Automated Software Engineering (2025) 32:54
https://doi.org/10.1007/s10515-025-00521-9

Continuous integration of architectural performance models with parametric dependencies – the CIPM approach

Manar Mazkatli[1] · David Monschein[2] · Martin Armbruster[1] · Robert Heinrich[3] · Anne Koziolek[1]

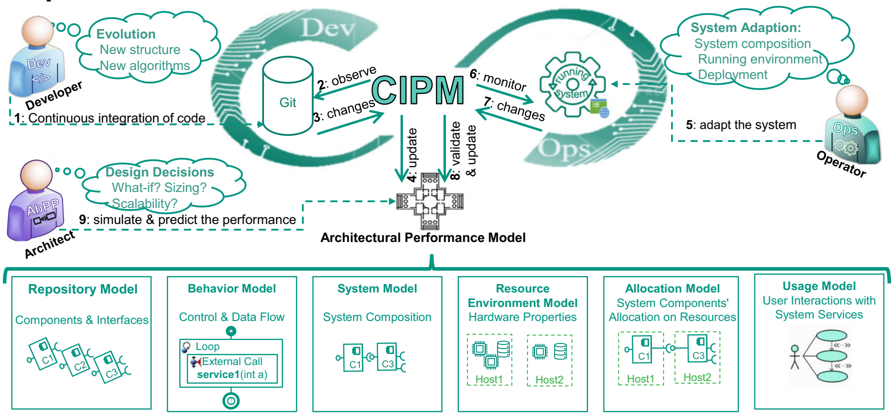
Received: 13 August 2024 / Accepted: 14 April 2025

Abstract

The explicit consideration of the software architecture supports system evolution and efficient quality assurance. 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 the performance (e.g., 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. Moreover, the extraction of a complete aPM after each impacting change causes unnecessary monitoring overhead and may overwrite previous manual adjustments. In this article, we present the Continuous Integration of architectural Performance Model (CIPM) approach, which automatically updates a parametric aPM after each evolutionary, adaptive, or usage change. To reduce the monitoring overhead, CIPM only calibrates the affected performance parameters (e.g., resource demand) using adaptive monitoring. Moreover, a self-validation process in CIPM validates the accuracy, manages the monitoring to reduce overhead, and recalibrates 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. We evaluate the applicability of CIPM in terms of accuracy, monitoring overhead, and scalability using six cases (four Java-based open source applications and two industrial Lua-based sensor applications). Regarding accuracy, we observed that CIPM correctly keeps an aPM up-to-date and estimates performance parameters well so that it supports accurate performance predictions. Regarding the monitoring overhead in our experiments, CIPM's adaptive instrumentation demonstrated a significant reduction in the number of required instrumentation probes, ranging from 12.6 % to 83.3 %, depending on the specific cases evaluated. Finally, we found out that CIPM's execution time is reasonable and scales well with an increasing number of model elements and monitoring data.

Extended author information available on the last page of the article

Graphical Abstract



Keywords Software architecture · Performance prediction · Model consistency · Parametric models · Self-validation · DevOps · Continuous integration · Consistency preservation

1 Introduction

Iterative software development approaches, such as agile methodologies, are commonly supported by Continuous Integration (CI) pipelines to ensure continuous integration and fast feedback during the development process. However, performance assurance during iterative software development faces several problems that we refer to by P. For example, the widely used application performance management (Heger et al. 2017) suffers from the required costs (P_{Cost}): Assessing the impact of design decisions on performance requires implementing them (Smith and Williams 2003) and setting up test environments and measurements for all design alternatives. As one effective way of helping solve P, software performance engineering (Woodside et al. 2007; Smith and Williams 2003) has strategic, cost-saving advantages to offer.

Software performance engineering reduce undue reliance on measurements in real environments by using models to predict the software performance and identify potential issues earlier (Balsamo et al. 2004). In particular, the approaches of architectural performance modeling, which model the system at the architecture level without implementation details, can build a good base for cost-effective Architecture-based Performance Predictions (AbPPs) of architectural design decisions (Reussner et al. 2016). Besides, the resulting architectural Performance Models (aPMs) increases the human understandability of the system and, consequently, the productivity of software development (Olsson et al. 2017).

Nevertheless, the application of AbPP in iterative development is a nontrivial task: Modeling is a time-consuming process (P_{Cost}), and developers do not trust models since they are approximations and difficult to validate (Woodside et al. 2007) ($P_{Inaccuracy}$).

Accurate and therefore trustworthy AbPP is challenging for various reasons. First, aPMs can be outdated due to frequent software changes, leading to inconsistencies between aPMs and the software system ($P_{Inconsistency}$). Here, changes of the source code at Development time (Dev-time) can affect the accuracy of aPMs. Similarly, adaptive changes at Operation time (Ops-time) (e.g., changes in the system composition and deployment) also affect aPMs. Second, the accuracy of the AbPP depends mainly on the estimated Performance Model Parameters (PMPs) (e.g., resource demand). These parameters can, in turn, depend on influencing factors, which may vary over the Ops-time (e.g., usage profile or execution environment). The parameterization of PMPs over these factors allows AbPP for unseen states (e.g., 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 become inaccurate as a system changes over time. Frequent re-estimations of all PMPs following every impacting change causes monitoring overhead ($P_{Monitoring-Overhead}$) because PMPs are calibrated mostly by dynamic analysis of the whole system (Spinner et al. 2015). Keeping the aPMs and their parameterized PMPs consistent with the running *and* continuously evolving system requires repeated manual effort (P_{Cost}). Hence, when there is an efficient maintenance of consistency between the *parameterized* aPMs and the related software system, not only does the comprehensiveness of the system's architecture improve but the performance management with AbPP also becomes proactive.

1.1 State of the art

In several approaches, it is suggested that the consistency maintenance between software artifacts is partially automated. These approaches can be divided into two categories.

Approaches of the one category reverse-engineer the current architecture based on the static analysis of the source code (El Hamdouni et al. 2010; Langhammer et al. 2016; Becker et al. 2010), the dynamic analysis (Brosig et al. 2011; van Hoorn 2014; Walter et al. 2017), or both (Konersmann 2018; Krogmann 2012). These approaches suffer from the following shortcomings. First, not all impacting changes at Dev-time or Ops-time are observed and addressed ($P_{Inconsistency}$). Second, the frequent extraction and calibration of aPMs may cause high monitoring overhead ($P_{Monitoring-Overhead}$). Third, possible manual modifications of the extracted aPMs would be discarded and should be repeated during the next extraction (P_{Cost}). Finally, fourth, since automatic validation is unavailable, the problem of uncertainty about aPM accuracy arises ($P_{Inaccuracy}$).

Approaches of the other category maintain the consistency incrementally, either at Dev-time (Langhammer 2017; Ding and Medvidovic 2001; von Detten 2012; Voelter et al. 2012; Jens and Daniel 2007; Buckley et al. 2013) based on consistency rules or at Ops-time (Heinrich 2020; Spinner et al. 2019; van Hoorn 2014) based on dynamic analysis. None of these approaches succeeds in updating aPMs according to evolution and to adaption ($P_{Inconsistency}$). The accuracy of the resulting aPMs and AbPP is uncertain and unreliable because no statement on the accuracy is provided ($P_{Inaccuracy}$). The same problem holds for approaches that estimate parametric dependencies to increase the accuracy of AbPP (e.g., Krogmann 2012; Grohmann et al. 2019).

1.2 Approach and contributions

In this article, we present the Continuous Integration of Performance Models (CIPM) approach. This approach maintains the consistency between an aPM and software artifacts (e.g., source code and measurements) (Mazkatli and Koziolek 2018). As shown in Fig. 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 uses adaptive monitoring to reduce the required overhead for updating and calibrating aPMs ($P_{Monitoring-Overhead}$) (Mazkatli et al. 2020; Voneva et al. 2020). Moreover, CIPM provides a statement on the accuracy of the AbPP and automatically recalibrates inaccurate parts ($P_{Inaccuracy}$) (Monschein et al. 2021). The upshot is, resulting aPMs allow accurate AbPP and also an improved comprehension of the system architecture, thereby enabling proactive actions for upcoming performance issues and assessment of design alternatives (P_{Cost}).

This paper is an extension of our work published in conference and workshop papers (Mazkatli et al. 2020; Monschein et al. 2021; Voneva et al. 2020). In this extension, we introduce a novel contribution to update aPMs and instrument source code changes based on version control commits. This innovative approach allows the seamless use of CIPM in modern Continuous Integration (CI) pipelines. This new version of CIPM is (to the best of our knowledge) the first approach for consistency preservation between source code and architecture models in a CI pipeline without the need for specialized source code annotations or specific frameworks/editors.

We make the following four contributions to an innovated CIPM approach:

1. *C1: Automated CI-based consistency maintenance at Dev-time.* A commit-based strategy (Section 5) is proposed to automatically update the models (e.g., aPMs) affected by the CI of the source code ($P_{Inconsistency}$). Unlike other methods, this approach uses standard version control commits as input, thus eliminating the need for specialized development editors to record source code changes and update aPMs accordingly. Our use of commit as input facilitates integration into CI pipelines while also minimizing the overhead that comes with keeping aPMs up-to-date with the latest code changes (P_{Cost}).

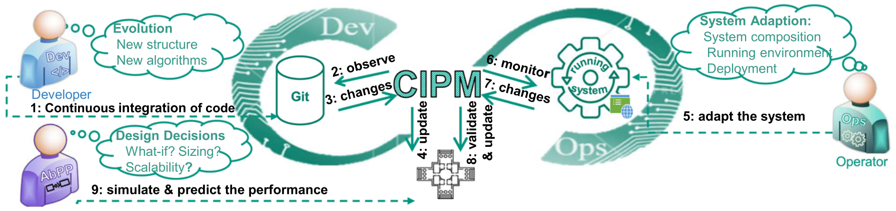


Fig. 1 Overview of the CIPM approach and the main actors

2. *C2: Automated adaptive instrumentation of source code changes.* We propose a CI-based, model-based instrumentation that targets the changed parts in the source code (Section 6). Unlike existing concepts, our instrumentation automatically detects where and how to instrument the source code to calibrate performance parameters. This reduces monitoring overhead ($P_{Monitoring-Overhead}$) and eliminates the need for costly, error-prone manual approaches (P_{Cost}).
3. *Comprehensive overview of entire approach and previously published contributions:* To help readers grasp the CIPM approach as well as all additional evaluations conducted on it, we present an overview of the approach in its entirety, including, as well, a concise description of previously published contributions:

   - C3: Incremental calibration. Our calibration of the PMPs is based on adaptive monitoring and uses statistical analysis to learn parametric dependencies (Mazkatli et al. 2020). If needed, our calibration optimizes the dependencies using a genetic algorithm (Voneva et al. 2020). Compared to existing approaches, our CIPM can calibrate PMPs at Ops-time, addressing $P_{Monitoring-Overhead}$ and $P_{Inaccuracy}$.
   - C4: Automated consistency maintenance at Ops-time. Our Ops-time calibration observes Ops-time changes based on dynamic analysis, in order to update the aPMs accordingly (Monschein et al. 2021) (Section 9). Unlike existing approaches, our CIPM automatically updates the aPMs using adaptive monitoring to include PMPs, system composition, and resource environment ($P_{Inconsistency}$,$P_{Monitoring-Overhead}$).
   - C5: Self-validation of updated aPMs. Our self-validation (Section 8) estimates the accuracy of AbPP against real measurements ($P_{Inaccuracy}$). It manages the adaptive monitoring to reduce the required overhead ($P_{Monitoring-Overhead}$) (Monschein et al. 2021). Our approach is (to the best of our knowledge) the first to enable self-validation of aPMs and dynamic management of monitoring overhead.
   - C6: Model-based DevOps pipeline. The proposed pipeline integrates and automates the CIPM activities during DevOps. In this paper, we refine the pipeline initially proposed in Mazkatli et al. (2020). Unlike existing pipelines, our pipeline maintains consistency during the whole DevOps life cycle, enabling AbPP.

4. *Additional evaluation for scalability of approach:* Besides evaluating the novel contributions (C1, C2) focusing on the updated models' accuracy and monitoring overhead, we also evaluate the scalability of the approach (Section 10.7).

This paper is organized as follows. We provide a background on the approaches used as a foundation in Section 2. We present a motivating example in Section 3 and give an overview of the CIPM approach in Section 4. Next, we describe the major contributions of the paper, namely, the automatic consistency preservation at Dev-time in Section 5 and the adaptive instrumentation in Section 6. In the sections that follow, we describe additional contributions pertaining to how our CIPM keeps the performance model consistent to the measurements at Ops-time through incremental calibration (Section 7), through self-validation (Section 8), and through maintenance by Ops-time calibration (Section 9). We present the evaluation results in Section 10 and discuss related research in Section 11. Finally, we summarize the article and discuss future work in Section 12.

2 Background

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

2.1 Software models

Models (Stachowiak 1973) abstract entities and relationships from the real world by capturing essential features and omitting unnecessary details. They provide structured representations of attributes and their relationships, and facilitate various operations and analyses for specific purposes.

Metamodels define the structure and relationships of different types of models within a specific domain, serving as higher-level abstractions for valid software models.

The Eclipse Modeling Framework provides a standardized framework for modeling, offering tools for creating, editing, manipulating, and transforming structured models (Eclipse Foundation 2024; Steinberg et al. 2009).

In CIPM's context, the Eclipse Modeling Framework is utilized for modeling software artifacts. This facilitates model-based transformations, such as propagating changes between models to resolve inconsistencies.

2.2 Palladio Component Model (PCM)

Palladio (Reussner et al. 2016) is a software architecture simulation approach that analyzes software at the model level for performance assessments (e.g., detecting bottlenecks or scalability problems).

The Palladio AbPP supports the proactive evaluation of design decisions to avoid high costs resulting from suboptimal decisions. The PCM consists of five sub-models, shown in Fig. 2. The `Repository Model` (A) contains a repository with components, interfaces, and services defined in the interfaces. It also describes which component provides or requires which interfaces. In addition, the `Repository Model` includes descriptions of the abstract behavior of services in the form of Service Effect Specifications (SEFFs) (A'). The `System Model` (B) describes the

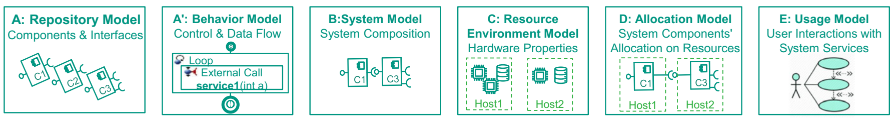


Fig. 2 Illustrations of PCM's five submodels. The `Repository Model` includes static structure and behavior (first two boxes)

composition of the software architecture based on the components and interfaces specified in the repository. The `Resource Environment Model` (C) reflects the actual hardware environment, which is composed of containers with 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` (D). Finally, the `Usage Model` (E) defines the behavior of users and how they interact with the system.

A SEFF describes the abstract behavior of a service in a component (Becker et al. 2009). It consists of ordered *actions* representing single process steps and focusing on the explicit modeling of interactions between components. There are different action types: *start actions*, *stop actions*, *external call actions* (calls to required services), *internal actions* (combines internal computations that do not include calls to required services), *loop actions*, and *branch actions*. Loop and branch actions contain at least one external call action to explicitly model control flow elements that influence external call actions. Remaining loops and branches in the source code are incorporated into internal actions for a higher level of abstraction. An illustrative example for a SEFF from our running example is shown in the left portion of Fig. 3.

To predict performance measures (response times, CPU utilization, and throughput), architects have to enrich SEFFs with PMPs. Examples of PMPs include resource demands (processing units that an internal action requests from specific active resources such as an CPU or hard disk), the probability of selecting a branch in a branch action, the number of loop iterations in a loop action, and the arguments of external call actions. Palladio employs stochastic expressions to define PMPs (Koziolek 2016) by means of random variables and empirical distributions. Additionally, a stochastic expression can express so-called *parametric dependencies*, which define how a given PMP depends on other parameters (e.g., input parameters of a service or

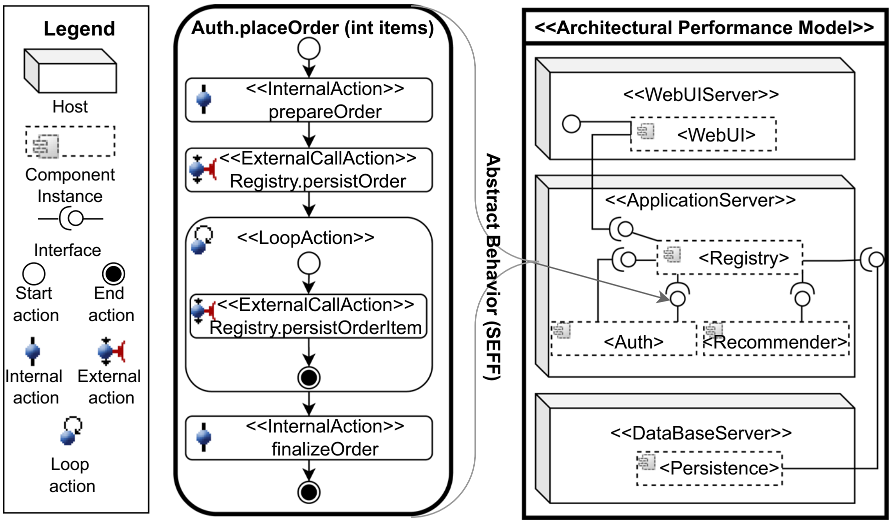


Fig. 3 Excerpt of TeaStore's architecture with notations from Palladio (Section 2.2)

on configuration parameters of a component). Dependencies cannot only be defined directly on parameter values, but also on other characterizations of parameter values, for instance, the number of elements in a collection or the size of a file.

The PCM is the preferred aPM for implementing CIPM due to its modular structure and parameterized PMPs that facilitate AbPP under varying conditions. This aligns with the goals of CIPM for the proactive identification of performance issues and evaluation of design alternatives. Moreover, CIPM reduces the overhead associated with PCM by automating both the modeling process and the updates of its six core models.

2.3 Vitruvius

The Vitruvius framework encapsulates heterogeneous models of a system and their semantic relationships in a so-called Virtual Single Underlying Model (VSUM) and keeps them consistent (Klare et al. 2021). Thus, the Reactions language of Vitruvius allows describing Consistency Preservation Rules (CPRs) at the metamodel level. CPRs, in turn, define the consistency preservation logic for each type of model change (i.e., which and how models must be changed to restore consistency after a change in a related model has occurred).

Vitruvius as a delta-based consistency approach (Diskin et al. 2011) propagates changes (i.e., deltas) in one model to derive changes for other models, updating models inductively and avoiding overwriting changes from other models (Wittler et al. 2023). For this purpose, Vitruvius stores a mapping between corresponding model elements in a correspondence model to reuse them during the consistency preservation.

Utilizing Vitruvius, the co-evolution approach (Langhammer 2017) keeps a Java source code model (Heidenreich et al. 2010) for Java 6 (Joy et al. 2000) and the PCM consistent. Therefore, the co-evolution approach employs specialized editors to record changes (in the form of deltas) that developers apply to the source code and propagate them to the PCM by executing CPRs. Langhammer defines CPRs that update the `Repository Model` of a PCM (i.e., the components and interfaces) and its behavior (SEFFs ***without*** PMPs by a full reconstruction) in response to the recorded changes in the source code. Changes in the PCM are also propagated to the code model.

In the context of CIPM, we utilize the Vitruvius platform to maintain consistency across software models, including PMPs, through incremental updates. In contrast to model-based batch transformations, these updates retain potential manual adjustments of the models and preserve architectural decisions of users.

2.4 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 a given architecture model, which is then applied to quality predictions. The PCM is 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 with Kieker (Jung et al. 2013) and applies necessary changes to the architecture model (PCM). 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 a so-called runtime architecture correspondence model.

In CIPM, we integrate iObserve's dynamic analysis to update the modeled usage and deployment parts based on monitoring data. CIPM updates remaining aspects, such as the system composition, performance parameters, and the resource environment, to ensure that the architectural performance model reflects the system's behavior.

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 evaluating 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 provides each microservice to all the other microservices. In that way, the Registry enables client-side load balancing. The communication between microservices is based on the widely used Representational State Transfer (REST) standard (Filho and Ferreira 2009). In Fig. 3, on the right-hand side, there is a picture of the `System Model` and `Allocation Model` of the TeaStore, as an illustration of the connections between its microservices. The left-hand side depicts the abstract behavior of the `placeOrder` service. This service begins with an internal action to prepare an order and calls an external service from the Registry to persist the order. Next, in a loop, calls to an external service persist each item of the order. The loop iterates until all items are persisted. Lastly, to conclude, an internal action finalizes the order.

TeaStore includes four different Recommender implementations that suggest related products to users. The developers implemented these versions along different development iterations. The four implementations have different performance characteristics. Performance tests or monitoring can be employed to discover these characteristics for the current state (i.e., current implementation, current deployment, current environment, current system composition, and current workload). However, predicting the performance for another state is challenging and expensive because it requires setting up and performing several tests for each implementation alternative. In our example, it is a challenge to answer the following questions based solely on application performance management (Heger et al. 2017): "Which implementation would perform better if the load or deployment were changed?" or "How well does the Recommender perform during unseen workload scenarios?" To illustrate this last question, consider an upcoming discount offer. The architects expect an increased number of customers and also a change in customer behavior because now each customer is expected to order more items.

Other challenging questions can be: “What is the current system composition?” or “How would the performance be if the system composition changes?” Changes in the landscape of the Teastore are common at Ops-time due to the load balancing required to conduct replications and de-replications at a minimum of effort. Consequently, it is inevitable that the associated architecture model will require constant updates if it should remain consistent with the system. Only an up-to-date architecture model can answer questions correctly about performance, scalability, and other quality characteristics. This is why our approach aims to provide an accurate architecture model at every and any point in time, while also keeping manual effort and monitoring overhead as low as possible.

4 Continuous integration of architectural performance models

To streamline delivery in iterative software development processes (e.g., agile or DevOps), developers rely on automated builds, test automation, CI, and continuous deployment. Here, CIPM reduces the need for manual modeling or updating by automatically integrating aPMs into the delivery pipeline and also keeping aPMs consistent with the evolving system. In CIPM, an up-to-date aPM is available at any time for AbPP at low costs as detailed in Mazkatli (2025).

This section provides an overview of CIPM by describing the models CIPM automatically updates (Section 4.1) and the integration of CIPM into the DevOps pipeline (Section 4.2).

4.1 Models to keep consistent

Performance predictions with architecture-based methods require modeling the software architecture from all perspectives of the aPMs, namely the perspectives of the static structure and behavior, the resource environment, the related resource demands, the workload, and the usage. The accuracy of the performance prediction depends on the accuracy of the aPM updated by CIPM.

In our CIPM approach, the PCM is used for AbPP because it simplifies modeling all the aforementioned perspectives of aPMs using the (A) `Repository`, (B) `System`, (C) `Resource Environment`, (D) `Allocation`, and (E) `Usage Model` (Section 2.2).

To update the `Repository Model` (A), CIPM proposes a CI-based consistency preservation. The CI-based consistency preservation of CIPM extracts changes from source code commits and applies pre-defined change-based consistency preservation rules to restore consistency between source code and the repository model (i.e., static structure and behavior models as SEFFs) (Section 5). To estimate the PMPs of the `Repository Model` and collect the required data while the application runs, CIPM uses, on the one hand, adaptive instrumentation that applies fine-grained instrumentation to the changed sections of the source code, and on the other, adaptive monitoring that reduces the monitoring overhead through deactivation after calibration (Section 6). The `Repository Model` is calibrated at Dev-time using test data

(Section 7) and refined at Ops-time using the monitoring data from the production environment (section 9). The incremental calibration (Section 7) estimates the PMPs (e.g., the resource demand) by considering parametric dependencies. If necessary, the adaptive optimization of PMPs can be activated to estimate complex dependencies (Voneva et al. 2020). All these processes here keep the `Repository Model` (A) up-to-date at both Dev-time and Ops-time.

Regarding the `System Model` (B), we provide both a semi-automatic extraction at Dev-time based on static analysis of source code (Section 5.5) and also automatic updates at Ops-time based on dynamic analysis of monitoring data (Section 9). Further, CIPM updates the `Resource Environment Model` (C) based on the dynamic analysis, and to update the `Allocation Model` (D) and `Usage Model` (E), CIPM integrates the dynamic analyses of the iObserve approach (Heinrich 2020) (Section 9).

Overall, CIPM continuously updates the aPM parts (A) - (E) to keep them consistent with the running system.

CIPM employs a high degree of automation, which is designed to leverage an up-to-date aPM. Nonetheless, it is possible that the following minimal manual interventions may be required in specific cases. Figure 1 shows the architects who may need to confirm component candidates during updates to the `Repository Model` (A), resolve conflicts during updating the `System Model` at Dev-time (B), or provide details about the available resource environment to the `Resource Environment Model` (C). These manual steps are straightforward and limited in scope, since the more complex tasks like performance model updates and parameter calibration are fully automated.

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 2021). To achieve a seamless workflow, we integrate and automate the CIPM processes into the DevOps pipeline, creating a Model-based DevOps (MbDevOps) pipeline (C6 in Fig. 1). Specifically, this integration ensures that the aPM is continuously updated as part of the automated MbDevOps, enabling AbPP during DevOps-oriented development while addressing $P_{Inconsistency}$ and $P_{Inaccuracy}$.

Figure 4 shows the MbDevOps pipeline where it starts on the development side with the 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 a source code model in the VSUM of Vitruvius (1.1) (Section 5.2). Next, in response to those changes in the source code, predefined CPRs in Vitruvius update the `Repository Model` (1.2) (Section 5.3). Similarly, CPRs update the Instrumentation Model (IM) (1.3) with new probes corresponding to recently updated parts of the `Repository Model` for subsequent calibration (Section 5.4). This first process also semi-automatically extracts the `System Model` (1.4) (Section 5.5). The second process, the *adaptive instrumentation (2)*, instruments the changed parts of the source code based on the instrumentation probes in the IM and also based on

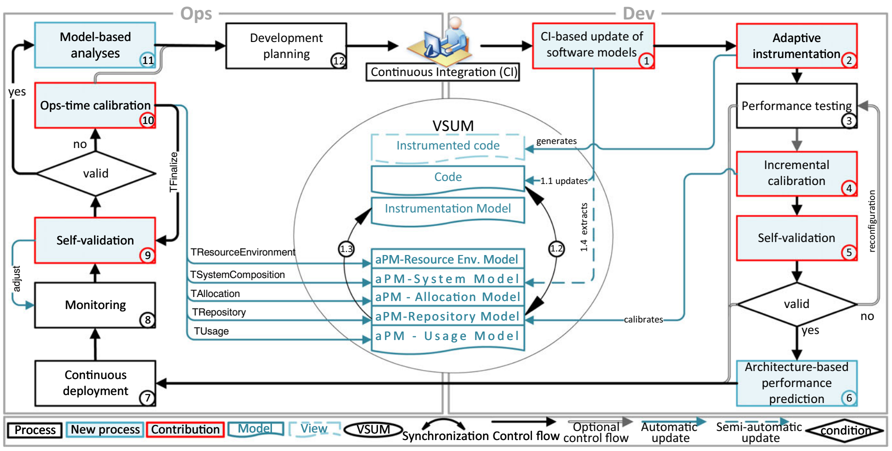


Fig. 4 Model-based DevOps pipeline, from Mazkatli (2025)

the source code model (Section 6). The third process is the *performance testing (3)* with the instrumented source code. The *performance testing (3)* generates the necessary measurements for calibration. For purposes of cross-validation, the pipeline divides these measurements into two sets, a training set comprising 80 % of the measurements and a validation set comprising 20 % (Xu and Goodacre 2018). The fourth process, the *incremental calibration (4)*, employs the training set to estimate the PMPs with parametric dependencies, which are then used to enrich `Repository Model` (Section 7). After that calibration process, the pipeline starts the fifth process, the *self-validation (5)*. Here, the calibration accuracy is evaluated using the validation set (Section 8). If the resulting aPM is deemed accurate, developers can use it to answer performance questions using the sixth process, the *AbPP (6)*. However, if not, developers can either change the test configuration to recalibrate the aPM again or wait for the Ops-time calibration. Answering what-if performance questions using AbPP in this way, as opposed to intensive performance tests, reduces the effort and costs of performing prediction before the operation.

The operation side of Fig. 4 starts at the *continuous deployment (CD) (7)* in the production environment, followed by the *monitoring (8)*. In DevOps, monitoring is a standard practice. Accordingly, to ensure the quality of monitoring and updates in our CIPM pipeline, it is beneficial to use a separate monitoring resource, distinct from the one used by the pipeline itself. Our *Monitoring (8)* in the production environment generates the required runtime measurements in a customizable time interval. These measurements are grouped and sent to the subsequent processes: *self-validation (9)* and *Ops-time calibration (10)*. The *self-validation (9)* is essential to improve the accuracy of the aPM and reduce the monitoring overhead (Section 8). The *self-validation (9)* compares the monitoring data and the monitored simulation results to validate the estimated aPM. The result of self-validation is used as input to the Ops-time calibration and also as a mechanism for managing the monitoring overhead. If the aPM is not accurate enough, the *Ops-time calibration (10)* recalibrates inaccurate parts based on the feedback of the self-validation (e.g., by updating PMPs of the `Repository Model`, `Resource Environment Model`, `System Model`, `Allocation Model`, or `Usage Model` (Section 9)). The self-validation also deactivates the fine-grained monitoring of accurate parts to reduce monitoring overhead. The *self-validation (9)* and *Ops-time calibration (10)* are (as explained above) triggered after a source code commit. However, the two processes are also triggered frequently according to a customizable trigger time to respond to possible Ops-time changes and improve the aPM accuracy with new monitoring data. The resulting model can be used to perform *model-based analyses (11)* (e.g., model-based auto-scaling). Up-to-date descriptive aPMs can support the *development planning (12)* by increasing the understandability of the current version, by modeling and evaluating design alternatives, and by answering what-if questions. More details on our CIPM approach and the MbDevOps pipeline are found in Mazkatli (2025).

Sections 5 through 9 describe the new processes marked as contributions in Fig. 4. Sections 5 and 6 highlight our primary contributions C1 and C2. Sections 7 through 9 delve into previously published contributions (C3, C4, and C5) to offer a comprehensive overview of the CIPM concept.

5 CI-based update of software models

Section 5.1 describes the concept of C1 that *automatically* maintains the *consistency* between an aPM and software models at Dev-time addressing P_{Cost} and $P_{Inconsistency}$. This includes the extraction of the initial models based on the considered initial commit of the CI and the automatic updates in response to subsequent commits.

The realization of C1's concept updates the following four models in response to the CI of source code by (A) statically analyzing the source code and (B) executing predefined CPRs within the VITRUVIUS platform.

- The Source Code Model in the VSUM of VITRUVIUS based on (A) (Section 5.2).
- The `Repository Model` in the VSUM of VITRUVIUS based on (B) (Section 5.3).
- The Instrumentation Model in the VSUM of VITRUVIUS based on (B) (Section 5.4).
- The `System Model` based on (A) (Section 5.5).

5.1 Concept of the CI-based update

In the following, we describe the concept behind C1 to maintain the consistency between software models after source code changes. The process steps are depicted in Fig. 5.

As mentioned in Section 2.3, VITRUVIUS itself is a delta-based consistency approach, i.e., it requires a sequence of changes (deltas) as input. Thus, so far, its direct use required specialized editors to record and output a sequence of changes. However, popular available code editors cannot produce such a change sequence. Therefore, CIPM extracts the source code changes from version control systems to

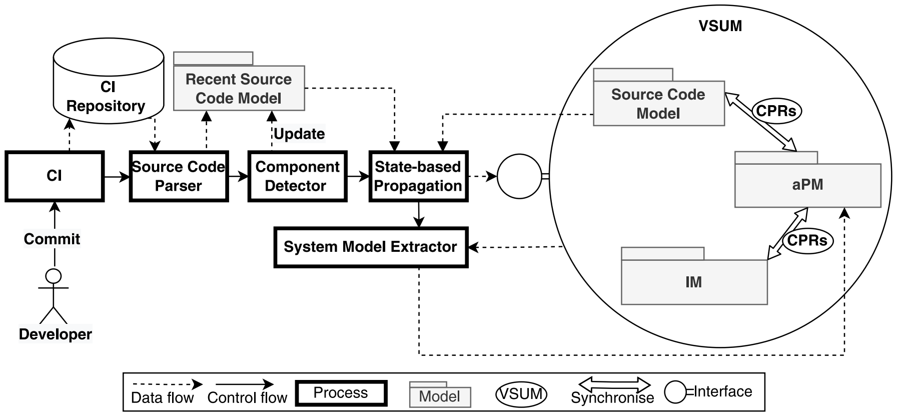


Fig. 5 CI-based update of the aPM

allow developers to use their own external and preferred tools for development and version management.

To the best of our knowledge, our approach is the first one to bridge this gap between state-based version control in source code repositories and delta-based multi-view consistency preservation, increasing the applicability of model-based consistency preservation for source code. In addition to the overall process detailed below, the conceptual novelties are specifically the *parsing*, *component detection*, *CPRs for the aPM*, and the *CPRs for the IM*.

Parsing

At first, developers commit their (source code) changes to a version control system's repository, which triggers the CI pipeline of CIPM. Next, a parser reads all source code files with the latest changes from the repository. It generates a source code model containing the developers' recent changes.

Usually, there are multiple source code files. If one source code model was created for every file, it would be challenging to extract the changes. For instance, changes in two models can have cyclic dependencies to each other so that the two models need to be considered at the same time. As a result and novelty, our approach consolidates the complete source code into one single source code model that allows to integrate it into the VITRUVIUS platform.

Component detection

The parsing process may not provide additional information from further sources (e.g., from file structures or configuration files), especially when a source code parser is applied. Such information is crucial for the model-based detection of components within VITRUVIUS because such additional information can support or enable the component detection. To tackle this challenge, we introduce a novel pre-processing step after parsing the recent commit. This component detector enhances the source code model by incorporating information about the components based on different factors, for example, the file structure of the source code or configuration files.

Change extraction

For the extraction of a change sequence in CIPM, the source code model extended in the last step is compared state-based to the source code model in VITRUVIUS's VSUM which corresponds to a previous commit. The state-based comparison consists of these extendable phases (Brun and Pierantonio 2008): *calculation* (divided into *matching* and *differencing*) and *representation*. In the *matching* phase, elements from both models are compared to find related elements. Afterward, the *differencing* phase calculates the changes between related model elements. These changes are *represented* as VITRUVIUS changes.

VITRUVIUS does not specify how the matching is performed (Klare et al. 2021). Therefore, we provide custom language-specific matching algorithms (Kolovos et al. 2009a) to VITRUVIUS. These algorithms consider the model structure and model element types, representing a novel advancement in accurately matching source code elements within VITRUVIUS.

Change propagation

The changes obtained from the state-based comparison consist of atomic edit operations on which the CPRs operate. Additionally, they describe how the source code model in the VSUM can be transformed into the recent source code model. Thus, the changes are sorted so that the creation of elements happens before references to the elements are added, and VITRUVIUS applies the sorted changes. Consequently, the change sequence is utilized to update the source code model in the VSUM of VITRUVIUS. This triggers the CPRs for the aPM and IM.

CPR for the aPM

The CPRs for the aPM focus on (1) the technology-specific detection of components and interfaces, (2) addition, update and deletion of elements, and (3) behavior reconstruction so that the aPM can represent the software architecture as close as possible imagined by the developers.

Regarding the detection of components and interfaces (1), our approach employs the information from the source code model (potentially enhanced by the component detector) to find the components and their interfaces. As these CPRs are only intended to detect new elements in the source code, further CPRs add them and other elements to the aPM (2). Within these CPRs, the components are connected with the interfaces they provide and require. Here, a challenge in preserving the consistency between the source code and aPM is the order of aPM changes: components and interfaces, for example, need to be created first before they are connected. In general, the structural parts are followed by the behavioral parts. Addressing this challenge, our CPRs ensure this order. To the best of our knowledge, this ordering of changes for the aPM is novel.

Moreover, there are CPRs which update or delete elements in the aPM (2) if the corresponding elements in the source code change. Finally, behavioral parts in the aPM are reconstructed (3). In addition, we maintain the mapping between changed source code statements and their related behavioral parts with VITRUVIUS.

CPR for the IM

Based on the behavior reconstruction, we propose novel CPRs that update the IM to capture all behavioral parts that have changed in the recent commit. As the CPRs react to the changes in the aPM, they are independent of the source code model. Again, we employ VITRUVIUS to establish a mapping between the behavioral parts and probes in the IM. This mapping in combination with the mapping between source code statements and behavioral parts enables the consistency preservation between an aPM and measurements generated by instrumented code.

The information from the IM is later employed in the adaptive instrumentation (Section 6). Additionally, software architects and the self-validation process can add deactivated probes to the IM. The self-validation can activate these new probes if their related PMPs are not accurate enough.

System model extraction

At last, the `System Model` is extracted, capturing the system's composition by detailing how the components in the `Repository Model` are instantiated and assembled. The `System Model` is essential for enabling AbPP to guide design decisions at Dev-time. The `System Model` extraction at Dev-time is semi-automated (Monschein et al. 2021), saving time and effort compared to manual creation.

First, we generate a unified structure called Service-Call-Graph (SCG) based on the models in the VSUM. The SCG describes "calls-to" connections between services, including associated resource containers on which the respective services are executed. It is a directed graph where a node consists of a service and a resource container. An edge indicates that one service on a particular container calls another service on another container. By introducing the SCG, we employ a data structure that is independent of the underlying programming languages and applicable in different situations.

Second, we extract the `System Model` based on the SCG. It starts by modeling the boundary interfaces that the system provides. The user (architect or developer) determines these interfaces. Then, the `Repository Model` is searched for components that provide the selected interfaces. If more than one component provides the same interface, the user is asked to choose the correct one. For a valid `System Model`, the required interfaces originating from the provided components need to be satisfied. Therefore, the SCG is traversed to detect all services called by the services of the provided interfaces. As before, the components providing the called services are detected based on the `Repository Model` so that the required interfaces can be connected with the related provided interfaces. Recursively, each required interface is satisfied until none is left.

5.2 Realization of the source code model update

CIPM employs source code models which represent the source code one-to-one so that they can be integrated into VITRUVIUS. Currently, we provide metamodels and support for the programming languages Java and Lua (Mazkatli et al. 2023). We focus on the Java model as an example because the TeaStore is implemented in Java (TeaStore-Git 2023). The following excerpts and examples are simplified.

Java programming language

For Java models, we build upon the existing Java Model Parser and Printer (JaMoPP) which provides a Java metamodel (Heidenreich et al. 2010). We extended this metamodel with new features to enable the support for the Java versions 7-15, including lambda expressions, modules, and others (Armbruster 2022). Moreover, we implemented a new parser to generate Java models from source code and a new printer to output the models as code again. More details on our JaMoPP extensions can be found in Armbruster (2022)[1].

Java source code contains references between different elements (e.g., a class references the interface it implements), which also extend into the source code's dependencies. References can be reflected in the Java models. In order to generate them, CIPM provides two options. When applying the first option, the source code is compiled before it is parsed. This allows us to obtain and include the dependencies. Alternatively, for cases where compiling the source code is not feasible, the second option uses a recovery strategy, which creates model elements for references to missing dependencies (Armbruster et al. 2023).

Figure 6 displays an excerpt from the Java metamodel. It contains elements to represent Java classes and interfaces as specializations of Java types. Types in turn include their declared members, for instance, methods or fields.

We illustrate the Java models with parts of the *Recommender* microservice code from the TeaStore (TeaStore-Git 2023). Listing 1 shows an excerpt of the code: The `IRecommender` interface allows to `train` a recommender and to `recommendProducts`. The `AbstractRecommender` serves as a default implementation, which delegates the actual recommendations to the `execute` method. For the presented code, Figure 7 displays an excerpt from the Java model conforming to the metamodel excerpt shown in Fig. 6.

To achieve the update of the source code model in the VSUM, we perform a state-based comparison of the Java model generated by the parser and the Java model in the VSUM. The first (i.e., *matching*) phase aims to find and relate all elements from both models which are considered equal. Because the default similarity-based matching algorithm (Kolovos et al. 2009a) in VITRUVIUS's implementation resulted in insufficient matches (e.g., elements without a match although there is an equal element, or matches in which the elements are not equal), we decided to apply a custom language-specific matching algorithm. For Java, we reuse an existing hierarchical Java-specific matching algorithm (Klatt 2014) that we extended to be compatible with Java

[1] The source code is available on https://github.com/MDSD-Tools/TheExtendedJavaModelParserAndPrinter.

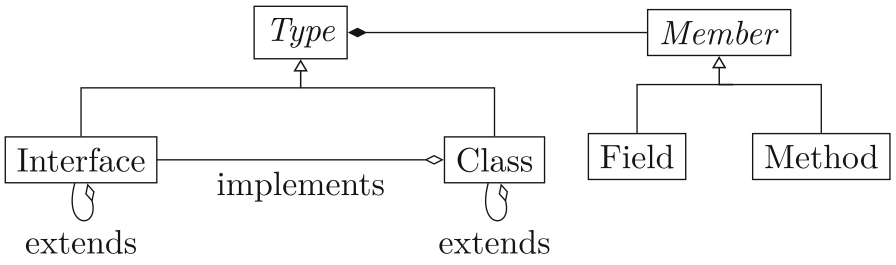


Fig. 6 Excerpt from the Java metamodel highlighting parts to represent interfaces, classes, and their members

7-15. This matching algorithm considers the specific properties and structures of the Java language to provide an accurate matching. For example, two classes are only equal if they are located in the same package and have the same name. After the elements have been matched, the differences and change sequence for VITRUVIUS are calculated.

```
public interface IRecommender {
    public void train(List orderItems, List orders);
    public List recommendProducts(Long userid, List
        currentItems); }

public abstract class AbstractRecommender implements
     IRecommender {
    public void train(List orderItems, List orders) {}
    public List recommendProducts(Long userid, List
        currentItems) {}
    protected abstract List execute(Long userid, List
        currentItems);
}
```

Listing 1 Excerpt from the Recommender code.

Lua programming language

For Lua, we extended an existing Lua grammar from which we obtain a metamodel, parser, and printer (Mazkatli et al. 2023). Additionally, we defined and implemented a custom language-specific matching algorithm for Lua-based applications.

5.3 Realization of the repository model update

In this process, CPRs update the `Repository Model` according to the changes on the source code model. We base the CPRs for Java on the CPRs defined by the Co-evolution approach (Langhammer 2017) that we extended for our applications. The extension is mainly related to the (1) technology-specific detection of components and interfaces, (2) addition, update, and deletion of elements, and (3) behavior reconstruction in the form of a SEFF reconstruction.

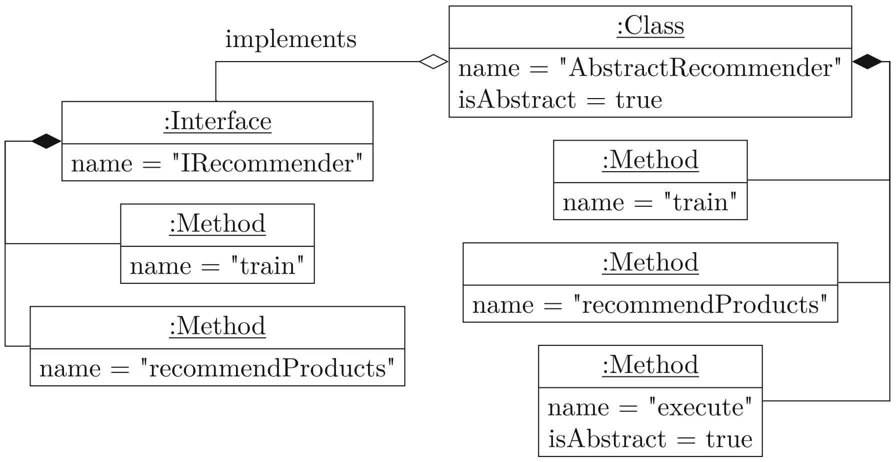


Fig. 7 Excerpt from the Recommender Java model

For the detection of components and interfaces (1), we consider technologies for the implementation of microservice-based applications. As a result, the component detector aims to find all microservices in order to generate a component for each microservice. Due to its access to the source code and configuration files, the component detector looks for sets of classes, which are complemented by a Docker and build file (e.g., a `pom.xml` file for Maven). In this case, it identifies a set of classes as a microservice since the classes are built into one artifact and deployed as one. Parts of the `Repository Model` affected by the discovered microservice can be extracted fully automatically.

In addition, CIPM also supports cases where information is missing and consistency can only be restored semi-automatically. If there is only a build file without a Docker file for a set of classes, 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. 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 stored and reused in subsequent executions of the CPRs to eliminate the need to ask the developer again.

While the CPRs focus on microservices, we allow the discovery of other notions of components. In our current implementation, if a set of classes resides in a specific pre-defined package, it is considered as a regular component.

After all components have been identified, they need to be encoded in the Java models so that the CPRs within VITRUVIUS can create the components in the `Repository Model`. As a consequence, a Java module is created for each component. Then, our CPRs map each Java module to a corresponding component in the `Repository Model` and, consequentially, generate a component for each new module as illustrated in Algorithm 1.

Moreover, we defined CPRs that detect the interfaces of microservices. Concretely, we implemented CPRs for two technologies as a first step because these technologies are used in our evaluation. One of the technologies is the Jakarta RESTful Web Services. In its context, Java classes that are annotated with `@Path` constitute an API (Contributors to Jakarta RESTful Web Services 2020) so that an interface in the `Repository Model` is created for such an annotated class. Regarding regular components, we use their public Java classes and interfaces to model the interfaces in the `Repository Model`. The pseudocode in Algorithm 2 represents the CPRs for the interface detection.

Algorithm 1 Pseudocode for the CPR updating the `Repository Model` after adding a new module.

```
Require: New module added to Java model
1: function UPATECOMPONENTS(module)
2:     component ← createComponent(module)
3:     component.name ← module.name
4:     addToRepositoryModel(component)
5: end function
```

Algorithm 2 Pseudocode for the CPR updating the `Repository Model` after adding a new class.

```
Require: New class added to Java model
1: function UPDATEINTERFACES(class)
2:     if class.isAnnotatedWith(Path) or
3:             (class.isPublic() and componentOf(class).isRegular()) then
4:         interface ← createInterface(class)
5:         interface.name ← class.name
6:         addToRepositoryModel(interface)
7:     end if
8: end function
```

For further illustration, we extend the simplified example from the Recommender microservice in Section 5.2. As depicted in Fig. 8, the Java model M_{J1} contains at least the *IRecommender* interface for which the pre-processing step detects the Recommender microservice because it has a Docker and `pom.xml` file (TeaStore-Git 2023). Thus, a new module in the Java model is generated. The state-based comparison of the Java models with and without the Recommender module results in the ordered changes C_1. These changes transform the Java model M_{J1} into M_{J2} with the Recommender module and include the creation of the module, adding it to the Java model, setting its name, and adding a reference to the *IRecommender* interface. The second change (addition of the module to the model) triggers the CPR described in Algorithm 1. Its execution induces the changes C_2 to keep the `Repository Model` consistent with the changes C_1. Per definition, the changes in C_2 create a component, assign the module name as the component name, and add it to the `Repository Model`. As a consequence, the previous `Repository Model` M_{P1} without a Recommender component is enhanced with such a component into the `Repository Model` M_{P2}.

While the addition of elements is mostly covered by the previously described CPRs, we also implemented CPRs for the update and deletion of elements (2). On one hand, if an element in the source code model is removed, corresponding elements in the `Repository Model` are also removed. For example, if a set of classes relating to a component is removed, the component and its interfaces are deleted. On the other

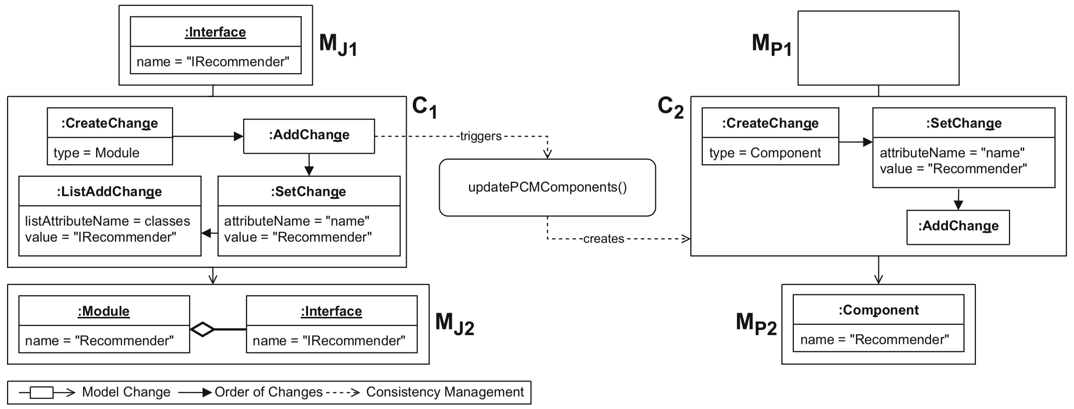


Fig. 8 Simplified example for the execution of the CPR which adds a component in the `Repository Model` for a Java module

hand, CPRs for changes consider, for instance, renames (and rename corresponding elements) or changes in methods with a corresponding SEFF to update it.

Regarding the SEFF reconstruction (3), CPRs respond to changes in method bodies (e.g., adding or changing statements) by reconstructing the SEFF with an existing reverse engineering tool (Krogmann 2012). Our extension to this tool extracts the mapping between source code statements and their related SEFF actions and stores them in the VITRUVIUS correspondence model. The execution of the CPRs also keep these mappings between the source code and the aPM up-to-date, which is necessary for the consistency preservation process in VITRUVIUS on one hand. On the other hand, we use the mapping for the following two processes: the Dev-time `System Model` extraction (Section 5.5) and the adaptive instrumentation (Section 6).

5.4 Realization of the instrumentation model update

For this step, we defined CPRs that react to changes in a SEFF. In these cases, the CPRs generate or update probes in the IM for every SEFF action that is newly added or whose corresponding source code statements have changed in the recent commit. For deleted SEFF actions, the probes in the IM are also removed.

5.5 Realization of the system model extraction

In Section 5.5.1, we describe the SCG extraction at Dev-time exemplified for Java. Subsequently, the example is extended to a more detailed `System Model` extraction from the SCG in Section 5.5.2.

5.5.1 SCG extraction at dev-time

The extraction of the SCG begins with a code analysis that indicates the invocation dependencies between Java methods (Vallée-Rai et al. 2010) and builds a method call graph. Utilizing the mapping between the source code model and the `Repository Model` stored in VITRUVIUS (Section 5.3), the resulting method call graph is transformed into an SCG including the service calls and their associated components. Figure 9 shows a truncated version of the SCG from our running TeaStore example.

It is important to note that uncertainties may arise during Dev-time. In particular, in scenarios involving inheritance or conditions, the determination of specific call paths becomes ambiguous. To address this issue, the SCG extraction at Dev-time considers all potential execution semantics, leading to *conflicts*. This happens, for example, if there are multiple components that provide a certain interface and it is uncertain at Dev-time which component is actually used at Ops-time (e.g., strategy pattern). Furthermore, during the development phase, the distribution of services and components on resource containers in the production environment remains unknown. Thus, this information is excluded. The conflicts require user intervention for resolution, as discussed in Section 5.5.2. This manual intervention is simplified by presenting all possible options for resolving a conflict to the user, which makes it easy to identify

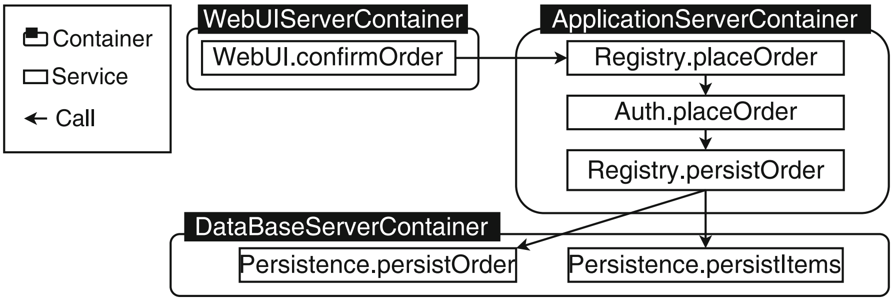


Fig. 9 SCG extracted from an excerpt of the TeaStore

potential consequences. For example, if there are several components that provide a particular interface, the user is given a description of the conflict and can choose from a list of possible components. Once the desired resolution is selected, it is automatically applied, eliminating the need for the user to perform the underlying modeling.

5.5.2 Exemplary system model extraction from an SCG

As outlined in Section 5.1, the `System Model` extraction starts with the selection of the system's interfaces. From our running example, we choose the `CartActions` interface, which exposes services for purchasing products and managing orders. Then, the `System Model` extraction searches the `Repository Model` for components providing the interface. In our example, the `WebUI` component provides the `CartActions` interface, and no conflict occurs because it is the only component providing the interface. After finding the components, instances of the components called *assembly contexts* are created and linked to the provided interfaces. Therefore, an instance of the `WebUI` component is created and added to the `System Model` as shown in Fig. 10.

For the completion of the `System Model`, the required interfaces of the added assembly contexts must be satisfied. Considering the SCG in Fig. 9, the service `confirmOrder` calls `placeOrder`. Here, the `Registry` component is detected based on the `Repository Model`. If more than one component had provided the same required interface, the user would have needed to resolve this *conflict*. This is not the case for the `Registry` component. Next, an assembly context for the component is created and added, as there is none available. Otherwise, the user must resolve another *conflict* by deciding whether to use the available assembly context or to add a new one. Afterward, the extraction continues to satisfy the required interfaces of the recently added assembly contexts. As a consequence, the required interfaces of the `Registry` are satisfied by adding instances for the components providing them (i.e., `Auth` and `Persistence`). Subsequently, the required interface of `Auth` should be satisfied since `Auth.placeOrder` calls `Registry.persistOrder` as shown in Fig. 9. In this case, a *conflict* occurs because an instance of the `Registry` component is already available. As shown in Fig. 10, the user can resolve this conflict by using the existing instance of the `Registry` instead of creating a new one. Subsequently, the extraction of the `System Model` (see Fig. 10) from the SCG (see Fig. 9)

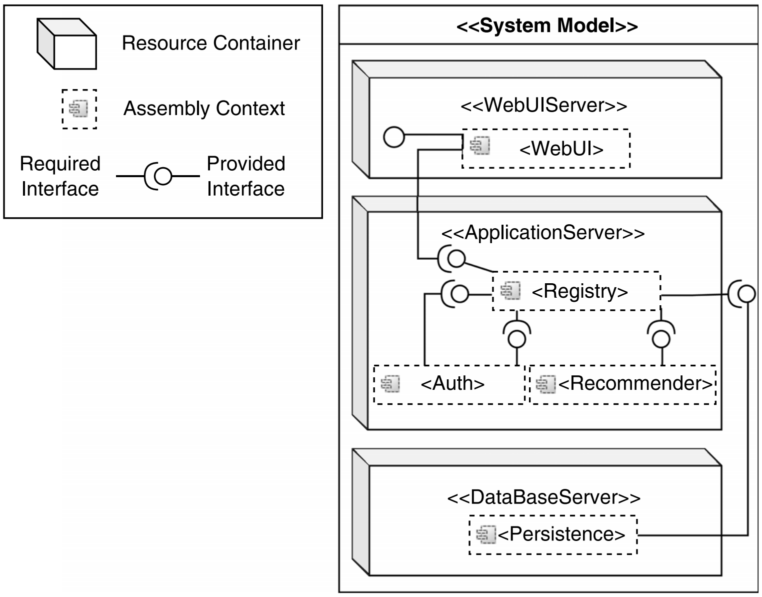


Fig. 10 Example `System Model` for the TeaStore extracted from its truncated SCG

is completed. Finally, the decisions made by the users are stored so that they can be reused in future iterations.

To sum up, the system extraction is executed automatically, except for three types of manual interventions that may occur:

- Selecting the system boundary: Users select the system's interface through the available provided interfaces of components.
- Selecting the component type: In cases where multiple component types provide the same interface, users choose the specific type to include.
- Determining new or existing component instances: Users decide whether to create a new instance of a component or use an existing one.

6 Adaptive instrumentation

The adaptive instrumentation automatically detects and instruments changed parts of the source code, enabling the calibration of related PMPs. Thus, it minimizes expenses and potential errors associated with manual instrumentation (P_{Cost}) while supporting the enhancement of PMP accuracy ($P_{Inaccuracy}$). The adaptive instrumentation generates coarse-grained probes at the service level, as commonly used in practice for performance monitoring and system resource tracking, in addition to fine-grained probes required by CIPM. In our running example (Section 3), if the `Auth.placeOrder` service is newly added, the adaptive instrumentation injects fine-grained monitoring probes to provide measurements for calibrating its PMPs: the resource demands of `prepareOrder` and `finalizeOrder`, the number of loop executions, and the parameters of the `Registry.persistOrder` and `Registry.persistOrderItem` external calls. If the source code of the remaining services has not been changed, the old estimations of their related PMPs remain valid, and there is no need for fine-grained instrumentation. Thus, only a coarse-grained instrumentation at the service level is performed for the self-validation and additional update of aPM. In the following, we expand upon the adaptive instrumentation concept introduced in Mazkatli et al. (2020) to (1) enhance its generalizability and (2) to propose general solutions for special cases that may cause compilation errors.

Regarding the enhancement of the generalizability (1), our extension injects instrumentation statements at the model level to separate the instrumentation process from implementation details such as a concrete monitoring tool. In order to support the monitoring tool independence, we define a measurements metamodel consisting of monitoring record types which correspond to the probe types defined in the IM:

- A *service record* monitors the input parameters for parametric dependency investigation, the caller for building an SCG, the current deployment for updating the `Allocation Model`, and the service execution time for the self-validation.
- The *internal action record* tracks the execution time for internal actions to estimate the related resource demands.
- A *loop action record* tracks the number of loop iterations.
- A *branch action record* monitors which branch is selected for conditional statements.

The measurement metamodel can be implemented for various monitoring tools. To implement our specific monitoring records, we use the instrumentation record language (Jung et al. 2013) of Kieker (van Hoorn et al. 2012). A detailed description and technical specifications can be found in the appendix (Appendix B).

The adaptive instrumentation process (presented as pseudocode in Algorithm 3) operates on the model level to generate instrumentation statements that are responsible for capturing the measurements and creating monitoring records. The process starts by generating the instrumentation code for each probe in the IM based on the probe type. Subsequently, it injects this instrumentation code into a copy of the source code model. To identify the appropriate locations for the instrumentation code, the process relies on the mapping stored in the VITRUVIUS correspondence model (Section 5.3). This mapping defines the relationship between the probes in the IM and SEFF elements as well as the connection between SEFF elements and their corresponding source code statements.

In certain scenarios, adjustments to the source code are necessary during the injection of the instrumentation statements to prevent compilation errors. Therefore, we identify these scenarios and define general solutions for each of them (2). For instance, instrumentation statements should surround the changed parts of the source code. Nevertheless, they cannot be added after a return statement. To address this issue, the return value is stored in a new variable by an assignment statement. Then, the instrumentation statements follow the assignment so that the final return statement only gives the new variable back (cf. Appendix C for a code example).

After completing the injection of the instrumentation code, the instrumented source code model is ready to be printed and deployed in the test or production environment to provide measurements for the calibration (Section 7).

Algorithm 3 Adaptive Instrumentation Process

```
1: function ADAPTIVEINSTRUMENTATION(IM, sourceCodeModel, correspondenceModel)
2:     codeModelCopy ← copy(sourceCodeModel)
3:     for all probe in IM do
4:         code ← generateInstrumentationCode(probe.type)
5:         locations ← findInstrumentationLocations(probe, codeModelCopy,
6:                         correspondenceModel)
7:         injectInstrumentationCode(codeModelCopy, code, locations)
8:     end for
9:     printModel(codeModelCopy)
10: end function

11: function INJECTINSTRUMENTATIONCODE(codeModelCopy, code, locations)
12:     if requiresAdjustments() then
13:         adjustSourceCode(codeModelCopy, code, locations)
14:                                   ▷ Handles cases such as post-return injections.
15:     end if
16:     addInstrumenationCode(codeModelCopy, code, locations)
17: end function
```

7 Incremental calibration of performance model parameters

After monitoring the adaptively instrumented source code, CIPM conducts the incremental calibration of PMPs by analyzing the resulting measurements. The calibration is incremental because only inaccurate PMPs are calibrated based on adaptive monitoring, instead of monitoring the entire system ($P_{Monitoring-Overhead}$) and calibrating all PMPs from scratch (P_{Cost}). Besides, this calibration aims to facilitate performance predictions for unseen states by considering impacting parameters ($P_{Inaccuracy}$). This capability enables the evaluation of design alternatives from a performance perspective, thereby supporting design decision-making.

The incremental calibration (Mazkatli et al. 2020), as described in Section 4.2, can be applied at Dev-time using measurements from a test environment and at Ops-time using measurements from the production environment. Applying this process at Dev-time allows for AbPP to assess design alternatives such as deployment plans instead of relying on expensive test-based predictions (P_{Cost}). However, the PMPs can be calibrated for the first time at Ops-time as Section 9 explains.

The incremental calibration of PMPs within abstract behaviors (SEFFs) considers the parametric dependencies described in Section 2.2. The current implementation considers the impact of the input parameters' properties, such as their values, the number of elements in a list parameter, or the size of a file parameter (Mazkatli et al. 2020). The incremental calibration starts with calibrating SEFF loops, branch transitions, and the parameters and return values of external calls. Calibration of internal actions follows and requires traversing the SEFF control flow, relying on the aforementioned estimated parameters. Resource demand estimations through adaptive monitoring may lack necessary measurements, as not all services are monitored. To overcome this challenge, the incremental calibration complements the available measurements with predicted values derived from the previously calibrated PMPs.

For instance, the calibration of the PMPs of `placeOrder` shown in Fig. 3 starts with calibrating the `LoopAction` and examining whether there is a correlation between the monitored iteration number and the input parameter `items`. In this scenario, a relationship is indeed identified, as the loop iterates based on the number of items. The calibration of both external call actions (`persistOrder` and `persistOrderItem`) follows. The exploration of parametric dependencies also covers the relation between the arguments of each `ExternalCallAction` and input data, as well as data flow (Voneva et al. 2020). Afterward, CIPM estimates the resource demands of both internal actions (`prepareOrder` and `finalizeOrder`) based on available monitoring data and previous predictions. Similar to other PMPs, CIPM investigates whether the estimated resource demands are dependent on the input parameter items. CIPM uses statistical analysis to learn parametric dependencies (Mazkatli et al. 2020). It also optimizes them based on a genetic algorithm if necessary (Voneva et al. 2020). As a result, CIPM calibrates all PMPs concerning influential variables expressed as stochastic expressions. This enables performance predictions for an unknown workload, such as estimating the performance during an upcoming offer of discounts where the order of more items is expected. This is feasible in our scenario since the performance parameters (e.g., the loop) are calibrated as a stochastic expression relating to the items variable.

8 Self-validation

The self-validation (Mazkatli et al. 2020; Monschein et al. 2021) aims at continuously evaluating the accuracy of the performance predictions related to the updated aPM ($P_{Inaccuracy}$) and reduce the monitoring overhead by deactivating unnecessary probes ($P_{Monitoring-Overhead}$).

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 in test environments at Dev-time and 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 align simulation results with monitoring data, we map the monitoring data to corresponding measuring points based on the mapping between the `Repository Model` and the source code. After this mapping, the self-validation compares the resulting two distributions with three metrics to assess the proximity of simulation results to actual measurements. For the comparison, we employ the Wasserstein distance (Santambrogio 2015), Kolmogorov-Smirnov test (Dodge 2008), and conventional statistical measures (e.g., average and quartiles). More details on the metrics are provided in Section 10.2.2.

The metrics give the user feedback on the accuracy of the PMPs of aPM. If the model is deemed accurate, developers can trust it to answer what-if performance questions. Additionally, the self-validation deactivates fine-grained probes related to accurate PMP to reduce the monitoring overhead. Otherwise, the self-validation identifies inaccurate parts that need to be recalibrated. The re-calibration can access the calculated metrics and use them to reduce the deviation and improve the accuracy of the aPM. If the self-validation is executed in a test environment and the re-calibration fails to increase the accuracy, the tester may change the test configuration to obtain 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 taken until the accuracy is acceptable.

The resulting metrics are also utilized to adapt the monitoring granularity. Fine-grained monitoring of specific services can be turned off if a predefined accuracy threshold is reached, reducing monitoring overhead. Contrarily, the self-validation may trigger the fine-grained monitoring of services if the accuracy of their performance parameters is deemed insufficient. If inaccurate parameters are not instrumented, new probes are added to the IM for recalibration after the next deployment. This ensures a balance between the aPM accuracy and necessary monitoring overhead.

9 Ops-time calibration

The goal of the Ops-time calibration (Monschein et al. 2021) is to keep the consistency between the aPM and monitoring data ($P_{Inconsistency}$) by updating the aPM based on monitoring data from the production environment. Thus, we utilize a transformation pipeline similar to iObserve (Heinrich 2016), which is summarized in Fig. 4. It consists of seven transformations that are organized in a tee and join pipeline architecture (Buschmann 1998).

The first transformation within the pipeline, $T_{Preprocess}$, filters monitoring data and converts them into suitable data structures. Besides, the monitoring data is divided into a set which is used as input for the following transformations (training set) and a second set that is used for the validations of the architecture model (validation set).

Afterward, $T_{ResourceEnvironment}$ updates the resource environment of the corresponding aPM with CPRs based on the VITRUVIUS platform (Klare et al. 2021). Next, $T_{SystemComposition}$ extracts a SCG from the monitoring data at first (Monschein et al. 2021). Then, using the SCG as input, it applies the same procedure as at Dev-time to extract a `System Model` (Section 5.5.2). The subsequent transformation $T_{Repository}$ calibrates inaccurate PMPs discovered by the self-validation. The calibration process is similar to the process at Dev-time (Section 7) and optimizes the PMPs according to the validation results either by regression analysis or by a genetic algorithm. In parallel to the $T_{Repository}$ transformation, T_{Usage} analyzes the user behavior and updates the `Usage Model` based on iObserve (Heinrich 2020).

The final transformation of the pipeline, $T_{Finalize}$, executes the self-validation (Section 8). Based on the obtained results and configurable criteria, the granularity of the monitoring is adjusted (i.e., fine-grained monitoring can be activated or deactivated).

10 Evaluation

In our evaluation, we primarily focus on evaluating the new contributions, namely C1, C2, and the scalability when applying C4 and C5. To avoid an excessively long paper, experiments evaluating C3, C4, and C5 in detail (Mazkatli et al. 2020; Voneva et al. 2020; Monschein et al. 2021) have been excluded from this paper. However, we provide a summary of their results regarding the prediction accuracy and monitoring overhead to discuss the applicability of the extended MbDevOps pipeline (C6).

We adopt the Goal-Question-Metrics approach of van Solingen et al. (2002). Based on defined evaluation goals, we derive evaluation questions (EQs) that can check whether the described goals are reached or not (cf. Section 10.1). In Section 10.2, we define metrics that can answer the EQs. In Section 10.3, we introduce the cases used for the evaluation and sum up the goals, questions, metrics, and cases for evaluating our contributions (C1-C6) (cf. Table 1). Afterward, we performed goal-oriented experiments to calculate the metrics and answer the EQs (cf. Section 10.4).

The results of our evaluation are described in three subsections that are related to the evaluation goals: the accuracy of AbPP with the updated aPMs in Section 10.5, the required monitoring overhead in Section 10.6, and the scalability of the approach in Section 10.7. Finally, we discuss the threats of validity in Section 10.8 and the results in a more general way in Section 10.9.

10.1 Evaluation goals and questions

The main goal of the evaluation is to investigate the applicability of the approach. First, to be applicable in practice, an approach has to make predictions that are reliably close to the actual performance to support developers in their decisions. Thus, our first goal is to assess the accuracy (Goal 1, G1) of the approach, which includes the accuracy of the models (G1.1) and the accuracy of the resulting AbPP (G1.2). Second, especially when integrating our approach into CI pipelines in practice, the monitoring overhead (G2) and scalability (G3) of the approach are essential to quickly remove possible inconsistencies and enable a fast reaction to potential performance issues without unnecessary strain on system resources.

We derive the following EQs to clarify whether the aforementioned goals are achieved:

- **G1.1: Accuracy of the incrementally updated models.** The accuracy of the incrementally updated models relates to whether the correct model elements have been created and updated by CIPM. Thus, **EQ-1.1**, **EQ-1.3**, and **EQ-1.4** ask for the accuracy of updating the Source Code Model (SCM), IM, and different models of the PCM. Regarding the novel CI-based consistency preservation (C1), we additionally investigate whether the granularity of the commits affects the accuracy in **EQ-1.2**. Finally, accurate instrumented source code is a precondition for accurate predictions. Thus, we investigate this aspect in **EQ-1.5**. The resulting EQs are:

- **EQ-1.1:** How accurately does CIPM update the structure of the `Repository Model` and its related models within the VSUM according to a Git commit?
- **EQ-1.2:** How accurately does CIPM update the VSUM models when it aggregates multiple commits and propagates them as a single commit? An accurate update of models by aggregating multiple commits allows users to choose less frequent executions of CIPM, for example, nightly builds only.
- **EQ-1.3:** How accurately does CIPM extract 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 IM?

- **G1.2: Accuracy of AbPP using the updated aPMs.** The main aim of CIPM is to enable accurate AbPP. Here, we study how accurate the predictions are when the models are calibrated at Dev-time (**EQ-1.6**) and at Ops-time where the self-validation updates PMPs based on measurements from production environments (**EQ-1.7**). Additionally, for more trustworthy models that are valid beyond the setting in which they have been calibrated, CIPM supports parametric dependencies so that we validate their impact on the accuracy (**EQ-1.8**). The resulting EQs are:
  - **EQ-1.6:** How accurate is the AbPP using the incrementally updated aPM at Dev-time?
  - **EQ-1.7:** How accurate is the AbPP using the incrementally updated aPM at Ops-time?
  - **EQ-1.8:** How accurately can CIPM identify parametric dependencies, and to what extent can their estimation improve the accuracy of the AbPP?
- **G2: Monitoring Overhead.** To reduce the monitoring overhead, CIPM uses adaptive instrumentation and adaptive monitoring. Thus, we investigate how well these two features can reduce the overall monitoring overhead. The resulting EQs are:
  - **EQ-2.1:** To what extent can the adaptive instrumentation reduce the instrumentation probes?
  - **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.** The scalability of the MbDevOps pipeline during Dev-time is not critical as it can be scheduled nightly. Thus, we only report how long the CIPM pipeline takes at Dev-time for various cases (**EQ-3.1**). In contrast, scalability during operation is crucial for the applicability of the CIPM approach, and its implementation should not impact the performance of the running system. Thus, we study the scalability of CIPM at Ops-time in more detail with varying numbers of model elements and monitoring records (**EQ-3.2**). The resulting EQs are:

- **EQ-3.1:** How long does the CIPM pipeline take to execute at Dev-time?
- **EQ-3.2:** How does the transformation pipeline of CIPM scale at Ops-time?

10.2 Evaluation metrics

The evaluation metrics can be divided into the following three categories. First, we need metrics to compare models to assess their accuracy, usually by comparing them against reference models (Section 10.2.1). Second, monitoring measurements need to be analyzed, which are mostly characterized as distributions of numerical values (Section 10.2.2). Third, we require a metric to estimate the quality of the update of the IM, which can be reduced to a classification problem (Section 10.2.3).

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:

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

The JC ranges from 0 to 1. A higher value indicates a greater similarity between the two sets. The JC is 1 if the sets are identical and 0 if they have no common element.

As models can be considered to be sets of elements for the purpose of evaluating accuracy (Heinrich 2020), we also apply this concept in our case. Therefore, we utilize algorithms that perform pairwise comparisons of model elements to find equal elements between two models A and B. This comparison determines the equivalence of two model elements based on different factors. At first, they need to have the same type (i.e., two elements with different types can never be equal). Then, depending on the type of the elements, further properties are checked. This can include: both elements need to have the same name in case of named elements, certain referenced elements need to be equal, or the elements need to be embedded in an equal model structure (e.g., the same position in a list). Consequently, the comparisons result in a set of equal model elements which we consider as the intersection of the models ($A \cap B$). In contrast, the models' union ($A \cup B$) consists of the equal elements and elements from A and B without an equal element in the other model. By calculating the intersection and union of two models, we can directly determine the JC for both models.

We implemented the JC to evaluate the equality of the `Java models` (cf. Section 5.2), `Lua models` (Burgey 2023, p. 31), `Repository Models` (Armbruster 2021, p. 37), `System Models`, `Allocation Models`, and `Resource Environment Models` (Monschein 2020, p. 69).

10.2.2 Distribution comparison

To compare distributions when evaluating the accuracy of performance predictions, we use three types of metrics: conventional statistical measures (Upton and Cook

2008), non-parametric tests (Kolmogorov-Smirnov-Test (KS)) (Sheskin 2007), and distance functions (Wasserstein) (Mémoli 2011). These types of metrics were chosen deliberately to address diverse aspects; each having unique advantages and flaws. This multi-metric approach ensures a comprehensive evaluation, considering interpretability, robustness, and sensitivity to different distribution characteristics. The combination of these types of metrics allows a thorough comparison of the distributions of monitored response times (reality) with simulated response times (prediction).

The Kolmogorov-Smirnov-Test (KS) (Dodge 2008) calculates the maximum distance between cumulative distribution functions. The minimum is 0 if both distributions are perfectly identical. When the value approaches 1, it signifies an increasing dissimilarity between the observed distributions. The KS test is sensitive to the shifts and shapes of the distributions, which may produce undesirable false positive alerts (high values) (Huyen 2022). For example, the KS test may result in a high value if two distributions with the same mean have different shapes. Thus, we use the KS test in combination with other metrics in our evaluation.

Wasserstein (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 the Wasserstein distance is computationally expensive to calculate, and its result is an absolute number that cannot be easily interpreted without a baseline.

Statistical measures, such as the mean or quartiles, are also calculated for both distributions.

In our research, these three metrics are employed to quantify the accuracy of AbPP by comparing predicted performance with measured performance. The comparison centers on the cumulative distribution functions, depicting the predicted service response times against the measured service response times. By employing these metrics, an overview of the dissimilarities between the predicted and actual distributions can be obtained, contributing to the assessment of AbPP accuracy.

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 if they belong to the given class and True Negatives if they do not belong to the class) and incorrectly classified entities (False Positives if they are assigned to the class and do not belong to the class and False Negatives if they are not assigned to the class and belong to the class):

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

We use the F1-Score (F-Score with $\beta = 1$) to evaluate the update of the IM. Thus, we take the `Repository Model` and IM after executing the CI-based update of the aPM. For these models, we automatically count every SEFF and updated action with a corresponding probe in the IM as True Positive. In contrast, every SEFF and updated action without a corresponding probe in the IM is counted as False Negative. At last, every probe in the IM without a corresponding SEFF or action is a False Positive. Based on these numbers, we can calculate the F1-Score.

Figure 11 displays an exemplary `Repository Model` with two SEFFs of which each has one SEFF action. All elements were added with the recent changes. The IM contains two probes related to elements in the `Repository Model` and one probe without any relation. This example has two True Positives because of the two corresponding SEFF elements in the `Repository Model` and probes in the IM. The number of False Negatives is two (Two SEFF elements without a corresponding IM probe), and the number of False Positives is one (One IM probe without a corresponding SEFF element). As a result, the approximate F1-Score is 0.5714.

10.3 Evaluation cases

In the following paragraphs, we provide details on the evaluation cases: TeaStore (von Kistowski et al. 2018) which is introduced in Section 3, Common Component Modeling Example (CoCoME) (Heinrich et al. 2015), the real-world cases "TEAMMATES" (TEAMATES-Git 2022) and Corona-Warn-App Server (CWA-Server 2023), and two Lua-based industrial applications (Mazkatli et al. 2023).

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 with a credit card. We used a cloud-based implementation of CoCoME where the enterprise server and 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 (TEAMATES-Git 2022), on which we focus in the evaluation.

The Corona-Warn-App Server is part of the German Corona-Warn-App infrastructure, which tracked the exposure of users to Corona-positive ones (CWA-Server 2023). The server handled encryption keys for the mobile apps and consists of five cloud services, communicating with further services including the mobile apps. As the mobile apps are not implemented in Java and CIPM's support is currently limited to one repository, we focus on the Corona-Warn-App Server.

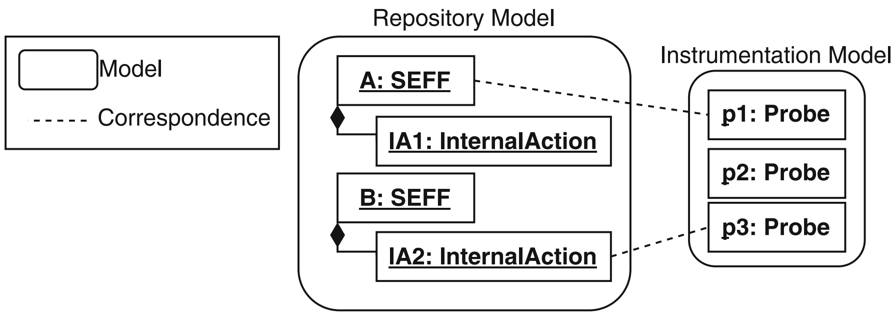


Fig. 11 Example of a `Repository Model` and IM with their correspondences

The first Lua-based sensor application, BarcodeReaderApp detailed in Section D.4, was developed by integrating four applications from the SICK AppSpace.[2] Its primary function is to capture images with a camera module, recognize barcodes within these images, and, subsequently, transmit the barcode information to a database. The second sensor application, ObjectClassifierApp detailed in Section D.5, is also from SICK AG and designed for the detection and sorting of objects from images based on their color. Table 1 shows in which experiment the cases were used.[3]

The selection of these cases considered (1) real-life contexts, (2) applicability, and (3) the objectives of our study. Regarding real-life contexts (1), we aimed to choose cases with real Git histories and available performance tests. The cases should represent realistic usage scenarios for benchmarking purposes and for the industry. This ensures that we evaluate the consistency preservation at Dev-time in real practical settings of software development. Additionally, we observed the consistency preservation at Ops-time with real measurements of the system. This provided us insights that are applicable to consistency preservation during real-world software operation.

For the applicability (2), we required cases written in programming languages for which we have compatible parsers and printers, as our approach currently operates on source code models. By extending the parser and printer for Java and implementing both for Lua, we were able to select cases based on Java and Lua.

Regarding (3), the chosen cases needed to align with the objectives of our study. The assessment of the CI-based update of aPMs (C1) required cases with real Git histories in diverse programming languages and technologies. Consequently, we selected TeaStore, TEAMMATES, the Corona-Warn-App Server, and the Lua-based sensor applications. For a comprehensive analysis of the incremental calibration (C3), we needed open source cases with performance tests representing realistic contexts. Therefore, we utilized TeaStore and CoCoME, as they are also well-suited for architecture-based performance prediction approaches. In the context of the Ops-time calibration (C4 and C5), the use of a distributed application with containerization facilitates the evaluation of the accuracy of updated models following changes in the running system architecture. Therefore, TeaStore as a microservice-based application using Docker was employed in more evaluation scenarios than CoCoME.

Overall, all aspects of CIPM were evaluated with the TeaStore. However, an exception is the evaluation of the scalability and adaptive optimization of PMPs. Their goals necessitated the assessment of worst-case scenarios. Consequently, artificial examples were employed in these scenarios.

10.4 Experiment setup

For the evaluation's primary focus on the new contributions, we conducted two experiments, E1 and E2. As shown in Table 1, experiments evaluating other aspects of C1, C3, C4, and C5 were conducted in previous work. Therefore, we briefly present their results to provide a comprehensive overview of the CIPM approach, covering the eval-

[2] See https://sick.com/.

[3] More information on the source code, replication packages, and experiments can be found on https://sdq.kastel.kit.edu/wiki/CIPM and https://doi.org/10.5281/zenodo.11236139.

Table 1 GQM-based experimental evaluation of contributions with related cases

Contribution		Experiment	Cases	GQM-G	GQM-Q
C1	CI-based Update of aPMs, including extended SCM, parser, and CPRs	E1	TeaStore, TEAMMATES, Corona-Warn-App Server, Two Lua-based sensor applications	G1.1	**EQ-1.1**, **EQ-1.2**
	`System Model` Extraction at Dev-time	Monschein et al. (2021)	TeaStore, CoCoME		**EQ-1.3**
C2	Adaptive Instrumentation	E1	TeaStore, TEAMMATES	G1.1, G2	**EQ-1.5**, **EQ-2.1**
C3	Incremental Calibration Process	Mazkatli et al. (2020), Voneva et al. (2020)	TeaStore, CoCoME, Artificial example	G1.2, G2	**EQ-1.6**, **EQ-1.8**
C4	Ops-time Calibration	Monschein et al. (2021)	TeaStore, CoCoME	G1	**EQ-1.4**, **EQ-1.7**
	Ops-time Calibration	E2	Artificial examples	G3	**EQ-3.2**
C5	Self-Validation	Monschein et al. (2021)	TeaStore	G1.2, G2	**EQ-2.2**
C6	MbDevOps Pipeline	All	All	All	All

uation of the applicability in terms of accuracy of aPMs (G1.1), prediction accuracy (G1.2), monitoring overhead (G2), and scalability (G3).

10.4.1 Experiment 1 (E1)

The objective of E1 is to assess the CI-based update of the aPM (C1) and adaptive instrumentation (C2) and to answer **EQ-1.1**, **EQ-1.2**, and **EQ-2.1**. E1 simulates the evolution at Dev-time by propagating parts of the Git history to CIPM, which in turn updates aPM accordingly. To achieve this goal, we utilized the five cases that have Git repositories with a sufficiently long development history: TeaStore, TEAMMATES, the Corona-Warn-App Server. and two Lua-based sensor applications. Table 2 shows the cases and the considered Git histories. We conducted our experiment on standard user computers for all cases. Specifically, we executed E1 for TeaStore on a computer with an Intel Core i7-4790K CPU, 32 GB RAM, and Windows 10, and for TEAMMATES and the Corona-Warn-App Server on a laptop with an Intel Core i5-7200U CPU, 8 GB RAM, and Windows 10.

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] (see Table 2). The first interval (I) consists of 50 commits, of which 27 commits change 144 Java files with overall 9,553 added and 7,908 removed lines. The remaining commits affect the build,

Table 2 The cases used by E1

Case	Interval	Commits	Added Lines	Removed Lines
TeaStore	(I)	50	9,553	7,908
	(II)	20	264	1
	(III)	11	121	134
	(VI)	100	215	227
	(I)-(IV)	181	10,153	8,270
TEAMMATES	(I)	17,832	114,468	0
	(II)	3	154	129
	(III)	2	3,249	2,978
	(IV)	2	502	340
	(V)	20	3,457	1,293
	(I)-(V)	17,859	121,830	4,740
Corona-Warn-App Server	(I)	1,232	52,163	0
	(II)	19	35	7
	(III)	18	126	147
	(IV)	2	0	0
	(V)	29	740	226
	(VI)	46	5,058	3,463
	(VII)	21	22	3
	(I)-(VII)	1,367	58,144	3,846
Lua-based BarcodeReaderApp	(I)	7	862	283
Lua-based ObjectClassifierApp	(I)	12	6,651	2,663

including configuration files. Interval (II) contains 20 commits, of which 12 commits affect five 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 four Java files with overall 121 added and 134 removed lines, while nine Java files with overall 215 added and 227 removed lines are affected by 12 of 100 commits in interval (IV).

The initial commit includes many architectural-relevant changes. By propagating this commit, an incremental reverse engineering of the source code's version 1.1 is performed. 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, (B) a method corresponding to a SEFF was extended by one statement, and (C), in a supporting service, a new class representing an interface 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. Including these intervals allows for evaluating the efficiency of adaptive instrumentation in reducing instrumentation probes for the commits without architecturally relevant changes, where fine-grained instrumentation is deactivated.

To initiate the propagation, we utilize the changes between an empty repository and version 1.1 allowing CIPM to integrate them within VITRUVIUS. Then, we perform the CI-based update of the models based on the commits that transform version 1.1 into 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. Afterward, we evaluate whether the instrumented source code is correctly generated (4). Finally, we evaluate the reduction of the monitoring overhead resulting from the adaptive instrumentation (5).

Regarding (1), an updated source code model in the VSUM shall be in the same state as if the complete source code model of a commit would have been integrated into the VSUM. Therefore, the source code of the last commit is parsed to generate a reference model for 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), a new interface is added, provided by the supporting service. For a further evaluation of the CPRs for the `Repository Model`, we also propagate version 1.3.1 as an initial commit to compare the resulting model with a completely manually created 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 to which extent 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 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 incrementally propagated commits ($RepM_{V_{inc}}$) and a manually updated `Repository Model` ($RepM'$) with the JC. The resulting source code model and IM are checked as in (1) and (3), respectively.

E1 on TEAMMATES Similar to TeaStore, we replicated E1 with the real Git history of TEAMMATES. The assessment covers 17,859 commits affecting 1,270 files, divided into five intervals (I)-(V) (see Table 2). The selected commits are propagated in five steps, where the commits of each interval are integrated and propagated as five separate commits. This approach simplifies the presentation of the results, aligning with **EQ-1.2**which evaluates the accuracy of integrating multiple commits. Consequently, we propagate commit 64842 (TM-0) as the initial commit integrating interval (I), followed by 48b67 (TM-1) for interval (II), 83f51 (TM-2) for interval (III), f33d0 (TM-3) for interval (IV), and ce446 (TM-4) for interval (V) as subsequent commits.

While TM-0 spans 17,832 commits and adds 114,468 code lines in 709 Java files, TM-1 spans three 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 direct field accesses to the new methods. TM-2 spans two commits with 3,249 added and 2,978 removed lines in 227 Java files. With TM-3, two commits affected 65 Java files, adding 502 lines and removing 340 lines. Static variables were made non-static while certain 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 3,457 and removing 1,293 lines in 147 Java files.[4] As TeaStore, the objective of E1 on TEAMMATES is to answer **EQ-1.1**, **EQ-1.5**, and **EQ-2.1**. To broaden the evaluation scope, we assess TEAMMATES under conditions different from those of TeaStore: the components of TEAMMATES are identified by a package-based strategy. Additionally, to expedite execution time, missing dependencies in the source code model are recovered with the recovery strategy (Armbruster et al. 2023), mentioned in Section 5.2, instead of building TEAMMATES.

E1 on Corona-Warn-App Server Analogous to TEAMMATES, we replicated E1 with the Corona-Warn-App Server. In total, it covers 1,367 commits affecting 816 Java files. We split these commits into seven intervals (I)-(VII) (cf., Table 2) and propagated each interval as one separate commit. Consequently, commit 7e1b6 (CWA-0) represents the initial commit for interval (I), followed by 6e970 (CWA-1) for interval (II), c22f9 (CWA-2) for interval (III), 33d1c (CWA-3) for interval (IV), 9323b (CWA-4) for interval (V), 206e8 (CWA-5) for interval (VI), and 3977e (CWA-6) for interval (VII).

While CWA-0 covers 1,232 commits with 52,163 added code lines in 597 Java files, CWA-1 spans 19 commits with 35 added lines and 7 removed lines in 4 Java files, and CWA-2 spans 18 commits with 126 added lines and 147 removed lines in 74 Java files. CWA-4 spans 2 commits with changes in one file. Between CWA-1 and CWA-2, CWA-2 and CWA-3, and CWA-3 and CWA-4, minor bugs and warnings were fixed. With CWA-5 (29 commits, 740 added lines, 226 removed lines, 54 affected Java files), a new operating mode was added. From CWA-5 to CWA-6, which spans 46 commits with 5,058 added lines and 3,463 removed lines in 82 Java files, self-report submissions were implemented beside a validation of a particular file. The last commit CWA-7 spans 21 commits with 22 added lines and 3 removed lines in 4 Java files and

[4] For more details, see https://sdq.kastel.kit.edu/wiki/CIPM_Evaluation_Details

extended test cases. The conditions for assessing the Corona-Warn-App Server differ again from TeaStore and TEAMMATES: while the components are identified with the Microservice-based strategy, missing dependencies in the source code model are recovered as in TEAMMATES. Moreover, we changed the evaluation of the automatically updated `Repository Model` (2). Instead of comparing the automatically updated `Repository Model` to a manually updated one, we manually compare the `Repository Model` changes and related source code changes to check if the `Repository Model` was accurately updated.

E1 on Lua-based Sensor Applications We consider two Lua-based sensor application cases to evaluate CIPM for a new programming language (Lua) and a new technology (sensor applications from SICK AG). Similar to the previous cases, we propagate commits and assess the models' accuracy and the monitoring overhead.

From the first application, *BarcodeReaderApp*, detailed in Section D.4, we propagate seven commits with 862 added and 283 removed lines, affecting one to four files. More details on the considered commits are provided in Table 10.

Subsequently, we propagate 12 commits from the second application, *ObjectClassifierApp*, detailed in Section D.5. These 12 commits include 6,651 added and 2,663 removed lines, impacting one to 13 files. Additional details on the commits are provided in Table 11.

In the next step, we evaluate the accuracy of the updated models, including the (1) SCM, (2) RepM, and (3) IM. Regarding (2), we implemented specific CPRs for sensor applications to expand the border of the evaluation.

However, we did not evaluate the adaptive instrumentation for Lua-based applications (4) since it is currently not implemented for a monitoring tool supporting Lua-based applications. In terms of the monitoring overhead, we assess the reduction in the number of instrumentation probes achieved by a potential adaptive instrumentation (5); thus, answering **EQ-2.1**.

10.4.2 Experiment 2 (E2)

In this experiment, we analyze the scalability at Ops-time. As mentioned in Section 9, the Ops-time calibration is achieved by a transformation pipeline that updates the `Repository Model`, `Resource Environment Model`, `System Model`, `Allocation Model`, and `Usage Model`. The scalability of these transformations is analyzed in this experiment in detail. It is important to note that there is no general benchmark for the investigation of scalability in the context of our approach due to the specific structure and parameters relevant in our approach. For this reason, we are establishing our own benchmark based on synthetically generated monitoring data in order to reflect worst-case scenarios. Thus, we first identify the parameters that influence the execution times and, subsequently, generate the monitoring data in such a way that it produces worst-case execution times of the transformation under observation. Here, we relied on synthetic data generation, as it is not possible to reliably enforce worst-case scenarios with real-world data. Based on this experiment, scalability questions with respect to the identified parameters of our approach can be answered (**EQ-3.2**).

10.5 Results for goal 1: accuracy

In this section, we present the results of evaluating the accuracy of the updated aPM in Section 10.5.1 and the related AbPP in Section 10.5.2.

10.5.1 Results for goal 1.1: model accuracy

In this section, we present the evaluation of the models' accuracy after changes at Dev-time (E1) and changes at Ops-time (Monschein et al. 2021).

Regarding Dev-time, Table 3 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. The results confirm the accuracy of the Java SCM, RepM, and IM, as indicated by JC values of one for all model comparisons and F-Scores of one for the IM.

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

Additionally, Table 4 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).

In Table 5, the results for the Corona-Warn-App Server are depicted. The source code models are nearly always accurately updated. For the intervals (III), (IV), and (VII), there is a slight deviation between the compared source code models. An investigation revealed that one or two static imports were not matched for intervals (III) and (IV) due to differing properties. For interval (VII), an additional classifier was present in one source code model, but not in the other one. These deviations had no effect on the update of the `Repository Models`. The IM is accurately updated, as indicated by F-Scores of one.

Comparing the `Repository Model` for the initial commit with the architecture documentation of the Corona-Warn-App Server (CWA-Server 2023), we observed that the `Repository Model` contains the components, interfaces, and services of the Corona-Warn-App Server as well as their connections. When we checked the source code changes and related `Repository Model` changes, we discovered that one SEFF was recovered between interval (II) and (III). After the recovery, the SEFF represented the method accurately. Before the recovery, the SEFF contained fewer actions than expected, not accurately representing the method. This deviation occurred

Table 3 Excerpt of E1's results for the TeaStore case

	JC(SCM, SCM$'$)	Resulting RepM	JC(RepM, RepM$'$)	F(IM, IM$'$)
(I)	1	RepM$_{I.3}$	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 incremental reverse engineering in average = 26.5 min
Update time average = 3.8 min
Instrumentation time average = 0.7 min

Table 4 E1's results for the 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 incremental reverse engineering = 9.39 min
Update time average = 2.05 min
Instrumentation time average = 0.45 min

Table 5 E1's results for the Corona-Warn-App Server

	JC(SCM, SCM$'$)	Resulting RepM	RepM accurately updated?	F(IM, IM$'$)
(II)	1	RepM$_{II}$	Yes	1
(III)	0.9999	RepM$_{III}$	Yes	1
(IV)	0.9999	RepM$_{IV}$	Yes	1
(V)	1	RepM$_{V}$	Yes	1
(VI)	1	RepM$_{VI}$	No	1
(VII)	0.9999	RepM$_{VII}$	Yes	1

Execution time of incremental reverse engineering = 10.95 min
Update time average = 2.80 min
Instrumentation time average = 0.08 min

during propagating the initial commit. A similar issue happened between interval (V) and (VI). There, one new SEFF contained fewer actions than expected, so that the `Repository Model` was not accurately updated in this case. In future work, we want to investigate the issue with the SEFF reconstruction.

Concerning the Lua-based applications, accurate updates are observed in the first application, BarcodeReaderApp, achieved through the propagation of seven commits. For all commits, the resulting JC values for the SCM and RepM are one, and the F-scores for the IM are one.

However, in the second application, ObjectClassifierApp, the accuracy of the Lua model is slightly less than 100 % for six out of 12 commits (cf. Table 6): the JC value is approximately 0.992 for commits 7 to 10 and 0.9472 for commits 11 and 12. Consequently, the Lua models are nearly identical. An examination of the models reveals that only a few elements are out of order in the VSUM source code model. This also impacted the subsequent updates of the RepM and IM for those six commits.

Based on the results of the five cases, we can address **EQ-1.1** as follows: CIPM updated the SCM, RepM, and IM successfully and accurately for the majority of commits related to the Java- and Lua-based cases. In future work, the implemented hierarchical matching for Lua models will be examined to understand the reasons for its inability to accurately match the order of elements in certain cases. Additionally, we will investigate the Corona-Warn-App Server case to find the reasons for the inaccuracies.

Beside the models, we checked the instrumented source code for Java-based applications (TeaStore and TEAMMATES) generated during E1 to ensure that it includes all probes that were present in the IM. We found out that the source code is correctly instrumented and has no compilation errors, 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 7. 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

Table 6 E1's results for the Lua-based sensor applications

Case	Commits	JC(SCM, SCM′)	JC(RepM, RepM′)	F(IM, IM′)
Lua-based BarcodeReaderApp	1-7	1	1	1
	Execution time of incremental reverse engineering = 0.269 seconds			
	Update time average = 0.49 seconds			
Lua-based ObjectClassifierApp	1-6	1	1	1
	7-10	0.99	0.99	1
	11	0.94	0.96	0.99
	12	0.94	1	1
	Execution time of incremental reverse engineering = 2.092 seconds			
	Update time average = 3.93 seconds			

Table 7 Evaluation results for propagating TeaStore's commits in the intervals as one

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

It compares resulting `Repository Models` ($RepM_{\sum V}$) to manual references ($RepM'$) and incrementally updated ones ($RepM_{V_{inc}}$)

contains all newly generated probes at once, while the IM is continuously updated during the propagation of multiple commits.

To answer **EQ-1.3**, we sum up the results of a previous experiment that extracts the `System Model` of CoCoME and TeaStore at Dev-time (Monschein et al. 2021). The findings in both cases show an identical model compared to the reference one, as the JC equals one. Additionally, 68.75% of the `System Model` elements in the CoCoME case and 72.3% in the TeaStore case were automatically extracted. Manual conflict resolution (cf. Section 5.5.1) was required for the remaining elements, where the user had to choose one of the suggested resolution choices. Existing methods do not support the extraction of a `System Model` at Dev-time, which is discussed in more detail in the following Section 11.1. Therefore, the automatic extraction of 68.75 % and 72.3 % of the model elements presents a significant reduction of the required modeling effort. In addition, our approach suggests potential options for the remaining model elements (cf. Section 5.5.2), which is why the modeling effort and the resulting costs can be further reduced. According to these results, **EQ-1.3** can be answered, as the system compositions were correctly reflected in the extracted `System Models`.

Regarding **EQ-1.4**, another experiment by Monschein et al. (2021) assessed the accuracy of aPMs after simulating change scenarios along the operation, such as replications, allocations, workload, and system composition changes. The accuracy of affected parts of the aPM (`System Model`, `Resource Environment Model`, and `Allocation Model`) is evaluated by comparing them to generated reference models. The results indicate that they are correctly updated by the Ops-time calibration (C4) to reflect the applied change scenarios, with a minimal JC equal to one. Consequently, it can be concluded for **EQ-1.4** that the change scenarios were recognized and correctly propagated to the models.

Summary of the results for Goal 1.1: Model Accuracy We conclude that CIPM (mainly C1, C2, and C4) can update the software models automatically and accurately. This applies to the following models: source code model (`JaMoPP`), instrumented source code, `Repository Model`, `System Model`, `Allocation Model`, and `Resource Environment Model`. An exceptional case was the update of the `System Model` and `Repository Model` at Dev-time, where the update process is not fully automatic: the user can be asked to confirm the detected components of the `Repository Model` or to decide whether to create or reuse available component instances in the `System Model`. In another exceptional case, the update of the Lua source code model encountered an implementation issue related to matching out-of-order elements in minor cases, preventing an identical update. Nevertheless, at least 94 % of the elements were accurately updated. It is important to note that the matching issue is attributed to the implementation of the hierarchical matching algorithm and is not inherent to the CIPM concept itself. While custom language-specific matching algorithms such as the one employed for Lua provide accurate matching, their implementation poses a considerable effort (Kolovos et al. 2009b) that can lead to deviations from the conceptual algorithms. It should also be noted that comparing our results with other approaches is challenging for several reasons. Firstly, the results depend on the experimental setup, which often varies between studies. Secondly, some approaches utilize batch reverse engineering techniques or do not incorporate a source code model, making direct comparisons difficult.

10.5.2 Summary of the results for goal 1.2: prediction accuracy

The prediction accuracy is an important aspect for the applicability of the CIPM approach within the proposed MbDevOps pipeline (C6). In previous work (Mazkatli et al. 2020; Voneva et al. 2020; Monschein et al. 2021), we conducted various experiments to evaluate the prediction accuracy of aPMs updated and calibrated by CIPM. These experiments compared the performance predictions against real measurements as a reference using the KS test, Wasserstein, and statistical measures. In this section, we provide a summary of the results to confirm the applicability and to answer the related EQs, namely **EQ-1.6**, **EQ-1.7**, and **EQ-1.8**.

In the study presented by Mazkatli et al. (2020), we assessed the accuracy of performance predictions employing aPMs that undergo incremental calibration at Dev-time (C3). Our experiment applied incremental evolution scenarios to CoCoME and TeaStore. According to these scenarios, the related aPMs were incrementally calibrated, and the accuracy was quantified. The obtained KS test values, on average, did not exceed 0.16 and Wasserstein distances remained below 39.6 on average. These relatively low values suggest a close alignment between the cumulative distribution functions of the predicted response times and measured response times. Based on this result, we can affirmatively answer **EQ-1.6**, stating that incremental calibration is effective in achieving accurate performance predictions.

Furthermore, we performed an assessment of the accuracy achieved through Ops-time calibration (C4) with self-validation (C5) (Monschein et al. 2021). This evaluation was conducted under worst-case scenarios, where aPMs were uncalibrated, and all instrumentation probes were activated. We observed the prediction accuracy along the execution time by comparison with measurements. Our findings confirm that the prediction accuracy of aPMs at Ops-time improves progressively with the accumulation of more monitoring data. After this learning process, the accuracy stabilizes at an acceptable level. For instance, the results of CoCoME demonstrated a KS test value below 0.1 and a Wasserstein distance under 20, with some fluctuations, after 20 minutes of the learning process. Results from the TeaStore case also confirm the prediction accuracy, even when simulating changes in deployment, workload, and system structure during the execution. The mean value of KS is 0.199 ($\sigma = 0.131$), while Wasserstein shows a mean of 70.99 ($\sigma = 68.542$). Notably, initial data insufficiency impacts the metric accuracy, resulting in higher values. However, the metrics reveal an increasing accuracy trend over time, highlighting the system's capacity to learn from additional monitoring data and adapt to changes in operation. Based on these findings, **EQ-1.6** and **EQ-1.7** can be answered: all metrics show that the derived models represent the already observed behavior well and can also be used to predict the performance for scenarios that have not been observed so far.

The experiments mentioned in this section also confirmed the detection of parametric dependencies. The advantage of parameterized models has been studied in detail by Mazkatli et al. (2020) and Mazkatli (2025), where the experiment compared the prediction accuracy of aPMs that CIPM parameterized with dependencies to the prediction accuracy of a non-parameterized aPM. The results show that prediction accuracy of a parameterized aPM is higher in the case of predicting the performance for unseen states. The accuracy is improved by 15.17 % considering the KS metric and by 22.85 % for Wasserstein (**EQ-1.8**). Additionally, the incremental optimization of PMPs is evaluated in more detail by Voneva et al. (2020). The results indicate that the optimization can improve the accuracy of AbPP for unseen states by approximately five times if the PMPs have non-linear dependencies.

10.6 Results for goal 2: monitoring overhead

Monitoring overhead reduction can be achieved by the adaptive instrumentation and adaptive monitoring.

Firstly, adaptive instrumentation decreases the number of instrumentation probes, focusing on monitoring only the necessary parts and, thereby, reducing overall monitoring overhead. This aligns with the first EQ (**EQ-2.1**). Here, we assess the reduction in monitoring overhead through adaptive instrumentation by calculating the ratio of reduced instrumentation probes during the final step of experiment E1 (5).

Table 8 summarizes the results obtained from the five cases: TeaStore, TEAMMATES, Corona-Warn-App Server, BarcodeReaderApp, and ObjectClassifierApp. Regarding the Java-based applications, Teastore, TEAMMATES, and Corona-Warn-App Server, the results demonstrate significant reductions in monitoring overhead through adaptive instrumentation. Specifically, the reduction ranges from 44.8 % to 83.3 % across all probes. Examining the fine-grained probes shows a reduction ranging from 61.9 % to 100 %. A reduction of 100 % means that no architectural changes require any fine-grained monitoring. Therefore, no fine-grained monitoring is activated in these iterations.

The Lua-based applications, represented by BarcodeReaderApp and ObjectClassifierApp, also exhibit notable reductions. The BarcodeReaderApp shows a reduction in the range of 27.27 % to 68.75 % for all probes and 52 % to 100 % for fine-grained probes among six commits. Similarly, ObjectClassifierApp demonstrates a reduction in the range of 12.6 % to 45.7 % for all probes and 26.2 % to 100 % for fine-grained probes among 11 commits. Detailed results about the Lua-based applications can be found in Tables 12 and 13. It is worth noting that we exclude the initial commits from

Table 8 Results for **EQ-2.1**. Reduction of the number of instrumentation probes in experiment E1

Case	Commit	Reduction of instrumentation probes	Reduction of fine-grained instrumentation 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 %
TEAM- MATES	TM-1	57.1 %	89.5%
	TM-2	47.3 %	73.2 %
	TM-3	64.2 %	99.5 %
	TM-4	62.0 %	96.5 %
Corona-Warn- App Server	CWA-1	83.3 %	100 %
	CWA-2	66.7 %	80 %
	CWA-3	83.3 %	100 %
	CWA-4	44.8 %	61.9 %
	CWA-5	72.4 %	100 %
	CWA-6	44.8 %	61.9 %
	CWA-7	72.4 %	100 %
Lua App.	BarcodeReaderApp	27.27 % - 68.75 %	52 % - 100 %
	ObjectClassifierApp	12.6 % - 45.7 %	26.2 % - 100 %

the discussion of the reduction of monitoring overhead, as these commits represent a state where the source code is fully instrumented.

In most cases, the number of fine-grained probes was significantly and consistently lower than the number of coarse-grained probes, typically used in iterative software development. For example, in TEAMMATES, the proportion of fine-grained probes on all probes was on average 9.9 %. Analogously calculated, the proportion was 39.4 % for the Corona-Warn-App Server, and 11.5 % and 2.5 % for both Lua-based sensor applications. The only exception was in the Corona-Warn-App Server. In some cases, the number of fine-grained probes equaled the number of coarse-grained probes, doubling the number of probes that are typically monitored. This was due to the complexity and detailed behavior of the involved services.

However, it is important to note that, in general, only a small number of services change per commit. As a consequence, fine-grained monitoring is only required for few parts of the system. Consequently, if a small fraction of the system is modified and, accordingly, fine-grained monitored, the overhead for the additional fine-grained monitoring can be comparable to or even lower than the overhead of coarse-grained monitoring. Additionally, fine-grained probes are not added when there are no structural changes, as in the third and fourth intervals of TeaStore.

In conclusion, the results highlight the effectiveness of the adaptive instrumentation in reducing monitoring overhead for both Java and Lua-based applications. The reduction across various commits promises a reduction of monitoring overhead while running the code at Ops-time. Furthermore, the self-validation process ensures that fine-grained probes are deactivated after calibration, preventing unnecessary overhead. It deactivates instrumentation probes for accurately calibrated parts.

To show the reduction due to the adaptive monitoring (**EQ-2.2**), the overall monitoring overhead is analyzed and observed over time in the worst case when the source code of TeaStore is fully instrumented and Ops-time changes are simulated (Monschein et al. 2021). The results show a reduction of **39.65 %**, after the deactivation of all fine-grained probes via adaptive monitoring and approximately 20 minutes of the experiment.

Summary of Results for Goal 2: Monitoring Overhead Based on the evaluation results, we can conclude that the adaptive instrumentation can remarkably reduce the number of instrumentation probes based on the source code changes extracted from CI, as shown in Table 8. Together with the evaluation results of the model and prediction accuracy, it can also be concluded that the self-validation successfully identified services that are accurately represented in the model and, then, reduced the monitoring granularity accordingly. Ultimately, this leads to a significant reduction of the monitoring overhead, even though the source code was fully instrumented for a worst-case analysis. With adaptive instrumentation, we can anticipate further reductions in monitoring overhead by decreasing the instrumentation probes.

10.7 Results for goal 3: scalability

The scalability of the MbDevOps pipeline during Dev-time is not critical and can be scheduled nightly. However, we delve into this aspect in Section 10.7.1 to address **EQ-3.1**. In contrast, scalability during Ops-time is crucial for the applicability of the CIPM approach, and its implementation should not impact the performance of the running system. Thus, to answer **EQ-3.2**, we conducted E2 to examine the scalability properties of the different Ops-time calibration transformations: `Repository Model` transformation (subsection 10.7.2), resource environment transformation (subsection 10.7.3), `System Model` and `Allocation Model` transformation (subsection 10.7.4), and `Usage Model` transformation (subsection 10.7.5).

10.7.1 EQ-3.1: scalability at development time

In this paper, we empirically identify the factors impacting scalability at Dev-time and envision approaches to improve it. We measure the execution time during experiment E1 and analyze the factors impacting it. Tables 3, 4, 5, and 6 contain the execution times for the initial commits and the average execution times for model updates and the instrumentation.

The execution times differ primarily based on three factors: the parser, commit size, and changes included in the commit. The incremental reverse engineering for the initial commit represents the worst-case scenario. If the initial commit is not the first one in the Git history, the size tends to be relatively large, accompanied by a high number of changes that CIPM should detect and process. Regarding Java-based applications, there is a notable difference in the execution time of the incremental reverse engineering between TeaStore (26.5 min) and TEAMMATES (9.39 min) / Corona-Warn-App Server (10.95 min): it takes 15.55-17.11 minutes longer in the case of TeaStore because the TeaStore code is built before it is parsed, which results in a larger source code model. In TEAMMATES and the Corona-Warn-App Server, we apply a recovery strategy that we implemented to reduce the execution time by resolving dependencies with synthetic elements (Armbruster et al. 2023). The implemented recovery strategy also represents a significant step towards an incremental parsing approach, which we envision for future work. In the current implementation, the entire system is parsed to detect changes, but our plan involves replacing this behavior with parsing only the modified files. Despite parsing the whole source code, the average execution time for propagating commits is 3.8 minutes for TeaStore, 2.05 minutes for TEAMMATES, and 2.8 minutes for the Corona-Warn-App Server. This time is not critical for the development phase and should be significantly reduced by implementing the suggested incremental parsing. The results from E1 also detect a correlation between execution time and the number of affected lines, as well as VITRUVIUS changes, for the three applications. If the volume of affected lines or VITRUVIUS changes increases, the execution time rises gradually, too. It should be noted that the impact of VITRUVIUS changes on the update time is not only a matter of the number of changes, but also influenced by the types of changes. The execution times of the CPRs vary based on the nature of the changes, indicating that the type of change is a crucial factor.

In the case of Lua-based applications, the execution time of the incremental reverse engineering is notably short, attributed to the efficiency of the Lua parser and the smaller size of the Lua applications. The investigation also indicates that the number of changes in a commit is an influential factor, as well as the applied changes. Since the aPM of the first App is more complicated, the execution time for App 1 is higher.

Summary of Results for EQ-3.1: Scalability at Development Time Based on the provided information, the observed relationships suggest a gradual and proportional impact of changes on execution time in the TeaStore, TEAMMATES, Corona-Warn-App Server, and Lua-based applications. Besides, a further reduction in execution time for Java-based applications is expected if the proposed incremental parsing is implemented. In general, the results can confirm the applicability of the CIPM approach at Dev-time.

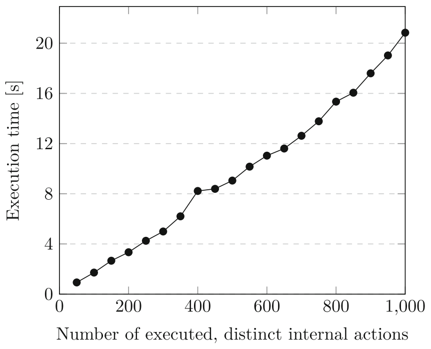
10.7.2 EQ-3.2: scalability of the repository model transformation

At first, the scalability of the `Repository Model` transformation has been analyzed. The transformation can be roughly 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.

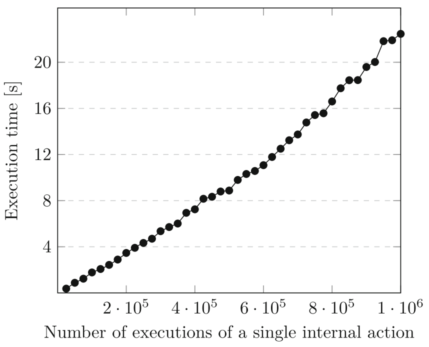
In our evaluation, the optimization is performed based on regression. However, if the genetic algorithm is applied (Voneva et al. 2020), it could be configured so that its execution time does not exceed a specific threshold. As a result, there is a tradeoff between the execution time and the accuracy 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 the regression.

The regression is performed for each stochastic expression that needs to be calibrated. The number of data points within a regression is variable and depends on the monitoring data. This can be well illustrated with 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 these factors, we built 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 Fig. 12.

In both scenarios, it is apparent that the transformation duration grows proportionally with increasing parameter values. Nevertheless, even a high number of internal actions (a) does not lead to exceptionally high execution times. The same situation can be observed when increasing the data points per internal action. In summary, it can be concluded that no unexpected side effects emerge.



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



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

Fig. 12 Scalability analysis of the `Repository Model` transformation

10.7.3 EQ-3.2: scalability of 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 CPRs based on VITRUVIUS, 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 of 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. An increasing number of new hosts, each of which has only one network connection (sparse meshed).
2. An increasing number of new hosts, each of which has a connection to all other hosts (fully meshed).

The left chart in Fig. 13 shows the scalability of sparse meshed Ops-time environments. The execution time scales almost perfectly linearly, with up to 180 new hosts. For realistic values of approximately 20 new hosts or fewer (within a single execution of the transformation), an execution time of two seconds is not exceeded in our test setup. In contrast, the execution time rises exponentially when adding fully meshed hosts, as the right chart in Fig. 13 shows. This comes from the connections between hosts that 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` indicate that the execution times scale appropriately for realistic use cases.

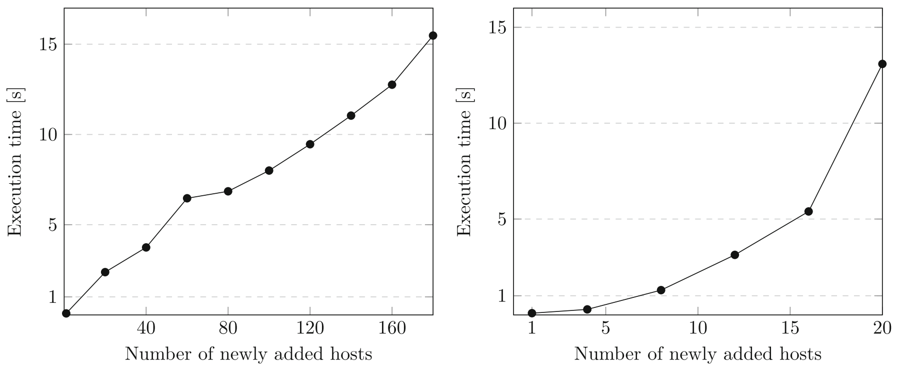


Fig. 13 Scalability analysis for transforming the `Resource Environment Model` with an increasing number of hosts. On the left side (sparse meshed), each host has only one network connection. On the right side (fully meshed), each host has a connection to every other host

10.7.4 EQ-3.2: scalability of system model and allocation model transformation

The transformations that are responsible for updating the `Allocation Model` and `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 14 shows the cumulated execution times 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 at around 500 changes and beyond. Even for a total number of 1,200 changes, the execution time is lower than four seconds. Because such a number of changes between two pipeline executions probably never occurs in practice, it can be concluded that both transformations scale well.

10.7.5 EQ-3.2: scalability of 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 for iObserve is in Heinrich (2020).

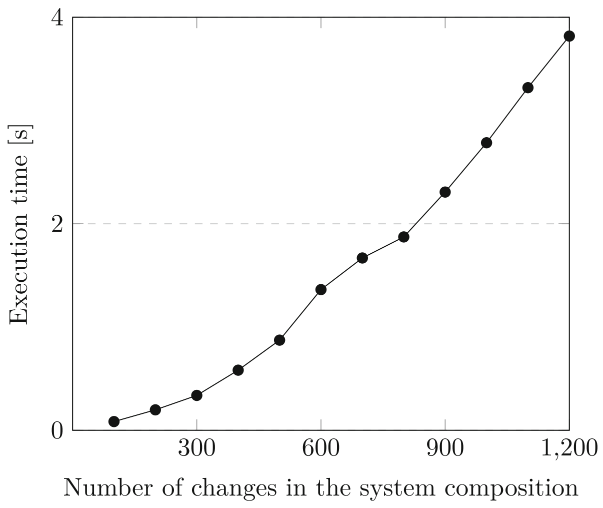
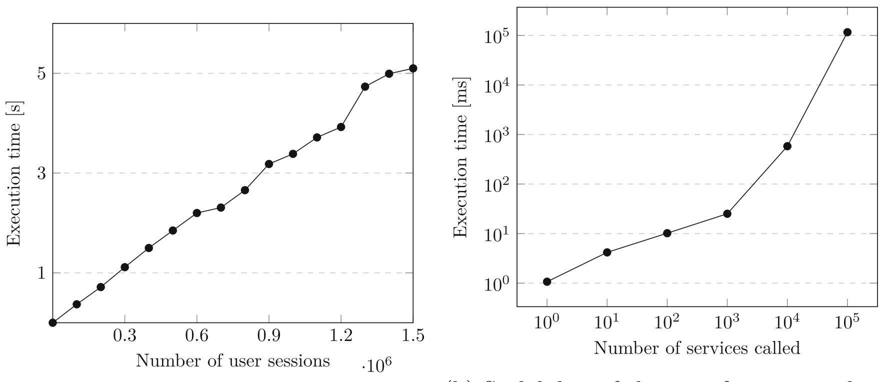


Fig. 14 Execution times of the `System Model` transformation with an increasing number of changes in the system composition

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 15 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 sub-figure (b), it should be noted that the axes are scaled logarithmically. In this way, we wanted to ensure that the results can be compared to those obtained from the iObserve scalability analysis.

The experiment shows that the execution time scales almost perfectly linearly 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 fewer initiated service calls, the execution time increases sublinearly and, thereafter, superlinearly, with a fast growth in execution time above 10,000 initiated service calls. In the superlinear segment, the execution time is dominated by the loop detection (Heinrich 2020). The extreme increase in the execution time with a high number of triggered service calls is not critical, because such a user behavior is unlikely in practice. 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.

Summary of the Results of EQ-3.2: Scalability of the Ops-time Pipeline Based on our findings, it can be inferred that all transformations demonstrate appropriate scalability across various scenarios, which positively answers **EQ-3.2**.



(a) Scalability of the transformation when the number of users increases.

(b) Scalability of the transformation when the number of triggered service calls by a single user increases.

Fig. 15 Scalability analysis of the `Usage Model` transformation

10.8 Threats to validity

The identification of the threats to validity is based on the guidelines by Wohlin et al. (2012). Therefore, we distinguish between three dimensions of validity: *internal validity*, *external validity*, and *construct validity*.

Internal validity

One concern is related to the selection of PCM reference models employed to evaluate the initial commits in experiment E1. Thus, two strategies were applied to ensure the accuracy of the selected models and to mitigate the subjectivity of the experiment executor. First, a reference model was manually modeled based on available documentation and reviewed by individuals who were not involved in the execution of the experiment. We applied this strategy to E1 on the TeaStore. Second, while we automatically extracted the initial PCM model for TEAMMATES and Corona-Warn-App Server and modeled them for the Lua applications, we later asked individuals who did not take part in the automatic extraction or modeling process to review the resulting models to ensure their quality. For instance, the automatically extracted PCM model from TEAMMATES was checked by comparing it with the extensively documented architecture of TEAMMATES (TEAMATES-Git 2022). Similarly, external individuals, such as those from SICK AG, reviewed the initial models of the Lua applications. Both strategies aimed to mitigate observer bias and increase the reliability of the evaluation process.

Another threat to internal validity concerns the evaluation of the `System Model` extraction at Dev-time. The conflicts occurring during the execution were resolved manually so that the outcome may depend on the person who conducted the experiment. If this person had not known the system composition well enough and had made an incorrect decision, the calculated JC would have been lower.

External validity

A threat to external validity is the selection of cases. It may be possible that the results obtained from the cases are not representative. To mitigate this threat, we selected CoCoME, TeaStore, TEAMMATES, and Corona-Warn-App Server, 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; Horn et al. 2022; Elsaadawy et al. 2021; Sokolowski et al. 2021; Liao et al. 2021; Flora et al. 2021; Torquato et al. 2021; Eismann et al. 2020; Caculo et al. 2020; Viktorsson et al. 2020; Martin et al. 2020; Hahner et al. 2023). Furthermore, we included two Lua-based applications to assess an industrial case based on a different programming language and technology. By combining these cases, the risk of non-representative results is further reduced.

Another threat to validity is the selection of the Git history. The Git history should be representative and cover the predefined CPRs. Therefore, we selected 17,859 commits of the real application TEAMMATES, 1,367 of the Corona-Warn-App Server, 181 of TeaStore, and 19 commits from the Git histories of the Lua-based applications. The incremental reverse engineering of the initial commits covers almost all CPRs adding elements. Propagating the following commits covers most of the remaining CPRs. For instance, propagation of the commits between TM-2 and TM-4 from TEAMMATES covers 41 % of the remaining CPRs.

Inconsistent experimental conditions can also impact the evaluation. Thus, a common protocol for evaluating the updated models across all cases defined the criteria for a consistent assessment of the models' accuracy using different metrics. Besides, identical experimental conditions were kept across all transformations of the Ops-time calibration. Ensuring consistency in model evaluation minimizes the variability in results, enhancing the reliability and validity of the research findings.

Construct validity

A threat to the construct validity is the selection of metrics for the evaluation. To compare distributions, we applied the Wasserstein distance, the KS test and conventional statistical measures. These have been used in related studies in similar scenarios (Eismann 2023; Mellit and Pavan 2010). 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 of us (Adeagbo 2019; Heinrich 2020). Besides, in the evaluation, we rely on a combination of synthetically generated monitoring data and monitoring data generated by directly 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 a previous project (van Hoorn et al. 2012). Additionally, the experiments assessing the performance prediction accuracy were repeated multiple times to mitigate outliers. Then, we utilized statistical analyses to enhance the results' confidence.

10.9 Discussion

In this section, we discuss the evaluation results and their implications on CIPM. Concretely, the evaluation results indicate that CIPM is applicable to software systems, requiring standard computers with acceptable processing time and memory usage, depending on the size and complexity of the system (cf. Section 10.7).

The current realization has some limitations, which constrain the applicability. At first, every programming language in which a part of a system is implemented requires a modeling environment for its integration into CIPM. Such a modeling environment consists of an explicit metamodel to express source code in a source code model, a parser to generate these models, and a printer to output the models as source code again. While we provide modeling environments for Java and Lua, other programming languages usually have none for direct usage in CIPM available. Thus, the modeling environments need to be developed. This development effort varies and depends on different factors, including the size of the language's features. In future work, we want to investigate how the development effort can be reduced and further programming languages can be integrated more easily into CIPM as a consequence. Secondly, the monitoring at Ops-time is required, which typically is one of the standard practices in modern iterative software development, such as DevOps. The instrumentation will be automatically performed with our adaptive instrumentation. Currently, CIPM supports instrumentation for Kieker monitoring tool (van Hoorn et al. 2012). If a different monitoring tool should be used for some reason, additional effort is required to implement an interface with the required instrumentation statements.

Moreover, in the evaluation, we considered systems, which are only implemented in one programming language and versioned in one repository. However, as our approach is based on Vitruvius, it allows to easily integrate further source code models for other programming languages and to use them at the same time. Then, if multiple programming languages or repositories are involved in a software system, the information for the aPM is distributed over the repositories and source code in different programming languages so that the information needs to be combined. There are different locations in the CIPM approach where this combination can take place. For example, the whole source code can be directly input into CIPM, resulting in one aPM for the complete system. However, this prevents the analysis of single components (e.g. single microservices). Therefore, as an alternative, disjoint parts of a system can be analyzed independently of each other in separate CIPM pipelines. Then, to assess the properties of the complete system, the partial aPMs are transformed into one. We leave the comparison and evaluation of these possibilities for incorporating multiple programming languages and repositories open for future work.

By employing technology-specific CPRs, CIPM supports multiple and different technologies. This is also demonstrated in the evaluation: both TeaStore and TEAMMATES apply a few technologies, of which one, for instance, is reused. In general, technology-specific CPRs allow their reuse in all cases, in which the technology is applied. Nevertheless, if there is a new technology without CPRs, the CPRs require an initial definition and implementation with associated effort and costs. Effort estimation for implementing new CPRs varies based on several factors, for example, the existing knowledge about the architecture, as observed in our experiments on our cases. In scenarios where there is an architect, up-to-date architecture documentation, or well-distributed knowledge about the code structure, the adoption process should take less than or equal to one week. Conversely, in cases where documentation is outdated, knowledge is lost, or an architect is missing, the adoption could extend to 1-3 weeks. Additionally, implementing CPRs requires the involvement of an architect to define the architecture or a detailed review of the documentation. It is advisable to have a second person review the defined rules to ensure correctness and completeness. Therefore, the human resource requirement at the beginning of the implementation process is typically two people. The discussion above outlines the requirements and costs associated with applying CIPM for the first time to new technologies. As part of future work, we want to look for general CPRs to reduce the effort, since general CPRs could be applied for multiple and different technologies and programming languages. Our initial results indicate a progress toward general CPRs (Weng 2024).

While we evaluated the scalability of the calibration transformations during Ops-time, we gained first insights into the scalability of the CI-based update of the aPM during Dev-time (cf. Section 10.7.1). TEAMMATES as a real case with 114,468 lines of code in its initial considered commit TM-0 suggests that the CI-based update scales well for larger systems. This is further confirmed by the real Corona-Warn-App Server. However, it requires further evaluation in the future with systems of varying sizes, multiple programming languages, and different technologies to analyze the scalability. Moreover, we plan to evaluate the scalability of our approach in regard to the optimization with the genetic algorithm, since there are trade-offs between the impact of the algorithm's configuration on PMPs (in particular, their accuracy) and the overhead required by this configuration.

For the adaptive instrumentation and monitoring at Dev-time, a system needs tests for a specified test environment. This allows to take measurements from the instrumented source code and to calibrate the aPM at Dev-time. Regarding Ops-time, a monitoring setup is typically one standard practice in modern iterative software development, such as DevOps. Therefore, no additional effort should be required.

To summarize, the evaluation suggests that CIPM is applicable to software systems. Several limitations constrain these systems. In future work, we want to address the limitations to improve the applicability of CIPM. The evaluation of Lua-based sensor applications, for example, is a first step towards this goal due to the usage of another programming language and other technologies.

11 Related work

Many approaches aim to achieve the consistency between software artifacts automatically. As explained in Section 1, these approaches belong to two main categories. In the first one (A1), approaches generate up-to-date artifacts as a batch process (marked as $\checkmark_B$ in Table 9), for example, reverse engineering approaches. In the second category (A2), approaches check the consistency and try to resolve inconsistencies (named as incremental approaches and marked as $\checkmark_{inc}$ in Table 9). Both A1 and A2 can also be classified into three subcategories based on the phase in which the consistency is maintained: at Dev-time (Section 11.1), at Ops-time (Section 11.2), or both (Section 11.3). Table 9 summarizes the related work and assigns them to these subcategories since the main category is labeled as $\checkmark_B$ for A1 and $\checkmark_{inc}$ for A2. Related work on resource demand, parametric dependencies, and instrumentation is discussed in Sections 11.5, 11.6, and 11.4.

11.1 Consistency management at dev-time

As shown in Table 9, many approaches focus on the consistency maintenance at Dev-time. They either extract an architecture model (A1) or maintain an existing one (A2). The reverse engineering approaches at Dev-time (A1) are based mainly on the static analysis of source code. For example, Qayum et al. (2025) identified 111 Software Architecture Recovery (SAR) approaches through a tertiary mapping study of 23 recent studies published after 2020. Their analysis aimed to uncover key characteristics for selecting suitable SAR approaches. Additionally, they proposed a

Table 9 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	SAR by Qayum et al. (2025) and ROMANTIC-RCA by El Hamdouni et al. (2010)	$\checkmark_B$								
	Extract by Langhammer et al. (2016)	$\checkmark_B$				$\checkmark_B$		✓		
	SoMoX+Beagle by Krogmann (2012)	$\checkmark_B$					$\checkmark_B$	✓	✓	
	Consistency checker by Jens and Daniel (2007), Buckley et al. (2013), von Detten (2012), and Lucas et al. (2009)	$\checkmark_{inc}$								
	Co-evolution by Langhammer (2017, p. 35)	$\checkmark_{inc}$					$\checkmark_B$	✓		
	Co-evolution: VAVE by Ananieva et al. (2022), Mbeddr by Voelter et al. (2012), Focus by Ding and Medvidovic (2001)	$\checkmark_{inc}$								
Ops-time	Brosig et al. (2011)	$\checkmark_B$	$\checkmark_B$	$\checkmark_B$		$\checkmark_B$	$\checkmark_B$	✓		
	PMX by Walter et al. (2017), Weber et al. (2024)	$\checkmark_B$	$\checkmark_B$	$\checkmark_B$			$\checkmark_B$	✓		
	Brunnert et al. (2013)	$\checkmark_B$	$\checkmark_B$	$\checkmark_B$	$\checkmark_B$		$\checkmark_B$	✓		
	SLAstic by van Hoorn (2014)	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_{inc}$			✓	✓		
	iObserve by Heinrich (2020)	$\checkmark_{inc}$		$\checkmark_{inc}$		$\checkmark_{inc}$		✓		
	Approach for CPE by Cortellessa et al. (2022)	$\checkmark_{inc}$					✓	✓		

Table 9 continued

Scope	Consistency Management Approaches	aPM					**PMPs**	**AbPP**	**Parametric dependencies**	**Self-validation**
		Repository	System	Allocation	Res. Env.	Usage				
Hybrid	EjbMoX by Langhammer (2017, P. 140)	$\checkmark_B$						✓		
	Konersmann (2018)	$\checkmark_B$		✓						
	PRISMA by Spinner et al. (2019)	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_B$	✓		
	CIPM	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_{inc}$	$\checkmark_{inc}$	✓	✓	✓

taxonomy defining the main SAR selection categories and developed a framework to support developers in making informed decisions. ROMANTIC-RCA (El Hamdouni et al. 2010) extracts component-based architectures from an object-oriented system based on relational concept analysis. Similarly, Extract (Langhammer et al. 2016) and SoMoX (Becker et al. 2010) extract parts of the PCM. A shortcoming of A1 approaches is that they ignore the possible manual optimization of the extracted model in the next extraction. This makes them unsuitable for continuous extraction after each iterative development cycle.

The incremental consistency maintenance at Dev-time (A2) includes approaches that minimize, prevent, or repair architecture erosion. de Silva and Balasubramaniam (2012) present a good summary of these approaches. Moreover, Jens and Daniel (2007) compare approaches that minimize architecture erosion by detecting architectural violations at Dev-time. The JITTAC tool (Buckley et al. 2013), for instance, detects 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 A2 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 (e.g., component-based architectures). Similarly, the Co-evolution approach uses a virtual single underlying model within the VITRUVIUS platform to allow the co-evolution of the PCM and source code. Ananieva et al. (2022) go beyond this approach by extending the VITRUVIUS platform to ensure consistency preservation during variability both in space, as found in software product line engineering (SPLE), and in time, as managed in software configuration management (SCM). Their approach enhances the VITRUVIUS platform by integrating these two dimensions of variability into a unified conceptual model (Ananieva et al. 2020). The evaluation demonstrates that architectural models, such as Unified Modeling Language (UML) class diagrams, and other affected artifacts, can be updated automatically in response to modifications in Java models.

The Focus approach by Ding and Medvidovic (2001) avoids inconsistencies by recovering the architecture and using it as a basis for the evolution of object-oriented applications.

Lucas et al. (2009) provide a systematic literature review to identify the current state-of-the-art in UML model consistency management, highlighting open issues, trends, and future research opportunities. Besides, they present a formal approach to resolve the inconsistencies based on model transformation techniques.

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 AbPP should be supported. Therefore, some of these approaches are extended to allow AbPP. For example, Langhammer calibrated co-evolved PCMs with approximated resource demands (response times) to demonstrate that they can be used for AbPP. Similarly, Krogmann et al. extended SoMoX with a calibration of PMPs with parametric dependencies based on dynamic analysis (Beagle approach Krogmann 2012). However, the approach of Krogmann requires high monitoring overhead, which restricts the collection of monitoring data from the production environment rather than from the test environment. Thus, 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 through incremental calibration. Moreover, all consistency management approaches at Dev-time ignore the effect of adaptions at Ops-time on the accuracy of aPMs.

In addition, the current CIPM implementation ensures consistency within the CI pipeline, primarily addressing variability over time. Supporting variability in space, as proposed by Ananieva et al. (2022), would require either integrating their approach with our incremental calibration to enable AbPP for each variant or considering multiple CI pipelines, each representing a variant, where the corresponding source code is managed in separate Git branches.

11.2 Consistency management at ops-time

Approaches that maintain the consistency at Ops-time are mainly based 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 (A1) that extract parts of the PCM based on dynamic analysis for AbPP. Furthermore, the SLAstic approach (van Hoorn 2014) extracts an aPM, can detect certain changes at Ops-time (e.g., migrations), and reflects them in the model (A2). Similar to CIPM, Weber et al. (2024) examine the integration of performability-model extraction and prediction in CI/CD pipelines, building upon PMX. They use monitoring infrastructure to collect traces and metrics. Traces reveal the control and data flow within applications. When combined with metrics, they enable the derivation of system models. One of our previous works, iObserve (Heinrich 2020), can also respond to changes in the deployment and usage by updating related parts in the PCM (A2), which we also integrated into CIPM. Similar to iObserve, Cortellessa et al. propose a model-driven integrated approach that utilizes traceability relationships between monitoring data and architectural models (UML profiles with MARTE) to identify design alternatives for addressing performance issues (A2). Their goal is to enhance system performance through continuous performance engineering based on an initial aPM. However, their approach is limited to microservice-based systems and lacks the ability to observe changes in the source code to update the UML model accordingly. Other approaches that extract or update performance models at Ops-time are summarized by Szvetits and Zdun (2016). Drawbacks of the Ops-time approaches are that a continuously high monitoring effort is required to extract or update models. They cannot model the system's parts that have not been called and, consequentially, are not covered by the monitoring. Besides, these approaches ignore source code changes and 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 a reverse engineering tool (EjbMox) (Langhammer 2017, P. 140) that extracts the behavior of underlying Enterprise JavaBeans source code, by analyzing it at Dev-time and calibrating it based on dynamic analysis at Ops-time. The approach of Konersmann (2018) integrates annotations about the architecture model into source code for a dynamic generation of an architecture model from the source code via transformations. The approach of Konersmann also synchronizes allocation models with running software (Konersmann and Holschbach 2016). Spinner et al. (2019) propose an agent-based approach to update aPMs (A2). Their approach is integrated with the CD to identify newly deployed Enterprise JavaBeans components and interfaces. The `Repository Model` does not capture non-deployed components, and PRISMA calibrates parameters as constant values without parametric dependencies. In summary, 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 enable performance analyses and usability evaluations. 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 an adaptable instrumentation of the services of the application under test to obtain more accurate measurements for estimating the resource demands of an aPM. Unlike existing work, our approach expects the source code changes from the CI as input and automatically detects which parts should be instrumented fine-grained. The measurements are collected from test or production environments.

11.5 Resource demands

The related approaches estimate 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 yield a higher accuracy, but suffer from the overhead of the instrumentation and monitoring. Our approach reduces the overhead through the automatic adaptive instrumentation and monitoring. Similar to CIPM, Grohmann et al. (2021) update the resource demand continuously at Ops-time. To achieve this, 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.

11.6 Parametric dependencies

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

12 Conclusion

In software development, understandability and productivity increase when the software architecture is considered (Olsson et al. 2017). Moreover, problems of expensive measurement-based performance prediction can be resolved by applying Architecture-based Performance Prediction (AbPP). Through simulation of the architectural Performance Model (aPM), AbPP promises the proactive detection of performance problems.

In this article, we presented the Continuous Integration of architectural Performance Models (CIPM) approach, which keeps aPMs continuously up-to-date. Our approach maintains the consistency between software artifacts at Dev-time and Ops-time by updating the aPM after the evolution and adaptation of the system.

At Dev-time, our novel commit-based strategy extracts source code changes from commits to apply a CI-based consistency preservation on software models. That process allows updating the aPM, including its structure, abstract behavior, and system composition. Moreover, the CIPM approach applies adaptive instrumentation to the changed parts of the source code. To achieve the simulation of the aPM, our calibration estimates the parameterized PMPs incrementally and uses also an incremental resource demand estimation based on adaptive monitoring. The calibration identifies the parametric dependencies and optimizes them based on a linear regression or genetic algorithm.

In addition to PMPs, the Ops-time calibration monitors the adaptive changes (whether in the deployment, resource environment, usage, or even the system composition) and updates the affected parts of the aPM accordingly. Our proposed self-validation continuously analyzes the accuracy of the AbPP. The results of the self-validation are utilized to manage the monitoring and calibration of the aPM at Ops-time.

We have implemented our approach for Java-based applications and the Palladio simulator. Additionally, we have tailored the CIPM approach to be applicable to Lua-based industrial sensor applications at SICK AG. These sensor applications serve our purpose of assessing CIPM's effectiveness in other programming languages and technologies. Our adaptations include the parsing of Lua source code, modifications of CPRs for the purpose of identifying SICK AppSpace apps as components, and lastly, updating the models in the VSUM.

For the evaluation, we performed experiments based on six cases: CoCoME (Heinrich et al. 2015), TeaStore (von Kistowski et al. 2018), TEAMMATES (TEAMATES-Git 2022), the Corona-Warn-App Server (CWA-Server 2023), and two Lua-based sensor applications from SICK AG (Mazkatli et al. 2023). We were able to update the structure of the aPM based on Git commits. We demonstrated the accuracy of the updated models and the applicability of the consistency maintenance process. Further, we demonstrated a decrease in monitoring probes accomplished by using our adaptive instrumentation. According to our cases, CIPM's adaptive instrumentation reduced the number of required instrumentation probes by between 12.6 % and 83.3 %. Our findings indicate that, beyond the influence of adaptive instrumentation, the overhead can be reduced through our adaptive monitoring, which dynamically adjusts the monitoring based on validation results. Finally, we analyzed the factors that affect the execution time at Dev-time and characteristics of scalability of the transformation pipeline at Ops-time. We found that the execution time of CIPM during development is acceptable and that CIPM scales adequately at operational time with an increasing amount of monitoring data.

Currently, we are completing the Lua-based prototype to enable the adaptive instrumentation, and we are conducting further evaluation at Dev-time and Ops-time. In future work, we plan to evaluate the scalability of our approach for different scenarios (e.g., for optimization with the genetic algorithm, for multiple programming languages, amongst others). One limitation of our implementation is that the CPRs are specific to programming languages and technologies. While our approach is based on the Vitruvius platform, which allows the integration of further metamodels for additional programming languages and CPRs for new technologies (Klare et al. 2021), we are investigating different approaches to simplify these integrations and hence, to improve the applicability of CIPM.

Appendix

Some parts of the appendix are from Mazkatli (2025), where more details on the CIPM approach can be found.

A: Acronyms

CD	continuous deployment	12
CI	Continuous Integration	2
CoCoME	Common Component Modeling Example	33
CIPM	Continuous Integration of Performance Models	4
PMP	Performance Model Parameter	3
aPM	architectural Performance Model	2
CPR	Consistency Preservation Rule	8
CPU	Central Processing Unit	7
Dev-time	Development time	3
Ops-time	Operation time	3
EQ	evaluation question	29
SCM	Source Code Model	29
RepM	Repository Model	37
IM	Instrumentation Model	11
JaMoPP	Java Model Parser and Printer	17
JC	Jaccard similarity Coefficient	31
KS	Kolmogorov-Smirnov-Test	32
AbPP	Architecture-based Performance Prediction	2
MbDevOps	Model-based DevOps	11
PCM	Palladio Component Model	
REST	Representational State Transfer	9
SCG	Service-Call-Graph	16
SEFF	Service Effect Specification	6
UML	Unified Modeling Language	59
VSUM	Virtual Single Underlying Model	8

B: Measurement metamodel

This section provides an in-depth description of the measurements metamodel introduced in Section 6. The metamodel comprises several monitoring record types, each aligned with the probe types defined in the IM (cf. Section 5.4). The graphical representation of the measurements metamodel (Fig. 16) shows the following records:

- `ServiceContextRecord` monitors the following features of a service:
  - `serviceID`: Identifier for the service.
  - `executionID`: Identifier for each service execution.

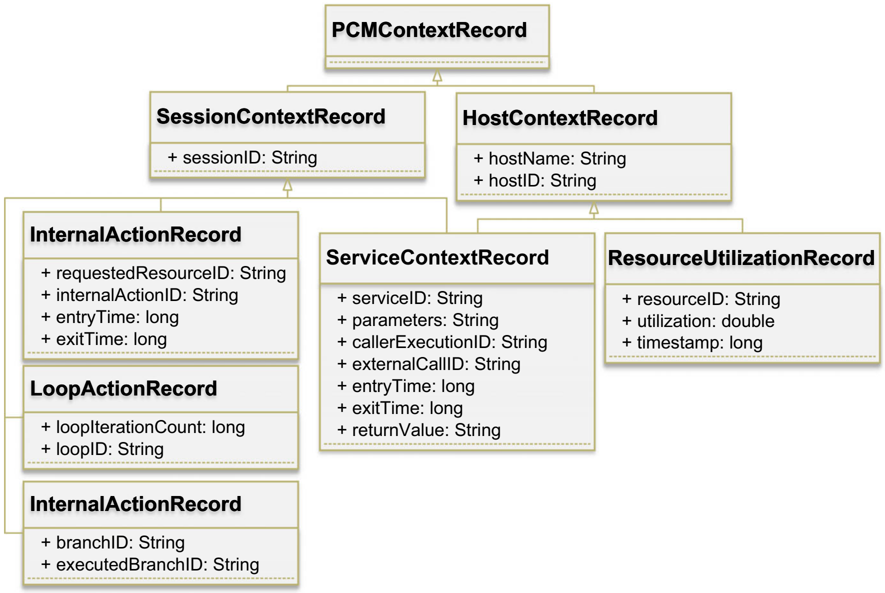


Fig. 16 The Measurements Metamodel

  - `parameters`: Input parameter properties (e.g., type, value, number of list elements, etc.) as candidates for parametric dependency investigation.
  - `callerExecutionID`: The caller of this service execution supporting the construction of an SCG.
  - `externalCallID`: Identifier of the external call to which this execution belongs. It is necessary for calibrating the SEFF external call. Note that a service can be called from multiple services using multiple external calls.
  - `entryTime, exitTime`: The service execution time serving as a reference for the self-validation.

- `HostContextRecord` monitors the host properties such as `hostID` or `hostName` determining the deployment location of the component offering this service. This information is later used for the Ops-time calibration.
- `ResourceUtilizationRecord` monitors the `utilization` of a resource (`resourceID`) at a specific time (`timestamp`).
- `InternalActionRecord` monitors the execution time of internal actions (`entryTime, exitTime`) to estimate their resource demands on a resource (`requestedResourceID`).
- `LoopActionRecord` monitors the number of loop iterations (`loopIterationCount`).
- `BranchActionRecord` monitors the selected branch (`executedBranchID`).

C: Example of the adaptive instrumentation

In this example, we consider the adaptive instrumentation of the `clearAllCaches` service from our running example TeaStore. Initially, the method's `InternalAction` is instrumented with enter and exit statements to monitor the response time and to report the "ID" of this internal action, as shown in Listing 2. During instrumentation, the unique identifier of an internal action (e.g., `_YHXHhwzdEeyhr8BpjC`) is typically included in the monitoring records. This ID serves as a reference point within the monitoring system, facilitating the mapping between monitoring data and correlated internal actions that need to be calibrated based on this data.

To ensure that the instrumented code compiles, adjustments are necessary since the exit statement is located after a return statement. Thus, as depicted in Listing 3, the return value is stored in a local variable (`resp1`) which is returned after the exit statement. This allows to compile the instrumented code while capturing the necessary monitoring data during the execution of the method.

```
monitoringController.enterInternalAction("
    _YHXHhwzdEeyhr8BpjC");//instrumentation code
return Response.ok("cleared").build();
monitoringController.exitInternalAction("
    _YHXHhwzdEeyhr8BpjC");//instrumentation code
```

Listing 2 Direct instrumentation of the `InternalAction` of `clearAllCaches`.

```
monitoringController.enterInternalAction("
    _YHXHhwzdEeyhr8BpjC");//instrumentation code
Response resp1 = Response.ok("cleared").build();
monitoringController.exitInternalAction("
    _YHXHhwzdEeyhr8BpjC");//instrumentation code
return resp1;
```

Listing 3 Corrected instrumentation of the `InternalAction` of `clearAllCaches`.

D: Evaluation cases

In this section, we provide more details on the evaluation cases, focusing on the goal of selecting the case, its structure, and the technologies used. We exclude TeaStore since it is presented in Section 3.

D.1: TEAMMATES

TEAMMATES is a tool for confidentially managing students' peer evaluations, instructor comments, and feedback. For this goal, hundreds of universities around the world use TEAMMATES as a free online cloud-based service. TEAMMATES is a realistic open source application with a long history, making it suitable for evaluating C1 and C2.

Goal: The reason for choosing TEAMMATES is to evaluate CIPM with a real tool and a real Git history. Specifically, evaluating the CI-based update of aPMs requires an existing real Git history. Therefore, TEAMMATES is used for evaluating the accuracy of the updated models (G1) and the reduced instrumentation probes (G2).

Structure and Technology: TEAMMATES consists of two parts: a web-based frontend and a Java-based backend. Our evaluation covers the backend part over a Git history of 17,859 real commits and changes related to 1,428 files. According to the well-documented architecture,[5] the backend of TEAMMATES consists of the following four main components: First, the `UI` represents the entry point for the application backend. The `UI` is based on a RESTful controller. It ensures the separation between access control and execution logic. The second component `Logic` handles the business logic of the tool. It processes various data and can access data from the `Storage` component. The third component `Storage` is responsible for CRUD (Create, Read, Update, Delete) operations on data entities, leveraging the Google Cloud Datastore. The last component `Common` includes required common utilities.

D.2: Corona-Warn-App server

As part of the German Corona-Warn-App infrastructure with which the exposure of users to Corona-positive ones could be tracked, the Corona-Warn-App Server handled the encryption keys for the mobile apps (CWA-Server 2023). It was developed open-source and operated by SAP SE and Deutsche Telekom on behalf of the German government.

Goal: Similar to TEAMMATES, we employ the Corona-Warn-App Server as a suitable additional real-world application to evaluate the accuracy of the updated models (G1) and the reduced instrumentation probes (G2).

Structure and Technology: The Corona-Warn-App Server consists of five cloud-based services (CWA-Server 2023). The `Submission` service is responsible for processing (verifying and storing) keys sent by the mobile apps, while the `Distribution` service publishes keys in a cloud storage. The `Upload`, `Download`, and `Callback` services interact with the Federation Gateway Service to exchange keys with other national app backends within the European Union. All services define REST interfaces or clients to communicate with other services or the mobile apps.

D.3: SICK appspace sensor applications

In this case, we examine the applicability of the CIPM approach in the context of two industrial applications from SICK AG.[6]

Goal: The goal of using Lua-based sensor applications as evaluation cases is to investigate the applicability of CIPM for an industrial case with a new technology and a new programming language. Hence, we evaluate CIPM based on the Git history

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

[6] See https://www.sick.com/.

of two sensor applications. Particularly, we evaluate the CI-based update of aPMs for evaluating the accuracy of updated models (G1) and the reduced instrumentation probes (G2). In general, applying CIPM for sensor applications allows developers to address the challenges facing the development of these applications by assessing design decisions through AbPP.

Technology: The sensors by SICK AG, both programmable and non-programmable, can be utilized to create customized solutions through Lua applications, known as `SensorApps`. The SICK AppSpace, a commercial sensor application ecosystem,[7] enables the development of individualized `SensorApps`. These `SensorApps` are specific software applications that are developed to work with sensors. Non-programmable sensors can also be integrated into sensor apps by employing a Sensor Integration Machine (SIM) and various network protocols.[8]

The SICK `AppEngine` executes the `SensorApps` and operates on programmable sensors and SIMs.[9] Beyond execution, the SICK `AppEngine` plays an essential role by providing an infrastructure for `SensorApps`, including low-level interfacing with sensor hardware and network communication with other devices. One important feature of this technology is "Common Reusable Objects Wired by Name" (CROWN), essentially serving as APIs by `SensorApps` and the `AppEngine`. CROWNs simplify the creation of more complex applications through dependency injection of `SensorApps`.

Using this technology, the following two applications were developed. These applications are the cases for our evaluation.

D.4: Barcode reader application

The BarcodeReaderApp was developed from sample apps. It consists of 7 commits, involving a total of 862 added lines and 283 removed lines across one to four files. The BarcodeReaderApp example combines existing SICK AppSpace samples to form a component-based architecture. Thus, it serves as a realistic demo application for the industrial Internet of Things (IoT) setting (Burgey 2023). The application involves capturing images with a camera module, detecting barcodes in these images, and sending the barcode information to a database via HTTP. The four related SICK AppSpace samples include a barcode scanner app, an HTTP client app, an HTTP server app, and a database app. While the barcode scanner app detects barcodes in images, the HTTP client app submits the information to a web server. The HTTP server app receives the information and forwards it to the database app, which inserts the information into a database and provides a web interface for users to view the scanned barcodes. To simplify the setup, a directory image provider was used instead of a camera module. More details on the example can be found in Burgey (2023).

[7] See https://www.sick.com/de/de/sick-appspace/c/g555725?q=:Def_Type:ProductFamily.

[8] See https://www.sick.com/de/de/integrationsprodukte/sensor-integration-machine/c/g386451

[9] See https://www.sick.com/de/en/sick-appspace/sick-appspace-software-tools/sick-appengine/c/g547567

D.5: Object classifier application

The ObjectClassifierApp represents a real-world app involving the detection and sorting of objects from images based on color recognition. The considered development history spans 12 commits and encompasses a significant codebase adjustment, with 6,651 added lines and 2,663 removed lines across one to thirteen files. Unlike BarcodeReaderApp, this real-world scenario offers a more complex and dynamic testing ground for CIPM. More information about this real-world application can be found in Burgey (2023).

D.6: CoCoME

CoCoME (Heinrich et al. 2015) is a trading system for handling sales in a supermarket chain. It supports several sales processes at each store of the supermarket chain, such as scanning products at a cash desk, processing sales using a credit card or inventory reporting.

Goal: There are two goals of using CoCoME in the evaluation: CoCoME is a distributed system, which makes it suitable for evaluating the extraction of the system composition. Therefore, it is used for the evaluation of the accuracy of the `System Model` (G1). The second reason for using CoCoME is that several quality properties can be affected by the evolution of such distributed systems. Therefore, it is suitable for evaluating the accuracy of the performance prediction (G1) and the required monitoring overhead (G2).

CoCoME is also used for evaluating several approaches, such as iObserve (Heinrich 2020) and the proposed platform for empirical research on information system evolution (Heinrich et al. 2015). Similar to TeaStore, CoCoME is suitable for the evaluation of C3, C4, and C5 since it includes the required benchmark and a manually modeled PCM.

Structure and Technology: In our evaluation, we used the cloud variant of CoCoME,[10] where the enterprise server and the database are running in the cloud. This version of CoCoME is based on the Java Enterprise Edition (Java EE) and uses Maven to manage sub-projects and configure the deployment.

CoCoME consists of three main layers: the `GUI` layer containing components for ordering and reporting sales, the `Application` layer containing components for the main logic, and the `Data` layer containing components for the communication with the database. The architecture is modeled with Palladio (Heinrich et al. 2015).

E: Additional details on Experiment 1

In this section, we provide more details on E1 based on Burgey (2023). In particular, Table 10 includes details on the considered commits of the BarcodeReaderApp. Similarly, Table 11 includes details on the commits of ObjectClassifierApp.

[10] See https://github.com/CIPM-tools/cocome-cloud-jee-platform-migration.

Table 10 The commits of the BarcodeReaderApp

Commit Index	Hash	Added Lines	Removed Lines	Changed Files	Description
1	e25fb6b	575	0	4	Adding BarcodeReader, HTTP Client, and HTTP Server.
2	7126aab	204	0	1	Adding DatabaseAPI.
3	d92b459	76	81	3	Removal of irrelevant functionality from DatabaseAPI.
4	e6d87e0	4	1	1	Conditional to prevent empty barcode API submission.
5	542d2e9	2	0	1	Message added for empty API objects.
6	6b7b35f	1	21	2	Removal of serve calls from DatabaseAPI.
7	1f2fb08	0	180	1	Complete removal of DatabaseAPI from the application.

Table 11 The selected Git commit history of ObjectClassifierApp

Commit Index	Hash	Added Lines	Removed Lines	Changed Files	Short Description
1	f713180	1855	0	7	Import of existing app
2	1466c57	3	1	2	Minor text changes
3	f57823c	10	4	3	Bug fix
4	40bf36e	451	122	7	New features and improvements
5	473c9d9	1	1	1	Minor bug fix
6	995ecc0	505	187	7	LED and camera mode changes
7	956aeb9	448	234	7	Minor bug fix for version 2.6.0
8	0da3169	88	19	3	Features and bug fixes for version 2.7.0
9	00b44f8	60	44	3	UI improvements for version 3.0.0
10	88d005a	41	14	1	Features and bug fixes for version 3.1.0
11	92cb3bc	3188	2036	13	Code refactoring and new features
12	916fc52	1	1	1	Minor bug fix

F: Detailed results of E1 on lua-based applications

The reduction of instrumentation probes by E1 on the Lua applications are provided in both Tables 12 and 13, based on Burgey (2023).

Table 12 The reduction of instrumentation probes in E1 on the BarcodeReaderApp

Commit	Overall Reduction (%)	Fine-Grained Reduction (%)
2	56.2	90.0
3	27.3	52.9
4	47.1	88.9
5	51.4	94.7
6	66.7	100.0
7	68.8	100.0

Table 13 The reduction of instrumentation probes in E1 on the ObjectClassifierApp

Commit	Total Reduction (%)	**Fine-Grained Reduction (%)**
2	36.17	98.55
3	35.64	97.10
4	22.79	62.82
5	36.28	100.00
6	20.94	56.98
7	21.43	60.00
8	31.66	88.17
9	33.21	92.55
10	32.95	94.51
11	12.62	26.21
12	45.79	95.15

Acknowledgements This work was funded by the topic Engineering Secure Systems of the Helmholtz Association (HGF). Furthermore, this work was supported by KASTEL Security Research Labs as well as Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1608 - 501798263 and FeCoMASS 499241390. This work received funding from the HGF under the pilot program Core-Informatics. This publication is also partially based on the project SofDCar (19S21002) that the German Federal Ministry for Economic Affairs and Climate Action funds. Thanks to our textician.

Author Contributions Manar Mazkatli, David Monschein, and Martin Armbruster wrote the main manuscript. All authors reviewed the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

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

Adeagbo, M.: Towards a job recommender model: An architectural-based approach. Intl. J. Adv. Trends Comp. Sci. Eng. (2019). https://doi.org/10.30534/ijatcse/2019/94862019

Ananieva, S., Greiner, S., Kühn, T., et al.: A conceptual model for unifying variability in space and time. Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A (2020). https://doi.org/10.1145/3382025.3414955

Ananieva, S., Kühn, T., Reussner, R.H.: Preserving consistency of interrelated models during view-based evolution of variable systems. Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (2022). https://doi.org/10.1145/3564719.3568685

Armbruster, M., Mazkatli, M., Koziolek, A.: Recovering missing dependencies in java models. In: Softwaretechnik-Trends Band 43, Heft 4. Softwaretechnik-Trends, Gesellschaft für Informatik e.V. (2023). https://dl.gi.de/handle/20.500.12116/43231

Armbruster, M.: Commit-based continuous integration of performance models. Master thesis, Karlsruher Institut für Technologie (2021). https://doi.org/10.5445/IR/100015458810.5445/IR/1000154588

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

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

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

Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. J. Syst. Softw. **82**, 3–22 (2009). https://doi.org/10.1016/j.jss.2008.03.066

Brosig, F., Huber, N., Kounev, S.: 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, pp. 183–192. IEEE Computer Society, ASE '11 (2011). https://doi.org/10.1109/ASE.2011.6100052

Brosig, F., Kounev, S., Krogmann, K.: 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. In conjunction with the 4th Intl. Conference on Performance Evaluation Methodologies and Tools. ACM (2009)

Brun, C., Pierantonio, A.: Model differences in the eclipse modeling framework. UPGRADE, Eur. J. Inf. Prof. **9**(2), 29–34 (2008)

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

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

Burgey, L.: Continuous integration of performance models for lua-based sensor applications. Master thesis, Karlsruhe Institute of Technology (2023). https://doi.org/10.5445/IR/1000166032

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

Caculo, S., Lahiri, K., Kalambur, S.: Characterizing the scale-up performance of microservices using teastore. In: 2020 IEEE Intl. Symposium on Workload Characterization, pp. 48–59. IEEE (2020). https://doi.org/10.1109/IISWC50251.2020.00014

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

Cortellessa, V., Di Pompeo, D., Eramo, R., et al.: A model-driven approach for continuous performance engineering in microservice-based systems. J. Syst. Softw. **183**, 111084 (2022). https://doi.org/10.1016/j.jss.2021.111084, https://www.sciencedirect.com/science/article/pii/S0164121221001813

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

CWA-Server: Corona-warn-app server (2023). https://github.com/corona-warn-app/cwa-server

de Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: A survey. J. Syst. Softw. **85** (2012). https://doi.org/10.1016/j.jss.2011.07.036

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

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

Diskin, Z., Xiong, Y., Czarnecki, K., et al.: From state-to delta-based bidirectional model transformations: The symmetric case. In: Model Driven Engineering Languages and Systems: 14th Intl. Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings 14, pp. 304–318. Springer (2011)

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

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

Eclipse Foundation: Eclipse Modeling Framework (2024). https://eclipse.dev/modeling/emf/. Accessed 21 May 2024

Eismann, S., Bezemer, C., Shang, W., et al.: Microservices: A performance tester's dream or nightmare? In: Proceedings of the ACM/SPEC Intl. Conference on Performance Engineering. ACM (2020). https://doi.org/10.1145/3358960.3379124

Eismann, S.: Performance engineering of serverless applications and platforms. doctoralthesis, Universität Würzburg (2023) . https://doi.org/10.25972/OPUS-30313

El Hamdouni, A.-E., Seriai, A-D., Huchard, M.: Component-based architecture recovery from object oriented systems via relational concept analysis. In: CLA'10: 7th International Conference on Concept Lattices and Their Applications, pp. 259–270. University of Sevilla (2010). https://hal-lirmm.ccsd.cnrs.fr/lirmm-00531804/

Elsaadawy, M., Lohner, A., Wang, R., et al.: Dymond: dynamic application monitoring and service detection framework. In: Proceedings of the 22nd Intl. Middleware Conference: Demos and Posters. ACM (2021). https://doi.org/10.1145/3491086.3492471

Filho, O.F.F., Ferreira, M.A.G.V.: Semantic Web Services: A RESTful Approach. In: IADIS Intl. Conference WWWInternet 2009. IADIS (2009)

Flora, J., Gonçalves, P., Teixeira, M., et al.: My services got old! can kubernetes handle the aging of microservices? In: IEEE Intl. Symposium on Software Reliability Engineering Workshops. IEEE (2021). https://doi.org/10.1109/ISSREW53611.2021.00042

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

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

Grohmann, J., Eismann, S., Bauer, A., et al.: Sarde. ACM Trans. Auton. Adaptive Syst. **15**(2), 1–31 (2021). https://doi.org/10.1145/3463369

Hahner, S., Heinrich, R., Reussner, R.: Architecture-based uncertainty impact analysis to ensure confidentiality. In: 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 126–132 (2023). https://doi.org/10.1109/SEAMS59076.2023.00026

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

Heidenreich, F., Johannes, J., Seifert, M., et al.: 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, pp. 374–383. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-12107-4_25

Heinrich, R.: Architectural runtime models for integrating runtime observations and component-based models. J. Syst. Softw. **169** (2020). https://doi.org/10.1016/j.jss.2020.110722

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

Heinrich, R., Gärtner, S., Hesse, T., et al.: The CoCoME platform: A research note on empirical studies in information system evolution. Int. J. Softw. Eng. Knowl. Eng. **25**(09&10), 1715–1720 (2015). https://doi.org/10.1142/S0218194015710059

Horn, A., Fard, H., Wolf, F.: Multi-objective hybrid autoscaling of microservices in kubernetes clusters. In: Euro-Par 2022: Parallel Processing, pp. 233–250. Springer (2022). https://doi.org/10.1007/978-3-031-12597-3_15

Huyen, C.: Designing Machine Learning Systems: An iterative process for production-ready applications, 1st edn. O'REilly Media, INC, USA (2022)

IEEE: IEEE Standard for DevOps:Building Reliable and Secure Systems Including Application Build, Package, and Deployment, pp. 1–91. IEEE Std 2675-2021 (2021). https://doi.org/10.1109/IEEESTD.2021.9415476

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

Joy, B., Steele, G., Gosling, J., et al.: The Java Language Specification, 3rd edn. Addison-Wesley Reading (2000)

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

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

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

Klare, H., Kramer, M.E., Langhammer, M., et al.: Enabling consistency in view-based system development – The Vitruvius approach. J. Syst. Softw. **171** (2021). https://doi.org/10.1016/j.jss.2020.110815

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

Kolovos, D.S., Di Ruscio, D., Pierantonio, A., et al.: 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 (2009a). https://doi.org/10.1109/CVSM.2009.5071714

Kolovos, D.S., Paige, R.F., Polack, F.A.: On the evolution of ocl for capturing structural constraints in modelling languages. In: Abrial, J.R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis, Lecture Notes in Computer Science, vol. 5115, pp. 204–218. Springer Berlin Heidelberg (2009b). https://doi.org/10.1007/978-3-642-11447-2_13

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

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

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

Krogmann, K.: 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 (2012). https://doi.org/10.5445/KSP/1000025617

Krogmann, K., Kuperberg, M., Reussner, R.: Using genetic search for reverse engineering of parametric behaviour models for performance prediction. IEEE Trans. Software Eng. **36**(6), 865–877 (2010). https://doi.org/10.1109/TSE.2010.69

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

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

Liao, L., Chen, J., Li, H., et al.: Locating performance regression root causes in the field operations of web-based systems: An experience report. IEEE Trans. Software Eng. (2021). https://doi.org/10.1109/TSE.2021.3131529

Lucas, F.J., Molina, F., Álvarez, J.: A systematic review of uml model consistency management. Inf. Softw. Technol. **51**, 1631–1645 (2009). https://doi.org/10.1016/j.infsof.2009.04.009

Martin, J., Kandasamy, A., Chandrasekaran, K.: Crew: Cost and reliability aware eagle-whale optimiser for service placement in fog. Softw.: Pract. Exper. **50**(12), 2337–2360 (2020). https://doi.org/10.1002/spe.2896

Mazkatli, M., Armbruster, M., Koziolek, A.: Towards continuous integration of performance models for lua-based sensor applications. In: Herrmann A (ed) Softwaretechnik-Trends Band 43, Heft 4. Gesellschaft für Informatik e.V., Softwaretechnik-Trends, pp. 41–43 (2023). https://dl.gi.de/handle/20.500.12116/43232

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

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

Mazkatli, M.: Continuous integration of architectural performance models with parametric dependencies. Phd thesis, Karlsruhe Institute of Technology (KIT), Faculty of Informatics, Karlsruhe, Germany (2025). https://doi.org/10.5445/IR/1000181611

Mellit, A., Pavan, A.M.: A 24-h forecast of solar irradiance using artificial neural network: Application for performance prediction of a grid-connected pv plant at trieste, italy. Sol. Energy **84**(5), 807–821 (2010). https://doi.org/10.1016/j.solener.2010.02.006

Mémoli, F.: Gromov-wasserstein distances and the metric approach to object matching. Found. Comput. Math. **11**(4) (2011)

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

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

Monschein, D.: Enabling consistency between software artefacts for software adaption and evolution. Master thesis, Karlsruher Institut für Technologie (2020). https://doi.org/10.5445/IR/1000166031

Olsson, T., Ericsson, M., Wingkvist, A.: Motivation and impact of modeling erosion using static architecture conformance checking. In: 2017 IEEE Intl. Conference on Software Architecture Workshops. IEEE (2017). https://doi.org/10.1109/ICSAW.2017.15

Qayum, A., Zhang, M., Colreavy, S., et al.: A framework and taxonomy for characterizing the applicability of software architecture recovery approaches: A tertiary-mapping study. Softw.: Pract. Exp. **55**(1), 100–132 (2025). https://doi.org/10.1002/spe.3364

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

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

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

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

Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison Wesley Longman Publishing Co., Inc. (2003)

Sokolowski, D., Weisenburger, P., Salvaneschi, G.: Automating serverless deployments for devops organizations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM (2021). https://doi.org/10.1145/3468264.3468575

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

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

Spinner, S., Grohmann, J., Eismann, S., et al.: Online model learning for self-aware computing infrastructures. J. Syst. Softw. **147**, 1–16 (2019)

Stachowiak, H.: Allgemeine Modelltheorie. Springer-Verlag (1973)

Steinberg, D., Budinsky, F., Paternostro, M., et al.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)

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

TEAMATES-Git: Teammates developer web site (2022). https://github.com/TEAMMATES/teammates

TeaStore-Git: Teastore (2023). https://github.com/DescartesResearch/TeaStore. Accessed 12 Oct 2023

Torquato, M., Maciel, P., Vieira, M.: Pymtdevaluator: A tool for time-based moving target defense evaluation: Tool description paper. In: IEEE 32nd Intl. Symposium on Software Reliability Engineering. IEEE (2021). https://doi.org/10.1109/ISSRE52982.2021.00045

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

Vallée-Rai, R., Co, P., Gagnon, E., et al.: Soot: A java bytecode optimization framework. In: CASCON First Decade High Impact Papers. IBM Corp., Riverton, NJ, USA, CASCON '10, pp. 214–224 (2010). https://doi.org/10.1145/1925805.1925818 van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proceedings of the 3rd ACM/SPEC Intl. Conference on Performance Engineering, pp. 247–248. ACM (2012)

van Hoorn, A.: 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 (2014)

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

Viktorsson, W., Klein, C., Tordsson, J.: Security-performance trade-offs of kubernetes container runtimes. In: 2020 28th Intl. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 1–4. IEEE (2020). https://doi.org/10.1109/MASCOTS50786.2020.9285946

Voelter, M., Ratiu, D., Schaetz, B., et al.: Mbeddr: An extensible c-based programming language and ide for embedded systems. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, pp. 121–140. Association for Computing Machinery, New York, NY, USA, SPLASH '12 (2012). https://doi.org/10.1145/2384716.2384767

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

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

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

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

Weber, S., Weber, T., Henss, J.: Integration of performability-model extraction and performability prediction in continuous integration / continuous delivery, to be published in Softwaretechnik-Trends (2024)

Weng, F.: Generalisation of consistency rules between architectural and code models. Master's thesis, Karlsruhe Institute of Technology (KIT), Department of Informatics (2024)

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

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

Wittler, J.W., Saglam, T., Kuhn, T.: Evaluating model differencing for the consistency preservation of state-based views. J. Obj. Technol. **22**, 1–14 (2023)

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

Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Proceedings of ICSE 2007, Future of SE, pp. 171–187. IEEE Computer Society, Washington DC, USA (2007)

Xu, Y., Goodacre, R.: On splitting training and validation set: A comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning. J. Anal. Test. **2**(3), 249–262 (2018). https://doi.org/10.1007/s41664-018-0068-2

Authors and Affiliations

Manar Mazkatli[1] · **David Monschein**[2] · **Martin Armbruster**[1] · **Robert Heinrich**[3] · **Anne Koziolek**[1]

✉ Manar Mazkatli
manar.mazkatli@kit.edu

David Monschein
david.monschein@h-ka.de

Martin Armbruster
martin.armbruster@kit.edu

Robert Heinrich
robert.heinrich@uni-ulm.de

Anne Koziolek
koziolek@kit.edu

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

2 Institute of Data-Centric Software Systems, Karlsruhe University of Applied Sciences, Moltkestrasse 30, 76131 Karlsruhe, Germany

3 Institute of Software Engineering and Programming Languages, Ulm University, Albert-Einstein-Allee 11, 89069 Ulm, Germany