next generation, and the process continues until a termination condition is met. The termination condition can be an optimal solution, defined as the best fitness being lower than a given minimum threshold, exceeding a maximum execution time, or evolving a fixed number of generations. Finally, the resulting AST is converted to a StoEx and transferred to `Repository Model`. The adaptive optimization of inaccurate PMPs is the last step of the incremental calibration and is followed by the self-validation process that informs 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. To determine the prediction accuracy of a model, a baseline is required. CIPM uses measurements from the real system as a reference, which are available from the monitoring, both from test environments at Dev-time and from the production environment at Ops-time.

By comparing the 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 obtain 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. For the comparison, we use the Wasserstein distance (Santambrogio, 2015), Kolmogorov–Smirnov test (Dodge, 2008) and conventional statistical measures (e.g., average and quartiles). More details on the used 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. 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. Another option is to wait for the Ops-time calibration, where measurements based on the real usage of the system can be continuously monitored until the accuracy degree is accepted.

The resulting metrics are also used to adjust the granularity of the monitoring: The monitoring for certain services can be deactivated If a predefined accuracy threshold is met. This help in reducing the monitoring overhead. However, the self-validation can activate the fine-grained monitoring for some parts if their accuracy degree is not enough. If the inaccurate parts are not instrumented, new probes for them are added to IM to recalibrate these parts after the next deployment. So the monitoring management balances the accuracy of aPM and the required monitoring overhead.

# 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 (Heinrich, 2016). To realize the transformation pipeline we used a tee and join pipeline architecture (Buschmann, 1998), 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 detail these transformations.

## 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 (Klare et al, 2021) to keep our REM consistent with the resource environment in the corresponding aPM. 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

The process of extracting a `System Model` at Ops-time is similar to Dev-time, and it requires a SCG. 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. Figure 5 and 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 (subsubsection 5.4.2). Moreover, we can attach the information about the host to the SCG nodes, because it 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 (Monschein, 2020).

#### *Updating the System Model*

The $T_{SystemComposition}$ updates the `System Model` based on the extracted SCG and the methodology in subsubsection 5.4.3. The only difference is that no conflict happens since 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, 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 (subsection 9.3). 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 (Monschein, 2020).

## 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 (section 7) and includes the validation results to optimize the PMPs. The simple optimization is based on the regression analysis. It adjusts the regressions' parameters based on the validation results to minimize the error between the measured response time and monitored one. If more optimization is required, the optimization using the genetic algorithm can also be triggered ( subsection 7.3). In parallel, $T_{Usage}$ analyses the user behavior and updates the `Usage Model` based on iObserve (Heinrich, 2020).

## 9.6 Finalization of Ops-time Calibration

The final step of Ops-time Calibration, $T_{Finalize}$, executes the self-validation (section 8). Based on the self-validation results and configurable criteria, the granularity of the monitoring is adjusted, i.e., fine-grained 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., the 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 adopt the Goal-Question-Metrics approach of van Solingen et al (2002): We define the evaluation goals and drive the Evaluation Questions EQs that can check whether the described goals are reached or not, cf. subsection 10.1. In subsection 10.2, we define metrics that can answer the EQs. Then, 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 has 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 update the `Repository Model` and its related models within VSUM after a git commit?
  - **EQ-1.2:** Can we aggregate commits and propagate them together without affecting the accuracy? If yes, users can also choose less frequent execution of CIPM, e.g. in nightly builds.
  - **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 AbPP accuracy?
- **G2: Monitoring Overhead**
  - **EQ-2.1:** To what extent can the adaptive instrumentation reduce 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-3.1:** How does the transformation pipeline of CIPM scale with an increasing number of monitoring records?

## 10.2 Evaluation Metrics

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

### 10.2.1 Model Conformity

The Jaccard similarity coefficient (JC) (Eckey et al, 2002) calculates the similarity of two sets ($A$ and $B$) as follows:

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

The resulting JC ranges between 0 to 1. The higher the value is, the more similar the two sets are. The JC is 1 if the sets are identical.

As models are sets of elements, we can also apply this concept to models. Therefore, we implemented matching algorithms that determine the identical elements (intersection) of two models $A$ and $B$ based on different factors: the types of elements, some properties, e.g., the name of named elements, some references as the implementing interface of a SEFF, and the model structure, for example, the position of SEFF actions. The matched elements will be considered as the intersection of the models ($A \cap B$) whereas the models' union ($A \cup B$) consists of the matched and unmatched elements.

We implemented the JC to evaluate the equality of `System Models`, `Allocation Models`, and `Resource Environment Models`[2] (Monschein, 2020, p. 69).

In the case of `JaMoPP Models` and `Repository Models`, we implemented matching algorithms based on EMF Compare (Langer, 2019) respecting the models' structures (see subsection 5.1 for JaMoPP and (Armbruster, 2021, p. 37) for more detail).

### 10.2.2 Distribution Comparison

To compare distributions, we use three types of metrics: conventional statistical measures (Upton and Cook, 2008), non-parametric tests (Kolmogorov–Smirnov test (KS test)) (Sheskin, 2007) and distance functions (Wasserstein) (Mémoli, 2011). These metrics can be used to compare the distributions of the monitored response times (reality) with the simulated response times (prediction).

KS test (Dodge, 2008) 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. KS test is sensitive to the shifts and shapes of distributions, which may produce undesirable false positive alerts (high values) (Huyen, 2022). 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 (Mémoli, 2011) measures the distance between distributions, describing the effort to transform one into the other. An advantage of this metric is that, unlike the KS test, it is insensitive to the distributions' shapes. A drawback, however, is that a result is an absolute number that cannot be easily interpreted without a baseline.

[2] For more detail, see https://github.com/CIPM-tools/CIPM-Pipeline/wiki/Model-Conformity.

Statistics metrics (e.g., mean or quartiles) are also 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.

### 10.2.3 F-Score

The F-Score is a measure to assess the quality of a binary classification (Derczynski, 2016). Its calculation (Goutte and Gaussier, 2005) is based on the number of correctly classified entities (True Positives (TP) if they belong to the given class and True Negatives (TN) if they do not belong to the class) and incorrectly classified entities (False Positives (FP) if they are assigned to the class and do not belong to the class and False Negatives (FN) if they are not assigned to the class and belong to the class):

$$F - Score = \frac{(1 + \beta^2) * TP}{(1 + \beta^2) * TP + \beta^2 * FN + FP} \tag{7}$$

We use the F1-Score (F-Score with $\beta = 1$) to evaluate the update of the IM. Thus, we count SEFFs and changed SEFF actions with a corresponding instrumentation point as TPs, SEFFs and changed SEFF actions without a corresponding instrumentation point as FNs, and instrumentation points without a corresponding SEFF or SEFF action as FPs. Based on these numbers, we can calculate the F1-Score.

## 10.3 Experiment Setup

The evaluation of our approach is based on three cases: TeaStore (von Kistowski et al, 2018) that is introduced in section 3, Common Component Modeling Example (CoCoME) (Heinrich et al, 2015), and the real-world case "TEAMMATES" [3].

CoCoME is a trading system for supermarkets (Heinrich et al, 2015). 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 students' feedback with a Web-based frontend and a Java-based backend, on which we focus 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 incremental calibrations of aPM (Mazkatli et al, 2020) and (B) after optimization of the PMPs (Voneva et al, 2020). The main reason for 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 cases 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 [4].

[3] See https://github.com/TEAMMATES/teammates

[4] 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** and **EQ-1.2**. To achieve this goal, we used two cases: TeaStore and TEAMMATES.

*E1 on TeaStore.* we propagate the changes between version 1.1 and 1.3.1 of the TeaStore. The commits from version 1.1 to version 1.3.1 can be split into four intervals (I) [1.1, 1.2], (II) [1.2, 1.2.1], (III) [1.2.1, 1.3], and (IV) [1.3, 1.3.1]. The first interval consists of 50 commits, of which 27 commits affect 144 Java files with overall 9553 added and 7908 removed lines. In contrast, interval (II) contains 20 commits, of which 12 commits affect 5 Java files with 141 added lines and one removed line. Three Java files (123 lines in total) were added. In interval (III), seven of 11 commits affect 4 Java files with overall 121 added and 134 removed lines while 9 Java files with overall 215 added and 227 removed lines are affected by 12 of 100 commits in interval (IV).

The initial commit includes a lot of architectural-relevant changes. By propagating this commit, an incremental reverse engineering of version 1.1 of the code is performed (IRE). Considering interval (I), it contains five architectural-relevant changes and no changes in the dependencies. The successive commits in interval (II), on which we concentrate in the following, 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. Besides, there are no changes in the dependencies. Both intervals (III) and (IV) contain no architectural-relevant changes and no changes in the dependencies.

Before the changes are propagated, we use the changes between an empty repository and version 1.1 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 1.1 into the version 1.2. We also execute the adaptive instrumentation after the update process and repeat the complete procedure for every interval. For every commit, components are detected by the microservice-based strategy, and a build of the TeaStore is performed. Only if the build succeeds, the commit is propagated.

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 (SCM), (2) the `Repository Model` (RepM), 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 one in the VSUM. We compare the updated source code model with the generated reference (SCM') by calculating the JC metric.

To evaluate the automatically updated `Repository Model` (2), we compare it with a manually updated `Repository Model` ($RepM'$) used as a reference. 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 also propagate the version 1.3.1 as an initial commit to compare the resulting model with a manual available one (Monschein, 2020).

The expected changes in the SEFFs should cause the generation of new probes in the IM (3). Thus, we calculate the F-Score for the IM.

Regarding (4), we first check that no compilation errors occur because of the instrumentation. Then, we check whether the instrumentation statements related to the IM probes are correctly injected into the source code (**EQ-1.5**).

Regarding (5), we check how much the adaptive instrumentation can reduce the monitoring overhead by calculating the ratio of adaptively instrumented probes to all probes required to calibrate the whole aPM (**EQ-2.1**). We also calculate the ratio of the fine-grained adaptively instrumented probes to all possible fine-grained probes.

To answer **EQ-1.2**, for each interval, we propagate all commits individually and then the commits between two Versions (V) as one commit, e.g., the 20 commits between version 1.2 and 1.2.1 are propagated as a single commit. Then, we compare the resulting `Repository Model` ($RepM_{\sum V}$) to the result of the propagation of multiple commits incrementally ($RepM_{V_{inc}}$) and a manually updated `Repository Model` ($RepM'$) using JC. The resulting source code model and IM are 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 impact 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**. 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 [5]. Like TeaStore, we repeat the E1 on TEAMMATES to answer **EQ-1.1**, **EQ-1.5** and **EQ-2.1**. However, different to TeaStore, the components of TEAMMATES are detected by the package-based strategy, and missing dependencies in the source code model are recovered instead of building TEAMMATES.

### *Experiment 2 (E2)*

In this experiment, we evaluate the extraction of a `System Model` at Dev-time that is explained in subsection 5.4. The input is the source code of TeaStore and CoCoME.

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

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)***

This experiment aims to evaluate the accuracy of Ops-time calibration (G1).E3 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 Heinrich addressed it extensively.

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 system. Here, we used the metrics introduced in subsubsection 10.2.2 to answer **EQ-1.6**, **EQ-1.7** and **EQ-1.8**. To quantify the accuracy of AbPP, the following procedure is used:

1. Execution of E4, storage of the derived models and the related 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 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 so that it correctly reflects the system at the respective point in time.

E4 was executed 10 times, each time with different changes. Every 5 minutes the next change is executed. In total, E4 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 E4, we only consider the TeaStore case 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. 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 ensure that our approach recognizes parametric dependencies (**EQ-1.7**) and the system composition (**EQ-1.4**).

In E4, we measure the monitoring overhead and data generated in the worst case, where the whole source code is fine-grained instrumented. This allows us to evaluate whether self-validation can reduce monitoring overhead, answering **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, that it produces worst-case execution times. By means of this experiment, scalability questions can be answered (**EQ-3.1**).

## 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 1 shows an excerpt of the evaluation results for the updated models of TeaStore in experiment E1. The excerpt includes the initial commits of each interval and commits whose changes contain architectural-relevant changes. The resulting RepMs are distinguished according to the version, interval and commit number.

Additionally, Table 2 displays the results for TEAMMATES. The results in both tables reveal that the calculated JC for the Java models is one, i.e., the source code models in the VSUM are correctly updated. 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., the F-Score is one. As a consequence for answering **EQ-1.1**, these results indicate that the source code model, `Repository Model` and IM were correctly updated. The results for the remaining commits of TeaStore are identical. We obtained JC values of one for all Java model and repository model comparisons and F-Scores of one for all IMs.

**Table 1**: Excerpt of E1 results by TeaStore case.

| | $JC(SCM, SCM')$ | Resulting $RepM$ | $JC(RepM, RepM')$ | $F(IM, IM')$ |
|---|---|---|---|---|
| I | 1 | $RepRepM_{I.3}$ | 1 | 1 |
| | 1 | $RepM_{I.18}$ | 1 | 1 |
| II | 1 | $RepM_{1.2.1}$ | 1 | 1 |
| | 1 | $RepM_{II.10}$ | 1 | 1 |
| | 1 | $RepM_{II.11}$ | 1 | 1 |
| | 1 | $RepM_{II.13}$ | 1 | 1 |
| | 1 | $RepM_{II.18}$ | 1 | 1 |
| III | 1 | $RepM_{1.3}$ | 1 | 1 |
| IV | 1 | $RepM_{1.3.1}$ | 1 | 1 |
| Execution time of IRE = 36.6 min | | | | |
| Update time average = 4.24 min | | | | |
| Instrumentation time average = 0.6 min | | | | |

Both tables 1 and 2 contain the execution time for the initial commit and the average execution time for the model update and instrumentation. Between TeaStore and TEAMMATES, there is a difference in the execution time of the IRE: it takes 27.21 minutes longer in case of TeaStore because the TeaStore code is built before it is parsed. This results in a higher number of source code model elements compared to the recovered elements in TEAMMATES which need to be propagated.

**Table 2**: E1 results by TEAMMATES case.

| | $JC(SCM, SCM')$ | Resulting $RepM$ | $JC(RepM, RepM')$ | $F(IM, IM')$ |
|---|---|---|---|---|
| II | 1 | $RepM_{II}$ | 1 | 1 |
| III | 1 | $RepM_{III}$ | 1 | 1 |
| IV | 1 | $RepM_{IV}$ | 1 | 1 |
| V | 1 | $RepM_{V}$ | 1 | 1 |
| Execution time of IRE = 9.39 min | | | | |
| Update time average = 2.05 min | | | | |
| Instrumentation time average = 0.45 min | | | | |

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 that are not present in the manually created one.

Additionally, we check the instrumented source code generated during E1 to ensure that it includes all probes that were represented in IM. We find that the source code is correctly instrumented and has no compilation error, which answers **EQ-1.5**.

The evaluation results for the propagation of the changes as one commit for answering **EQ-1.2** 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 cases 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

**Table 3**: Evaluation results for propagating TeasStore versions' commits as one, comparing resulting `Repository Model` ($RepM_{\sum V}$) to manual reference ($RepM'$) and one updated incrementally ($RepM_{V_{inc}}$).

| Versions (V) | Version 1.1 | Version 1.2.1 | Version 1.3 | Version 1.3.1 |
|---|---|---|---|---|
| $JC(SCM, SCM')$ | 1.0 | 1.0 | 1.0 | 1.0 |
| $JC(RepM_{\sum V}, RepM')$ | 1.0 | 1.0 | 1.0 | 1.0 |
| $JC(RepM_{\sum V}, RepM_{V_{inc}})$ | 1.0 | 1.0 | 1.0 | 1.0 |
| $F(IM, IM')$ | 1.0 | 1.0 | 1.0 | 1.0 |

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

| Open Source System | JC | Model Elements | Conflicts Number |
|---|---|---|---|
| CoCoME | 1.0 | 16 | 2 |
| TeaStore | 1.0 | 18 | 5 |

manually during the process. According to these results, **EQ-1.3** can be answered, as the system compositions were reflected correctly in the extracted `System Models`.

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

**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 |

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 cases, 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.

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

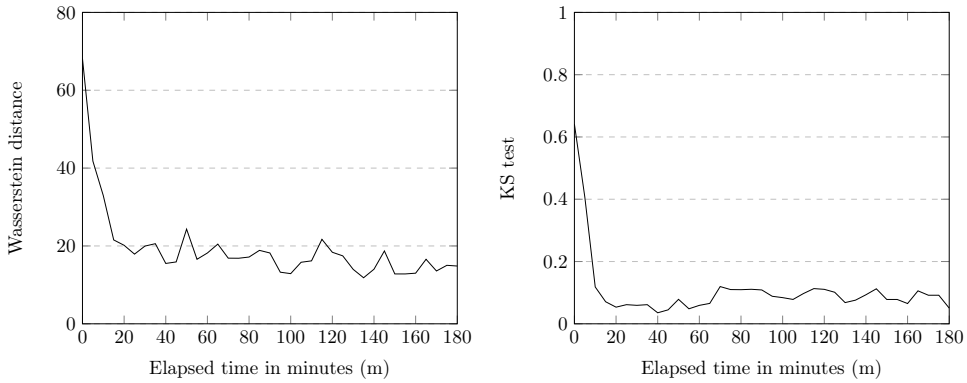


**Fig. 6**: Overview of the metrics over time for the CoCoME case (comparing the distributions which result from the analysis and the monitoring)

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.

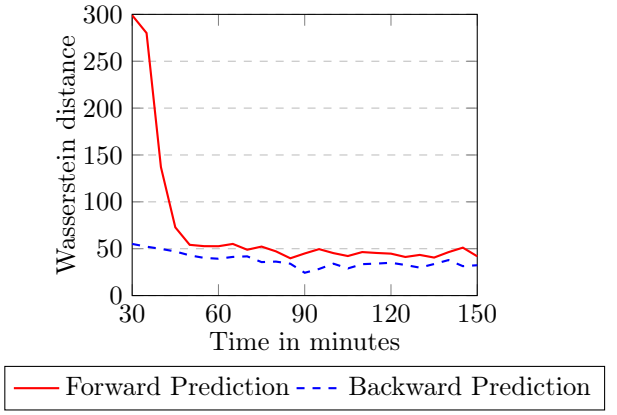


**Fig. 7**: Median Wasserstein distance for forward and backward prediction of TeaStore's *confirmOrder* service response times using models derived at a given point in time.

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 (Mazkatli et al, 2020). 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 (Mazkatli et al, 2020) (**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 (Voneva et al, 2020). 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.

**Table 6**: Aggregated metrics of the forward and backward prediction over time

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

The observations can be confirmed using additional metrics (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. Besides, it is important to note that the average and standard deviation of the forward prediction metrics are strongly affected by the high values at the beginning of the experiment. Based on 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 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 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 (Monschein, 2020, 86). Applying E4 on TeaStore also confirmed the accuracy of aPMs despite of the simulated Ops-time changes.

Additional experiments on TeaStore and CoCoME in Mazkatli et al (2020) studied the incremental calibration based on adaptive instrumentation in more detail and also confirmed the accuracy of AbPP (**EQ-1.6**). The experiments resulted in KS test values not exceeding on average 0.16 and Wasserstein distances lower than 39.6 on average.

## 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 69% (**EQ-2.1**). On the level of the fine-grained probes, the reduction of the monitoring overhead is between 72.8% and 100%. In Mazkatli et al (2020), we obtained similar results in evaluating the reduction of monitoring overhead in the case of adaptive instrumentation. Even if the developer decides to instrument the whole source code, the monitoring overhead will be reduced because only the probes for parts have been changed after the last commit are activated.

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

| Case | Commit | The reduction ratio of probes | The reduction ratio of fine-grained probes |
|---|---|---|---|
| Teastore | I.3 | 60.5% | 85.2% |
| | I.14 | 67.7% | 95.5% |
| | II.10 | 67.8% | 96.3% |
| | II.11 | 67.8% | 96.3% |
| | II.13 | 67.8% | 96.3% |
| | II.18 | 68.1% | 97.6% |
| | III | 69% | 100% |
| | IV | 69% | 100% |
| Teammate | TM-1 | 55.4% | 88.9% |
| | TM-2 | 46.0% | 72.8% |
| | TM-3 | 62.8% | 99.5% |
| | TM-4 | 60.7% | 96.4% |

Second, 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 parts of the monitoring that are independent of the granularity of the monitoring, such as observing resource utilization. Figure 8 shows the results obtained by forming the median from multiple experiment executions. The dashed line in the graph highlights the point in time when the first switch from fine-grained monitoring to coarse-grained monitoring happened.

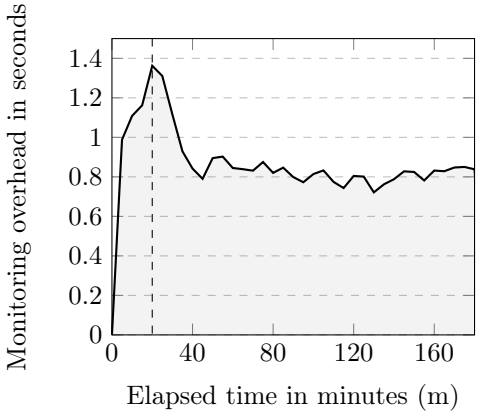


**Fig. 8**: Median of the monitoring overhead over time for five minute intervals

Based on the graph above, 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%**.

***Summary:*** Together with the evaluation results of the model and the prediction accuracy, it can be concluded that the self-validation 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 addition to the reduction can be achieved by the adaptive instrumentation.

## 10.6 Scalability

We address **EQ-3.1** 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 evaluation, 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 future work. For these reasons, we focused here on the scalability of the transformation when using 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. Two factors 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.

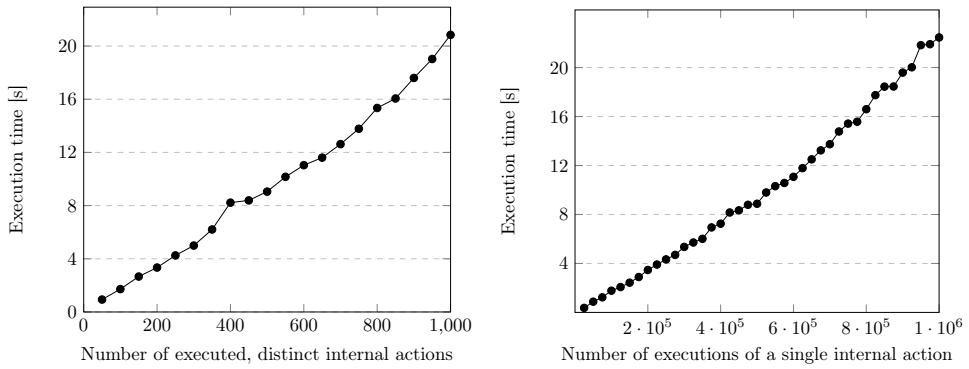


**Fig. 9**: Scalability analysis of the repository transformation under various circumstances

Even for a high number of internal actions (a), the growth of the execution time remains linear. 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.

### 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

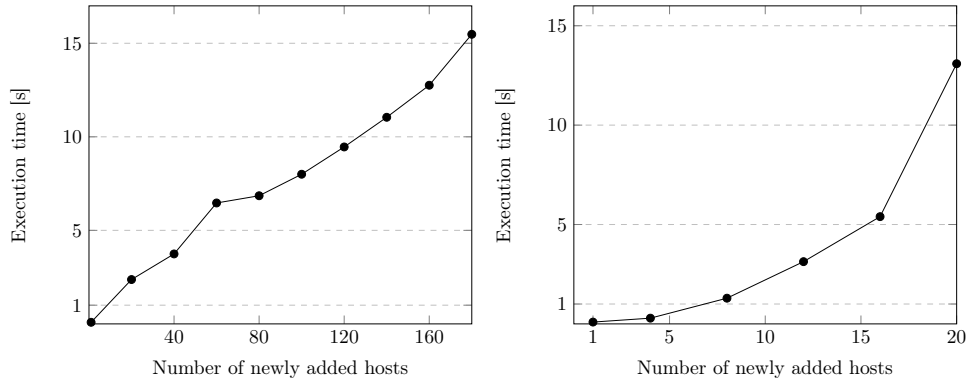


**Fig. 10**: Scalability analysis of the transformation of `Resource Environment Model` in different scenarios

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

To recap, the analysis results for the `Resource Environment Model` transformation indicated that the execution times scale is 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 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.

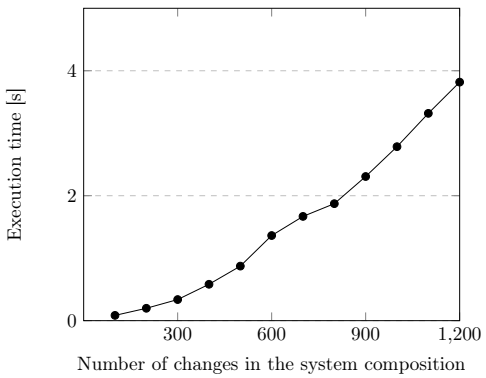


**Fig. 11**: Execution times of `System Model` transformation with increasing changes in the system composition.

### 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 (Heinrich, 2020).

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.

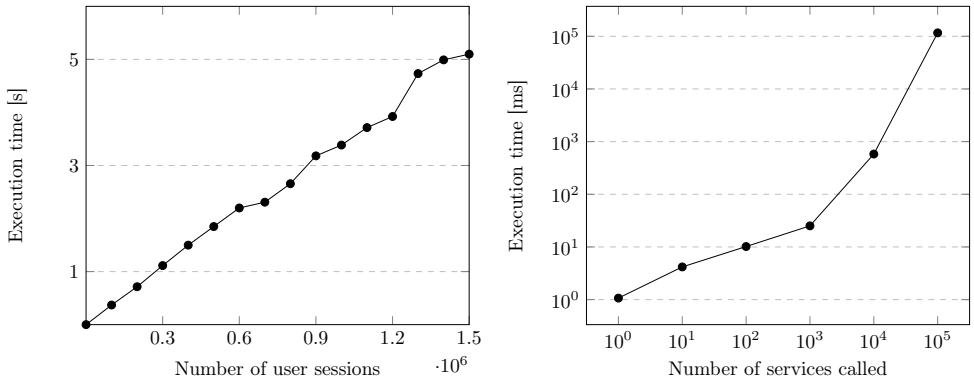


**Fig. 12**: Scalability analysis of the `Usage Model` transformation

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 (Heinrich, 2020). 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. In the superlinear segment, the execution time is dominated by the loop detection (Heinrich, 2020). 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.

***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 (Wohlin et al, 2012). Therefore, we distinguish between three dimensions of validity: *internal validity*, *external validity*, and *construct validity*.

***Internal Validity:*** The first threat to validity 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 an incorrect decision, the calculated JC would be lower. The second threat to validity is by applying E1 on Teammates where we do not have a reference PCM to evaluate the first PCM obtained by propagating the initial commit. However, we evaluated the resulting PCM by the comparison with the available well-documented architecture[6].

***External Validity:*** An external threat to validity is the selection of cases. It may be possible that the results obtained from the cases are not representative. To avoid this, we selected CoCoME, TeaStore and Teammates, which are widely used in research and address common business use cases (Reussner et al, 2019; Keim et al, 2021; Grohmann et al, 2019; von Kistowski et al, 2018). By combining the several cases, the risk of non-representative results is further reduced.

***Construct 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 (Heinrich et al, 2017; Mazkatli et al, 2020; Voneva et al, 2020) 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 (Heinrich, 2020). Besides, in the evaluation, we rely on a combination of synthetically generated monitoring data and monitoring data generated directly by executing a system. 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 system, 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 (van Hoorn et al, 2012; Mazkatli et al, 2020).

# 11 Related Work

There are a lot of approaches that aim to achieve the consistency between the software artifacts automatically. As explained in section 1, 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 tries to eliminate it (called as incremental approach and marked as $\checkmark_{inc}$ in Table 8). Both C1 and C2 can be also classified into 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.

## 11.1 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

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

one (C2). The reverse engineering approaches at Dev-time (C1) is based mainly on static analysis of source code. For example, SoMoX (Becker et al, 2010) and Extract (Langhammer et al, 2016) extract parts of PCM. Similarly, the ROMANTIC-RCA (Hamdouni et al, 2010) 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. de Silva and Balasubramaniam (2012) present good summary of these approaches. Moreover, Jens and Daniel (2007) compare the approaches that minimize the architecture erosion by detecting architectural violations at Dev-time. JITTAC tool (Buckley et al, 2013), for instance, detects the inconsistencies between architecture models and source code, but does not eliminate them automatically. Archimetrix (von Detten, 2012) also detects 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 mbeddr approach of Voelter et al (2012) and the Co-evolution approach of Langhammer (2017). The mbeddr approach uses a single underlying model for implementing, testing and verifying system artifacts like component-based architecture. Similarly, the Co-evolution approach uses a virtual single underlying model to allow the co-evolution of PCM and source code. The Focus approach of Ding and Medvidovic (2001) avoids the inconsistencies by recovering the architecture and using it as a basis for the evolution of object-oriented applications. 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 supported. Therefore, some of these approaches are extended to allow AbPP. For example, Langhammer calibrated the co-evaluated approach with an approximation of resource demand (response times) to show that it can be used for AbPP. Similarly, Krogmann et al extended SoMoX with a calibration of PMPs with the parametric dependencies based on dynamic analysis (Beagle approach (Krogmann, 2012)). However, the approach of Krogmann requires high monitoring overhead which restricts collecting the monitoring data from the production environment rather than from the test environment. Therefore, we assign Krogmann's approach to the Dev-time approaches rather than 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 incremental calibration. Moreover, all the consistency management approaches at Dev-time ignore the effect of adaption at Ops-time on the accuracy of aPMs.

## 11.2 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 approaches of Brosig et al (2011) , (Walter et al, 2017) and (Brunnert et al, 2013) are reverse engineering approaches (C1) that extract parts of the PCM based on dynamic analysis for AbPP. Furthermore, the SLAstic approach (van Hoorn, 2014) extracts aPM and can detect some changes at Ops-time like migrations and reflect them in the model (C2). A

previous work of ours, iObserve (Heinrich, 2020), can also respond to changes in deployment and usage by updating the related parts in PCM (C2), which we also integrated into CIPM. Other approaches that extract/ update performance models at Ops-time are summarized by Szvetits and Zdun (2016). The Drawbacks of the Ops-time approaches are that continuously high monitoring effort is required to extract/ update models. They cannot also model the system's parts that have not been called and consequentially not covered by the monitoring. Besides, these approaches ignore the source code changes do not validate the accuracy of the resulting models.

## 11.3 Hybrid approaches

The scope of hybrid approaches spans Dev-time and Ops-time. For example, Langhammer introduces also a reverse engineering tool (EjbMox) (Langhammer, 2017, 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 approach of Konersmann (2018) integrates information about the architecture model into the code via annotations for a dynamic generation of an architecture model from the source code via transformations. Moreover, the approach of Konersmann synchronizes allocation models with running software (Konersmann and Holschbach, 2016). Spinner et al (2019) propose an agent-based approach to update architectural performance models (C2). In their 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).

## 11.4 Instrumentation

Similar to CIPM, Kiciman and Livshits (2010) propose a platform (AjaxScope) for the instrumentation of JavaScript code 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 (Wert et al, 2015) provides adaptable instrumentation of the services of the 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-grained. The measurements are collected then from test or production environments.

## 11.5 Resource Demands

The related approaches estimate the resource demands either based on coarse-grained monitoring data (Spinner et al, 2015, 2014) or fine-grained data (Brosig et al, 2009; Willnecker et al, 2015). The latter approaches give a higher accuracy but have a downside effect because of the overhead of the instrumentation and the monitoring. Our approach reduces the overhead by the automatic adaptive instrumentation and monitoring. Similar to CIPM, Grohmann et al (2021) 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 estimated stochastic expression at Ops-time.

**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 Langhammer (2017, p. 35) | ✓$_{inc}$ | | | | | ✓$_{B}$ | ✓ | | |
| | Mbeddr Voelter et al (2012), Focus Ding and Medvidovic (2001) | ✓$_{inc}$ | | | | | | | | |
| | Consistency checker Jens and Daniel (2007), Buckley et al (2013),von Detten (2012) | ✓$_{inc}$ | | | | | | | | |
| | Extract Langhammer et al (2016) | ✓$_{B}$ | | | | ✓$_{B}$ | | ✓ | | |
| | ROMANTIC-RCA Hamdouni et al (2010) | ✓$_{B}$ | | | | | | | | |
| | SoMoX+Beagle Krogmann (2012) | ✓$_{B}$ | | | | | ✓$_{B}$ | ✓ | ✓ | |
| Ops-time | Brosig et al (2011) | ✓$_{B}$ | ✓$_{B}$ | ✓$_{B}$ | | ✓$_{B}$ | ✓$_{B}$ | ✓ | | |
| | PMXWalter et al (2017) | ✓$_{B}$ | ✓$_{B}$ | ✓$_{B}$ | | | ✓$_{B}$ | ✓ | | |
| | Brunnert et al (2013) | ✓$_{B}$ | ✓$_{B}$ | ✓$_{B}$ | ✓$_{B}$ | | ✓$_{B}$ | ✓ | | |
| | SLAstic van Hoorn (2014) | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{inc}$ | | | ✓ | ✓ | | |
| | iObserve Heinrich (2020) | ✓$_{inc}$ | | ✓$_{inc}$ | | ✓$_{inc}$ | | ✓ | | |
| Hybrid | EjbMoX Langhammer (2017, P. 140) | ✓$_{B}$ | | | | | | ✓ | | |
| | Konersmann (2018) | ✓$_{B}$ | | ✓ | | | | | | |
| | PRISMA Spinner et al (2019) | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{B}$ | ✓ | ✓ | |
| | CIPM | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{inc}$ | ✓$_{inc}$ | ✓ | ✓ | ✓ |

### 11.6 Parametric dependencies

In addition to the approach of Krogmann (2012) (SoMoX+Beagle), the works of Ackermann et al (2018) and Courtois and Woodside (2000) characterize the parametric dependencies. Grohmann et al (2019) also consider the characterization of parametric dependencies in performance models at Ops-time. Similarly, CIPM characterizes the parametric dependencies during updating and calibrating aPMs at Ops-time.

## 12 Conclusion

Considering the software architecture model increases the understandability as well as the productivity of software development (Olsson et al, 2017). 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 to update the aPM including the structure, abstract behavior and 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 cases: CoCoME (Heinrich et al, 2015) and TeaStore (von Kistowski et al, 2018). 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, which allows defining domain-specific metamodels and consistency preservation rules (Klare et al, 2021). The required overhead to adjust metamodels and CPRs for other domains in future works should be acceptable compared 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 (Werle et al, 2020). 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 to be applicable to LUA-based industrial applications by SICK[7]. Our adaption already covers parsing LUA source code, adjusting the CPRs to detect the SICK AppSpace apps as components and update of the models in VSUM. The initial findings (Burgey, 2023) indicate that the approach is applicable to real-world applications from the industrial sector. The future work is aimed on completing the prototype and resolving minor technical constraints.

## 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 German Research Foundation – project number 432576552, HE8596/1-1 (FluidTrust). This publication is also based on the research project SofDCar (19S21002) that the German Federal Ministry for Economic Affairs and Climate Action funds.

[7]See https://www.sick.com/gb/en/

# References


Ackermann V, Grohmann J, Eismann S, et al (2018) Black-box learning of parametric dependencies for performance models. In: Proceedings of 13th Workshop on Models@run.time (MRT), co-located with MODELS 2018

Armbruster M (2021) Commit-based continuous integration of performance models

Armbruster M (2022) Parsing and printing java 7-15 by extending an existing metamodel. Tech. rep., Karlsruhe Institut for Technology, https://doi.org/10.5445/IR/1000149186

Balsamo S, Di Marco A, Inverardi P, et al (2004) Model-based performance prediction in software development: a survey. IEEE Transactions on Software Engineering 30(5):295–310. https://doi.org/10.1109//TSE.2004.9

Becker S, Koziolek H, Reussner R (2009) The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82:3–22. https://doi.org/10.1016/j.jss.2008.03.066

Becker S, Hauck M, Trifu M, et al (2010) Reverse Engineering Component Models for Quality Predictions. In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track. IEEE, pp 199–202, URL http://sdqweb.ipd.kit.edu/publications/pdfs/becker2010a.pdf

Brosig F, Kounev S, Krogmann K (2009) Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In: Proceedings of the 1st Intl. Workshop on Run-time mOdels for Self-managing Systems and Applications (ROSSA 2009). In conjunction with the 4th Intl. Conference on Performance Evaluation Methodologies and Tools. ACM, New York NY, USA, pp 10:1–10:10

Brosig F, Huber N, Kounev S (2011) Automated extraction of architecture-level performance models of distributed component-based systems. In: Proceedings of the 2011 26th IEEE/ACM Intl. Conference on Automated Software Engineering. IEEE Computer Society, ASE '11, p 183–192, https://doi.org/10.1109/ASE.2011.6100052

Brunnert A, Vögele C, Krcmar H (2013) Automatic performance model generation for java enterprise edition (ee) applications. In: Computer Performance Engineering, Springer, pp 74–88

Buckley J, Mooney S, Rosik J, et al (2013) Jittac: A just-in-time tool for architectural consistency. 2013 35th Intl Conference on Software Engineering (ICSE) pp 1291–1294

Burgey L (2023) Continuous integration of performance models for lua-based sensor applications

Buschmann F (1998) Pattern-orientierte Software-Architektur: ein Pattern-System. Professionelle Softwareentwicklung, Addison-Wesley

Contributors to Jakarta RESTful Web Services (2020) Jakarta RESTful Web Services. URL https://jakarta.ee/specifications/restful-ws/3.0/jakarta-restful-ws-spec-3.0.pdf

Courtois M, Woodside M (2000) Using regression splines for software performance analysis. In: Proceedings of the 2nd Int. Workshop on Software and Performance

Derczynski L (2016) Complementarity, F-score, and NLP evaluation. In: Proceedings of the Tenth Intl. Conference on Language Resources and Evaluation (LREC'16). European Language Resources Association (ELRA), Portorož, Slovenia, pp 261–266, URL https://aclanthology.org/L16-1040

von Detten M (2012) Archimetrix: A tool for deficiency-aware software architecture reconstruction. In: WCRE 2012. IEEE, https://doi.org/10.1109/wcre.2012.61

Ding L, Medvidovic N (2001) Focus: a light-weight, incremental approach to software architecture recovery and evolution. In: Proceedings Working IEEE/IFIP Conference on Software Architecture, pp 191–200, https://doi.org/10.1109/WICSA.2001.948429

Dodge Y (2008) Kolmogorov–Smirnov Test, Springer New York, New York, NY, pp 283–287. https://doi.org/10.1007/978-0-387-32833-1_214

Eckey HF, Kosfeld R, Rengers M (2002) Multivariate Statistik. https://doi.org/10.1007/978-3-322-84476-7

Filho OFF, Ferreira MAGV (2009) Semantic Web Services: A RESTful Approach. In: IADIS Intl. Conference WWWInternet 2009. IADIS, pp 169–180

Goutte C, Gaussier E (2005) A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In: Losada DE, Fernández-Luna JM (eds) Advances in Information Retrieval. Springer, Berlin, Heidelberg, pp 345–359

Grohmann J, Eismann S, Elflein S, et al (2019) 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

Grohmann J, Eismann S, Bauer A, et al (2021) Sarde. ACM Transactions on Autonomous and Adaptive Systems 15(2):1–31. https://doi.org/10.1145/3463369

Hamdouni AEE, Seriai AD, Huchard M (2010) Component-based architecture recovery from object oriented systems via relational concept analysis. University of Sevilla, pp 259–270, URL https://hal-lirmm.ccsd.cnrs.fr/lirmm-00531804/

Heger C, van Hoorn A, Mann M, et al (2017) 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. ACM, New York, NY, USA, ICPE '17, pp 429–432, https://doi.org/10.1145/3030207.3053674

Heidenreich F, Johannes J, Seifert M, et al (2010) Closing the gap between modelling and java. In: van den Brand M, Gašević D, Gray J (eds) Software Language Engineering, Lecture Notes in Computer Science, vol 5969. Springer Berlin Heidelberg, p 374–383, https://doi.org/10.1007/978-3-642-12107-4_25

Heinrich R (2016) Architectural run-time models for performance and privacy analysis in dynamic cloud applications. ACM SIGMETRICS Performance Evaluation Review 43(4):13–22. https://doi.org/10.1145/2897356.2897359

Heinrich R (2020) Architectural runtime models for integrating runtime observations and component-based models. Journal of Systems and Software 169. https://doi.org/https://doi.org/10.1016/j.jss.2020.110722

Heinrich R, Gärtner S, Hesse T, et al (2015) 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. https://doi.org/10.1142/S0218194015710059

Heinrich R, Merkle P, Henss J, et al (2017) Integrating business process simulation and information system simulation for performance prediction. Software & Systems Modeling 16(1):257–277. https://doi.org/10.1007/s10270-015-0457-1

van Hoorn A (2014) Model-Driven Online Capacity Management for Component-Based Software Systems. No. 2014/6 in Kiel Computer Science, Department of Computer Science, Kiel University, dissertation, Faculty of Engineering, Kiel University

van Hoorn A, Waller J, Hasselbring W (2012) Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proceedings of the 3rd ACM/SPEC Intl. Conference on Performance Engineering. ACM, pp 247–248

Huyen C (2022) DESIGNING MACHINE LEARNING SYSTEMS: An iterative process for production-ready applications, first edition edn. O'REILLY MEDIA, INC, USA

IEEE 2675 (2021) IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment. https://doi.org/10.1109/IEEESTD.2021.9415476

Jakarta EE Platform Team (2019) Jakarta EE Platform. URL https://jakarta.ee/specifications/platform/8/platform-spec-8.pdf

Jakarta Servlet Team (2020) Jakarta Servlet Specification. URL https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.pdf

Jens K, Daniel P (2007) A comparison of static architecture compliance checking approaches. In: 2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07), p 12, https://doi.org/10.1109/WICSA.2007.1

Joy B, Steele G, Gosling J, et al (2000) The Java Language Specification, Third Edition. Addison-Wesley Reading

Jung R, Heinrich R, Schmieders E (2013) Model-driven instrumentation with kieker and palladio to forecast dynamic applications. In: Symposium on Software Performance, vol 1083. CEUR, pp 99–108, URL http://ceur-ws.org/Vol-1083/paper11.pdf

Keim J, Schulz S, Fuchss D, et al (2021) Tracelink recovery for software architecture documentation. In: Biffl S, Navarro E, Löwe W, et al (eds) Software Architecture. Springer Intl. Publishing, Cham, pp 101–116, https://doi.org/10.1007/978-3-030-86044-8_7

Kiciman E, Livshits B (2010) Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. ACM Trans Web 4(4). https://doi.org/10.1145/1841909.1841910

von Kistowski J, Eismann S, Schmitt N, et al (2018) Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In: 2018 IEEE 26th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, pp 223–236

Klare H, Kramer ME, Langhammer M, et al (2021) Enabling consistency in view-based system development – The Vitruvius approach. Journal of Systems and Software 171. https://doi.org/10.1016/j.jss.2020.110815

Klatt B (2014) Consolidation of customized product copies into software product lines. PhD thesis, Karlsruhe Institute of Technology (KIT)

Kolovos DS, Di Ruscio D, Pierantonio A, et al (2009) Different models for model matching: An analysis of approaches to support model differencing. In: 2009 ICSE Workshop on Comparison and Versioning of Software Models, pp 1–6, https://doi.org/10.1109/CVSM.2009.5071714

Konersmann M (2018) Explicitly integrated architecture - an approach for integrating software architecture model information with program code. PhD thesis, University of Duisburg, URL https://duepublico2.uni-due.de/receive/duepublico_mods_00045949

Konersmann M, Holschbach J (2016) Automatic synchronization of allocation models with running software. Softwaretechnik-Trends 36(4). URL https://mkonersmann.de/perm/publications/KonersmannHolschbach2016SSP.pdf

Koziolek H (2016) Modeling Quality. In: Modeling and simulating software architectures: the Palladio approach. MIT Press, Cambridge, Massachusetts

Kramer O (2017) Genetic Algorithms, Springer Intl. Publishing, Cham, pp 11–19. https://doi.org/10.1007/978-3-319-52156-5_2

Krogmann K (2012) Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis, The Karlsruhe Series on Software Design and Quality, vol 4. KIT Scientific Publishing, https://doi.org/10.5445/KSP/1000025617

Krogmann K, Kuperberg M, Reussner R (2010) Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. IEEE Transactions on Software Engineering 36(6):865–877. https://doi.org/http://doi.ieeecomputersociety.org/10.1109/TSE.2010.69

Langer P (2019) Emf compare. URL https://wiki.eclipse.org/EMF_Compare

Langhammer M (2017) Automated coevolution of source code and software architecture models. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, https://doi.org/10.5445/IR/1000069366

Langhammer M, Shahbazian A, Medvidovic N, et al (2016) Automated extraction of rich software models from limited system information. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE

Mazkatli M, Koziolek A (2018) Continuous integration of performance model. In: Companion of the 2018 ACM/SPEC Intl. Conference on Performance Engineering. ACM, ICPE '18, pp 153–158, https://doi.org/10.1145/3185768.3186285

Mazkatli M, Monschein D, Grohmann J, et al (2020) Incremental calibration of architectural performance models with parametric dependencies. In: IEEE International Conference on Software Architecture (ICSA 2020), Salvador, Brazil, pp 23–34, https://doi.org/10.1109/ICSA47634.2020.00011

Menasce DA, Almeida VA, Dowdy LW, et al (2004) Performance by design: computer capacity planning by example. Prentice Hall Professional

Meyer M (2014) Continuous integration and its tools. IEEE software 31(3):14–16

Monschein D (2020) Enabling consistency between software artefacts for software adaption and evolution

Monschein D, Mazkatli M, Heinrich R, et al (2021) Enabling consistency between software artefacts for software adaption and evolution. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA), pp 1–12, https://doi.org/10.1109/ICSA51549.2021.00009

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

Olsson T, Ericsson M, Wingkvist A (2017) Motivation and impact of modeling erosion using static architecture conformance checking. In: 2017 IEEE Intl. Conference on

Software Architecture Workshops. IEEE, https://doi.org/10.1109/ICSAW.2017.15

Reussner R, Goedicke M, Hasselbring W, et al (2019) Managed Software Evolution. Springer, Cham, https://doi.org/10.1007/978-3-030-13499-0

Reussner RH, Becker S, Happe J, et al (2016) Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press, Cambridge, MA

Santambrogio F (2015) Optimal Transport for Applied Mathematicians: Calculus of Variations, PDEs, and Modeling. Progress in Nonlinear Differential Equations and Their Applications, Springer Intl. Publishing

Sheskin DJ (2007) Handbook of Parametric and Nonparametric Statistical Procedures, 4th edn. Chapman and Hall/CRC

de Silva L, Balasubramaniam D (2012) Controlling software architecture erosion: A survey. Journal of Systems and Software 85. https://doi.org/10.1016/j.jss.2011.07.036

Smith CU, Williams LG (2003) Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA

van Solingen R, Basili V, Caldiera G, et al (2002) Goal question metric (gqm) approach. In: Marciniak JJ (ed) Encyclopedia of Software Engineering. John Wiley & Sons, New York, https://doi.org/10.1002/0471028959.sof142

Spinner S, Casale G, Zhu X, et al (2014) Librede: A library for resource demand estimation. In: Proceedings of the 5th ACM/SPEC Int. Conference on Performance Engineering. ACM, ICPE '14

Spinner S, Casale G, Brosig F, et al (2015) Evaluating approaches to resource demand estimation. Performance Evaluation 92

Spinner S, Grohmann J, Eismann S, et al (2019) Online model learning for self-aware computing infrastructures. Journal of Systems and Software 147:1–16

Szvetits M, Zdun U (2016) Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. Softw Syst Model 15(1):31–69. https://doi.org/10.1007/s10270-013-0394-9

Upton G, Cook I (2008) A Dictionary of Statistics. Oxford University Press, https://doi.org/10.1093/acref/9780199541454.001.0001

Vallée-Rai R, Co P, Gagnon E, et al (2010) Soot: A java bytecode optimization framework. In: CASCON First Decade High Impact Papers. IBM Corp., Riverton, NJ, USA, CASCON '10, pp 214–224, https://doi.org/10.1145/1925805.1925818

Voelter M, Ratiu D, Schaetz B, et al (2012) 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. Association for Computing Machinery, New York, NY, USA, SPLASH '12, p 121–140, https://doi.org/10.1145/2384716.2384767, URL https://doi.org/10.1145/2384716.2384767

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

Voneva S, Mazkatli M, Grohmann J, et al (2020) Optimizing parametric dependencies for incremental performance model extraction. In: Muccini H, Avgeriou P, Buhnova B, et al (eds) Software Architecture. Springer Intl. Publishing, Cham, pp 228–240

Walter J, Stier C, Koziolek H, et al (2017) An expandable extraction framework for architectural performance models. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. ACM, ICPE '17 Companion, pp 165–170, https://doi.org/10.1145/3053600.3053634

Werle D, Seifermann S, Koziolek A (2020) Data stream operations as first-class entities in component-based performance models. In: Proceedings of the 14th European Conference on Software Architecture, Lecture Notes in Computer Science, vol 12292. Springer, pp 148–164, https://doi.org/10.1007/978-3-030-58923-3_10

Wert A, Schulz H, Heger C (2015) Aim: Adaptable instrumentation and monitoring for automated software performance analysis. In: 2015 IEEE/ACM 10th Intl. Workshop on Automation of Software Test. IEEE, https://doi.org/10.1109/AST.2015.15

Willnecker F, Dlugi M, Brunnert A, et al (2015) Comparing the accuracy of resource demand measurement and estimation techniques. In: European Workshop on Performance Engineering, Springer

Wohlin C, Runeson P, Höst M, et al (2012) Experimentation in software engineering. Springer Science & Business Media

Woodside M, Franks G, Petriu DC (2007) The Future of Software Performance Engineering. In: Proceedings of ICSE 2007, Future of SE. IEEE Computer Society, Washington DC, USA, pp 171–187

Xu Y, Goodacre R (2018) 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. https://doi.org/10.1007/s41664-018-0068-2