



Continuous Integration of Architectural Performance Models with Parametric Dependencies

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Manar Mazkatli

aus Aleppo (Syrien)

Tag der mündlichen Prüfung: 23. Januar 2025
Erste Gutachterin: Prof. Dr.-Ing. Anne Koziolek
Zweiter Gutachter: Prof. Dr. Wilhelm Hasselbring

Abstract

Incorporating the software architecture model during agile software development enables a better understanding of the rapidly evolving system and efficient prediction of quality attributes. In particular, Architecture-based Performance Prediction (AbPP) enables performance assessments across various what-if scenarios without the need for expensive measurements across all design alternatives, such as alternative workloads, designs, and deployments.

However, achieving accurate AbPP in agile software development presents significant challenges, requiring an up-to-date architectural Performance Model (aPM) parameterized over factors impacting performance. Frequent code changes within short development iterations can lead to architectural drift and erosion. This demands continuous updates to the aPM's structure, behavior, and performance parameters, such as resource demands. Additionally, adaptive operational changes— such as system composition adjustments and deployment reconfigurations— introduce further inconsistencies between the aPM and the running system, reducing the model's accuracy in representing the system architecture.

Accurate AbPP also necessitates accurate calibration of the performance parameters over influencing factors like usage-driven input data, hardware configurations, and workload characteristics. Ignoring parametric dependencies limits predictive power for unforeseen states and restricts the assessment of a broad range of what-if scenarios. While frequent calibration is often necessary due to ongoing changes, it can also introduce unnecessary monitoring overhead that may negatively impact system performance. Hence, efficient consistency maintenance between a parametrized aPM and the software system is essential for enabling accurate AbPP for proactive performance assessment, especially during agile software development.

Existing approaches struggle to update architecture models in response to all impactful changes, making the accuracy of AbPP uncertain. Even reverse engineering approaches that use static or dynamic analysis or a combination of both to extract up-to-date aPMs encounter challenges in maintaining AbPP accuracy, primarily due to the absence of automatic re-extraction after evolutionary and adaptive changes. Furthermore, frequent reverse engineering can be costly and may overwrite possible manual adjustments to the previously extracted model.

In this thesis, we present our approach, Continuous Integration of architectural Performance Models (CIPM), to support performance management during agile software development with AbPP. CIPM continuously updates architectural performance

models in response to evolutionary, adaptive, or usage changes. During development, it utilizes standard version control commits from the continuous integration (CI) pipeline to monitor source code changes and to update the aPM based on static analysis of these changes. After the continuous deployment (CD), CIPM calibrates affected performance parameters with parametric dependencies using measurements from adaptive monitoring targeting the changed parts of source code to reduce monitoring overhead. To enable adaptive monitoring, CIPM automatically instruments source code changes using specific configurable monitoring probes. Furthermore, CIPM introduces a self-validation process that ensures the accuracy of the updated aPM, manages the monitoring probes, and recalibrates any identified inaccuracies. At operation time, CIPM also continuously updates aPMs in response to operational changes, enhancing model accuracy.

As a result, CIPM automates the preservation of consistency between the aPM and the software system during both the development and operational phases throughout seamless integration into the CI/CD pipeline, keeping aPM up-to-date. This consistency supports a deeper understanding of evolving software systems and enables AbPP to proactively identify potential performance issues and assess design alternatives at minimal cost in preparation for the next evolutionary iteration.

Our evaluation follows mainly a measurement-based approach for validating aPMs at two levels [BR08]: Level I focuses on validating the accuracy of aPM and the associated AbPP by comparing performance predictions with actual performance measurements, while level II assesses the applicability of CIPM in terms of required monitoring overhead and scalability. In addition to empirical experiments at both levels, we furthermore conduct experiments validated CIPM by mining real software repositories containing approximately 18,000 commits. In our validation, we use a Java-based application, two benchmarks, and two industrial Lua-based sensor applications from SICK AG [AG24a].

Regarding accuracy, our findings show that CIPM effectively keeps the aPM up-to-date and accurately estimates performance parameters, thereby enabling accurate performance predictions. Besides, our experimental results indicate that calibrating the performance parameters considering the parametric dependencies significantly improves the predictive power of AbPP. Regarding monitoring overhead, CIPM's adaptive instrumentation significantly reduces the number of required probes, with reductions ranging from 12.6% to 69%, depending on the specific cases examined. Furthermore, adaptive monitoring can reduce up to 40% of the monitoring overhead, according to our experiments. Lastly, CIPM demonstrates reasonable execution times and scales well with the increasing number of model elements and monitoring data.

Zusammenfassung

Softwarearchitekturmodelle ermöglichen es agilen Teams, das System in Entwicklung besser zu verstehen und Qualitätseigenschaften effizient vorherzusagen. Insbesondere erlaubt Architektur-basierte Performance-Vorhersage (AbPV) die Durchführung von Performance-Bewertungen für verschiedene Szenarien, ohne dass teures Monitoring für alle Entwurfsalternativen, wie alternative Arbeitslasten, Entwürfe oder Deployments, erforderlich ist.

Allerdings ist eine Umsetzung von akkuraten AbPV in der agilen Softwareentwicklung mit erheblichen Herausforderungen verbunden, da ein stets aktuelles, auf performance-beeinflussende Faktoren parametrisiertes, architektonisches Performance-Modell (aPM) erforderlich ist. Häufige Quellcodeänderungen in kurzen Entwicklungszyklen können zu architektonischen Drifts und Erosionen führen. Dies verlangt eine kontinuierliche Aktualisierung der Struktur, des Verhaltens oder der Performance-Parameter des aPM, z.B. des Ressourcenbedarfs. Zusätzlich erzeugen adaptive Änderungen zur Laufzeit – etwa Anpassungen in der Systemzusammensetzung und Re-Konfigurationen der Deployment-Umgebung – weitere Inkonsistenzen zwischen dem aPM und dem laufenden Softwaresystem, wodurch die Genauigkeit des Modells bei der Repräsentation der Systemarchitektur sinkt.

Eine akkurate AbPV erfordert auch eine präzise Kalibrierung der Performance-Parameter über beeinflussende Faktoren wie Eingabedaten, Hardwarekonfigurationen und Arbeitslastcharakteristika. Das Ignorieren dieser Abhängigkeiten begrenzt die Vorhersagekraft für unvorhergesehene Zustände und schränkt die Bewertung eines breiten Spektrums an verschiedene Szenarien ein. Während häufige Kalibrierungen aufgrund häufiger Änderungen oft notwendig sind, können sie auch einen unnötigen Monitoring-Aufwand einführen, der sich negativ auf die Systemperformance auswirken könnte. Eine effiziente Konsistenzerhaltung zwischen einem parametrisierten aPM und dem Softwaresystem ist daher essenziell, um eine präzise AbPV zur proaktiven Performance-Bewertung während der agilen Softwareentwicklung zu ermöglichen.

Existierte Ansätze haben Schwierigkeiten, Architekturmodelle auf alle relevanten Änderungen hin zu aktualisieren, was die Genauigkeit der AbPV unsicher macht. Selbst Reverse-Engineering-Ansätze, die statische oder dynamische Analysen nutzen, um ein aktuelles aPM zu extrahieren, stoßen bei der Sicherung der AbPV-Genauigkeit auf Herausforderungen, insbesondere durch das Fehlen einer automatischen Neu-Extraktion nach evolutionären und adaptiven Änderungen. Darüber hinaus kann häufiges Reverse

Engineering kostspielig sein und möglicherweise manuelle Anpassungen am zuvor extrahierten Modell überschreiben.

In dieser Arbeit stellen wir unseren Ansatz, engl. Continuous Integration of architectural Performance Models (CIPM), vor, der das Performancemanagement während der agilen Softwareentwicklung mit AbPV unterstützt. CIPM ermöglicht die kontinuierliche Aktualisierung von Architektur-Performance-Modellen als Reaktion auf evolutionäre, adaptive oder nutzungsbezogene Änderungen. Während der Entwicklung nutzt es Standard-Commits der Versionskontrolle aus der Pipeline für kontinuierliche Integration"(engl. continuous integration (CI)), um Quellcodeänderungen zu überwachen und das aPM basierend auf einer statischen Analyse dieser Änderungen zu aktualisieren. Kontinuierliches Deployment (engl. continuous deployment (CD)) kalibriert CIPM die betroffenen Performance-Parameter mit parametrischen Abhängigkeiten durch Messungen aus adaptivem Monitoring, das durch das Beobachten der geänderten Teile des Quellcodes den Monitoring-Aufwand reduziert. Um adaptives Monitoring zu ermöglichen, instrumentiert CIPM automatisch Quellcodeänderungen mit spezifischen konfigurierbaren Monitoring-Probes. Weiterhin führt CIPM eine Selbst-Validierung ein, die die Genauigkeit des aktualisierten aPM sicherstellt, die Monitoring-Probes verwaltet und erkannte Ungenauigkeiten neu kalibriert. Zur Laufzeit aktualisiert CIPM außerdem kontinuierlich die aPMe als Reaktion auf Laufzeit-Änderungen und erhöht so die Modellgenauigkeit.

Als Ergebnis automatisiert CIPM die Konsistenzerhaltung zwischen dem aPM und dem Softwaresystem während der Entwicklungs- und Betriebsphase durch eine Integration in die CI/CD-Pipeline, wodurch das aPM stets auf dem neuesten Stand bleibt. Diese Konsistenz unterstützt ein tieferes Verständnis des Softwaresystems und ermöglicht es AbPV, potenzielle Performance-Probleme proaktiv zu identifizieren und Entwurfsalternativen mit minimalem Aufwand zu bewerten, um die nächste Entwicklungsiteration vorzubereiten.

Unsere Evaluierung basiert hauptsächlich auf einem messbasierten Ansatz zur Validierung der aPMen auf zwei Ebenen [BR08]: Ebene I konzentriert sich auf die Validierung der Genauigkeit des aPM und der zugehörigen AbPV durch den Vergleich von Performance-Vorhersagen mit tatsächlichen Performance-Messungen, während Ebene II die Anwendbarkeit von CIPM hinsichtlich des erforderlichen Monitoring-Aufwands und der Skalierbarkeit bewertet. Neben empirischen Experimenten auf beiden Ebenen führen wir außerdem Experimente zur Validierung von CIPM bei der Analyse echter Software-Repositories mit etwa 18.000 Commits durch. Für unsere Validierung verwenden wir eine Java-basierte Anwendung, zwei Benchmarks und zwei industrielle, Lua-basierte Sensoranwendungen von SICK AG [AG24a].

Unsere Ergebnisse zur Genauigkeit zeigen, dass CIPM das aPM effektiv aktuell hält und die Performance-Parameter präzise schätzt, wodurch genaue Performance-Prognosen ermöglicht werden. Darüber hinaus belegen unsere experimentellen Ergeb-

nisse, dass die Kalibrierung der Performance-Parameter unter Berücksichtigung der parametrischen Abhängigkeiten die Vorhersagekraft der AbPV erheblich verbessert. In Bezug auf den Monitoring-Aufwand reduziert die adaptive Instrumentierung von CIPM die Anzahl der erforderlichen Probes signifikant, wobei je nach spezifischem Fall eine Reduktion von 12,6 % bis 69 % erreicht wird. Zudem kann das adaptive Monitoring gemäß unseren Experimenten den Monitoring-Aufwand um bis zu 40 % verringern. Abschließend zeigt CIPM angemessene Ausführungszeiten und skaliert gut mit der zunehmenden Anzahl von Modellelementen und Monitoring-Daten.

Contents

Abstract i

Zusammenfassung iii

List of Figures xv

List of Tables xix

I. Preliminaries 1

1. Introduction **3**
1.1. Motivation 3
1.2. State of the Art 6
1.3. Problem Statement 7
1.4. Goals and Research Questions 9
1.5. Approach and Contribution 10
1.5.1. Context and Assumptions 10
1.5.2. Approach 11
1.5.3. Contributions 12
1.6. Application Scenario 13
1.6.1. Continuous Documentation 14
1.6.2. Software Performance Engineering 14
1.6.3. Extension and Reuse of Components 15
1.7. Thesis Structure 15

2. Background **17**
2.1. DevOps 17
2.1.1. Definition 18
2.1.2. Monitoring 18
2.1.3. Kieker Monitoring Tool 20
2.1.4. Measurement-based Performance Assessments 21
2.2. Modeling Terms and Standards 22
2.2.1. Model Definition 23

2.2.2. Metamodel Definition . . . 23
2.2.3. Meta Object Facility . . . 24
2.2.4. Eclipse Modeling Framework . . . 24
2.2.5. Tools for Defining Textual Domain-Specific Languages . . . 25
2.2.6. Source Code Model . . . 26
2.2.7. Model Differences . . . 27
2.3. Software Architecture . . . 28
2.3.1. Architecture Model . . . 29
2.3.2. Software Components . . . 30
2.3.3. Microservices . . . 30
2.4. Palladio Component Model . . . 31
2.4.1. Repository Model . . . 31
2.4.2. Behavior Model . . . 31
2.4.3. System Model . . . 32
2.4.4. Allocation Model . . . 33
2.4.5. Usage Model . . . 33
2.4.6. Resource Environment Model . . . 34
2.4.7. Performance Model Parameters . . . 34
2.5. Machine Learning Techniques . . . 35
2.5.1. Regression Analysis . . . 35
2.5.2. Decision Tree . . . 36
2.5.3. Genetic Algorithm . . . 36
2.5.4. Feature Selection . . . 37
2.6. Consistency Maintenance . . . 37
2.6.1. Consistency Definition . . . 37
2.6.2. Fine-grained Formalization of Consistency Relations . . . 38
2.6.3. Vitruvius Platform . . . 41
2.6.4. Consistency Preservation Rules . . . 42
2.6.5. Co-Evolution Approach . . . 43
2.6.6. iObserve Approach . . . 43
2.7. Evaluation Foundation . . . 44
2.7.1. Evaluation Levels . . . 44
2.7.2. Classification of Software Engineering Research . . . 45
2.7.3. Research Strategies: Case Study and Experiments . . . 46
2.7.4. Goal Question Metric . . . 47
2.8. Evaluation Metrics . . . 47
2.8.1. F-score . . . 48
2.8.2. Jaccard Similarity Coefficient . . . 49
2.8.3. Kolmogorov-Smirnov Test . . . 50
2.8.4. Wasserstein Distance . . . 51

3. Motivation Example **53**
3.1. Background 53
3.2. The Performance Management at Dev-Time 55
3.3. The Performance Management at Ops-Time 56
3.4. The Importance of Architectural Performance Models 57

II. Continuous Integration of Performance Models **59**

4. Consistency Preservation between Software Artifacts **61**
4.1. Consistency Maintenance 63
4.1.1. Consistency Definition 63
4.1.2. Consistency-Affecting Changes 65
4.1.3. Characteristics of Consistency Relations 66
4.1.4. Consistency Preservation 68
4.2. Models to Keep Consistent 69
4.2.1. Architectural Performance Model 69
4.2.2. Measurement Model 70
4.3. Model-based DevOps Pipeline 71
4.3.1. Development Stage of the MbDevOps Pipeline 72
4.3.2. Operation Stage of the MbDevOps Pipeline 74
4.4. Assumptions and Limitations of MbDevOPs Pipeline 75
4.4.1. Assumptions 75
4.4.2. Limitations 76
4.5. Discussion 76

5. Consistency Preservation at Development Time **79**
5.1. CI-based Update of Software Models 80
5.1.1. Overall Process 81
5.1.2. Realization of CI-based Source Code Model Update 87
5.1.3. Realization of CI-based Repository Model Update 92
5.1.4. Realization of CI-based Instrumentation Model Update 94
5.1.5. Realization of CI-Based System Model Update 97
5.2. Adaptive Instrumentation 101
5.2.1. Running Example 102
5.2.2. Overall Process 104
5.3. Realization of the Adaptive Instrumentation 107
5.4. Measurement Metamodel 107
5.5. Assumptions and Limits 108
5.6. Discussion 111

6. Incremental Calibration of Performance Model Parameters 115
6.1. Running Example 116
6.2. Overall Process 117
6.3. Calibrating Loop Actions 119
6.4. Calibrating Branch Transitions 120
6.5. Calibrating External Call Parameters and Return Values 120
6.6. Calibrating Resource Demands 121
6.6.1. Basis: Non-Incremental Resource Demand Estimation 121
6.6.2. Incremental Estimation of Resource Demands 122
6.6.3. Identification of Parametric Dependencies for Resource Demands 124
6.7. Adaptive Optimization of Parametric Dependencies 125
6.8. Self-Validation at Development Time 126
6.9. Assumptions and Limitations 126
6.9.1. Assumptions 127
6.9.2. Limitations 128
6.10. Discussion 128

7. Consistency Preservation at Operation Time 131
7.1. Overall Process 132
7.2. Monitoring 134
7.3. Ops-time Calibration 135
7.3.1. Pre-processing of Ops-time Calibration 136
7.3.2. Ops-time Updating of Resource Environment 137
7.3.3. Ops-time Updating of System Model 139
7.3.4. Ops-time Updating of Allocation Model 140
7.3.5. Ops-time Updating of Repository Model 140
7.3.6. Ops-time Updating of Usage Model 142
7.3.7. Ops-time Finalization 142
7.4. Self-Validation 143
7.5. Assumptions and Limitations 144
7.5.1. Assumptions 144
7.5.2. Limitation 145
7.6. Discussion 147

III. Validation and Discussion 149

8. Validation Strategy 151
8.1. Goal-oriented Evaluation 151
8.1.1. Evaluation Goals 152
8.1.2. Evaluation Questions 154

8.1.3. Evaluation Metrics . . . 156
8.2. The Classification Scheme of CIPM Contributions . . . 161
8.2.1. C1: Automated Consistency Maintenance at Dev-Time . . . 161
8.2.2. C2: Adaptive Instrumentation . . . 162
8.2.3. C3: Incremental Calibration with Parametric Dependencies . . . 164
8.2.4. C4: Ops-Time Calibration . . . 165
8.2.5. C5: Self-Validation . . . 165
8.2.6. C6: MbDevOps Pipeline . . . 166
8.2.7. Summary of Contributions Classification . . . 168
8.3. Case Study . . . 170
8.3.1. Case Study Process . . . 171
8.3.2. Case Selection . . . 172
8.3.3. TeaStore . . . 173
8.3.4. CoCoME . . . 178
8.3.5. TEAMMATES . . . 179
8.3.6. Sick Sensor Applications . . . 181
8.3.7. Artificial Examples . . . 183
8.4. CIPM Tool . . . 184
8.5. Experiments . . . 185

9. Validation of Consistency Preservation at Development Time . . . 187
9.1. Experiment1: CI-based Update of the Performance Model . . . 187
9.1.1. Goal . . . 188
9.1.2. Scenario . . . 189
9.1.3. Setup on TeaStore . . . 190
9.1.4. Setup on TEAMMATES . . . 193
9.1.5. Setup on Lua-based Applications . . . 195
9.1.6. Results . . . 196
9.1.7. Discussion . . . 205
9.1.8. Limitation . . . 207
9.1.9. Threats of Validity . . . 208
9.2. Experiment2: System Model Update at Development Time . . . 209
9.2.1. Goal . . . 209
9.2.2. Scenario . . . 210
9.2.3. Setup . . . 210
9.2.4. Results . . . 211
9.2.5. Discussion . . . 212
9.2.6. Limitation . . . 212
9.2.7. Threats of Validity . . . 213
9.3. Summary . . . 213

10. Validation of Incremental Calibration . . . 215
10.1. Experiment3: Incremental Calibration of Performance Model . . . 215
10.1.1. Goal . . . 215
10.1.2. Scenario . . . 216
10.1.3. Setup on Example1 . . . 218
10.1.4. Setup on Example2 . . . 219
10.1.5. Setup on CoCoME . . . 220
10.1.6. Setup on TeaStore . . . 220
10.1.7. Results . . . 222
10.1.8. Discussion . . . 229
10.1.9. Limitation . . . 231
10.1.10. Threats of Validity . . . 231
10.2. Summary . . . 232

11. Validation of Consistency Preservation at Operation Time . . . 233
11.1. Experiment 4: Updating Performance Models during Operation . . . 233
11.1.1. Goal . . . 233
11.1.2. Scenario . . . 234
11.1.3. Evaluation of Performance Prediction . . . 234
11.1.4. Evaluation of Model Accuracy . . . 235
11.1.5. Setup on CoCoME . . . 237
11.1.6. Setup on TeaStore . . . 237
11.1.7. Results . . . 238
11.1.8. Discussion . . . 246
11.1.9. Threats of Validity . . . 247
11.2. Experiment 5: Updating Performance Models with Increasing Monitoring Data . . . 248
11.2.1. Goal . . . 249
11.2.2. Scenario . . . 249
11.2.3. Setup for Repository Model . . . 250
11.2.4. Setup for System Model and Allocation Model . . . 251
11.2.5. Setup for Usage Model . . . 251
11.2.6. Setup for Environment Model . . . 252
11.2.7. Results . . . 252
11.2.8. Discussion . . . 255
11.2.9. Limitation . . . 256
11.2.10. Threats of Validity . . . 257
11.3. Summary . . . 258
11.3.1. Summery of Experiment4 . . . 258
11.3.2. Summery of Experiment5 . . . 258

IV. Reflection **259**

12. Related Work . . . **261**
12.1. Performance Assessment in Agile Development . . . 261
12.1.1. Application Performance Monitoring and Management . . . 261
12.1.2. Performance Testing . . . 263
12.1.3. Performance Assessment during DevOps . . . 264
12.2. Modeling during Agile Development . . . 265
12.3. Consistency Management Approaches in Software Development . . . 266
12.3.1. Consistency Management at Dev-time . . . 267
12.3.2. Consistency Management at Ops-time . . . 269
12.3.3. Hybrid Approaches . . . 270
12.4. Instrumentation Approaches . . . 271
12.5. Resource Demand Estimation Approaches . . . 272
12.6. Parametric Dependencies . . . 274

13. Outlook . . . **277**
13.1. Future Work . . . 277
13.1.1. Investigation the Cost and Trust in CIPM . . . 277
13.1.2. Version Controlling within CIPM . . . 278
13.1.3. Generalizing Consistency Preservation Rules for CIPM . . . 279
13.1.4. Improvement of Scalability . . . 280
13.1.5. Enhancing CIPM with Round-Trip Engineering . . . 280
13.1.6. Performance Analysis for Automobile Applications . . . 281
13.1.7. Support Diverse Version Control Systems . . . 281
13.1.8. Adaptive Performance Testing for MbDevOps Pipeline . . . 282
13.1.9. Expanding the Consistency Maintenance Range of CIPM . . . 283
13.1.10. Enhancing the Abstract Behavior . . . 284
13.1.11. Combining Static and Dynamic Analysis for Incremental Calibration . . . 286
13.1.12. Multi-Resource Demand Calibration for Internal Actions . . . 286
13.2. Conclusions . . . 287

List of Acronyms . . . **289**

Bibliography . . . **293**

Miscellaneous . . . **317**

Appendix 323

A. Appendix **323**
A.1. Palladio Metamodel 324
A.1.1. Repository Model 324
A.1.2. System Model 324
A.1.3. Allocation Model 324
A.1.4. Resource Environment Model 327
A.2. TeaStore case study 328
A.2.1. Git History 328
A.3. Example of the Adaptive Instrumentation 337
A.4. Teammates case study 339
A.4.1. Git Histoy 339
A.5. Lua-based sensor applications 341
A.5.1. Git History of BarcodeReaderApp 341
A.5.2. Git History of ObjectClassifierApp 341

List of Figures

1.1. Overview on CIPM approach and the main actors. Red boxes show the problems that CIPM addresses. The lower part of the figure abstracts some aspects of aPM (architecture and deployment on the right side and the behavior on the left side) using the TeaStore example [Kis+18] (c.f. Chapter 3) and the annotations of Palladio component model [Reu+16] (c.f. Section 2.4). 12
1.2. Relationships between problems, research questions, and scientific contributions of this thesis. 14

2.1. An example of the system model. 32
2.2. An example of allocation model and resource environment model, adjusted from an example in [Reu+16]. 33
2.3. Feature model explaining the classification schema of consistency relations, from [Küh+23]. 40
2.4. Classification of software engineering research from [Kap+21b]. 46
2.5. The intersection of two sets at the left side and the union of them at the right side. 50
2.6. The KS-test: the maximum absolute difference between two cumulative distributions. 51

3.1. Microservice-based architecture of the TeaStore [Kis+18]. 54
3.2. Excerpt of TeaStore's architecture with notations from `Palladio` (Section 2.4). 55

4.1. Excerpt of the measurement metamodel. 71
4.2. Model-based DevOps pipeline. 72

5.1. CI-based Update of the aPM. 82
5.2. Excerpt from the Java metamodel highlighting parts to represent interfaces, classes, and their members, from [Maz+25]. 89
5.3. Excerpt from the Java metamodel highlighting parts to represent interfaces, classes, and their members, from [AMK23] based on [Bur23]. 90
5.4. Excerpt from the Recommender Java model, from [Maz+25]. 91

5.5. Excerpt of the source code and JaMoPP model for the Recommender component in the TeaStore case study. 91
5.6. Simplified example for the CI-based update of the Repository Model according to change in the Java model, created based on [Maz+25]. . . 95
5.7. Instrumentation metamodel for providing the required measurements to calibrate Palladio Component Model (PCM). 96
5.8. The metamodel of Service-Call-Graph from [Mon20]. 97
5.9. An excerpt of TeaStore Service-Call-Graph (SCG), based on [Mon+21]. 98
5.10. Simplified excerpt of the extracted TeaStore System Model. 100
5.11. PlaceOrder SEFF. 102
5.12. Extension of Instrumentation Model for providing instrumentation code for monitoring Tools. 106
5.13. The measurements metamodel. 113

6.1. The incremental calibration process. 117
6.2. Learning the parametric dependencies. 118

7.1. The process of Ops-time calibration and self-validation. To enhance readability, data flows between sub-processes are omitted. 133
7.2. Collecting and processing monitoring data in intervals over time: window size defines interval duration, trigger time sets start and sliding interval. 136
7.3. Visualization of monitoring trace connected based on the caller execution identifier. 138

8.1. Goals, evaluation questions, and metrics employed to validate the CIPM approach. Gray-highlighted goals and their corresponding evaluation questions are explored through a survey. 152
8.2. Classification of the first contribution: CI-based update of software models. 162
8.3. Classification of the second contribution: automatic adaptive instrumentation. 163
8.4. Classification of the third contribution: incremental calibration of performance models with parametric dependencies. 165
8.5. Classification of the fourth contribution: Ops-time calibration of performance models. 166
8.6. Classification of the fifth contribution: self-validation of performance models. 167
8.7. Classification of the sixth contribution: MbDevOps Pipeline. 168
8.8. Microservice-based architecture of the TeaStore [Kis+18]. 174
8.9. The manually modeled Repository Model of TeaStore. 176

8.10. The abstract behavior of `ConfirmOrder` service using `Palladio` annotation (SEFF diagram). . . . 177
8.11. The abstract behavior of `train` service based on [Maz+20]. . . . 178
8.12. The abstract behavior of bookSale service. . . . 179
8.13. High Level Architecture of TEAMMATES [Kur24]. . . . 180
8.14. The simple structure of example1. . . . 183
8.15. The behavior of `service1` using `Palladio` SEFF diagram. . . . 183

9.1. GQM plan of Experiment1. . . . 188
9.2. The scenario of Experiment1. . . . 189
9.3. On the left side, the relation between the number of affected lines in TeaStore and the update time. On the right side, the relation between the number of Vitruvius changes and the update time. . . . 199
9.4. On the left side, the relation between the number of affected lines in TEAMMATES and the update time. On the right side, the relation between the number of Vitruvius changes and the update time. . . . 201
9.5. On the left side, the relation between the number of affected lines in BarcodeReaderApp and the update time. On the right side, the relation between the number of Vitruvius changes and the update time. . . . 204
9.6. On the left side, the relation between the number of affected lines in ObjectClassifierApp and the update time. On the right side, the relation between the number of Vitruvius changes and the update time. . . . 205
9.7. GQM plan of Experiment2. . . . 209
9.8. The scenario of Experiment2. . . . 210
9.9. Excerpt of the extracted CoCoME SCG, adjusted from [Mon20]. . . . 211
9.10. Simplified excerpt of the extracted CoCoME `System Model`. . . . 211

10.1. GQM plan of Experiment3. . . . 216
10.2. The scenario of Experiment3. . . . 217
10.3. Cumulative Distribution Functions (CDFs) for the response time of "bookSale" service. . . . 224
10.4. The Kolmogorov-Smirnov-test (KS-test) result and the absolute Wasserstein distance (WS-distance) between the monitoring data and the simulation results (100 repetitions) for CoCoME's "bookSale" service. 226
10.5. The average values of KS-test for the iterative evolution of TeaStore are presented on the left side, while the average values of WS-distance are shown on the right side. . . . 228

11.1. GQM plan of Experiment4. . . . 234
11.2. The scenario of Experiment4. . . . 235
11.3. The results of Experiment4 on the CoCoME based on [Mon20]. . . . 239

11.4. CDF of response time of “bookSale” service, from [Mon20]. 241
11.5. The results of the first stage of Experiment4 on the TeaStore based on [Mon20]. 242
11.6. Median Wasserstein distance of the forward and backward prediction for a model derived at a given point in time - regarding the response times of TeaStores *confirmOrder* service, from [Mon+21]. 243
11.7. Median of the arising monitoring overhead over time when considering five minute intervals, adjusted from [Mon+21]. 245
11.8. GQM plan of Experiment5. 249
11.9. The scenario of Experiment5. 250
11.10. Exploration of the scalability of the $T_{RepositoryModel}$ that updates `Repository Model`, based on results from [Maz+25]. 253
11.11. Execution times of $T_{SystemModel}$ and $T_{AllocationModel}$ with an increasing number of changes in the system composition [Maz+25]. 253
11.12. Scalability analysis of the $T_{UsageModel}$ that updates `Usage Model`, based on results from [Maz+25]. 255
11.13. Scalability analysis of the $T_{ResourceEnvironmentModel}$ that updates `Resource Environment Model`, based on results from [Maz+25]. 255

A.1. Excerpt of Palladio repository metamodel. 324
A.2. Excerpt of Palladio system metamodel. 325
A.3. Excerpt of Palladio allocation metamodel. 325
A.4. Allocation class diagram with a reference to the repository model . . . 326
A.5. Excerpt of Palladio resource environment metamodel. 327

List of Tables

2.1. The confusion matrix for an information retrieving problem. 48

4.1. Classification of consistency relations within CIPM 68

8.1. Classification of C1: consistency maintenance at Dev-time 162
8.2. Classification of C2: automatic adaptive instrumentation 163
8.3. Classification of C3: incremental calibration. 164
8.4. Classification of C4: Ops-time calibration. 165
8.5. Classification of C5: Self-validation. 166
8.6. Classification of C6: MbDevOps pipeline. 168
8.7. Summary of contributions including their classification, the applied GQM plan, and validation methods. 169
8.8. The case studies used for validating the contributions. 173
8.9. Summary of experiments and validation. 185

9.1. TeaStore commits that are considered in the evaluation. 191
9.2. TEAMMATES commits that are considered in the evaluation. 195
9.3. Commits of Lua-based sensor applications that are considered in the evaluation. 195
9.4. Excerpt of Experiment1.1 results based on [Arm21]. Experiment1.1 applies sequential update of TeaStore Virtual Single Underlying Model (VSUM) in four intervals. 198
9.5. The results of Experiment1.2 based on [Arm21]. Experiment1.2 updates the VSUM of TeaStore considering the *aggregated* commits of the intervals I-IV. 198
9.6. The reduction of the instrumentation probes that the adaptive instrumentation achieved during applying Experiment1 on TeaStore, based on [Arm21]. 199
9.7. The result of applying Experiment1 to update the VSUM of TEAMMATES considering the aggregated commits of the intervals I-V, based on [Maz+25]. 200
9.8. The reduction of the instrumentation probes that the adaptive instrumentation achieved by applying Experiment1 on TEAMMATES, based on [Maz+25]. 200

9.9. Results of Experiment1 on Lua-based BarcodeReaderApp, based on [Bur23]. . . . 202
9.10. Results of Experiment1 on Lua-based ObjectClassifierApp, based on [Bur23]. . . . 202
9.11. The reduction of the instrumentation probes that the adaptive instrumentation achieved during Experiment1 on BarcodeReaderApp, based on [Bur23]. . . . 203
9.12. The reduction of the instrumentation probes that the adaptive instrumentation achieved during Experiment1 on ObjectClassifierApp, based on [Bur23]. . . . 204
9.13. Results for deriving the `System Model` at Development time (Dev-time) 212

10.1. The setup of Experiment3 on Example1: $M1$ represents the PCM of the source code from the first iteration, and $M2$ represents the PCM from the second one. . . . 218
10.2. Results of Experiment3 on Example1, based on data from [Jäg19]. $M1$ is the PCM related to the first iteration, $M2$ is the PCM related to the second one. . . . 223
10.3. Comparison of non-parameterized, parametrized and optimized models' predictive accuracy with monitoring data from [Von+20]. . . . 223
10.4. Comparison of the predictive accuracy of non-parameterized, parametrized, and optimized model of Example2 using KS-test and WS-distance. Aggregated results based on 100 simulations from [Von+20]. 224
10.5. Quartiles for the probability distributions of a single simulation and the monitoring for the "bookSale" service ignoring the outlier, based on [Maz+20]. . . . 225
10.6. Comparison of the accuracy metrics (KS-test and WS-distance) for CoCoME's "bookSale" with two reference models based on the comparison to monitoring data with and without outliers. . . . 225
10.7. KS-tests results conducted over the iterative evolution of TeaStore. . . . 227
10.8. WS-distance results conducted over iterative evolution of TeaStore. . . . 227
10.9. Comparison of the predictive accuracy of the parameterized (PM) and non-parameterized (NPM) models, aggregated over 100 simulation repetitions. The table shows the KS-test and WS-distance for the known load of 20 users and the unseen load of 40 users, based on [Maz+20]. . . . 229

11.1. Quartiles for the randomly selected probability distributions of a single simulation and the monitoring for the "bookSale" service, from [Mon20]. 241
11.2. Model accuracy during the second stage of Experiment4 that simulates adaptation scenarios on TeaStore. . . . 243

11.3. Aggregated accuracy metrics related to the second step of Experiment4 on TeaStore [Mon+21]. 244

12.1. An overview of the related work. 267

A.1. The four intervals and information on their Commits. The commits are numbered continuously to ease readability. The information in this table is extracted from [Arm21]. 328
A.2. The five intervals and information on their commits. The commits are numbered continuously to ease readability. 340
A.3. The commits of the BarcodeReaderApp. 341
A.4. The selected Git commit history of ObjectClassifierApp. 341

Part I.

Preliminaries

1. Introduction

This thesis discusses how to keep software artifacts consistent with an up-to-date architectural performance model, especially during agile software development. We propose the CIPM approach to automatically update, parametrize, and validate architectural performance models by addressing observed changes in the version controller system and in the running system using reasonable adaptive monitoring.

In Section 1.1, we motivate the importance of maintaining an up-to-date performance model in modern software development, emphasizing its role in enhancing the understanding of software architecture, supporting design decisions, and managing quality attributes. Then, we highlight the main issues of the state-of-the-art works to formulate the problems that this thesis addresses (Section 1.2). Building on this general motivation and review of the state of the art, we identify specific problem statements, focusing on the challenge of keeping the architectural performance model up-to-date, which is critical for effective performance management (Section 1.3). Then, we define a research goal along with corresponding research questions (Section 1.4). In Section 1.5, we introduce our approach and contributions aimed at addressing these problems. The scenario applications of our approach follow in Section 1.6. Finally, we outline the structure of this thesis in Section 1.7.

1.1. Motivation

Agile methodologies [Bec+01] are widely employed in the development of modern complex systems [Ull+22] since they facilitate iterative development of dynamic requirements considering user feedback [Bec+01]. These methodologies, like DevOps [BWZ15], often use continuous integration (CI) and continuous deployment (CD) to quickly integrate, test, deploy, and review source code changes. Although agile is widely applied, scaling agile practices to large-scale distributed scenarios poses challenges to a common understanding of the complex software system at Development time (Dev-time) and ensuring its quality at the Operation time (Ops-time).

Software architecture—the overall structure of a system defined by its components, the connections among them, and their mapping to the execution environment [Has18; Reu+16]— plays a pivotal role in agile software development. It ensures a shared vision and understanding of complex systems during agile software development for establishing a shared vision and understanding of complex software systems [Amb22]

and can serve as a basis for quality assurance of architectural decisions. In agile software development, software architecture is collaboratively evolved to adapt to project requirements [Amb22]. The dynamic and rapid nature of agile practices often leads to knowledge exchange during meetings [AM14; Our+18]. However, the documentation or formal modeling of software architecture is often underestimated, leading to outdated representations over the software lifecycle [Tai+23]. This complicates the understanding of complex software architecture, particularly during personnel turnover. The problem (P) of understanding the current software architecture while the system evolves is referred to as $P_{ArchUnderstanding}$.

An architecture's quality characteristics are critical for the overall success of software systems. Even so, ensuring functional requirements in agile are often prioritized over quality ones, which can lead to compromising quality aspects such as performance, security, and usability [Beh+20]. Design decisions that are implemented through continuous evolution and adoption of software systems may adversely affect the software quality, especially performance. Software systems that do not meet performance objectives can fail or not be used as expected [SW03, p. 172]. Despite the importance of performance, agile teams often use Application Performance Management (APM) [Heg+17] to monitor the current state of the system's performance at Ops-time and react to detected performance issues utilizing costly ad hoc ways, making adjustments throughout the implementation process [SW03]. We refer to the issue of costly processes, such as application performance management, as P_{Cost}.

To avoid costs stemming from performance issues at the Ops-time, Software Performance Engineering (SPE) incorporates engineering activities like performance testing and modeling, early at Dev-time to ensure performance requirements [WFP07a]. However, SPE, similar to general engineering practices for quality requirements, struggles to adapt to the short iteration cycles of agile development without incurring additional costs [Beh+20]. To illustrate, performance testing for evaluating design decisions includes setting up test execution environments, executing the entire software system or specific components under anticipated workloads and conducting measurements for all design alternatives, including different deployments, designs, resource environments, and workloads [SW03; WFP07a]. Thus, conducting intensive performance tests during agile approaches' rapid iterative development cycles is also costly and may be impractical in CI/CD development due to the high frequency of releases [Beh+20] (P_{Cost}).

As an alternative to solely relying on measurements from production or test environments, SPE leverages predictive models to address performance issues early in the software development life cycle [WFP07a; SW03]. In particular, modeling system architectures abstractly from various viewpoints— such as structure, abstract behavior, and deployment— enables the analysis of performance characteristics through simulation [Reu+16]. Furthermore, architectural models enhance human understandability and

thus improve productivity in software development [OEW17]. These models also guide architectural decisions, ensuring continuous fulfillment of performance requirements [Reu+16; BHK11; SW03].

However, modeling is a time-consuming process in itself (P_{Cost}) and agile developers often do not trust performance models as they are approximations and challenging to validate [WFP07b]. We refer to the uncertainty of performance models' validity and accuracy in representing the current state of the real-world software system with $P_{Uncertainty}$.

Additionally, maintaining an up-to-date architectural Performance Model (aPM) throughout Dev-time and Ops-time is challenging, particularly in CI/CD environments [Tai+23]. Frequent changes in the source code during Dev-time can cause drift or erosion in the architecture [WA20], necessitating adjustments to the static structure or abstract behavior of aPMs ($P_{DevInconsistency}$). Similarly, adaptive changes during Ops-time, such as system composition and deployment modifications, also impact aPMs ($P_{OpsInconsistency}$). These changes often lead to inconsistencies between the software system and aPMs —referred to as $P_{Inconsistency}$ —reducing the models' effectiveness in representing the current architecture and complicating accurate AbPP. Ignoring these inconsistencies and simulating with outdated aPMs results in inaccurate performance predictions, thereby undermining the reliability of AbPP.

Furthermore, the accuracy of AbPP depends significantly on the estimated Performance Model Parameters (PMPs), such as resource demand. These parameters can rely on influencing factors varying over the Ops-time (e.g., usage profile and execution environment). The parameterization of PMPs over these factors allows AbPP for unseen states, for instance, unseen workloads. Ignoring the so-called *parametric dependencies* [BKR09] can lead to inaccurate AbPP for design alternatives, referred to as $P_{Inaccuracy}$. The parameterized PMPs may become inaccurate as the system evolves. Frequently re-estimating all PMPs after each impactful change introduces significant monitoring overhead, as PMPs are primarily calibrated through dynamic analysis of the entire system [Spi+15]. High monitoring overhead can impact system performance during operations [Bru+15], reducing measurement accuracy and slowing down the measurement process [RKH23], negatively affecting the accuracy of PMPs. We indicate the monitoring overhead problem as $P_{Monitoring-Overhead}$.

Given the challenges discussed above, maintaining consistency between the parametrized aPM and the continuously evolving software system in a CI/CD pipeline is costly and unsuitable for the rapid nature of Agile development. Therefore, the goal of the approach proposed in this dissertation is to reduce this cost by automating the update process of aPMs, enabling software performance engineering during agile development, and increasing the understandability of the software architecture.

In the following sections, we provide an overview of the limitations in the existing research and focus on approaches that maintain consistency between the software architecture and the evolving software system.

1.2. State of the Art

Managing shared architecture descriptions is a critical aspect of large-scale agile development [Kas+21]. Interpreting these shared objects as boundary objects can link individual teams to a shared vision of the entire system, leading to better knowledge management at scale. However, one of the main challenges in software engineering is the degradation of artifacts, particularly architecture models and requirements [Woh+19]. Therefore, maintaining an up-to-date architectural model is essential to ensure continuous alignment between evolving requirements and architectural decisions.

Several approaches have been proposed to partially automate the consistency maintenance between software artifacts and architectural models, addressing $P_{ArchUnderstanding}$ to some extent. These approaches can be divided into two main categories (A1 and A2):

The first category (A1) includes approaches that reverse-engineer the current architecture based on static analysis of source code [AAM10; Lan+16; Bec+10], dynamic analysis [BHK11; MHH11; Wal+17], or both [Kon18; Kro12; Lan17, p. 137]. These approaches suffer from four shortcomings: First, the accuracy of an extracted aPM is uncertain ($P_{Uncertainty}$) since not all impacting changes at Dev-time or Ops-time are observed and addressed ($P_{Inconsistency}$). Second, extracting and calibrating aPMs frequently would cause high monitoring overhead ($P_{Monitoring-Overhead}$), mainly if A1 approaches were used by iterative software development like DevOps, where the extraction of the entire model would have to be repeated after each iteration (or in larger intervals) and each adaptation at Ops-time. Third, the possible manual modifications of the extracted aPMs would be discarded by the subsequent extraction and should be repeated (P_{Cost}). Fourth, most of the existing work, with the exception of [Kro12; Gro+19], does not consider the parametric dependencies, resulting in inaccurate AbPP for design alternatives ($P_{Inaccuracy}$).

The second category (A2) includes approaches that maintain the consistency incrementally, either at Dev-time based on consistency rules [LK15; DM01; Det12; Voe+12; KP07; Buc+13] or at Ops-time based on dynamic analysis [Hei20; Spi+19; Hoo14a]. Although the incremental nature of A2 approaches lowers costs by preserving manual adjustments of models, they fail to update aPMs according to evolution and adaptation ($P_{Inconsistency}$). Thus, the accuracy of the resulting architecture model is still uncertain and cannot be trusted because the accuracy of aPMs and its AbPP are not provided ($P_{Uncertainty}$). This applies as well to approaches that estimate parametric dependencies to address $P_{Inaccuracy}$ (e.g., [Gro+21]).

1.3. Problem Statement

To address the problems identified in the motivation and state of the art, this section details these problems and their implications for software architecture and performance prediction during agile development processes.

$P_{ArchUnderstanding}$: Understanding the architecture of a complex system during agile development poses challenges:

- Agile developers primarily rely on informal documentation and exchange information through agile meetings [AM14; Our+18]. This approach can lead to a significant amount of knowledge being lost or not correctly transferred to others. Consequently, understanding the current system architecture, which is the first step for developers who join working on an unfamiliar system [Has18], would be challenging in the case of missing knowledge about architecture.

- The architecture models often become outdated as the project progresses, or they are not developed at all, resulting in a lack of architectural clarity [Tai+23].

P_{Cost}: Evaluating design alternatives in agile Software Development Life Cycle (SDLC) and suboptimal decisions are costly:

- Suboptimal design decisions can affect the quality of the software system and may lead to unexpected violations of the non-functional requirements [Bri+98; SW03]. Such violations cause high costs to resolve issues and maintain the system.

- Since agile developers usually apply design decisions throughout the development [SW03], they use APM to measure the current state of the software system. Evaluating design alternatives means implementing and measuring all of them. Moreover, APM cannot extrapolate the unforeseen states.

- Manual modeling and updating architecture for supporting design decisions and performance prediction are error-prone and expensive [Can+18].

- Automatic extraction of aPMs for applying AbPP overwrites the potential manual adjustments of the extracted model and does not reuse feedback from the previous model.

$P_{Uncertainty}$: Agile teams lack confidence in aPMs due to uncertainties in their accuracy:

- Manual models are approximations that may not reflect available implementations, and validating their accuracy is challenging [WFP07b].

- Existing works do not validate the accuracy of architecture models that can become outdated due to drift or erosion [WA20] along the SDLC, leading to uncertainty about how well they represent the system's current state.

- The automatic extraction of aPM also does not guarantee its validity for performance prediction, as the extraction process may not utilize all available resources, such as source code measurements. Besides, there is usually no observation of the state of the source code and running system, allowing the extraction of changes and the update of aPM.

$P_{Inconsistency}$: The continuous changes in development and operation can cause drift and erosion of aPM:

- The source code changes at Dev-time may implement an architecture that is not part of the system's intended architecture or violate the intended one [WA20] ($P_{DevInconsistency}$).

- Changes at Ops-time, such as alterations in deployment, system composition, or execution environments, lead to inconsistencies between the running system and the modeled operational architecture ($P_{OpsInconsistency}$).

$P_{Monitoring-Overhead}$: Frequent extraction and calibration of aPMs following dynamic changes result in monitoring overhead:

- Monitoring overhead can impact system performance during operations [Bru+15].

- Monitoring overhead reduces measurements' accuracy and slows down the measurement process [RKH23].

$P_{Inaccuracy}$: Accurate AbPP is challenging:

- The inconsistency between aPM and related software artifacts ($P_{Inconsistency}$) impacts the accuracy of AbPP.

- Parameter characterization and parametric dependencies' identification are challenging. The required overhead for the source code instrumentation, system monitoring, and aPM calibration is not negligible.

- Both the aPM and related PMPs become outdated due to dynamic changes during DevOps, resulting in inaccurate AbPP.

1.4. Goals and Research Questions

In the previous section, various issues were highlighted, stemming from inadequate architecture documentation and challenges in managing performance throughout the agile SDLC. This is because the source code evolves incrementally, and the software undergoes adaptations during its runtime. Consequently, the primary goal of our research revolves to the following:

> **Research Goal** Define a methodology for systematically integrating architecture models into agile software development and operations to enable accurate architecture-based performance prediction. This methodology should be able to automatically keep architectural performance models consistent with evolving software artifacts during both development and operations by updating them in response to software changes, ensuring reliability and scalability, while minimizing monitoring overhead. It should also enhance the trust in aPMs and ensure usability in modern agile software development without requiring specific development environments. The final goal is to enable AbPP for proactive issue identification and informed decision-making along SDLC.

This goal involves several sub-objectives. First, observing software systems to detect potential changes during development and operations and to update aPM accordingly. For this, we should consider the nature of modern software development, where developers evolve software source code and release it through CI pipelines to ease integration and testing of their changes while maintaining version history.

Second, we aim to ensure that our observations during the operations stage do not result in high monitoring overhead. Instead, they should be focused on identifying any changes in system composition, deployment, and production environments. This will allow us to update our aPM based on the detected changes.

Third, our goal is to enhance the trust in AbPP by validating the accuracy of the updated aPMs and related predictions. This should be based on real measurements to detect structural and prediction errors. The validation process should determine whether aPMs accurately reflect the behavior of the software system.

Fourth, reducing manual effort is another objective. This can be achieved by keeping manual changes and automating recalibration and self-validation, utilizing minimal monitoring overhead.

Our main objective is to devise a tool-supported method that ensures uninterrupted software engineering activities without delaying delivery and can even help accelerate the process. The method should be applicable for developers who may not be performance experts, visualizing the architect and performance. It should not be expensive and should be platform-agnostic, not requiring a specific development platform. This

tool should not cause high monitoring overhead and should be performant, while minimizing the time required for updating aPMs according to evolutionary changes and calibrating them.

Based on our research goals above, the primary focus of our research revolves around the following central research question:

> **Research Question** How can we accurately update and automatically validate a descriptive aPM with low overhead within CI/CD pipelines?

From the central research question, we derive the following sub-research questions:

RQ1 How can we efficiently update aPMs in response to continuously evolving source code within CI pipelines?

RQ2 How can we efficiently observe the required information about the running software system to update aPMs accordingly without introducing significant monitoring overhead?

RQ3 How can we calibrate the performance parameters with low monitoring overhead, considering parametric dependencies to support design decisions?

RQ4 How can we update aPMs according to adaptation changes in running software?

RQ5 How can we enhance trust in aPMs validity while reducing the required monitoring overhead?

RQ6 How can we efficiently integrate the continuous update process of aPMs with DevOps practices to support AbPP?

1.5. Approach and Contribution

In previous sections, we have identified specific problems within the agile SDLC and defined our research goal accordingly. From this, we derived also the research questions that this thesis addresses.

Subsequently, we summarize the context and assumptions underpinning our work in Section 1.5.1. Then, we provide an overview of our approach in Section 1.5.2. The contributions that address the defined research questions are in Section 1.5.3.

1.5.1. Context and Assumptions

In the context of the SDLC, particularly agile SDLC, we assume that developers use version control systems for CI of source code. Developers, the first actors, evolve system

features by adding or modifying code, while operators, the second actors, configure and deploy the running system.

To achieve our research goal and enable AbPP along SDLC, we assume that our approach has access to the source code repository. Additionally, we assume that an architecture modeling tool is available to model the system from different perspectives and to enable performance prediction. The required tool should enable modeling the software architecture from the following views [Reu+16]: a structural view, which includes a software repository encompassing components and their interfaces; a behavioral view, which abstracts behaviors to simulate system operations and predict performance metrics; and a deployment view, which illustrates the composition and assembly of components along with their allocation to resources environments. Based on such an architectural model, simulations can be conducted to predict system performance, such as response time and resource utilization. We also assume the presence of an architect, the third actor, who can use the tool to adjust the aPM and predict the performance of an alternative design without implementing it, such as evaluating an alternative deployment. This enables informed decision-making. Moreover, the model provides a comprehensive overview, detailing the software architecture. Figure 1.1 visualizes the three actors assumed to be available for applying our approach. Additionally, the lower part of Figure 1.1 illustrates an example of the views represented by the aPM

1.5.2. Approach

In this thesis, we present the Continuous Integration of architectural Performance Models (CIPM) approach [MK18; Maz+20]. This approach aims to maintain the consistency between the aPM and software artifacts (source code and measurements) along SDLC. CIPM targets agile SDLC to update aPMs automatically according to observed Dev-time and Ops-time changes ($P_{Inconsistency}$ and P_{Cost}).

At Dev-time (left side of Figure 1.1), developers integrate their source code into a CI repository. CIPM observes changes in a CI repository and updates aPMs based on predefined consistency rules [Maz+25; AMK23]. Moreover, CIPM plans what to monitor at the next phase (Ops-time) through automatic adaptive instrumentation of source code (P_{Cost}, $P_{Monitoring-Overhead}$) [Maz+25]. Then, CIPM calibrates PMPs with parametric dependencies ($P_{Inaccuracy}$) while reducing the required overhead through adaptive monitoring ($P_{Monitoring-Overhead}$) [Maz+20].

At Ops-time (right side of Figure 1.1), CIPM monitors the running system adaptively, extracts changes from measurements, and updates aPM accordingly. CIPM also provides a statement about the accuracy of the AbPP and automatically recalibrates inaccurate parts [Mon+21] ($P_{Uncertainty}$). In this way, the resulting aPMs can allow an accurate AbPP (P_{Cost}, $P_{Inaccuracy}$) and support understanding the system architecture ($P_{ArchUnderstanding}$), see the example in the lower part of Figure 1.1, which visualizes the software components, their interdependencies, and allocation on the right side, while

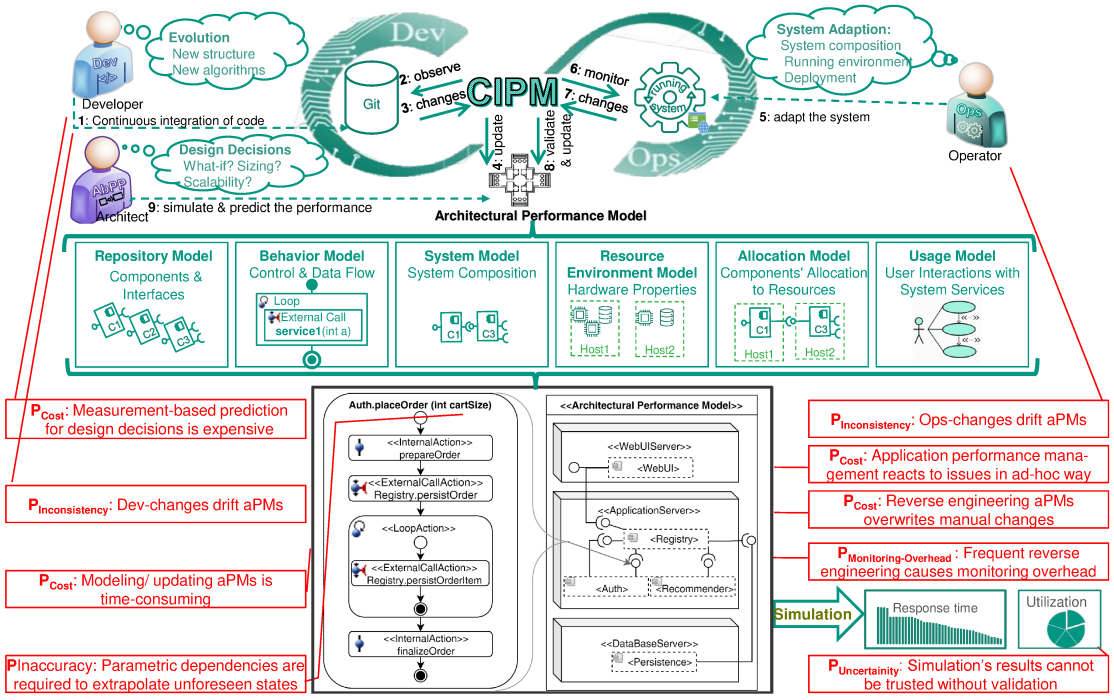


Figure 1.1: Overview on CIPM approach and the main actors. Red boxes show the problems that CIPM addresses. The lower part of the figure abstracts some aspects of aPM (architecture and deployment on the right side and the behavior on the left side) using the TeaStore example [Kis+18] (c.f. Chapter 3) and the annotations of Palladio component model [Reu+16] (c.f. Section 2.4).

the left side illustrates the abstract behavior of one service, depicting the calls between components. Such architecture supports architects to adopt proactive actions and to answer what-if questions about design alternatives.

1.5.3. Contributions

The contributions of our approach can be summarized as follows and shown in Figure 1.2:

C1 *Automated consistency maintenance at Dev-time:* For that, we propose a CI-based strategy (Section 5.1) to automatically update software models (e.g., aPMs) that are affected by the CI of the source code ($P_{Inconsistency}$). Unlike other methods, this approach uses standard version control commits as input, eliminating the need for specialized development editors to record source code changes and update aPMs accordingly. This facilitates easier integration into CI pipeline and minimizes overhead to keep the aPMs up-to-date with the latest codebase changes (P_{Cost}).

C2 *Automated adaptive instrumentation:* We propose CI-based, model-based instrumentation that targets the changed parts in source code (Section 5.2). 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}).

C3 *Incremental calibration:* We propose a novel incremental calibration of the PMPs based on adaptive monitoring [Maz+20]. Our calibration uses statistical analysis to learn about parametric dependencies. It also optimizes them using a genetic algorithm if necessary [Von+20]. Compared to existing approaches, CIPM can be performed at Ops-time and addresses P_{Cost}, $P_{Monitoring-Overhead}$, and $P_{Inaccuracy}$.

C4 *Automated consistency maintenance at Ops-Time:* The Ops-time calibration observes Ops-time changes based on dynamic analysis and updates the aPMs accordingly [Mon+21] (cf. Section 7.3). In comparison to existing approaches, CIPM automatically updates the aPMs including PMPs, system composition and resource environment using adaptive monitoring ($P_{ArchUnderstanding}$ and $P_{OpsInconsistency}$).

C5 *Self-validation of updated aPMs:* The self-validation (cf. Section 6.8) estimates the accuracy of AbPP compared with real measurements ($P_{Uncertainty}$). It manages the adaptive monitoring by activating and deactivating monitoring probes based on the validation results. This reduces the required overhead to the minimum ($P_{Monitoring-Overhead}$). According to our knowledge, our approach is the first approach that enables self-validation of aPMs and dynamic management of monitoring overhead.

C6 *Model-based DevOps pipeline:* The proposed pipeline [Maz+20] integrates and automates the CIPM activities to enable continuous AbPP during DevOps. To implement it, we designed and implemented a transformation pipeline [Mon+21] based on [Hei20] using the tee and join pipeline architecture [Bus98]. In contrast to existing pipelines, our pipeline automates consistency maintenance during the whole DevOps life cycle and enables AbPP.

1.6. Application Scenario

The CIPM approach aims to support various applications by providing an up-to-date performance model. In the following, we present some of the application scenarios:

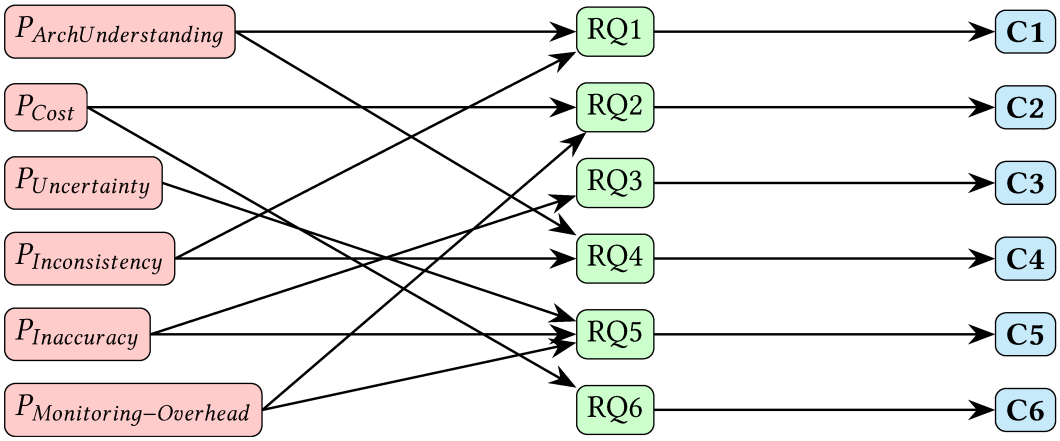


Figure 1.2: Relationships between problems, research questions, and scientific contributions of this thesis.

1.6.1. Continuous Documentation

CIPM facilitates sharing knowledge about the architecture and system performance characteristics. By updating the performance model at the architecture level alongside code changes or operational updates, the documentation of architecture changes remains up-to-date. This dynamic documentation aims to aid in system understanding and maintenance, providing developers and stakeholders with up-to-date insights into the evolving architecture and its performance implications. This, as a result, supports teams to make design decisions effectively throughout the software development lifecycle.

1.6.2. Software Performance Engineering

CIPM promises to improve performance management by integrating SPE activities at Dev-time and reducing monitoring overhead at Ops-time, thus considering performance throughout the software development lifecycle. With the predictive capabilities of CIPM, developers can simulate various scenarios and predict performance before implementing changes. This supports proactive optimization, helping teams avoid performance bottlenecks and make informed design decisions.

By continuously refining the performance model based on real-world data, CIPM also supports continuous performance management and optimization for operators. It enables model-based analysis of the performance, such as predicting the performance for unseen workload or supporting autoscaling, where CIPM can predict the performance based on the current workload before the dynamic adjustment of resources.

For performance engineering, CIPM can support a proactive approach to ensure that software remains efficient and reliable over time, enhancing its performance and

reliability. In the following, there are some performance engineering scenarios that CIPM works toward supporting:

Proactive Evaluation of Design Decisions CIPM enables proactive assessment of the planned design decision and the alternative designs based on the simulation of the aPM, aiming to anticipate and mitigate potential performance issues before they impact users.

Deployment Plan By providing updated performance models after each evolution, CIPM can assist in creating deployment plans that consider performance implications. Teams can then assess the impact of deployment strategies on system performance and make informed decisions to optimize deployment processes.

Sizing With up-to-date performance models, CIPM can aid in sizing infrastructure resources appropriately. Based on the simulation, teams can accurately estimate the resource requirements of the evolving system, optimizing resource allocation and avoiding over-provisioning or under-provisioning.

Design Optimization CIPM can support design optimization by providing insights into the performance implications of architectural decisions. Developers can experiment with different design alternatives and evaluate their impact on system performance, leading to more efficient and scalable designs.

1.6.3. Extension and Reuse of Components

For legacy software systems or component reuse scenarios, CIPM seeks to facilitate seamless integration of new functionalities while ensuring performance integrity. By continuously updating the performance model, teams can assess the impact of new components or changes on overall system performance, ensuring compatibility and efficiency.

These application scenarios demonstrate the expected benefits and value of the CIPM.

1.7. Thesis Structure

Part I. The first part introduces the preliminaries of the thesis. Chapter 2, provides background information on relevant concepts, including the continuous integration of performance models and consistency preservation between software artifacts. Chapter 3 introduces the running example and uses it to illustrate the problem of performance management at Dev-time and Ops-time.

Part II. The second part presents the CIPM approach, explaining the consistency preservation process for performance prediction: An introduction to the overall concept of consistency preservation between software artifacts is given in Chapter 4. It also introduces an overview of CIPM processes and how they can be integrated into DevOps pipeline for maintaining the consistency between software artifacts (**C6**). The detailed description of CIPM is divided according to the phase of the system's life cycle: Techniques for maintaining consistency during the development stage are described in Chapter 5, including the CI-based update of software models (**C1**) and the adaptive instrumentation of source code (**C2**). In Chapter 6, we describe the incremental calibration of performance models (**C3**) that can be executed at both Dev-time and Ops-time. The consistency preservation during system operation is explained in Chapter 7, which includes the Ops-time calibration of models (**C4**) and the self-validation of them (**C5**).

Part III. The fourth part presents the validation strategy and discusses the results. The overall validation approach is outlined in Chapter 8. The detailed validation and conducted Experiments (Es) for the contributions are also divided according to the phase of the system's life cycle: In Chapter 9, the evaluation of consistency preservation during Dev-time includes two experiments. The first experiment assesses the CI-based update of the performance model (**C1**) along with the adaptive instrumentation (**C2**), while the second experiment (**E2**) focuses on evaluating a specific aspect of **C1**: the system model update at development time. Chapter 10 validates the incremental calibration (**C3**), including experiment results for the incremental calibration of the performance model (**E3**). Chapter 11 provides the validation of the consistency preservation at Ops-time, including experiment **E4** to evaluate of the update process of performance models (**C4**) with self-validation (**C5**) during operation and experiment (**E5**) evaluating the scalability of the pipeline at Ops-time.

Part IV. The reflection part discusses the related works in Chapter 12 and provides an outlook, including future work, in Chapter 13.

2. Background

In this chapter, we introduce the foundations, upon which we build throughout this dissertation.

First, in Section 2.1, we introduce DevOps as an agile SDLC, which CIPM extends by incorporating new practices to enable AbPP and to reduce performance assurance costs P_{Cost} during DevOps SDLC.

Next, we define modeling terms in Section 2.2, as the main goal of our approach is to maintain consistency between software models, addressing $P_{Inconsistency}$.

Since the architectural performance model is one of the models we consider in consistency maintenance, we define software architecture in Section 2.3 and describe `Palladio`, the architectural performance model that CIPM updates, in Section 2.4. For calibrating performance models parameters of aPM considering parametric dependencies ($P_{Inaccuracy}$), we utilize machine learning techniques to detect and identify these dependencies based on some algorithms described in Section 2.5.

In Section 2.6, we discuss consistency maintenance between various software models, including concrete approaches for ensuring consistency between the code model, architecture model, and measurements, upon which CIPM relies.

Finally, in Section 2.7, we introduce aspects that we use for validating our approach. We then discuss evaluation metrics used to assess the effectiveness of our approach in Section 2.8.

2.1. DevOps

DevOps is a collaborative approach that combines software development (Dev) and IT operations (Ops) to streamline the software delivery process.

In Section 2.1.1, we provide a detailed definition of the DevOps approach. Then, in Section 2.1.2, we delve into one of the practices within DevOps, named monitoring, explaining its importance and characterizations. Monitoring enables measurement-based performance management at Ops-time. There is also a measurement-based performance assessment that can be utilized during the development, as we describe in Section 2.1.4. However, some of these assessments can be expensive and not suitable for DevOps.

2.1.1. Definition

According to an IEEE Standard for DevOps, the DevOps approach can be defined as follows: DevOps is a *"set of principles and practices which enable better communication and collaboration between relevant stakeholders for the purpose of specifying, developing, and operating software and systems' products and services, and continuous improvements in all aspects of the life cycle" [Soc21].*

A DevOps process consists of various stages and phases repeated periodically for software development and operation. These stages include planning, code development, building in a test environment, testing, releasing, deploying in a production environment, operating, and monitoring [CDM20]. Besides, the DevOps approach tends to automate the software lifecycle into an automated pipeline, in which the output of one process serves as an input for the next process [Soc21]. The automation is primarily achieved through continuous delivery, continuous integration, test automation, continuous deployment practices, and monitoring:

- **Continuous Delivery**: a *"software engineering practices that allow for frequent releases of new systems (including software) to staging or various test environments through the use of automated tools" [Soc21].*
- **Continuous Integration**: *"a technique that continually merges artifacts, including source code updates from all developers on a team, into a shared mainline to build and test the developed system" [Soc21].*
- **Continuous Deployment**: *"an automated process of deploying changes to production by verifying intended features and validations to reduce risk" [Soc21].*
- **Monitoring**: *Determining the status of a system, a process, or an activity [Soc21].*

The DevOps approach is well-suited for agile software development, as stated in the DevOps standard [Soc21]. So, the DevOps team begins implementing their systems by creating a continuous delivery pipeline that automates the entire workflow, from code management to deployment and operations. As a result, multiple developers integrate code continuously and utilize test automation frameworks. The continuous deployment in the production environment eases the operation.

Monitoring the system at Ops-time ensures a reliable and efficient development process [Bru+15; Soc21]. The monitoring concept is elaborated in Section 2.1.2, followed by an introduction to the `Kieker` monitoring tool utilized in our approach in Section 2.1.3. Lastly, measurement-based performance assessments are discussed in Section 2.1.4.

2.1.2. Monitoring

Monitoring is crucial in today's rapidly evolving business and development processes. It is particularly important in agile environments where continuous practices are common

[Bru+15]. Monitoring involves continuously observing systems to determine whether they are performing as expected and exhibiting the desired behavior.

Monitoring plays a crucial role in APM approaches [Heg+17; Bru+15], as it measures and analyzes the performance characteristics of the running system to ensure its performance.

Monitors are the tools used for collecting the measurements, which can be classified into two main categories. *Hardware monitor* collects low-level metrics from electrical signals and hardware registers. For instance, hardware monitors can capture characteristics of components such as the Central Processing Unit (CPU), memory, and hard disk drives. In contrast, *software monitor* collects the measurements from the running software based on software routines embedded within the target software. The *instrumentation* is the process of injecting this routine into the software system.

In the following, we provide an overview of the instrumentation process that is necessary for monitoring in Section 2.1.2.1. Then, we introduce monitoring triggers in Section 2.1.2.2. In Section 2.1.2.3, we discuss the characteristics essential for successful monitoring in Agile software development. Finally, we describe the monitoring tool utilized by CIPM in Section 2.1.3.

2.1.2.1. Instrumentation and Monitoring Techniques

Instrumentation is the process of incorporating software monitors into an application. The instrumentation enables the injection of software monitors into the application code at the source, object, or bytecode level, or into the underlying runtime environment, encompassing components such as the operating system, middleware, or application server [Hoo14b]. Thus, there are various techniques for instrumentation, including direct code modification, indirect code modification using aspect-oriented programming (AOP), compiler modification, and middleware interception [Bru+15].

Instrumentation can be classified as static if it occurs during design or compile time. It is also called dynamic instrumentation if it occurs at runtime without a system restart.

It is important to note that executing the instrumented application can influence the execution workflow for system characterization. This influence may affect the response times or CPU utilization due to the execution of the instrumentation code. This effect is referred to as *monitoring overhead*. Monitoring overhead is associated with the granularity of the instrumentation, aspects monitored, and the frequency of monitoring.

2.1.2.2. Monitoring Triggers

Monitoring can be categorized from trigger perspectives, either *event-driven* or *sampling-based* [Hoo14b]. Event-driven measurements capture measurements when specific events occur within the system, such as calls to software operations or exception

occurrences; a simple example is an event counter-update that counts how often the specific event occurs. A more detailed approach, tracing, captures more information, such as timestamps and transaction identifiers, for each event.

Alternatively, sampling-based measurements are set to execute at fixed intervals, gathering periodic snapshots of system performance independent of specific events, such as measuring the CPU utilization every minute [Bru+15].

2.1.2.3. Essential Monitoring Characteristics

Successful monitoring has four key characteristics [Sch18]. Firstly, it emphasizes the importance of monitoring the right aspects of the system. Monitoring should focus on relevant metrics and behaviors to ensure that the system is functioning correctly.

Secondly, the use of mathematics and statistical analysis is essential in processing the monitoring data effectively. By applying mathematical techniques, such as aggregating and analyzing data, it becomes easier to extract meaningful insights and understand the performance delivered by the system.

Thirdly, data retention is an important consideration. Although monitoring data is often discarded, the decreasing cost of data storage makes it more feasible to retain data for longer periods. Retaining monitoring data enables DevOps teams to learn from historical patterns and trends, leading to valuable insights and the ability to make better decisions.

Lastly, to achieve a successful monitoring system, it is essential to articulate what success looks like by using language to define clear goals and expectations. For instance, service-level indicators, service-level objectives, and service-level agreements play a crucial role in this process. Understanding the system at the service level. Besides, setting goals and utilizing histograms helps align monitoring efforts with business objectives.

By adhering to these characteristics, organizations can establish successful monitoring practices that enable them to ensure system performance, detect anomalies, and make continuous improvements.

2.1.3. Kieker Monitoring Tool

`Kieker` [HWH12a] is an open-source extensible framework that monitors distributed software systems for APM and architecture discovery.

The `Kieker` framework has two main parts: monitoring and analysis. The monitoring part gathers and saves measurements from software systems, while the analysis part examines these measurements.

On the monitoring side, `Kieker` allows defining custom *Monitoring Probes*. The monitoring probes are data structures to capture desired information. The monitoring probes collect measurements, which are called *Monitoring Records*. These records are

serialized and converted into a specific format by a monitoring writer, which and stores them in a monitoring log or stream.

On the analysis side, `Kieker` filters the monitoring data or captures additional information. For that, there are monitoring readers that read and interpret the relevant monitoring records to fill the customizable probes with the desired information. These records are then sent to analysis plugins to perform specific analyses and answer questions about performance. For example, `Kieker` allows regression analyses to predict upcoming performance issues. Moreover, an analysis of historical monitoring data enables the operator to detect the reason for a bottleneck issue. Besides, `Kieker` allows *tracing* the control flow, which supports extracting and visualizing architectural models.

Both the monitoring and analysis parts have controllers: the Monitoring Controller and the analysis controller. These controllers help manage and coordinate the monitoring and analysis processes.

With `Kieker`, developers can instrument their code statically and dynamically. The instrumentation of `Kieker` enables monitoring of specific data, such as method invocations, parameter values, and timing information. This data is recorded as execution traces, which can be analyzed to gain a deeper understanding of the system's behavior. `Kieker` provides an Instrumentation Record Language (IRL) [JHS13], which enables the description of specific probes, i.e., specific data structures for the monitored information.

`Kieker` supports multiple programming languages and frameworks, making it suitable for a wide range of application environments. `Kieker` also offers a variety of analysis and visualization tools to help users interpret the collected data and gain actionable insights.

Additionally, `Kieker` supports dynamic and adaptive monitoring to reduce the monitoring overhead. The term adaptive monitoring means that not all parts of the source code are monitored. `Kieker` can activate or deactivate probes during run time to allow dynamic monitoring. This flexibility enables users to focus their monitoring efforts on specific components or scenarios, reducing overhead and improving performance.

Due to the fact that `Kieker` is an open-source tool that allows customizable monitoring probes, tracing workflow, and adaptive monitoring, it is considered a suitable candidate for implementing the CIPM.

2.1.4. Measurement-based Performance Assessments

Measurement-based performance assessment aims to ensure that the software system meets its performance requirements. These assessments, part of the SPE discipline, include activities such as benchmarking and performance tests to evaluate and validate performance attributes.

Performance benchmarks are a key component of measurement-based approaches in Software Performance Engineering [SW03], contributing to the improvement of

successive system generations [Vie+12]. They assess metrics such as response time, throughput, and resource utilization.

Moreover, performance benchmarks can be integrated into CI environments, enabling their automation for consistent and efficient monitoring of performance over time. Waller et al. explore this integration, reporting on regression benchmarks for application monitoring frameworks and illustrating how automated performance benchmarks can be incorporated into CI setups to support DevOps practices [WEH15].

Performance tests execute a part or the complete software with different workloads to measure its performance [WFP07a]. Performance testing is one of these activities [Bon14]. The term "performance testing" means planning performance tests, defining the tests, modeling the required workload, executing the defined tests, analyzing the resulting test data and reporting the findings [Bon14]. Moreover, setting up a realistic test environment is required for performance testing. Hence, performance tests focus on assessing how a system performs under specific, real-world scenarios or varying conditions.

An intensive performance test is not a trivial process, it may take several hours [Bru+15]. Therefore, intensive performance testing is unsuitable for agile SDLC, which supports continuous deployment for quick releases like DevOps SDLC. In such SDLC, several deployments can be performed per day. Existing approaches suggest selecting the required performance tests dynamically and semi-automatically to enable usability for DevOps [Fer21].

2.2. Modeling Terms and Standards

Modeling is a general process used in various domains to represent, analyze, and understand complex systems or phenomena. In the Model-Driven Development (MDD) paradigm [Sel03], models are the primary development artifacts, unlike traditional processes where models play a secondary role, such as documentation. Hence, MDD considers that "*everything is a model* [Béz05]" and, therefore, represents all concepts and entities as models. For this purpose, MDD utilizes domain-specific language (DSL) to simplify the expression of domain concepts and transformation engines to convert models into other domain-specific languages or textual representations.

The term Model-Driven Software Development (MDSD) [SV06] refers specifically to the application of MDD in software development, focusing on the use of models to generate executable source code. While often used interchangeably, MDSD emphasizes automation in implementation through model transformations.

Model-Driven Engineering (MDE) generalizes the MDD paradigm by applying model-based techniques across a broader range of engineering activities beyond system development. It encompasses activities such as reverse engineering, modernization of legacy systems, and automation of system evolution tasks [RRM13]. In this broader context,

the principle that “everything is a model” [Béz05] remains central, requiring ensuring consistency and traceability across heterogeneous models.

To understand these concepts in more detail, we first define models and metamodels in Section 2.2.1 and Section 2.2.2. We then introduce a standard for metamodeling in Section 2.2.3 and discuss a modeling tool based on this metamodeling standard in Section 2.2.4. Tools for defining textual domain-specific languages are introduced in Section 2.2.5. These tools are applied in source code modeling, which is a central element of MDSD, as discussed in Section 2.2.6. To maintain consistency and enable integration across such models—particularly within the scope of MDE—model matching becomes essential. Therefore, in Section 2.2.7, we define model matching and examine relevant approaches.

2.2.1. Model Definition

The term model is "*an abstraction of reality according to a certain conceptualization*" according to Guizzardi [Gui05].

Stachowiak suggests in his work on general model theory [Sta73] that models must meet three criteria: abstraction, representation, and Pragmatics.

- **Representations** refers to the fact that the model represents some attributes of the original. This means the model exhibits a structural equivalence with another model, even if the individual features within the models are labeled differently. This phenomenon, known as isomorphism between model objects, allows for the establishment of a structural correspondence between the model and the original, which could also be a model itself. As a result, statements made about elements within the model can also be applicable to their corresponding real-world entities.
- **Abstraction** is a formal representation of entities and relationships in the real world. The abstraction excludes the unnecessary details that do not serve the purpose of the model to simplify the comprehensibility or analysis.
- **Pragmatics** means that the model can replace the original within certain time intervals under certain conditions for answering certain mental or actual operations.

Based on the aforementioned criteria of Stachowiak. The ***model*** can be defined as *formal representation of the real world (abstraction) with a certain correspondence (isomorphism) for a certain purpose (pragmatics)* [Gol11] via [Sta73].

2.2.2. Metamodel Definition

A metamodel is a higher-level abstraction that defines the structure and relationships of different types of models within a certain domain. In essence, a metamodel is also a

model. It precisely defines the model elements and the required rules for creating valid models.

The metamodel's primary function is to establish the framework for a model's structure, concentrating on its layout and relationships. However, it refrains from assigning meaning to individual elements or detailing their interactions in real-world contexts.

As a result, models qualify as instances of a metamodel when they conform to the structure prescribed by the metamodel.

The metamodel consists of an *abstract syntax* and at least one *concrete syntax*. The abstract syntax includes the model elements, their properties and the relation between them. The concrete syntax represents the abstract syntax textually or graphically.

Metamodels, in turn, are also defined using metametamodels. If two domain-specific languages (metamodels) are defined using a common metametamodel, then transformations at the metamodel level can be defined to transfer the models from one domain to another.

2.2.3. Meta Object Facility

Meta Object Facility (MOF) [OMG15] is a standardized *metamodel* from Object Management Group OMG (an international, open membership, not-for-profit technology standards consortium)[1]. The MOF is based on a four-layer architecture. These layers can be described as follows:

- M0 is the lowest layer that reflects the reality. For example, the object level in Unified Modeling Language (UML).
- M1 is the model layer that describes and classifies the structure and semantics of the reality in an abstract way, for example, the classes in the source code of the software.
- M2 is the metamodel layer that describes the structure and semantics of the model (the lower level), e.g., the UML class diagram of the software.
- M3 is the metametamodel layer that models the metamodel level. Similar to above, this layer is an abstraction of the lower one. This level should be able to describe itself. In other words, M3 should be described by the entities and rules of M3.

2.2.4. Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [Ecl24a][2] supports modeling and code generation. EMF unifies three technologies: Java, XML, and UML [BBM03].

[1] See `http://www.omg.org/`

[2] See `https://www.eclipse.org/modeling/emf/`

Ecore serves as the core metamodel in EMF, which is compatible with the essential MOF. Ecore is basically a subset of *UML* Class diagrams and can be defined in terms of itself. EMF provides an editor for creating and validating Ecore models. It allows them to be instantiated into *XML Metadata Interchange (XMI)* format [Wei09], which can be saved and reloaded again. Additionally, EMF supports code generation since it converts the models into *Java*. The standard Ecore metamodel facilitates the interoperability between EMF-based tools and applications.

2.2.5. Tools for Defining Textual Domain-Specific Languages

As aforementioned, domain-specific languages (DSLs) is important in MDD as they represent specific domains, enabling model transformation and analysis. DSLs textual representations are important because they provide an intuitive, human-readable format that is easier to edit, document, and review. Tools like EMFText and Xtext support the efficient creation and management of textual DSLs, offering integration with the Eclipse IDE for advanced editing capabilities and seamless model transformations, as described in Section 2.2.5.1 and Section 2.2.5.2.

2.2.5.1. EMFText

EMFText[3] utilizes standard Eclipse concepts, such as Ecore, to define the abstract syntax for DSL. Then, EMFText enables the definition of concrete syntax specifications (textual syntax) to outline how EMF model elements are represented in text form. Consequently, EMFText generates parsers and printers that facilitate the conversion between EMF-based models and their textual representations.

For instance, in reverse engineering, EMFText can be used to define a corresponding DSL for legacy configuration files with a custom format. By creating an Ecore metamodel and defining the concrete syntax that represents the current textual format, EMFText generates the necessary parser and printer. The parser reads the legacy configuration files and produces EMF models, facilitating further analysis and transformation. The printer then converts these EMF models back into the original textual format.

2.2.5.2. Xtext

Xtext [Ecl24b; Bet16] offers a comprehensive framework for developing DSLs with advanced editor features such as syntax highlighting, code completion, and validation. It combines the definition of both abstract and concrete syntax in a single grammar file. Then, Xtext can generate an Ecore model from the grammar or use an existing one.

Using the same example from the reverse engineering context mentioned with EMFText, Xtext can also define a corresponding DSL for legacy configuration files, but

[3] See `https://devboost.github.io/EMFText/`

using a different mechanism. Xtext allows for the creation of a grammar definition that specifies both the abstract and concrete syntax of the legacy format. This means defining the Ecore metamodel is not required, as Xtext can generate it if it does not exist. Xtext generates Eclipse-based tools, including advanced editor functionalities, a parser, and a printer. The parser converts the old configuration files into EMF models for analysis/ transformation, and the printer converts these EMF models back into the original text format, ensuring consistency with the original files.

2.2.6. Source Code Model

Since MDE considers that "*Everything is a model*" [Béz05], it is important to model the software source code. The source code is also a model with its own structured grammar. This facilitates the model transformation and manipulation of source code since the direct manipulation of the textual form of source code can be challenging. Therefore, a parser is used to convert the source code into a specific structure that is easier to work with. Conversely, a printer converts this structured representation back into text. The general concept of source code parser is explained in Section 2.2.6.1, while a concrete example of a Java parser and printer is provided in Section 2.2.6.2.

2.2.6.1. Source code Parser

A parser for source code reads the code and converts it into a structured representation, such as an Abstract Syntax Tree (AST), based on the code's grammar. This structured representation is utilized for various purposes, including code analysis, transformation, and generation.

Parsers are crucial in compilers and other source code editors. They provide an understanding of the code's structure and semantics, facilitating tasks such as syntax checking, refactoring, and performance optimization.

Parsers can be categorized based on their output into two categories: AST-based parsers and parsers using a unified metamodel, such as the EMF.

AST-Based Parsers AST-based parsers create an abstract syntax tree that represents the hierarchical structure of the source code. The AST captures the syntactical structure of the source code in a tree format, where each node represents a construct occurring in the source code. Examples of AST-based parsers include:

Eclipse JDT Core: The Eclipse Java Development Tool (JDT) Core provides an AST parser that transforms Java source files into an abstract syntax tree. The AST can be used for various purposes, such as code analysis, refactoring, and new code generation. JDT provides detailed information on the code structure and allows reference resolution and binding creation.

ANTLR: ANTLR (Another Tool for Language Recognition) is a parser generator that can be used to build parsers for any programming language. It creates a parser that generates an AST from the source code, which can then be traversed and manipulated.

EMF-based Parser EMF-based parsers use EMF to define a metamodel for the source code. These parsers generate models that conform to this metamodel, providing a high-level abstraction of the source code. An example of EMF-based parsers is in the following section (Section 2.2.6.2).

2.2.6.2. JaMoPP: Java model parser and printer

Java Model Parser and Printer (JaMoPP) is an EMF-based environment for modeling Java source code [HKP11]. JaMoPP defines an Ecore-based Java metamodel conforming to the third edition of Java language specifications, belonging to Java 5 and Java 6. Then, the concrete textual syntax of Java is defined in EMFText's specification language. After that, EMFText generates code for a parser and a pretty printer. The parser creates EMF-based models from source code files, while the pretty printer converts such models back into source code files.

JaMoPP also incorporates a mechanism to resolve references between different Java models, such as those introduced by imports. References use proxy objects, which are resolved to actual objects based on their uniform resource identifier when accessed. EMFText provides a set of default resolvers integrated into EMF's proxy resolution mechanism, which JaMoPP extends to handle Java-specific properties.

In the context of this thesis, we extend the JaMoPP metamodel to enable the modeling of Java versions 7-15 [Arm22]. Additionally, we provide a parser and a printer that are independent of EMFtext for the extended metamodel [Arm22], as described in Section 5.1.2.1.

2.2.7. Model Differences

Model matching aims to identify the common elements of two or more models. It is basically used to determine the differences of models. Brun and Pierantonio describe three fundamental steps to determine differences between two distinct models [BP08]: The first step, "calculation", matches model elements and calculates the differences. In the second step, "representation", the outcome of the calculation is represented in a different model. In the last step, "visualization", the represented differences are visualized in a human-readable notation.

Kolovos et al. [Kol+09] provide also an overview of the model differencing process and evaluate existing model matching approaches. Current approaches to model matching can be classified into four types:

- Static Identity-Based Matching: These approaches are fast but can be applied only to the models, whose elements have persistent and non-volatile unique identifiers.

- Signature-Based Matching: Instead of relying on identifiers, these approaches use predefined signatures. They define a series of functions to calculate the signature for the model's elements based on the type of the element and its values features.

- Similarity-Based Matching: These approaches consider the models as typed attribute graphs and calculate the matching elements based on the aggregated similarity of their features. similarity-based algorithms typically need to be provided with a configuration that specifies the relative weight of each feature. For example, the "name" feature of named elements is relatively more weighed than other features. *EMF Compare* [Lan19] is an example of these approaches. It enables defining the similarity and weights of features to allow more accurate results than the signature-based approaches. Similarity-based algorithms typically require a configuration that specifies the relative weight of each feature. For example, classes with the same name should be matched more likely than classes with the same values of the abstract feature. However, fine-tuning the weights empirically can be an error-prone process. Moreover, calculating the similarity for the modeling languages does not consider the semantics of these languages, as we will see in the next category.

- Custom Language- Specific: The approaches of this category implement matching algorithms for particular modeling languages. For that, they may incorporate the semantics of the language to provide more accurate matching and to reduce the cost of finding the matched elements. EMFCompare [Lan19] can also be used to implement custom language matching algorithms. However, the required implementation effort is still considerable.

2.3. Software Architecture

The architecture definition according to ISO/IEC_42010 is *"fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution"* [ISO11]. According to this definition, the architecture of the system can be understood as fundamental system concepts encapsulated in its elements, relationships, and design principles, guiding both the system's current structure and its evolution over time.

Taylor et al. [TMD09] defined the software architecture as *set of principal design decisions made about the system.* This perspective emphasizes the role of significant design choices in shaping the architecture, underscoring the decision-making aspect.

Reussner et al. [Reu+16] combine the understanding mentioned above of the software architecture. They described software architecture as *"the result of a set of design decisions relating to the structure of a system with components and their relationships and their mapping to execution environments"*. This extension highlights that architecture encompasses not just design decisions but also the structural arrangement of system components and their interactions.

The rational unified process defines software architecture as the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed together with their behavior as specified in the collaboration among those elements, the composition of these elements into progressively larger subsystems, the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition. Software architecture is concerned with not only structure and behavior but also usage, functionality, performance, resilience, reuse, comprehensibility, economic and technological constraints and trade-offs, and aesthetics.

2.3.1. Architecture Model

We introduce some fundamental definitions from ISO/IEC_42010 [ISO11] to describe the architecture model. ***Architecture view*** *addresses one or more of the concerns held by the system's stakeholders* [ISO11]. An architecture description can include more than one architecture view. For instance, the architectural views, according to Kruchten [Kru95], can be a logical view that describes the system functionality or a deployment view that describes the physical deployment of a system component. Architectural viewpoints are required to express these views. *The* ***viewpoint*** *establishes the conventions for constructing, interpreting and analyzing the view to address concerns framed by that viewpoint. Viewpoint conventions can include languages, notations, model kinds, design rules, and/or modeling methods, analysis techniques and other operations on views* [ISO11]. For instance, the UML is a viewpoint that enables visualizing different views of the system: structural views, behavioral views, environmental views and implementation views.

Based on the definition of the views, the architecture models can be also defined. The architecture models are parts of an architecture view. *An* ***architecture model*** *uses modeling conventions appropriate to the concerns to be addressed. These conventions are specified by the model kind governing that model* [ISO11]. For instance, the logical view according to Kruchten [Kru95] can be modeled using some architectural models of UML like package, class, and interaction diagrams. Similarly, the deployment view can be visualized using an architectural model of UML "Deployment diagram". It should be noted that an architecture model can be a part of more than one architecture view. for instance, the class diagram can be a part of the logical view and the structural view that represent the parts of the software system.

2.3.2. Software Components

According to Reussner et al. [Reu+16], *A **component** is a contractually specified building block for software that can be composed, deployed, and adopted without understanding its internals.* The component-based development enables the decoupling of functionalities' development and reusing the developed components. The component can be reused as a black-box if its internals are hidden. Otherwise, it is called white-box component. The component-based development aims to reduce the required effort for composing, deploying, adopting and reusing the components.

2.3.3. Microservices

Microservices are an architectural style for developing software systems as independently deployable services. The main goal of this style is to decouple the functionalities and increase the software system's scalability. Hence, the microservice-based architecture [LF14] divides the software system into different services that perform specific functionalities. These services are independently deployable, and they communicate with each other using lightweight communication like Hypertext Transfer Protocol (HTTP). Each service exposes its functionality through an interface, typically in the form of application programming interface (API). The services that belong to a specific deployment unit consist of one microservice. The microservices can be implemented using different programming languages and technologies. Since microservice-based architecture divides the functionality into fine-grained units "microservices", more effort will be required for monitoring and managing microservices. Therefore, the automation of integration, testing, and deployment is necessary.

To support standardized and scalable communication between services, many microservice systems adopt REST principles and implement them using RESTful APIs:

REST Representational State Transfer (REST) [NF11] is an architectural style that defines a set of principles and guidelines for building distributed systems. REST emphasizes a stateless client-server communication model, uniform interfaces, and resource-based interactions.

RESTful API RESTful [Bie16] is a HTTP-based implementation of REST. It implements API for Web services. It uses HTTP requests to access and use data, which allows the development of semantic Web services.

RESTful uses existing HTTP methods that are defined by the RFC 2616 protocol [FIE99] for accessing and using data. These methods are `GET` for retrieving data, `PUT` for updating the state of data resources, `POST` for creating data resources and `DELETE` for deleting resources. RESTful focuses on user-generated content and transfers messages defining actions to be executed using HTTP protocol.

2.4. Palladio Component Model

Palladio [Reu+16] is a software architecture simulation approach that analyses the software at the model level for performance bottlenecks, scalability issues, reliability threats and allows a subsequent optimization.

Regarding the performance assessment, PCM predicts the performance by the simulation. So, the architect can evaluate the performance and detect possible issues like bottlenecks and scalability problems. Since Palladio assesses the performance by AbPP, it can support a proactive evaluation of design decisions before applying them. The proactive assessment of design decisions can avoid high costs and efforts required to repair the wrong decisions [SW03].

The PCM can be divided into five different models that describe the software architecture from various viewpoints shown in Figure 1.1: Repository Model (Section 2.4.1) that includes the Behavior Model (Section 2.4.2), System Model (Section 2.4.3), Allocation Model (Section 2.4.4), Usage Model (Section 2.4.5) and Resource Environment Model (Section 2.4.6). In the following, we introduce more details on PCM models. After that, we introduce the PMPs in Section 2.4.7.

2.4.1. Repository Model

The Repository Model represents the static architecture of the software. Figure A.1 shows an excerpt of the Ecore metamodel of the Repository Model. According to its name, the Repository Model represents a repository that includes the software data types, components, interfaces, and the relations between them. For example, the relationship between the components and interfaces is determined by this component's role. If a component provides an interface, then a *ProvidedRole* of the component points to the provided interface. Besides, the component should provide an abstract behavior of all *Signatures* that its provided interfaces include. For that, the Service Effect Specifications (SEFFs) is used. In the next section (Section 2.4.2), we describe the structure and the purpose of SEFF.

2.4.2. Behavior Model

The Palladio SEFF [BKR09] is part of Repository Model. SEFF describes the behavior of a component service on an abstract level. Consequentially, SEFF can describe the inner behavior of the components, i.e., the relationship between services that a component provides and requires.

The structure of SEFF consists of a control flow of different elements called *actions* [Reu+16, p. 108]. In the following, we introduce the most essential SEFF actions:

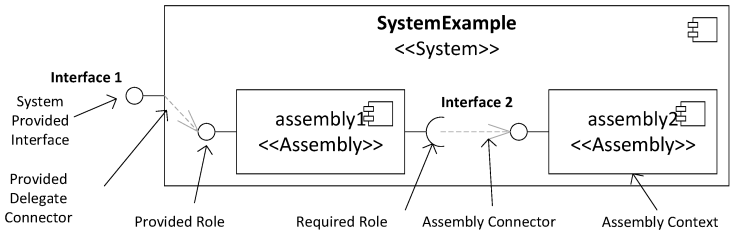


Figure 2.1: An example of the system model.

- *Internal action* is a combination of internal computations that do not include calls to required services, i.e., services belonging to other components. The internal action is usually annotated with Resource Demand (RD).
- *Internal call* is a call to a method within the component. The behavior of internal calls can be modeled by separate SEFF.
- *External call actions* are calls to required services. The behavior of the required services is modeled in the components that provide these services.
- *Loop action* represents a loop behavior that includes at least one external call. The loop should be repeated during the simulation according to an annotated loop count. To increase the level of abstraction, the loops in the source code that do not include external calls are not modeled in SEFF. Instead of that, they are combined into internal actions.
- *Branch action* is an abstraction of *switch* condition in source code. The branch action can split the control flow into different *transitions* according to the branch condition. The branch condition can be a conditional state or a fixed probability.

2.4.3. System Model

System Model describes the structure of the system based on the components and interfaces specified in the Repository Model. A system instantiates the components and wires the required interfaces of the instances with the provided ones. Mainly, System Model consists of *Assembly Contexts*, which are instances of the components. The component can be instantiated several times. These assembly contexts are connected with each other by *Assembly Connectors*. The connectors link the required role of an assembly context with the provided role of another one, see an example of a System Model in Figure 2.1. A *delegation connector*

In this example, two component instances (Assembly contexts) are connected with each other using an Assembly Connector that links the Required Role of assembly1 with the Provided role of assembly2. The provided role of assembly1 includes the functionality that the end user of this system obtains. Therefore, it is connected to the system-provided interface using A `Provided Delegation` Connector. For more details, see the metamodel of `System Model` in Figure A.2.

2.4.4. Allocation Model

The `Allocation Model` describes the mapping from the system composition (`System Model`) to the model of the resources (`Resource Environment Model`). In other words, `Allocation Model` models the system deployment on the execution environments. *Allocation Contexts* save the mapping between assembly contexts and one of the available *Resource Containers*. Each resource combines the processing resources that model either CPU or hard disk (HD) and their properties like processing rate, see Figure 2.2.

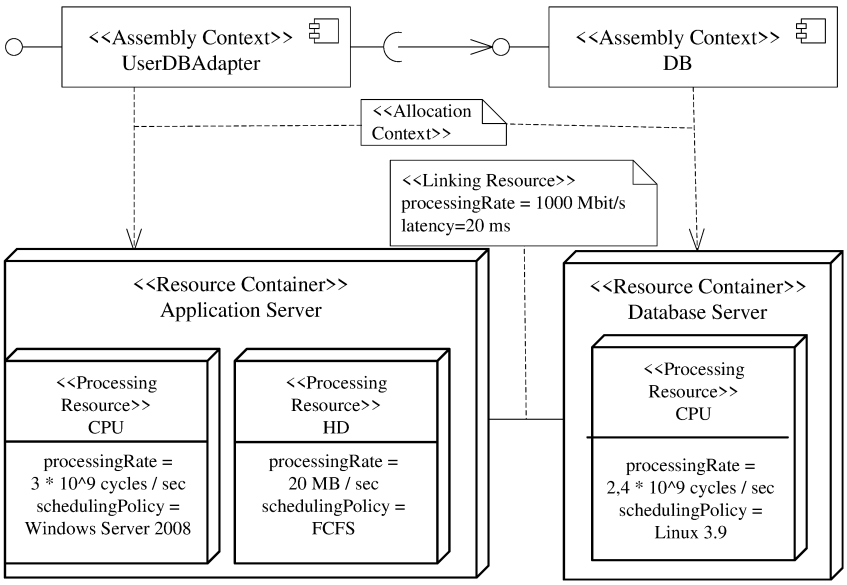


Figure 2.2: An example of allocation model and resource environment model, adjusted from an example in [Reu+16].

2.4.5. Usage Model

The `Usage Model` defines the usage profile of the software system, i.e., users' behavior in terms of how the users call the system. Thus, the `Usage Model` includes a specification of the order, frequency, and parameter values of component services called by users or

external systems at the system boundaries. It defines the interactions with the system and includes information about how components are used in real-world scenarios.

2.4.6. Resource Environment Model

The Resource Environment Model (`Resource Environment Model`) captures the physical aspect of the system. It encompasses the tangible hardware environment `"ResourceContainer"`, consisting of containers housing processing resources (e.g., CPU). Besides, `Resource Environment Model` represent the interconnecting links between resource containers using `"LinkingResource"`. For more detail, Figure A.5 shows an excerpt of the Ecore metamodel of the `Resource Environment Model`.

2.4.7. Performance Model Parameters

`Palladio` facilitates quality analysis by simulating PCM. This capability allows, for example, the prediction of performance metrics like response times, CPU utilization, and throughput. To achieve this, it is essential to specify concrete values for a range of parameters that significantly impact a software system's behavior and performance. We commonly refer to these parameters as Performance Model Parameters (PMPs). The primary PMPs include `SEFF` Parameters that specify the component quality and `Usage Model` Parameters that specify the usage profile.

`SEFF` Parameters define the behavior and quality of component services. Enriching `SEFF` parameters with specific values is essential for behavior simulation. Main `SEFF` parameters are the following:

1. **Resource Demand:** This entails characterizing the amount of processing resources required by internal actions, such as those requested from a CPU or a hard disk. By assigning values to these demands, the model can estimate the system's resource utilization and response time.

2. **Branch Probability:** Characterizing the probability of branch transitions within Branch actions enables the capture of the likelihood of various paths and allows selection of one of them during simulation. The branch transition probability can involve random variables, depend on an input parameter reflecting real-world usage scenarios, or follow a distribution.

3. **Loop Iterations:** Characterizing the number of loop iterations in a model allows for a more precise representation of repetitive operations for accurate prediction of the performance. Similar to branch probability, the count of loop iterations can be constant, depend on input parameters, or follow a distribution.

4. **External Call Arguments:** Specifying the arguments passed to external calls allows a comprehensive understanding of interactions between components. This, in turn, enables the prediction of the performance of the external component, which can depend on the passed arguments. This facilitates the analysis of the entire system's behavior and quality.

`Usage Model` parameters depict user interactions with the system. These parameters define how the software system is utilized in practical scenarios. For example, the parameters define the sequence and frequency of services used by the end users or systems. Besides, the parameter values offer actual input values for services, mirroring real-world situations. Additional parameters specify the workload and clarify the usage nature as open or closed, determining frequency or user circulation with defined pauses representing thinking time. Incorporating control constructs like branches and loops enhances realism in complex user scenarios.

`Palladio` employs the stochastic expression (StoEx), as introduced in [Koz16], to PMPs as expressions that incorporate random variables or empirical distributions. This approach enables the representation of dependencies between PMPs and impactful parameters through calculations and comparisons. These impactful parameters can be characterizations of input data like value, type, and size (e.g., the number of list elements or the size of files). By characterizing PMPs with impacting factors (the parametric dependencies [BKR09]), simulations are empowered to provide more accurate predictions of system performance.

2.5. Machine Learning Techniques

Machine learning techniques are a broad set of computational methods and algorithms designed to facilitate automated learning from data, enabling the generation of predictions based on the attained models. They include techniques like regression for predicting numerical values(Section 2.5.1), decision trees for classification, prediction or creating interpretable models (Section 2.5.2), genetic algorithms for optimization problems (Section 2.5.3) and feature selection for choosing relevant variables among others (Section 2.5.4).

2.5.1. Regression Analysis

Regression analysis [DS98] is a statistical technique to examine the relationship between one or more input variables (X-variables) and output variables (Y-variables) by fitting a mathematical model to the data. It is commonly employed for making predictions and understanding the strength and nature of relationships between variables.

Regression analysis encompasses a diverse range of techniques, each designed to address specific data patterns and research objectives. Linear regression, the most

basic form, is used to model straight-line relationships between variables. Multiple regression extends this to analyze the impact of multiple predictors on a single outcome. Logistic regression, on the other hand, is well-suited for predicting binary outcomes or probabilities. Nonlinear regression is a suitable framework for nonlinear relationships.

2.5.2. Decision Tree

A decision tree [Jam+13] is a machine learning algorithm used for classification and regression tasks. It is represented as an upside-down tree, with each node denoting a decision based on a specific attribute, and each branch leading to an outcome or further decision. Decision trees can be employed for prediction by navigating the tree structure with input features, starting at the root node and progressing through intermediate nodes until reaching a leaf node. The path taken through the tree determines the predicted class (in classification) or the numerical value (in regression) for the input data. Additionally, decision tree branches can be associated with probabilities, quantifying the likelihood of a data point belonging to a particular class or category, adding a probabilistic dimension to the decision-making process in predictive modeling. Decision trees are known for their interpretability, making them valuable for understanding and explaining complex decision patterns.

2.5.3. Genetic Algorithm

The genetic algorithm [Kra17] is a heuristic algorithm to resolve optimization problems. It is based on the evolution theory. It starts from initial solutions as candidates for optimum solutions and evolves them. The algorithm is shown in Listing 2.1 and starts with ‘initialize population’ step. This step initializes a set of solutions or uses a solution modeled by experts, which represents the first population. After that, the main evolutionary loop of the genetic algorithm generates new offspring based on two major operators: ’crossover’ and ’mutation’. These operators generate new candidates for the new offspring population in two different ways. The crossover generates the candidate solutions by combining genetic materials from the parents, whereas mutation generates them by applying random changes. After crossover and mutation, the new candidates must be evaluated based on their ability to solve the optimization problem. This evaluation is done by ‘phenotype mapping’ of candidates (genotype) to the actual solution in order to avoid introducing a bias. This step can be escaped if the genotype represents the solution itself. After that, the new offspring population is evaluated by the ’fitness function’. The goal of the fitness function is to evaluate the quality of the solutions in order to select the best offspring to be parents in the new parental population. This achieves the progress towards finding the optimal solution. However, the main evolutionary loop should have a termination condition that determines when the search for optimal solutions must be stopped. The condition can be a predefined

number of generations or an acceptable quality degree of the produced solutions. Usually, the termination condition is related to the time and cost of the designed fitness function.

Listing 2.1: Basic Genetic Algorithm from [Kra17]

```
initialize population
   repeat
      crossover
      mutation
      phenotype mapping
      fitness computation
   until population complete
   selection of parental population
until termination condition
```

2.5.4. Feature Selection

Feature selection [VR14] is the process of choosing a subset of relevant attributes or variables from a larger dataset. Its primary goals include improving model performance, reducing overfitting, enhancing interpretability, and minimizing computational complexity. In simpler terms, it streamlines important information from a dataset to create a more efficient and accurate predictive model, without unnecessary details. Feature selection is a crucial step in the machine learning pipeline to ensure effective and interpretable models.

2.6. Consistency Maintenance

The term consistency is used by various contexts for various purposes [Küh+23]. In this section, we describe the term in the context of this thesis, i.e., the models' consistency during model-based engineering, cf. Section 2.6.1. Then, we describe the definition of the consistency relation and its properties in Section 2.6.2.

In Section 2.6.3, we describe a consistency preservation platform that we use in our approach. In this section, we clarify the meaning of consistency preservation rules (Section 2.6.4) in addition to an application of consistency preservation in the context of software evolution (Section 2.6.5).

2.6.1. Consistency Definition

In model-based engineering, consistency is defined as a binary relation between two models [AC08]. For models M_1 and M_2, this relation R can be classified as follows:

(1) bijection, a one-to-one correspondence where each element of M_1 maps to exactly one element of M_2, and vice versa; (2) total and surjective function, a one-to-many relationship where each element of M_1 maps to exactly one element of M_2, and every element of M_2 is mapped by at least one element of M_1; and (3) total relation, a many-to-many relationship where multiple elements of M_1 may map to multiple elements of M_2, and vice versa.

Klare defines the multiplicity of consistency relations that exists and can be split into multiple binary relations [Kla21] where the consistency is the absence of contradictions between the models.

Definition 1 (Consistency Relation Definition according to [Kla21])**.** *Given a tuple of metamodels $\langle M = M_1, \ldots, M_n \rangle$, a consistency Relation R is a relation for instances of the metamodels $R \subseteq I_M = I_{M_1} \times \cdots \times I_{M_n}$. For a tuple of models $x \in I_M$, we say that:*

$$x \text{ is consistent to } R \iff x \in R.$$

This definition above suggests understanding consistency as a mathematical relation. In practice, consistency can be defined extensionally by listing all consistent models or intentionally by specifying constraints that models must satisfy. Hence, a consistency specification can take various forms, such as predicates that define specific constraints, checkers that validate the consistency of models, or formal semantic specifications used to evaluate whether models exhibit non-contradictory semantics.

Klare also describes two dimensions of consistency: first, according to the consistency notion, the consistency can be specified as *perspective* if we intend to keep the models consistent in a predefined way. This means the models are considered to be consistent if they adhere to specific specifications, e.g., a specified model transformation. The *descriptive* consistency assumes that there is always an explicit or implicit notion of consistency. So, it is not enough that the models adhere to a consistency specification; they should adhere to the existing notion of this consistency. Whether a specification is normative or descriptive depends on its intent: a normative specification establishes consistency without reference to an existing notion or specification, while a descriptive specification conforms to or approximates an already defined notion or specification of consistency.

The second dimension of the consistency, according to Klare, is related to the pragmatic of the models [Sta73], cf. Section 2.2.1. Similar to the UML notion, Klare distinguishes between the *structural* consistency, whose pragmatic has no execution semantics, and *behavioral* that has execution semantics.

2.6.2. Fine-grained Formalization of Consistency Relations

As described in the previous section, the consistency relation is a connection between models sharing common semantics to ensure they adhere to predefined consistency

rules or constraints, maintaining non-contradictory semantics. According to Klare's formulation, models are considered consistent if their combination satisfies the rules defined by the relation R between models. Perdita [Ste20] also defines the consistency relation between models using hyper-graphs.

In our chairs, we generalize the formalizations above to be more fine-granular, defining the relations among the elements of the models instead of focusing on the models themselves [Küh+23]. Moreover, we define the main properties of the consistency relations. For the formalization of such consistency relation, we use hyper-graphs. A hyper-graph G consists of a set of vertices V and hyper-edges $\mathcal{E}$. Contrary to other graphs, the edges can join more than two elements:

Definition 2 (Hypergraph from [Küh+23])**.** *A* simple hyper-graph $G := (V, \mathcal{E})$ *consists of a set of vertices V and a set of hyper-edges $\mathcal{E} \subseteq \{X \mid X \subseteq V \wedge |X| \geq 2\}$.*

Hence, the consistency relation over multiple models can be formalized by the hyper-graph as follows:

Definition 3. *Given a finite non-empty set of mutually disjoint models $\mathcal{M} := \{M_1, \ldots, M_n\}$ and the corresponding set $\Omega := \{m \in M \mid M \in \mathcal{M}\}$ of all model elements in $\mathcal{M}$. Then a* consistency relation *$R = (\mathcal{M}, \mathcal{E})$ defines a simple hyper-graph $G = (\Omega, \mathcal{E})$ that reflects a joint purpose of the connected model elements [Küh+23].*

A consistency relation $R := (\mathcal{M}, \mathcal{E})$ is non-trivial *iff $\mathcal{E} \neq \emptyset$.*

According to the definition above, the consistency relations connect elements across multiple models ($\mathcal{M}$) to ensure their compatibility. The related models' elements are linked to each other through the connections of the graph ($\mathcal{E}$), which are called "*correspondences*". Each correspondence links at least two model elements. The corresponding elements adhere to consistency rules/ constraints defined at the metamodel, maintaining non-contradictory semantics. Utilizing this definition enables us to explain the properties of consistency relations shown in Figure 2.3.

2.6.2.1. Abstraction

Models vary in abstraction levels. For example, in software development, different models at varying levels of abstraction represent a software system. Requirements or architectural models are more abstract, focusing on high-level structures and behaviors, whereas concrete source code models provide more detailed representations, capturing the specifics of the implementation. Determining the abstraction level can be challenging when dealing with models from different domains. Consistency relations are categorized into vertical and horizontal classifications based on the abstraction levels of the models that they consider.

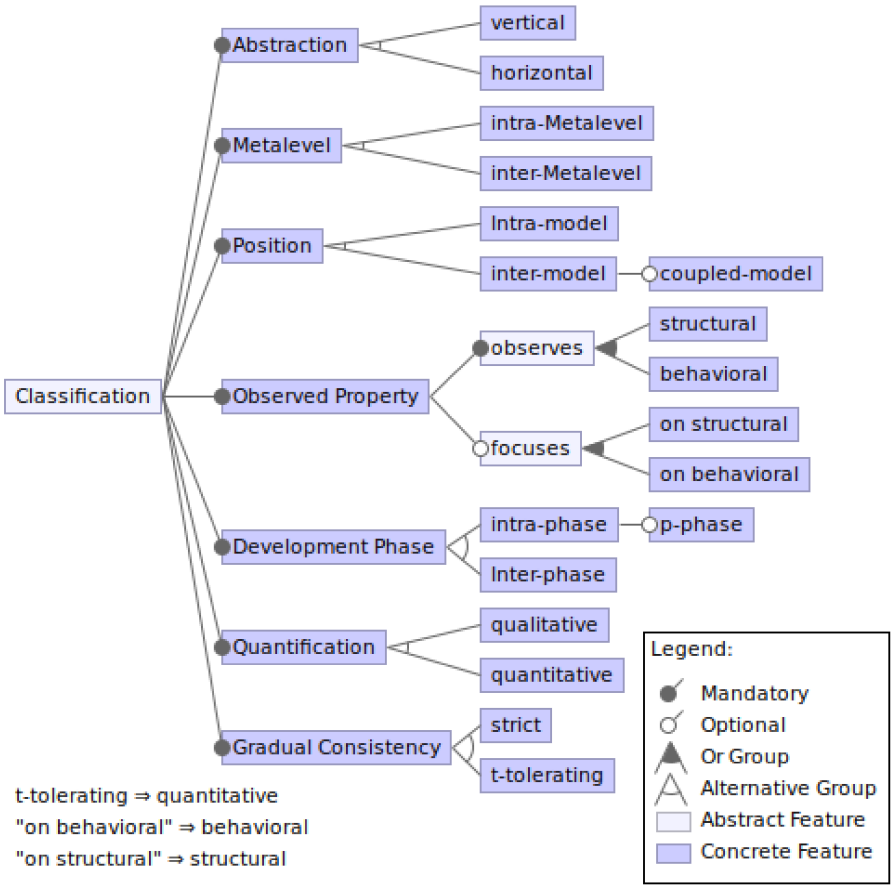


Figure 2.3: Feature model explaining the classification schema of consistency relations, from [Küh+23].

2.6.2.2. Metalevel

Model elements are associated with precisely one metalevel. Intra-metalevel consistency relations connect model elements that are at the same metalevel, while inter-metalevel consistency relations connect model elements at different metalevels.

2.6.2.3. Position

Consistency can be defined within or between models. Intra-model consistency ensures consistency within a single model, while inter-model consistency ensures alignment between elements across different models.

2.6.2.4. Observed Property

The observed properties of consistency relations can be categorized as either structural or behavioral. Structural properties, such as the presence of elements or their attributes, can often be statically checked, while behavioral properties, such as temporal or dynamic aspects, typically require dynamic analysis techniques like model checking or co-simulation. The relevance of these properties depends on the nature and purpose of the model being analyzed. Consistency relations observe specific properties if they connect model elements exhibiting those properties and focus on a property if all connected elements possess it.

2.6.2.5. Development Phases

In the context of software development, consistency relations can be associated with different phases of the system's lifecycle, such as specification, design, and runtime. Therefore, consistency relations are categorized as intra-phase when they connect elements within the same phase and inter-phase when they connect elements across different phases.

2.6.2.6. Quantification

Generally, most of relations determine whether models are consistent or not. However, in some cases the consistency can be quantified to determine to what degree the connected model elements can be seen as consistent. Thus, the relation can be either qualitative, indicating a binary state of consistency, or quantitative, providing a measure of the degree of consistency achieved.

2.6.2.7. Gradual Consistency

Generally, there are two types of gradual consistency: tolerating and strict: In tolerating consistency, a certain threshold is set, indicating the acceptable level of inconsistency between model elements. If the inconsistency score of connected model elements exceeds this threshold, they are considered consistent enough. On the other hand, strict consistency is characterized by a zero-tolerance policy towards inconsistencies. In strict consistency, all connected model elements must exhibit perfect consistency, without tolerance towards any level of discrepancy.

2.6.3. Vitruvius Platform

Vitruvius [Kla+21] is a view-based framework that encapsulates the heterogeneous models of a system and the semantic relationships between them in a VSUM in order

to keep them consistent. The Vitruvius approach [Kla+21] is a view-based framework that addresses the challenge of information fragmentation and inconsistency across heterogeneous modeling languages. It involves creating a Virtual Single Underlying Model (VSUM) metamodel, which consists of multiple metamodels, consistency preservation rules, and view types that derive information from them. In the next step, Vitruvius enables the consistency preservation of instances of the metamodels within a VSUM metamodel.

For that, a methodologist should create the VSUM metamodel. This includes defining CPRs at the metamodel level as we describe in Section 2.6.4 Then, developers instantiate the VSUM metamodel and use the resulting VSUM to derive views and perform modifications to them. The approach enables developers to work with different models supporting the preservation of consistency between views.

The Vitruvius approach applies delta-based consistency preservation [Dis+11] in which changes (i.e. deltas) in one model are propagated to become changes in other models. This approach updates models inductively and avoids overwriting changes from other models [WSK23]. For the delta-based consistency management, Vitruvius defines and utilizes two declarative languages: *mappings* and *reactions* languages. The first one specifies the bidirectional consistency relation on the metamodel-level, which is transformed into imperative unidirectional specifications using the *reactions* language. The *reactions* language describes then the consistency preservation rules that describe how to restore the consistent state. Thus, Vitruvius stores the mapping between corresponding model elements, in a *correspondence model* to reuse them by the consistency preservation process.

The approach has been implemented in the Eclipse Modeling Framework and has shown completeness, correctness, and applicability in maintaining consistency in different domains [Lan17; Maz+17; Maz+18; BMK16].

In the following section (Section 2.6.4), we explain the means of Consistency Preservation Rule (CPR) that methodologists define for keeping the consistency between the heterogeneous models. In Section 2.6.5, we introduce the co-evolution approach, which relies on Vitruvius for evolving the source code and PCM consistently.

2.6.4. Consistency Preservation Rules

Consistency Preservation Rules (CPRs) are used to define the consistency repair logic at the metamodel level for repairing consistency with respect to different types of changes, thus restoring the consistent state between the models. In other words, CPRs define which and how the artifacts of a metamodel must be changed to restore the consistency after a change in a related metamodel has occurred. CPRs can be defined using the Reactions and the Mappings languages of Vitruvius.

The top-level structure of the Reactions language, depicted in Listing 2.2, encompasses three fundamental steps. Initially, reactions are triggered after specific types of changes,

delineated by the "*after*" section. Subsequently, specific conditions or restrictions defined in the "*check*" block are evaluated to determine whether the subsequent actions in the reaction should be executed. Finally, a routine is invoked to address the consistency, initiated through a "call" operation. The "*match*" block of the called routine is used to retrieve corresponding elements, while the "*action*" block is used to perform actions restoring consistency. Additionally, "*action*" block manages correspondences by storing retrieved and altered elements in the correspondence model.

Listing 2.2: Reaction stub illustrating defining change-driven consistency preservation rules, from [Kla+21].

```
reaction aReaction {
   after // trigger definition
   check { /* further restrictions */ }
   call aSpecificRoutine(...)
}

routine aSpecificRoutine(...) {
   match { /* retrieve corresponding elements */ }
   action { /* perform actions and manage correspondences */ }
}
```

2.6.5. Co-Evolution Approach

The co-evolution approach [LK15; Lan17] keeps the architecture model and the source code consistent during software system evolution. It defines change-driven CPRs to propagate changes in source code to the architecture model and vice versa using model-based transformations.

Using Vitruvius, the co-evolution approach keeps Java source code (using an intermediate model [Hei+10] for Java 6 [Joy+00]) and PCM consistent. The CPRs update the `Repository Model` of a PCM model and its behavior (SEFFs *without* PMPs) as a reaction to changes in the source code. Similarly, changes in PCM model are propagated to the Java source code. Although the co-evolution approach allows updating the `Repository Model` manually, it does not support the consistency management of the PMPs.

2.6.6. iObserve Approach

The `iObserve` considers the adaptation and evolution of cloud-based systems as two interwoven processes throughout the application life-cycle to close the gap between Ops-time and Dev-time [Hei16; Hei20]. Thus, `iObserve` aligns architectural models that are manually modeled in development and measurements from operation to integrate

modeling concepts that bridge the gap between these two phases. The main idea is to use Ops-time observations to detect changes during the operation and to reflect them by updating an architecture model, which is then applied for quality predictions. The PCM is used as the basis for the quality predictions, and the Kieker monitoring tool (Section 2.1.3) is used for monitoring the system during operation [HWH12b; Hei16]. The main aspects that `iObserve` updates are the usage profile, usage intensity, and allocation of the components on the resources. For that, `iObserve` utilizes a correspondence model that links the divergent levels of abstraction between implementation artifacts and component-based architectural models. Besides, a transformation pipeline utilizes the information stored in the correspondence model to update the architectural models based on changes during operation.

Briefly summarized, `iObserve` collects monitoring data at Ops-time with the help of `Kieker` and applies necessary changes to the architecture model (PCM instance) which originated at Dev-time. The mapping between elements in the architecture model and corresponding elements in the source code is based on the runtime architecture correspondence model [Hei14; Hei20].

In CIPM adjusts and integrates iObserve's dynamic analysis to update the modeled usage and deployment parts based on monitoring data. Moreover, 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.

2.7. Evaluation Foundation

In this section, we outline the foundational concepts for evaluating software systems and the methodologies employed in this research. In Section 2.7.1, we discuss the levels of measurement-based evaluation as introduced by [BR08]. Then, in Section 2.7.2, we explore a statement-based approach for classifying the software engineering research, as proposed by Kaplan et al. [Kap+21a]. Section 2.7.3 further elaborates on the use of case studies and experiments. Finally, in Section 2.7.4 we introduce the Goal Question Metric (GQM) paradigm, a widely used method for evaluating software systems.

2.7.1. Evaluation Levels

The analytical metrics used for predicting a software system's quality attributes have to be validated. A comparison with independent measurements can be used for the validation of these metrics. Böhme and Reussner [BR08] suggest three validation levels for validating prediction models that include such analytical metrics:

- The first level, "metric validation," considers the validation of the metrics of models through the comparison of predictions and measurements.

- The second level, "applicability validation", validates the applicability of the prediction model. The validation investigates the reliability of the input data and the interpretability of the metric.

- The third level, "benefit validation", validates the prediction model's claimed benefit compared to competing approaches.

2.7.2. Classification of Software Engineering Research

Kaplan et al. [Kap+21a] propose a statement-based approach for software engineering research. This approach aims to link each statement in software engineering research to arguments, evidence (data), and related statements. They introduce a multi-dimensional classification system for these statements, as shown in Figure 2.4. The classification system categorizes statements into three dimensions:

1. **Research Object**: This dimension categorizes the object of research to be investigated. It can be a problem, method, model, metamodel, or automation.

2. **Kind of Statement**: This dimension classifies statements into three classes:

   - *Property-as-such*: Statements where the research object is not dependent on external factors.
   - *Property-in-Relation*: Statements related to the research object's context. For example, the performance of a method is influenced by the size of the input data.
   - *Relevance*: Statements assessing the importance or applicability of a research object within a specific context. In other words, relevance encompasses the scope of applicability, the magnitude of its impact, and the depth of originality. In a scientific context, relevance may also encompass the depth of intellectual engagement and the novelty of the idea.

3. **Evidence**: This dimension categorizes the type of evidence supporting a statement. It includes various categories such as argumentation, motivation examples, case studies, surveys, experiments, verification, and mining software repositories.

Further details and descriptions of the classes within each dimension are in [Kap+21a].

The authors evaluate this classification scheme through a survey sent to researchers in the field of software engineering, as reported in [KK21]. The survey results demonstrate the practical applicability of this classification system.

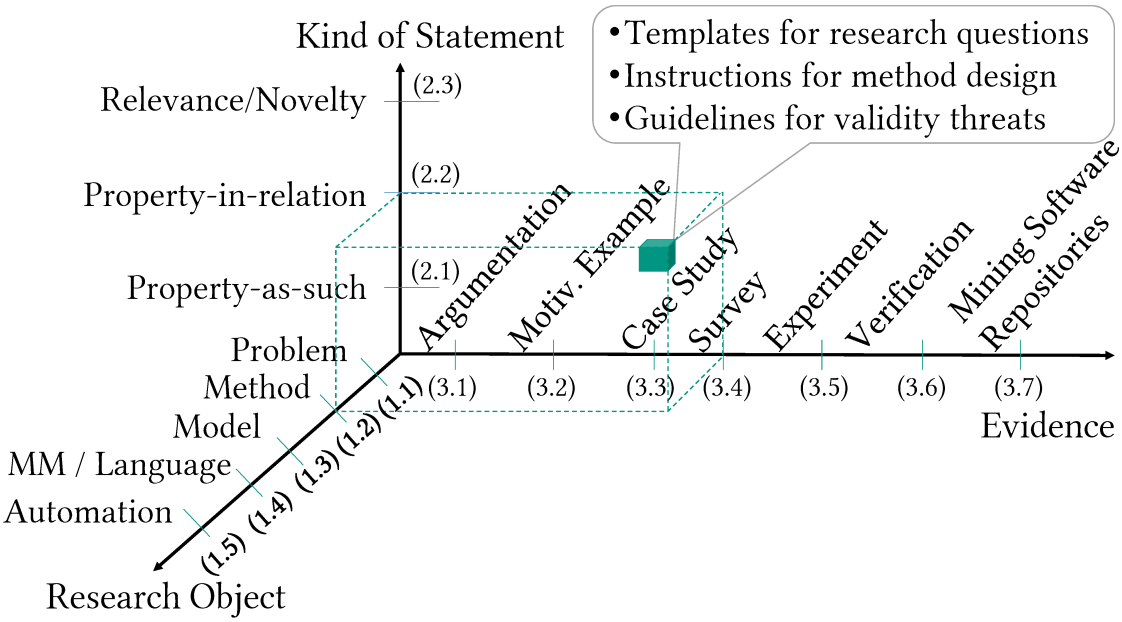


Figure 2.4: Classification of software engineering research from [Kap+21b].

2.7.3. Research Strategies: Case Study and Experiments

Wohlin has recently provided the following definition for a case study: " *A case study is an empirical investigation of a case, using multiple data collection methods, to study a contemporary phenomenon in its real-life context, and with the investigator(s) not taking an active role in the case investigated" [Woh21].* In this definition, Wohlin emphasized two points as essential criteria for case study: a contemporary phenomenon and the presence of a real-life context.

An *experiment* is a controlled and systematic scientific procedure in which researchers manipulate variables and observe their effects to test hypotheses, theories, or gather empirical data. Experiments aim to establish causal relationships, validate theories, or explore phenomena under controlled conditions to draw meaningful conclusions [Woh+12].

Case studies and experiments differ in several key aspects. Case studies are primarily exploratory, aiming to provide in-depth insights into complex phenomena, often conducted in natural settings with limited control over variables. They yield context-specific findings and rich qualitative data. In contrast, experiments are typically explanatory, focusing on hypothesis testing and causal relationships. Researchers have a control over variables, and data collection is structured and quantitative. Experiments aim for generalizability and replicability of results, making them suitable for validating theories

and drawing broader conclusions. The choice between these approaches depends on research objectives and the level of control needed.

We suggest to incorporate experiments within a case study, since experiments mostly isolate specific aspects of reality to enhance control over the situation, albeit often at the cost of realism, as noted by Runeson [Run+12]. In this proposed approach, researchers conduct a case study to gain a deep understanding of a complex phenomenon in its real-life context, e.g., the consistency preservation between software artifacts and performance models. Within the case study, they embed experiments to validate some aspects under controlled conditions. As a result, the researcher can generalize the result to environments similar to the experimental setting.

Case study Process: The case study process encompasses the definition of objectives, meticulous planning, precise formulation of data collection procedures, the execution of these procedures on the chosen cases, thorough data analysis to gather compelling evidence, and the conscientious packaging of study findings and conclusions into accessible formats suitable for reporting.

2.7.4. Goal Question Metric

Basili et al. introduce the Goal Question Metric (GQM) approach for evaluation through goal-oriented software measurements [BCR94].

For a purposeful measurement, the GQM plan consists of three steps: First, defining the "goal" of the evaluation at the conceptual level. To define these goals operationally, they must be traced with data. The objects of measurement can be: (1) products produced during the system life cycle like program, (2) processes related to the time like testing and (3) resources used by processes like hardware.

Second, the evaluation "questions" should be determined. These questions try to answer whether or not the evaluation goal has been achieved. The questions assess the goals at the operational level, considering specific quality attributes from particular viewpoints.

Third, metrics are defined to quantify the answers of evaluation questions. Metrics can be either objective if they depend only on the measured object, e.g., size of the program, or subjective if the considered viewpoint is measured also, e.g., the executability of the program". In the following section (Section 8.1.3), we introduce metrics that we used in our evaluation.

2.8. Evaluation Metrics

In this section, we describe the metrics used in our evaluation. These metrics are divided into two categories. The first category includes metrics for evaluating the accuracy

of models, such as the F-score (Section 2.8.1) and the Jaccard similarity coefficient (Section 2.8.2). The second category includes metrics for comparing two cumulative distribution functions such as the Kolmogorov-Smirnov test (Section 2.8.3) and the Wasserstein distance (Section 2.8.4), which can be used to evaluate the accuracy of AbPP by comparing the CDFs of measured response times and simulated ones.

2.8.1. F-score

The F-score is an accuracy metric. It is often used to measure the accuracy of information retrieval and binary classification tests. In such fields, the relevant cases have some required detail or fulfill some predefined conditions. The tests should retrieve the relevant cases and exclude the remaining cases. Retrieved cases are called *positive cases*. The rejected cases are *negative cases*.The results of an information retriever or a binary classifier can be divided into four cases that are shown in Table 2.1:

TP The true positive cases are cases that fulfill the conditions and correctly retrieved by the test.

FP The false positive cases are positive cases that do not fulfill the conditions and incorrectly retrieved by the test.

TN The true negative cases are cases that do not fulfill the conditions and the test excluded them correctly.

FN The false negative cases are cases that fulfill the conditions and the test incorrectly excludes them.

Table 2.1: The confusion matrix for an information retrieving problem.

		Predicted Condition	
		Positive	Negative
Actual Conditions	Positive	True Positive, TP True predicted	False Negative, FN False predicted
	Negative	False Positive, FP False rejection	True Negative, TN Correct rejection

Based on the above-defined cases, we can measure the accuracy of the test using the following metrics:

Precision The precision measures the relevance of retrieved items as a proportion of correctly identified relevant cases among all predicted cases, see Equation 2.1.

$$Precision = \frac{TP}{TP + FP} \tag{2.1}$$

Recall The recall indicates the sensitivity through calculating the ratio of relevant items retrieved by the test, see Equation 2.2.

$$Recall = \frac{TP}{TP + FN} \tag{2.2}$$

Accuracy The accuracy can be defined as the proportion of correct predictions (both TP and TN) among the total number of cases, see Equation 2.3. The accuracy ignores the effect of the false decision (FP and FN).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{2.3}$$

F-score This score combines both *precision* and *recall* in one metric. $F1$ score represents the harmonic means of *Precision* and *Recall* [Sas+07], see Equation 2.4.

$$F_\beta = \frac{(1 + \beta^2) \cdot \text{Precision} \cdot \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}} = \frac{(1 + \beta^2) \cdot \text{TP}}{(1 + \beta^2) \cdot \text{TP} + \beta^2 \cdot \text{FN} + \text{FP}} \tag{2.4}$$

In our Validation, we use the F1-Score (F-Score with $\beta = 1$), see Equation 2.5.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \tag{2.5}$$

2.8.2. Jaccard Similarity Coefficient

The Jaccard similarity Coefficient (JC) [EKR02] measures the similarity and diversity between two sets.

The JC is defined as the follows:

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.6}$$

where A and B are sets, $A \cap B$ is the intersection of the sets (A, B) and $A \cup B$ is the union of these sets, see the Figure 2.5.

Normally, the JC can quantify the similarity of sets. If the resulting JC is equal to 1, the two sets under investigation are considered to be fully identical.

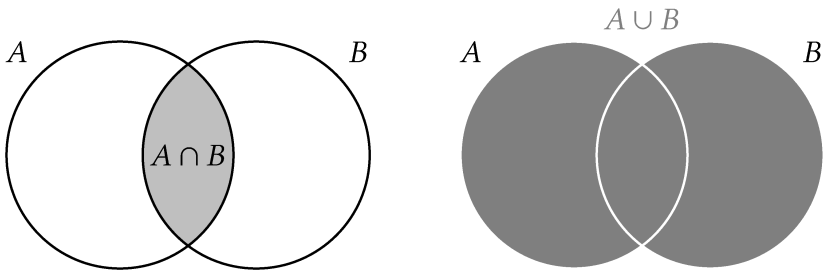


Figure 2.5: The intersection of two sets at the left side and the union of them at the right side.

Model conformity JC can be used for checking model conformity since the models are a set of elements that represents some knowledge. For that, a matching algorithm that determines the identical elements (intersection) in two models A and B is required. The required matching algorithm can be based on different factors: the types of models' elements, some of their properties like the name of named elements, some references and the model structure. See Section 2.2.7 for more details on the model matching approaches.

After applying a suitable matching algorithm, the elements matched in this method are considered to be equal for the intersection of the models A and B ($A \cap B$), i.e. the number of elements matched in this method becomes the numerator in Equation 2.6. The unmatched elements with the matched will represent the sets' union ($A \cup B$). The number of elements in the union will be the denominator of Equation 2.6.

2.8.3. Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov-test (KS-test) is a non-parametric test that quantifies the similarity of two arbitrary distributions. It is used for calculating the significant difference between two empirical data distributions, or for determining whether an empirical distribution originates from a reference one. The KS-test is non-parametric because the data does not require following a normal distribution.

The KS-test calculates the Maximum absolute difference between two Cumulative Distribution Functions (CDFs). CDF indicates the probability of a variable being less than or equal to a specified value. For example, the red CDF in Figure 2.6 shows that 50% of the value in the data set is less than zero.

The KS-test gives the term D for the maximum absolute difference between the CDFs, see Figure 2.6. If the two CDFs come from the population with the *same* distribution, D should technically be zero. Of course, in reality, the two samples are randomly taken from their populations, so they should differ somewhat. In general, the lower value of

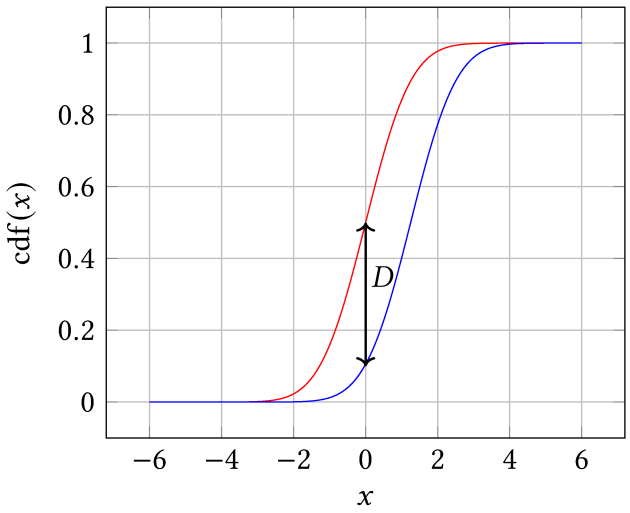


Figure 2.6: The KS-test: the maximum absolute difference between two cumulative distributions.

KS-test, the more similar distributions. If two distributions have no overlap at all, the maximum difference D will be then 1, i.e., when one CDF is 1 and the other is 0.

The KS-test can be used as a metric to measure the similarity between two distributions. The results of KS-test will be between 0 and 1. The lower values indicate that the distributions are similar. However, it should be noted, that KS-test is sensitive to the distribution shape. This means that two distributions with the same mean and significantly different shapes will produce a large value of D.

2.8.4. Wasserstein Distance

The Wasserstein metric, also known as the Earth Mover's Distance (EMD), has gained popularity as a distance metric for measuring the dissimilarity between probability distributions [Vil+09]. The EMD is a measure of how much mass must be moved from one distribution to transform it into another distribution. It has been used in various applications, such as image retrieval, machine learning, and computer vision.

In recent years, the Wasserstein metric has been proposed as an accuracy metric in machine learning tasks [ACB17; Cou+17; DR18; Gen+18]. For example, in the context of generative adversarial networks (GANs), which are a popular type of generative model, the Wasserstein metric has been shown to be a more reliable and stable measure of accuracy compared to traditional metrics [ACB17]. It captures the quality of generated samples by measuring the distance between the true distribution and the generated one, allowing for a more objective evaluation of the performance of the model.

One limitation of the Wasserstein metric is that it is computationally expensive to compute when dealing with high-dimensional distributions. However, there are approximations and modifications of the Wasserstein metric that can make it more feasible to use in practice. For instance, WS-distance can be efficiently computed using linear optimization algorithms [PW10; Cut11].

A disadvantage of using the WS metric is that it produces an absolute number, which can be difficult to interpret without a baseline or comparison to other results. However, a lower Wasserstein distance would indicate that the compared distributions are similar to each other. In other words, in the accuracy context, the predicted distribution is closer to the real data.

This metric is also known in the computer science under the name of Earth Mover's Distance [RTG00].

3. Motivation Example

The microservice architectural style has been widely adopted in DevOps SDLC because it improves dependability and modifiability [BWZ15]. Consequently, a microservice-based case study is a suitable example for illustrating the concept of CIPM within the DevOps SDLC. We present the TeaStore case study [Kis+18] to highlight the challenges of performance management in a DevOps context and to describe and validate all aspects of our CIPM approach.

The TeaStore is a web-based application, which implements a shop for tea. The application is designed to be suitable for the evaluation of approaches in the field of performance modeling and testing. For example, experiments on TeaStore show that performance testing of microservice-based applications is challenging [Eis+20]. It requires additional care when setting up large-scale test environments and enough repetitions of the performance tests for performance regression.

3.1. Background

The application is based on a distributed microservice architecture. The TeaStore consists of six microservices: `Registry`, `WebUI`, `Auth`, `Persistence`, `Image-Provider`, and `Recommender`. The main functionality of TeaStore microservices are:

- `Registry` makes the other microservices available to each other.
- `WebUI` provides the user interface.
- `Auth` is responsible for the authentication of users.
- `Persistence` is responsible for storing the data and retrieving it again.
- `Image-Provider` delivers the products' images to be displayed in the store.
- `Recommender` provides the advertised products based on the viewed products and the old orders.

As shown in Figure 3.1, the last five microservices include the main functionality of the TeaStore and register themselves at the `Registry` microservice. This makes them available for each other and enables client-side load balancing, as it will be explained later.

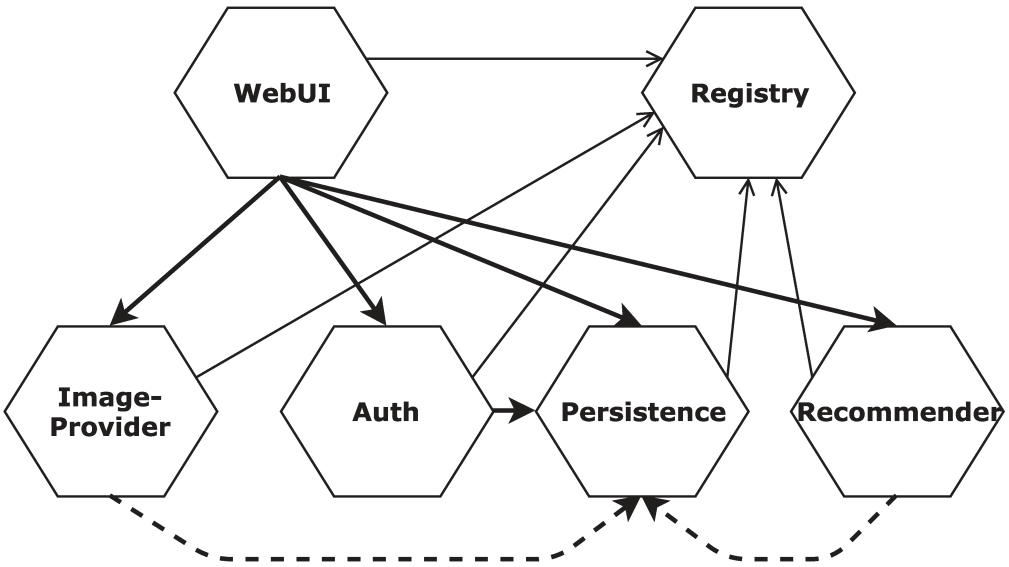


Figure 3.1: Microservice-based architecture of the TeaStore [Kis+18].

`WebUI` relies on the remaining functional microservices (bold communication lines) to view products and suggested products, authenticate users, order products and store the shopping details in the database.

The dashed communication lines show that both the `Image-Provider` and `Recommender` communicate with the `Persistence` on startup to provide the required images for the interface and to train the recommender according to the history data.

Communication between the microservices is based on the widely used REST standard [FF09]. The services are deployed as web services on Apache Tomcat or on a virtual Docker image that contains a Tomcat stack. Then, they communicate with each other using REST-calls.

Load-balancing aims to distribute the load on the available services in order to improve the performance, e.g., the resource utilization [Ala09]. For client-based load-balancing, TeaStore uses Ribbon[1] that distributes REST calls on the running instances of the services. For that, Ribbon can query the Registry to get the running instances, since all instances are registered in the registry. The registered instances are removed if a service is unregistered or the Registry doesn't receive the keep-alive message within a determined time. By overloading cases, new instances can be added. This leads to a dynamic change of the system architecture at Ops-time due to adding/ removing instances.

On the right side of Figure 3.2, there is a depiction of the `System Model` and `Allocation Model` of the TeaStore, illustrating the dependencies between its microser-

[1] `https://github.com/Netflix/ribbon`

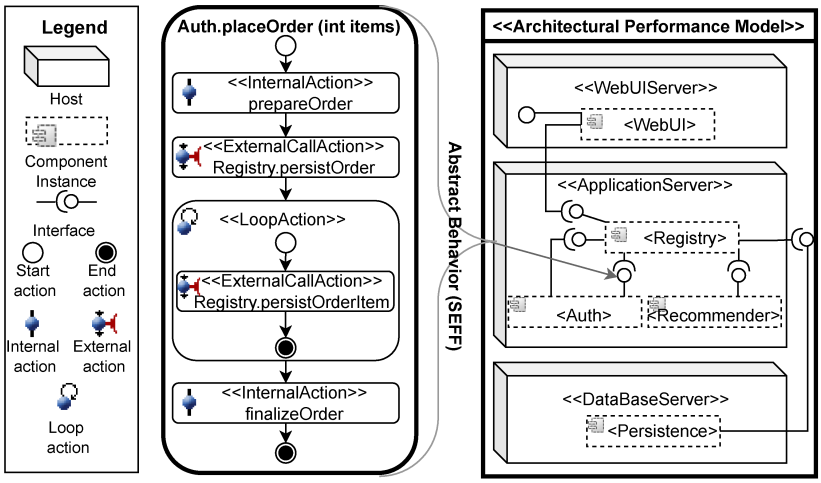


Figure 3.2: Excerpt of TeaStore's architecture with notations from `Palladio` (Section 2.4).

vices in a simplified manner. On the left side, the abstract behavior of the `placeOrder` operation is simplified by modeling the communication between the microservices via REST as a direct external call service begins with an internal action that prepares an order and calls an external service from the `Registry` to persist the order. Then, a loop includes a call to an external service of `Registry` to persist each order item (`persistOrderItem`). This loop iterates until all items are persisted. Finally, the order is finalized.

3.2. The Performance Management at Dev-Time

The TeaStore application recommends other products to customers based on their own and others' historical orders. There are different strategies to train this recommender system, which developers can develop. At least, there are four different Recommender implementations, which are used to suggest related products to the users. They each offer two different services, `train` and `recommend`. The `train` service trains the recommender with orders from the past, whereas the `recommend` service suggests products to the users that are related to their current shopping cart.

The developers implement these versions of `recommend` and `train` along different development iterations. These implementations have different performance characteristics. Performance tests or monitoring can be used to measure these characteristics for the current state, i.e., for the current implementation, current deployment, current environment, current system composition and the current workload. This requires,

however, effort and costs to set up the execution environment (test environment or production environment) and may require buying some hardware. In general, architects may be interested in several what-if questions and scenarios regarding the load, the user behavior, the available resources, and, last but not least, new architecture design alternatives they want to evaluate. It would be more expensive to predict the performance for various what-if scenarios, such as predicting the performance for another state (e.g., different deployment, workload, environment ...). In our example, answering the following questions is challenging based on application performance management [Heg+17]: "Which implementation would perform better if the load or the deployment is changed?" or "How well does the training service perform during yet unseen workload scenarios?" An example for the latter question would be an upcoming offer of discounts, where architects expect an increased number of customers and also a changed behavior of customers, where each customer is expected to order more items. The problem from a performance perspective is the expensive cost of evaluating the alternative algorithms of Recommender using performance tests or monitoring. In any case, it requires setting up and performing several tests/ measurements for each design alternative.

An aPM can answer questions regarding performance, scalability, and other quality aspects. Regardless of how the aPM will be built and updated (reverse engineering extraction or manual/ automatic update), all available approaches recalibrate the whole model by monitoring all parts of the source code instead of recalibrating only the model parts affected by the last changes in source code. In other words, it is sufficient to monitor the changed parts of the source code and calibrate the related parts in aPM that may be affected by these changes. The remaining PMPs should be valid. Recalibrating the whole PMPs loses potential previous manual changes and causes unnecessary monitoring overhead. Additionally, not all the available approaches detect the parametric dependencies.

3.3. The Performance Management at Ops-Time

TeaStore is a microservices-based application, cf. Section 3.1. It supports client-based load balancing. This means that the frontend requests are distributed automatically to the available deployed instances of the backend microservices. This enables automatic balancing of the overhead and automatic management of performance.

Hence, the load-balancing can cause changes in the system composition. Because the process of load-balancing can lead to replications and de-replications of the microservices. Moreover, the operator of the TeaStore may apply other changes at Ops-time like changes in the execution environment, deployment, or system composition (e.g., using another instance of an abstract microservice). Besides, the workload and user behavior

may change at Ops-time. For instance, the sales of tea during the winter season or discounts may be increased.

The problem from a performance viewpoint is that such changes at Ops-time can lead to performance issues at Ops-time. Avoiding serious performance issues at Ops-time requires careful attention to the consequences of the aforementioned changes [Tru+19]. Like evolutionary changes, predicting the impact of the changes at Ops-time using application performance management is expensive [Heg+17; Bez+19b].

Having an aPM can enable proactive performance management at Ops-time. For that, a reasonable AbPP can evaluate the impact of the expected/ planned changes at Ops-time. However, an accurate AbPP requires that aPM represents the current state of the system, i.e., the current system composition, deployment, etc. This means that aPM should be updated after each change at Ops-time in order to provide an accurate AbPP for model-based performance management at Ops-time.

3.4. The Importance of Architectural Performance Models

As explained in Section 3.2 and Section 3.3, performance management based on performance tests or measurements would be expensive, especially for the proactive evaluation of design alternatives. Otherwise, performance bottlenecks could happen.

An up-to-date aPM can answer questions regarding performance, scalability, and other quality aspects. Therefore, the goal of our approach is to provide an accurate up-to-date architecture model at any time point in the software development lifecycle. Moreover, the required manual effort and monitoring overhead for keeping the aPM should be as low as possible.

Part II.

Continuous Integration of Performance Models

4. Consistency Preservation between Software Artifacts

In this part of our thesis, we present an approach for enabling MDE during CI/CD of agile software development. The proposed approach seeks to maintain consistency across various software artifacts throughout the entire lifecycle, spanning both the development (Dev-time) and operational phases (Ops-time) of the system. The main aim is to facilitate the Continuous Integration of architectural Quality Models (CIQM) to enhance the understandability of the architecture ($P_{ArchUnderstanding}$) and to enable architecture-based quality prediction and engineering along agile SDLC (P_{Cost}). To achieve this aim, our approach observes the software system at both Dev-time and Ops-time, transforms the changes in the software models, and applies delta-based and rule-based consistency preservation of software models to update related models accordingly ($P_{Uncertainty}$, $P_{Inconsistency}$).

In modern agile and DevOps methodologies, automated processes like CI, automatic builds, automatic tests, and CD are applied. Hence, the CIQM approach can be seen as an extension of iterative software development that increases the automation level by automating the update process of aPM to enable architecture-based quality prediction for assessing design alternatives from a quality viewpoint. This addresses the P_{Cost} that is related to modeling, model updating, and quality engineering.

The current implementation of our approach considers the important quality property, the performance. Therefore, we call it "Continuous Integration of architectural Performance Models (CIPM)" to emphasize the fact that this *architectural* model enables performance prediction. However, the approach can be generalized and implemented to achieve CIQM for architecture-based prediction of other quality properties.

The novelty of CIPM is the automated consistency preservation between software artifacts for enabling Architecture-based Performance Prediction (AbPP). CIPM focuses on maintaining consistency among the source code from Dev-time, measurements from Ops-time, and primarily the architectural Performance Model (aPM). However, CIPM can be extended in future works to cover additional artifacts, such as documented architecture or configuration files stored in central repositories for supporting automatic builds, tests, or deployment.

For consistency preservation, CIPM mainly updates the aPM in response to changes in the source code and measurements, addressing $P_{Uncertainty}$ and $P_{Inconsistency}$.

The update process covers the performance parameters to allow accurate AbPP for evaluating design alternatives and enhancing the overall quality of the software system that considers $P_{Inaccuracy}$, $P_{Monitoring-Overhead}$.

The automatic performance feedback can be seen as a novel extension of the automation level achieved by iterative software development methodologies like agile or DevOps, which mainly address P_{Cost}.

In this chapter, we describe how CIPM maintains consistency between the aPM and software artifacts. For this, we define the consistency relations between the aPM and relevant software artifacts, detailing their main characteristics. Consequently, we outline a roadmap to update the aPM within a CI/CD pipeline by introducing the CIPM processes that ensure consistency and adhere to these defined relationships. These processes correspond to the first five contributions of this thesis (**C1**-**C5**).

Besides, this chapter focuses on addressing the following research question:

RQ6 How can we efficiently integrate the continuous update process of aPMs with DevOps practices to support AbPP?

To answer this question, we explain how to integrate the CIPM processes (contributions **C1**-**C5**) into the automated CI/CD DevOps pipeline to enable AbPP within DevOps-driven software development. We refer to this extended DevOps pipeline as the Model-based DevOps (MbDevOps) pipeline, identifying it as the sixth contribution of this thesis (**C6**). We describe the sixth contribution first in this chapter because it provides an overview of the CIPM approach, introducing the remaining five contributions integrated into it and helping to understand an overview of the entire approach.

MbDevOps pipeline pipeline (**C6**) primarily addresses $P_{ArchUnderstanding}$ and P_{Cost}, as it automates the preservation of consistency and ensures the aPM remains up-to-date, which helps understand the current software architecture, reducing manual effort.

In [Küh+23], we contribute to the formalization of consistency within model-based engineering. This paper, though not yet peer-reviewed, presents a shared understanding among eleven researchers from our chair, introducing a classification schema specifically structured for model consistency. In this paper, we analyze CIPM as one of the studied domains, providing a brief overview of the consistency relations relevant to CIPM and their characterizations. Besides, the concept of MbDevOps pipeline pipeline is published in [Maz+20], and a refined version is available in [Maz+25].

The structure of this chapter provides a detailed explanation of how CIPM ensures consistency and supports AbPP within DevOps environments. The subsequent section (Section 4.1) describes how CIPM maintains consistency between software artifacts and the aPM. In Section 4.2, we detail the models that CIPM keeps consistent for executing AbPP using the `Palladio` simulator.

In Section 4.3, we explain how to integrate CIPM processes into the DevOps pipeline to enable AbPP during DevOps software development (**C6**). The assumptions and limitations regarding MbDevOps pipeline are described in Section 4.4.

The remaining contributions (**C1**-**C5**) of this thesis are presented in Chapter 5, Chapter 6, and Chapter 7.

4.1. Consistency Maintenance

In this section, we examine the various aspects of maintaining consistency between the aPM and software artifacts. First, we define the meaning of consistency according to our approach in Section 4.1.1. Following this, we introduce the changes that affect consistency considered by CIPM and occur at development and operations in Section 4.1.2. Next, we establish the consistency relation and its characteristics in Section 4.1.3. Lastly, in Section 4.1.4, we explain strategies and methodologies for ensuring and maintaining consistency.

4.1.1. Consistency Definition

Consistency refers to the conformance of models to predefined consistency relations [Küh+23]. These relations verify if corresponding elements that share semantics across different models adhere to predefined consistency rules or constraints at the metamodel level, maintaining non-contradictory semantics. According to Klare's formulation Definition 1, models are considered consistent if their combination satisfies the rules defined by the relation R between models.

Therefore, consistency can be achieved through three mechanisms: defining the consistency relations and their characteristics, observing changes affecting them to detect possible inconsistencies, and synchronizing the models to resolve detected inconsistencies.

Regarding consistency relations, CIQM should adhere to at least three central relations. These relations differ mainly according to their pragmatics.

The first consistency relation is between the architecture model and software models representing the software system during development, such as the source code model. These models share semantics (designed architecture and implemented architecture) and must be free of contradictions to enhance understanding of the current software architecture and support design decisions during development.

The second one defines the consistency between the architecture model and software artifacts representing the software system during operation, such as measurements. This consistency ensures alignment between the designed architecture and the observed behavior of the system, enabling understanding of the current architecture of the running system and supporting design decisions at Ops-time.

The remaining relations define the consistency between the architecture model and software models, enabling the extraction of quality-related knowledge, such as insights into software system behavior from source code models and quality metrics from

measurements. According to this consistency relation, the architecture model must align with the software models to represent the software system accurately, thereby supporting accurate architecture-based quality prediction through the simulation of the software system's behavior.

All relations above should be checked and resolved continuously throughout the SDLC, to achieve their goals by enhancing the understandability of the system architecture, supporting design decisions, and enabling proactive identification of quality issues.

In the scope of this thesis, CIPM maintains these essential consistency relations among three models: Source Code Model (SCM), architectural Performance Model (aPM), and Measurements Model (MM). Both SCM and aPM are introduced in Section 2.2.6 and Section 2.4. Regarding MM, we define a measurement metamodel describing how measurements are organized, stored, and used for consistency preservation; see Section 4.2.2.

Like CIQM, we define the consistency relations of CIPM according to their purpose. The first one is the consistency relation at Dev-time (R_{Dev}). This relation ensures that the aPM is aligned with the implemented architecture in source code. In other words, aPM should reflect any changes occurring during software development to avoid contradictions.

The consistency relation at Ops-time (R_{Ops}) guarantees that the aPM aligns with the system architecture captured by measurements taken during operation. Operational changes can impact the aPM, causing contradictions between the real architecture captured from measurements and the modeled one aPM. Hence, R_{Ops} refers that the aPM should be updated according to the architecture driven from measurements.

The last consistency relation (R_{AbPP}) verifies the accuracy of Architecture-based Performance Prediction (AbPP) by keeping elements from the three models consistent: changed parts of source code (`SCM`), the abstract behavior that models these parts (aPM) and measurements related to these parts (`MM`).

Depending on the formalization of consistency relations as simple hyper-graphs, as presented in Definition 3 in Section 2.6.2, we can follow this concept to define consistency between software artifacts within CIPM approach based on the three consistency relations above, as follows:

Definition 4. *Given a set of software models* $\mathcal{M} := \{SCM, aPM, MM\}$ *and* $\Omega := \{m \in M \mid M \in \mathcal{M}\}$ *representing the corresponding set of all model elements* m *in* $\mathcal{M}$.

Then, according to CIPM, the consistent state adheres to the specification of the following three consistency relations*:*

- $R_{Dev} = (\mathcal{M}_{Dev}, \mathcal{E}_{Dev})$*, where* R_{Dev} *consists of the set of models* $\mathcal{M}_{Dev}$ *as well as a set of correspondences* $\mathcal{E}_{Dev}$*, whereas*
  - $\mathcal{M}_{Dev} \subseteq \mathcal{M}$ *and* $\mathcal{M}_{Dev} := \{SCM, aPM\}$*, and*

  - $\mathcal{E}_{Dev} \subseteq \{X \mid X \subseteq \Omega \wedge \forall x \in X : x \in \bigcup_{M \in \mathcal{M}_{Dev}} \wedge |X| \geq 2\}$.

- $R_{Ops} = (\mathcal{M}_{Ops}, \mathcal{E}_{Ops})$, *where* R_{Ops} *consists of the set of models* $\mathcal{M}_{Ops}$ *as well as a set of correspondences* $\mathcal{E}_{Ops}$, *whereas:*
  - $\mathcal{M}_{Ops} \subseteq \mathcal{M}$ *and* $\mathcal{M}_{Ops} := \{aPM, MM\}$, *and*
  - $\mathcal{E}_{Ops} \subseteq \left\{X \mid X \subseteq \Omega \wedge \forall x \in X : x \in \bigcup_{M \in \mathcal{M}_{Ops}} M \wedge |X| \geq 2\right\}$.

- $R_{AbPP} = (\mathcal{M}, \mathcal{E}_{AbPP})$, *where:*
  - $\mathcal{E}_{AbPP} \subseteq \{X \mid X \subseteq \Omega \wedge |X| \geq 2\}$.

The definition above states that software models are considered consistent if they adhere to the specified consistency relations. Thus, the correspondences should be checked, and any contradictions should be resolved to adhere to the joint purposes of the relations, enhancing the understandability of the system architecture, supporting design decisions, and enabling proactive identification of quality issues.

For each model involved, its elements must fulfill the joint purposes outlined by the correspondences in the consistency relations. This means a consistent state is achieved when the elements of the models align effectively according to the defined correspondences within the consistency relations. This ensures that the models accurately reflect the various aspects of the software system and can be relied upon for further analysis and decision-making processes.

The consistent state is affected by software changes occurring during the development and operation that we describe in Section 4.1.2. These changes should be observed to ensure/ restore a consistent state after these changes.

Detecting and resolving the potential inconsistencies requires understanding the characteristics of the consistency relations mentioned in Definition 4. These relations have different characteristics. Therefore, we classify the relation and their characteristics according to our schema described in [Küh+23] and introduced in Section 2.6.1. The characteristics of the consistency relations are described in Section 4.1.3. To adhere to these relations, we introduce how CIPM preserves consistency during the development and operation of software systems in Section 4.1.4.

4.1.2. Consistency-Affecting Changes

Various factors influence the consistent state during both the development and operation phases. In agile software development, frequent iterations and updates cause changes to the codebase, architecture, and requirements. These changes can impact the alignment between software models and lead to inconsistencies.

At Dev-time, one of the primary changes considered by CIPM is source code modifications. These alterations directly affect the `SCM` and can propagate to the aPM. Therefore, CIPM focuses on observing changes in the main repository of the CI process.

During the Ops-time phase, changes in system composition, execution environment, deployment configuration, and workload can significantly impact the consistent state. For instance, scaling up or down the number of servers, updating system composition, modifying deployment configurations, or altering the workload patterns can all lead to inconsistencies between the running software system in the operational environment and its architectural model. Such changes can be detected by monitoring during operation. Therefore, CIPM utilizes custom adaptive monitoring to adhere to R_{Ops} by detecting and applying these changes.

While CIPM focuses primarily on detecting and adapting to changes in source code at Dev-time and changes in system characteristics at Ops-time, certain types of changes at Dev-time are beyond its current scope. For instance, modifications to configuration files, changes in documented architecture, updates to requirements, or alterations in global configurations fall outside the scope of this thesis. Most of these changes are not reflected in source code changes. However, these changes remain essential considerations for future works aiming to expand CIPM consistency management and further achieve CIQM as described in Section 13.1.9.

In summary, understanding and managing consistency-affecting changes are related to the purpose of the architecture model. For the aPM, changes that affect the software performance should be reflected in the aPM for maintaining the accuracy and reliability of software performance prediction throughout the software development and operation lifecycle. CIPM addresses the main changes and acknowledges the expected improvement through further exploration and integration of the remaining changes.

In the following section, we introduce the characteristics of consistency relations and discuss how inconsistencies resulting from the considered changes are detected based on these characteristics.

4.1.3. Characteristics of Consistency Relations

According to our classification schema (Section 2.6.2), the consistency relations within CIPM can be classified as follows:

- **Abstraction**: All CIPM consistency relations are vertical, as the aPM serves as an abstraction of the `SCM` and the `MM`. For instance, R_{Dev} is vertical since the aPM represents the specification of the software system, while the `SCM` represents the concrete implementation of the system. Similarly, R_{Ops} and R_{AbPP} consider measurements (`MM`) as the current state of the running software system, i.e., the model with the lowest level of abstraction compared to both the `SCM` and the aPM.

- **Metalevel**: CIPM consistency relations span multiple metalevels: the aPM resides at the highest level, representing the abstract specification of the software system, followed by the `SCM`, which represents the concrete implementation. The `MM`

corresponds to the lowest level, reflecting the actual measurements of the running software system and representing the object level. Therefore, CIPM consistency relations (R_{Dev}, R_{Ops} and R_{AbPP}) between the aPM, the SCM, and the MM are inter-metalevel.

- **Position**: R_{Dev}, R_{Ops} and R_{AbPP} are inter-model relations, ensuring consistency between elements from various models. Additionally, CIPM addresses consistency within the aPM, where elements representing the static architecture, such as components, correspond to those representing the running system and deployment.

- **Property**: Consistency relations in CIPM observe both structural and behavioral properties, with R_{Ops} focusing on structural aspects such as system composition and deployment, R_{AbPP} concentrating on behavioral properties representing system behavior and performance, and R_{Dev} addressing both properties. Specifically, R_{Dev} observes structural properties like software components and behavioral ones representing the abstract behavior of provided interfaces (referred to as SEFF in Palladio terminology (Section 2.4.2)).

- **Phase**: Consistency relations in CIPM belong to different phases of the system's life cycle. While R_{Dev} and R_{Ops} are intra-phase, with R_{Dev} in the development phase and R_{Ops} in the operation phase, R_{AbPP} in inter-phase connecting elements across the development, testing and operation phase.

- **Quantification**: All consistency relations of CIPM are quantifiable.

  For both R_{Dev}, we can use JC to quantify the consistency (accuracy of the aPM) between the aPM and the implemented architecture and behavior in the SCM, see Section 8.1.3.1. Similar metrics are used by R_{Ops} for assessing the consistency between the aPM and the architecture retrieved from analyzing the measurements in the MM.

  Regarding R_{AbPP}, additional metrics such as Statistical Metrics (SMs), KS-test, and WS-distance can be employed to quantify consistency (the prediction accuracy of the aPM). These metrics can measure the discrepancy between predicted and measured performance and offer insights into the prediction accuracy.

- **Tolerance**: R_{AbPP} exhibits a tolerating relation with $0 < t < 1$, indicating tolerance for inconsistency within a certain threshold t. This tolerance is necessary because predicting performance with absolute accuracy is often impossible due to factors related to workload or environmental conditions that lead to occasional outliers in monitoring data. Therefore, R_{AbPP} allows for a margin of error within the specified threshold to adjust such variations without compromising the overall accuracy of performance predictions.

In contrast, R_{Dev} and R_{Ops} are strict, enforcing zero tolerance for inconsistencies. Any deviation from the expected architecture or configuration, whether in the software architecture (R_{Dev}) or the operational environment (R_{Ops}), is promptly identified and addressed to ensure the integrity and reliability of the system.

Table 4.1 summarizes the classification of consistency relations within CIPM based on different properties.

Table 4.1: Classification of consistency relations within CIPM

Relations	**Properties**					
	Abstraction	Metalevel	Position	Property	Phase	Quantification
R_{Dev}	Vertical	Multiple	Inter-Model	Structural & Behavioral	Development	Strict
R_{AbPP}	Vertical	Multiple	Inter-Model	Behavioral	Development/ Testing/ Operation	$0 < t < 1$
R_{Ops}	Vertical	Multiple	Inter-Model	Structural	Operation	Strict

4.1.4. Consistency Preservation

Our approach ensures consistency between software models by observing changes mentioned in Section 4.1.2, defining inconsistencies according to the characteristics explained in Section 4.1.3, and resolving them as we explain in the following.

First, we introduce CI-based consistency preservation at Dev-time (**C1**) in Section 5.1, which monitors changes in the main CI repository and automatically updates the aPM to maintain consistency at Dev-time, conforming to R_{Dev}. Here, we define consistency preservation rules at the metamodel level, responding to changes in the source code model by adjusting the aPM accordingly.

Second, **C1** updates the abstract behavior to ensure the consistency between the implemented and modeled behavior that R_{AbPP} maintains to simulate the behavior and predict the performance. Besides, our adaptive instrumentation (**C2**), described in Section 5.2, strategically inserts monitoring points into the `SCM` to achieve consistency management of R_{AbPP} across multiple phases, ensuring that the measurements required to check the consistency between PMPs of the aPM and software system are collected during testing/operation. To adhere to R_{AbPP}, the incremental calibration with parametric dependencies (**C3**), described in Chapter 6, dynamically analyzes measurements from the test environment, calibrating performance parameters to resolve inconsistencies within R_{AbPP} at Dev-time. Additionally, both Ops-time calibration and self-validation (**C4** and **C5**) also detect inconsistencies within R_{AbPP} at Ops-time. They utilize measurements collected from the operation (`MM`) to resolve the inconsistency by improving the accuracy of AbPP through recalibrating inaccurate PMPs.

Third, the Ops-time calibration (**C4**) detects and resolves the potential inconsistencies between the running system's architecture retrieved from measurements (`MM`) and the aPM, conforming to R_{Ops}. For that, **C4** applies the dynamic analysis of collected measurements to retrieve the architecture, current deployment, used resources, and load. Consequently, **C4** update the aPM, considering the measurements as the dominant model that represents the reality.

4.2. Models to Keep Consistent

In the context of consistency management within CIPM, the current implementation of the consistency relations involves three essential models: $\mathcal{M} := \{SCM, aPM, MM\}$. Future work can extend this scope to include additional models in the consistency management process (Section 13.1.9), e.g., models covering informal documentation or the configuration of used technologies.

Regarding the first model, `SCM`, CIPM assumes the presence of a source code parser that converts source code files into models, as described in Section 2.2.6. This facilitates consistency preservation at the model level.

For the remaining models (aPM, `MM`), the following paragraphs provide further details.

4.2.1. Architectural Performance Model

This subsection elaborates on the architecture performance model that CIPM maintains. The CIPM approach is adaptable to various implementations of the aPM that facilitate performance prediction through simulation. Examples of such implementations include modeling aPM using `Palladio` (Section 2.4), MARTE [OMG06][1], Descartes Modeling Language [Hub+17; Hub14; KBH14][2], ASCET [GmB24][3] or among other architecture modeling tools, which are capable of performance prediction. Generally, these models should encompass aspects like static architecture, behavior, performance parameters, deployment, usage, resource environment, and workload. To facilitate comprehension of these aspects, we describe them using terminologies from the sub-models of Palladio Component Model (PCM), given its utilization in our implementation of CIPM.

Executing AbPP using the `Palladio` simulator requires an up-to-date PCM containing the sub-models outlined in Section 2.4: (A) `Repository`, (B) `System`, (C) `Resource Environment`, (D) `Allocation`, and (E) `Usage Model`.

[1] MARTE is OMG standard that extends UML to include constructs for modeling real-time and embedded systems, including performance-related aspects.

[2] DML is an architecture-level modeling language for quality-of-service and resource management.

[3] ASCET is a tool used for modeling and simulation of embedded systems, including performance aspects.

For updating the `Repository Model` (A) and `System Model` (B), CIPM applies **C1**, CI-based consistency preservation (Section 5.1), to update these models utilizing predefined consistency preservation rules and static analysis of the source code. The update process adheres to the consistency relation R_{Dev}, which includes the `Repository Model` and `System Model` from the aPM. Regarding the PMPs of `Repository Model`, CIPM updates them to adhere to R_{AbPP}. For that, CIPM employs incremental calibration with parametric dependencies at Dev-time and self-validation at Ops-time. These processes aim to ensure that PMPs within the `Repository Model` (A) remain up-to-date at both Dev-time and Ops-time (R_{AbPP}).

To adhere to R_{Ops}, CIPM updates the following parts of the aPM: `System Model` (B), `Resource Environment Model` (C), `Allocation Model` (D), and `Usage Model` (E). To resolve potential inconsistencies, CIPM applies dynamic analysis of monitoring data. As a results, it automatically updates `System Model` (B) and the `Resource Environment Model` (C) based on the architecture retrieved from measurements, cf. Section 7.3.3 and Section 7.3.2. To update the `Allocation Model` (D) and `Usage Model` (E), CIPM integrates the dynamic analyses of the `iObserve` approach [Hei20] (Section 7.3.4 and Section 7.3.5).

As a result, CIPM processes continuously update the parts of the aPM (A-E) to keep them consistent with the running system and to enable accurate AbPP based on the updated aPM (R_{AbPP}).

4.2.2. Measurement Model

The goal of the measurement model is to preserve the monitoring data in a structural form, facilitating the consistent preservation between software artifacts. In this section, we define an excerpt of our measurements metamodel that we propose for the goal above. This metamodel, shown in Figure 4.1, describes how measurement models are managed and preserved by CIPM.

The *Measurements* root represents all measurements belonging to a software system. It enables the initialization of multiple measurement repositories for each committed version of the related software system. Each repository, termed `MeasurementsRepository`, refers to the physical storage location (`uri`) containing the measurement files, the version control system (`vcsUri`) of the related source code system, and the related version (`commit`).

The measurement repository comprises different measurement blocks, referred to as `MeasurementsBlock`, which represent a block or collection of measurements.

Each measurement is collected using a record called `MeasurementRecord`. In Figure 4.1, we present the abstract measurement record, which can be extended to gather diverse information aiding in calibrating various quality parameters. The defined measurement records related to CIPM will be explained and visualized later in Section 5.4.

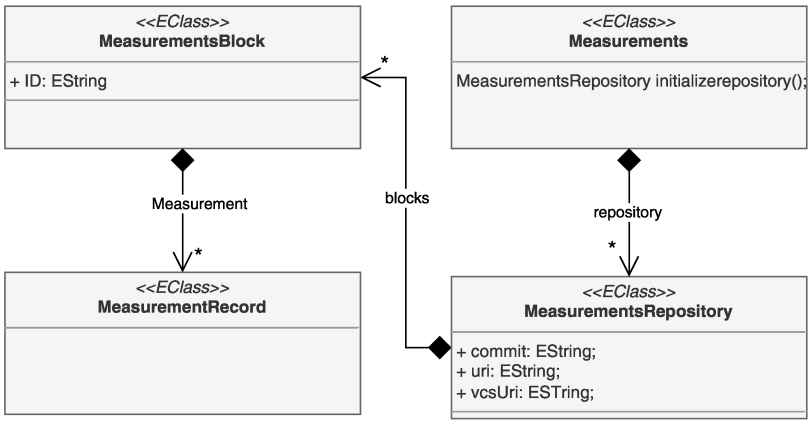


Figure 4.1: Excerpt of the measurement metamodel.

After detailing the models that CIPM keeps consistent, we describe in the following section (Section 4.3) how we integrate CIPM processes within the DevOps pipeline to automate the consistency maintenance of the aPM.

4.3. Model-based DevOps Pipeline

DevOps practices [Soc21; Bru+15] aim to bridge the gap between development and operations by integrating them into a reliable process, as we explained in Section 2.1.

In this contribution (**C6**), we extend DevOps practices to integrate and automate the CIPM approach, including (**C1**-**C5**). We named the extended pipeline "*Model-based DevOps (MbDevOps) pipeline*" because the models plays an important role within the extended DevOps SDLC, enabling the application of MDE while keeping the aPM up-to-date to address P_{Cost}, $P_{Uncertainty}$ and $P_{Inconsistency}$. This enables AbPP during DevOps-oriented development, allowing for more informed performance prediction, addressing $P_{Inaccuracy}$ and $P_{Monitoring-Overhead}$.

The novelty of the MbDevOps pipeline pipeline lies in its integration of several innovative concepts (**C1**-**C5**), along with adapted existing concepts, aimed at achieving the primary research goal: consistency preservation between software artifacts to facilitate Architecture-based Performance Prediction (AbPP). This integration is designed with an extendable strategy, allowing each concept to be further refined, either to enhance model accuracy and their AbPP capabilities or to enable proactive management that incorporates additional quality attributes.

In the following sections, we outline the main processes in the proposed MbDevOps pipeline as shown in Figure 4.2. We specifically refer to the contributions in this dissertation that are highlighted as red processes in Figure 4.2. In Section 4.3.1, we

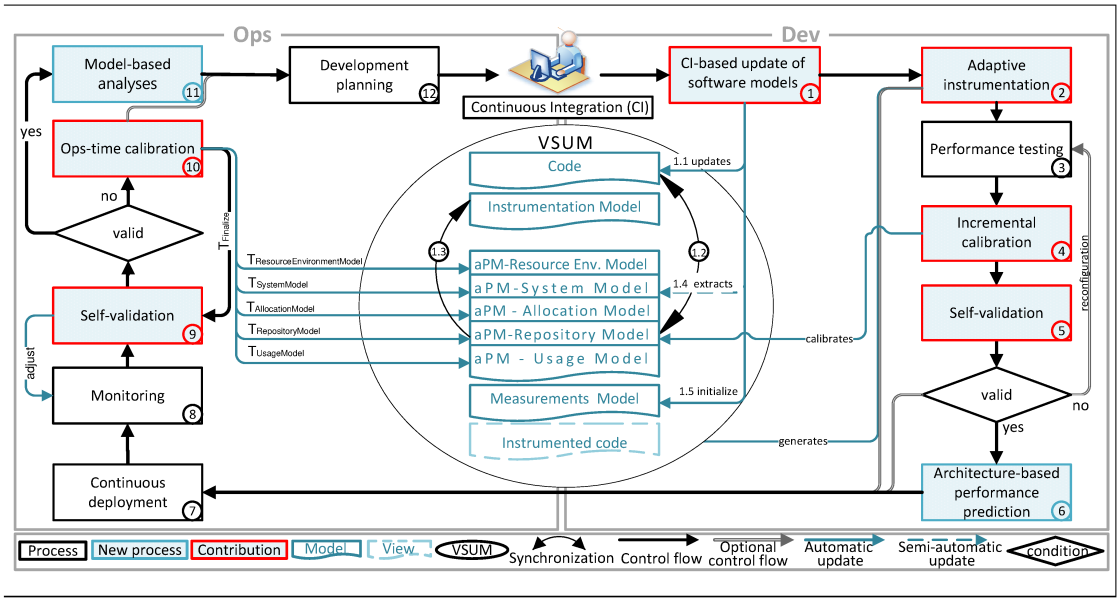


Figure 4.2: Model-based DevOps pipeline.

discuss the development phase, detailing how the processes contribute to consistency preservation based on CI-pipeline. Section 4.3.2 focuses on the operational phase of the MbDevOps pipeline pipeline, explaining how adaptive monitoring and automated calibration maintain the accuracy of the aPM and support proactive performance management.

4.3.1. Development Stage of the MbDevOps Pipeline

The development stage is the first stage of the MbDevOps pipeline shown on the right side of Figure 4.2. The development team commits the source code to a CI source control management system, such as Git.

1. The CI process triggers the first new process proposed from CIPM approach: *CI-based update of software models (1)* (Section 5.1). This process is the first contribution in this thesis (**C1**), which keeps the consistency relation at Dev-time, i.e., R_{Dev}. **C1** updates the source code model in the VSUM of Vitruvius (1.1) (Section 5.1.2). Then, the predefined CPRs in Vitruvius respond to the changes in source code by updating the `Repository Model` (1.2) (Section 5.1.3). Similarly, CPRs update the Instrumentation Model (IM) (1.3) with new probes corresponding to the recently updated parts of `Repository Model` to calibrate them later (Section 5.1.4). Besides, the first process extracts the `System Model` semi-automatically (1.4) (Section 5.1.5). Finally, a repository is created to store the measurements, and

the version of the VSUM containing the updated software models is preserved in a digital twin repository [SG22] in alignment with the corresponding commit. As a result, this process, i.e., **C1**, addresses the $P_{DevInconsistency}$.

2. The second process of MbDevOps pipeline, the *adaptive instrumentation (2)*, automatically instruments the changed parts of source code. For that, it uses the instrumentation probes collected in IM (Section 5.2), which target the changed parts in the source code. This process represents this thesis's second contribution (**C2**) and addresses P_{Cost} and $P_{Monitoring-Overhead}$.

3. The performance testing phase of the MbDevOps pipeline involves selectively conducting automated tests using the instrumented source code. This targeted approach avoids the need for comprehensive performance tests and instead focuses on the specific source code features undergoing modification. Based on our VSUM, it is possible to identify the components that require testing and test them automatically. In modern software development, tests and their configuration can be automated to ensure functional and quality attributes. Additional studies have also developed domain-specific languages for configuring and selecting the types of tests and target components [Fer21]. Based on these approaches and the knowledge of changed parts, an adaptive approach can generate and automate a minimal set of tests, as we explain in Section 13.1.8. Otherwise, benchmarking can also be executed. Previous studies have demonstrated the feasibility of incorporating automated performance benchmarks into continuous integration workflows [WEH15].

   Such automated tests can generate the necessary measurements for calibration, enabling AbPP at Dev-time. If such automated tests are unavailable, the architectural performance can be calibrated for the first time using measurements from the operational stage.

   Our MbDevOps pipeline divides the measurements resulting from performance tests into 80% training set and 20% validation set [XG18]. The training set is used in the fourth step for the calibration, whereas the validation one is used for the self-validation in the fifth step.

4. The *incremental calibration (4)* at Dev-time estimate the PMPs considering the parametric dependencies using the training set from the previous step. Then, the incremental calibration enriches the PMPs of the `Repository Model` with the estimated prediction models. This process is the third contribution of this thesis (**C3**) that addresses $P_{Inaccuracy}$ and $P_{Monitoring-Overhead}$, as we explain in Chapter 6.

5. After the calibration, the pipeline starts the *self-validation (5)* with the test data, which uses the validation set to evaluate the calibration accuracy. This contri-

bution addresses $P_{Uncertainty}$, where its results determine the level of trust in the models.

6. The sixth process in MbDevOps pipeline, *AbPP (6)*, is based on the results of the previous one, the self-validation. If the aPM is deemed accurate, it can be used to answer the performance questions using AbPP (6). If not, the user can either change the test configuration to recalibrate aPM again or wait for the Ops-time calibration.

   Answering the what-if performance questions using AbPP instead of the test-based performance prediction reduces both the effort and cost of performing this prediction before the operation, addressing P_{Cost}.

4.3.2. Operation Stage of the MbDevOps Pipeline

The second stage of MbDevOps pipeline is the operation in the production environment. This stage is shown on the right side of the MbDevOps pipeline and includes the following processes:

7. The operation starts with the *continuous deployment (7)* in the production environment, which is automated in the modern software SDLC.

8. The *monitoring (8)* in the production environment generates the required run-time measurements. The measurements in a customizable time interval are used for the subsequent processes: the self-validation (9) and Ops-time calibration (10).

9. The *self-validation (9)* is an essential process to improve the accuracy of the aPM. It compares the monitoring data with simulation results to validate the estimated aPM. The results of self-validation are used as input to the Ops-time calibration (10), addressing $P_{Inaccuracy}$. Besides, the self-validation deactivates the fine-grained monitoring of the accurate parts to manage the monitoring overhead, addressing $P_{Monitoring-Overhead}$.

10. Based on the results from the self-validation process, if the aPM is not accurate enough, the *Ops-time calibration process (10)* (Section 7.3) recalibrates the inaccurate parts. This involves updating PMPs of `Repository Model`, `Resource Environment Model`, `System Model`, `Allocation Model`, or `Usage Model` (Section 7.3). This mainly addresses the $P_{OpsInconsistency}$ by keeping the consistency between the architecture captured from measurements and the aPM (R_{Ops}).

    To adhere to R_{Ops}, self-validation (9) and calibration at Ops-time (10) are triggered frequently according to customizable trigger time to react to the new monitoring data, improve the accuracy of the aPM and to respond to the possible Ops-time changes.

11. The developers can perform more *model-based analyses (11)* on the resulting model, e.g., model-based auto-scaling and answering what-if questions based on AbPP.

12. Finally, having an up-to-date descriptive aPM supports the development planning (12). This is due to the advantages of models: increasing the understandability of the current version, modeling and evaluating design alternatives.

4.4. Assumptions and Limitations of MbDevOPs Pipeline

The effectiveness of the MbDevOps pipeline is based on several assumptions and faces certain limitations that impact its applicability in some cases. This section outlines the assumptions underpinning the pipeline's operation, followed by a discussion of the limitations that may constrain its use.

4.4.1. Assumptions

In this section, we outline the assumptions underlying the proposed MbDevOps pipeline. These assumptions are critical for ensuring the effective functioning and accuracy of CIPM in maintaining up-to-date aPMs.

1. **Availability of Testing and Monitoring Infrastructure**: The pipeline assumes the existence of a testing framework at Dev-time or a robust monitoring infrastructure at Ops-time to collect runtime measurements. These measurements are necessary for estimating PMPs such as resource demands. In cases where these infrastructures are unavailable, CIPM can allow for integrating approaches that estimate resource demand through static code analysis tools like ByCounter [KKR08]. Additionally, parametric dependencies can sometimes be estimated through static source code analysis. However, this is not always possible. For instance, if dependencies are not detected, CIPM typically builds a probability distribution based on measurement data.

2. **Availability of Performance Test Cases**: It is assumed that test cases for performance tests are available. These test cases are required to accurately assess the parametric dependencies. Otherwise, the parametric dependencies will be better learned through real measurements.

3. **Stakeholder Collaboration**: The effectiveness of the MbDevOps pipeline depends on the communication and collaboration between development, testing, operations, and quality assurance teams. This collaboration ensures that performance metrics and requirements are accurately captured and integrated into

the MbDevOps pipeline. If the aPMs are not calibrated accurately at Dev-time, testers may adjust the performance tests. Furthermore, based on AbPP results, the operations team can configure aspects like deployment. Similarly, the development team should be informed about components with low performance to evolve them in the next iteration.

4. **Digital Twin Repository Availability and Management** CIPM assumes that digital twin repositories are available for storing and managing different versions of VSUM. These repositories can support version control and archiving of model states, enabling efficient analysis and integration, particularly when diverse versions of the model run concurrently.

4.4.2. Limitations

1. **Limitation of Performance Prediction Focus**: The MbDevOps pipeline is currently limited to the processes needed for performance prediction. Extending the pipeline to other quality properties for CIQM may require additional processes.

2. **Source Code Availability**: The pipeline is constrained by the availability of source code and focuses on maintaining consistency during development. Using MbDevOps pipeline to update aPMs without access to the source code for adaptive instrumentation can lead to less efficient management of monitoring overhead, compared to the scenario where both adaptive instrumentation and adaptive monitoring are applied to reduce monitoring overhead.

3. **Challenges with Technology and Repository Diversity**: In highly distributed projects involving multiple programming languages, maintaining an accurate aPM is challenging in the current implementation of MbDevOps pipeline, as detailed in the assumptions of **C1** in Section 5.5. To address this limitation, a future work focuses on handling multiple repositories and various source code languages sequentially (Section 13.1.9.3).

4.5. Discussion

This chapter details how CIPM ensures consistency across aPM, `SCM`, and `MM` through three consistency relations: R_{Dev}, R_{Ops}, and R_{AbPP}, maintaining the consistency of the models above at different stages of the software lifecycle. The implementation of these relations demonstrates the feasibility of maintaining consistency across coupled models at different abstraction and modeling levels, capturing both structural and behavioral properties. Our implementation also indicates the feasibility of quantifying the consistency relations, including the use of measurements, throughout the development

and operational phases. This opens opportunities for defining additional consistency relations, broadening the scope to cover more quality aspects for CIQM.

To preserve consistency, CIPM monitors changes in the codebase and operational environment, addressing deviations to improve the accuracy of performance predictions. This approach can be expanded to include other change types and models into the consistency preservation process, such as configuration updates or documented architecture changes. Hence, CIPM can be considered as a foundational proof of concept for CIQM, offering practical consistency mechanisms that can support a wider range of quality attributes across software artifacts throughout the software lifecycle.

In contrast to other model-based approaches that may become outdated due to their inability to address ongoing changes, CIPM automates the update process to respond to these changes. This makes it well-suited for agile and DevOps environments, where source code is frequently pushed into CI/CD-pipeline and frequent adjustments are applied to the running software system.

To answer RQ6, **C6** integrates CIPM into DevOps workflows to reduce the cost of performance management by proactive management using AbPP on the continuously updated aPM.

The structure of the MbDevOps pipeline pipeline can be extended to support CIQM. For that, additional processes and models may be integrated, extending beyond performance to encompass other quality attributes such as security and usability.

However, there are still challenges in adapting CIPM to more complex, heterogeneous environments, including varying technologies and programming languages, as we discuss in future work Section 13.1.9.3.

The following chapters (Chapter 5, Chapter 6 and Chapter 7) describe the new processes in MbDevOps pipeline that are marked as contributions in Figure 4.2.

5. Consistency Preservation at Development Time

In this chapter, we introduce our first two contributions, **C1** and **C2**, which focuses on automating consistency maintenance at Dev-time. This automation aims to support the development of performant software in agile environments. Therefore, these contributions seek to answer the following research question:

RQ1 How can we efficiently update aPMs in response to continuously evolving source code within CI pipelines?

RQ2 How can we efficiently observe the required information about the running software system to update aPMs accordingly without introducing significant monitoring overhead?

To answer RQ1, we propose **C1** to provide a CI-based consistency preservation process that adheres to the consistency relation at Dev-time (R_{Dev}), ensuring the consistency between aPMs and the evolving source code. This is especially important in agile development environments, where frequent and rapid changes can lead to architectural drift ($P_{DevInconsistency}$). Such drift creates discrepancies between the current system state and the corresponding aPM.

To answer RQ2, we introduce **C2**, a process for adaptive instrumentation of changed parts of the source code. This contribution aims to reduce the cost of instrumenting and monitoring the source code, addressing P_{Cost} and $P_{Monitoring-Overhead}$.

The chapter begins by introducing the concept of **C1**, the CI-based update of software models (aPMs), outlining how the CI pipeline can be leveraged to ensure that aPMs and certain intermediate models remain consistent throughout the development process (Section 5.1). Then, we explain the update process for software models in Section 5.1.1. This process is realized to update the following models: the source code models in Section 5.1.2, the `Repository Model` in Section 5.1.3, the instrumentation model in Section 5.1.4, and the `System Model` in Section 5.1.5. Updating these models according to source code changes in CI-pipeline resolves inconsistencies and maintains the R_{Dev},

In Section 5.2, the chapter introduces the second contribution, the adaptive instrumentation, to inject instrumentation probes in source code to monitor the changed source code, thereby minimizing overhead while ensuring data collection for maintaining R_{AbPP} and R_{Ops}, as the following chapters explain. The implementation of adaptive

instrumentation is detailed in Section 5.3, while Section 5.4 introduces the measurements metamodel, defining the measurement records related to the probes injected by the adaptive instrumentation.

The chapter also describes the assumptions and limitations of the proposed CI-based update approach and the adaptive instrumentation in Section 5.5.

Finally, we summarize the proposed contributions and answer the research questions in Section 5.6.

5.1. CI-based Update of Software Models

The goal of this contribution, **C1**, is to adhere to R_{Dev} by updating aPM in response to continuously evolving source code.

C1 aims to ensure the consistency between aPM and the evolving software system at Dev-time. This is especially important in agile development environments, where frequent and rapid changes can lead to architectural drift ($P_{DevInconsistency}$). Such drift creates discrepancies between the current system state and the corresponding aPM. For that, **C1** updates aPM throughout the development process according to source code changes.

To address the uncertainty of aPM validity in representing the current system ($P_{Uncertainty}$), **C1** observes CI-pipelines to detect changes in the code and reflect them in the architectural performance models. This ensures the aPM remains up-to-date and addresses concerns about aPM accuracy to enhance developer trust in using it for performance-related decisions.

As a result, **C1** first addresses $P_{ArchUnderstanding}$ by providing aPM that helps understand software architectures, especially during personnel turnover or architectural evolution. Second, **C1** reduces modeling and updating costs by providing automated updates of aPM, which can be leveraged for proactive performance management rather than relying on costly, ad-hoc performance monitoring and post-development adjustments.

The main novelty compared to existing approaches is that **C1** efficiently addresses $P_{Uncertainty}$, enabling seamless integration into modern software development: **C1** provides a tool-agnostic process that does not require special development editors nor framework, facilitating seamless integration of our CIPM approach into modern software development practices, where CI-pipelines are widely used. Consequentially, developers should not adopt additional tools for consistency preservation, thus reducing the overhead associated with maintaining the aPM up-to-date (P_{Cost}).

In existing approaches for delta-based consistency preservation of software system artifacts, such as Vitruvius described in Section 2.6.3, the sequence of impacting changes, deltas, is required for maintaining the consistency. Therefore, these approaches are limited to using specialized editors to record the required sequence of source code

changes, as popular existing code editors cannot produce them. Besides, other existing approaches, as we discuss in related works (Section 12.3.1), require injecting specific annotations in the source code to extract the architecture model, limiting their usability. To avoid the limitations above, the first contribution of CIPM extracts the source code changes from the widely used version control systems, which allows developers to use their own preferred tools for development and version management.

As far as we know, our approach is the first to bridge the gap between state-based version control in code repositories and delta-based multi-view consistency preservation. This increases the feasibility of using model-based consistency preservation for source code because our contribution no longer requires change-tracking specialized editors, manual input or editors.

Hence, the first contribution of this work is inherently novel and includes advances that can be leveraged to support other scientific methods. These innovations include improving source code parsing techniques, enhancing the detection of code changes necessary for consistency preservation, and facilitating the propagation of these changes into delta-based consistency preservation tools. Additionally, our contribution proposes technology-specific detection of components and interfaces for updating the aPM and establishes appropriate consistency preservation rules for maintaining consistency in instrumentation models.

This contribution, **C1**, is published in [Maz+25]. The detailed process including advances is described in the next paragraph.

5.1.1. Overall Process

The process of **C1** involves a CI-triggered automatic process that updates various software models when source code changes are detected. The structure of this process is shown in Figure 5.1. When the developers commit their changes in the version control repository, the **C1** process is triggered. The source code parser and component detector identify relevant changes, while the state-based propagation and system model extractor ensure these changes are reflected in aPM. The VSUM of the consistency preservation platform, Vitruvius in our implementation, acts as the synchronization core and update mechanism that maintains the consistency between the software models, aided by consistency preservation rules (CPRs). The main steps of the **C1** are detailed in the following.

1. *CI-based Source Code Model Update:* This process includes several sub-processes, starting with the CI process that updates the source code repository. It is followed by parsing the updated state, enhancing the parsed source code model with information about the components, extracting the source code changes between the current and previous states, and finally updating the source code model in the

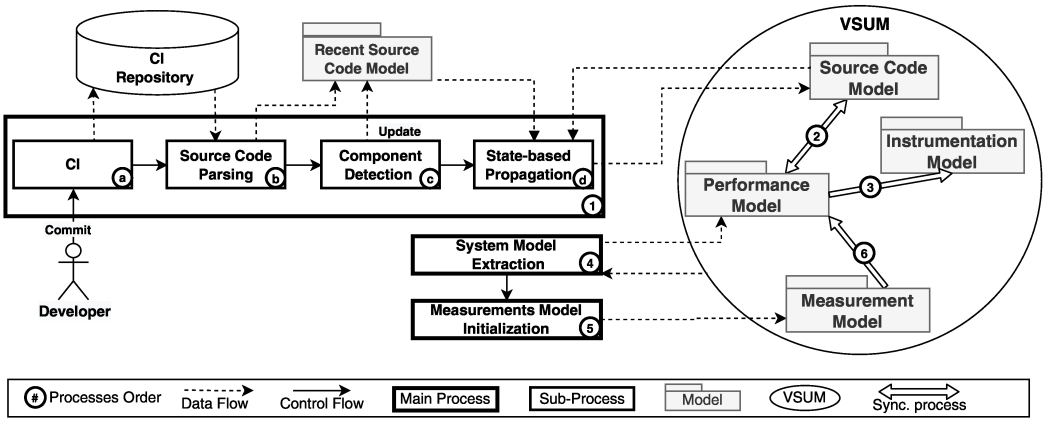


Figure 5.1: CI-based Update of the aPM.

VSUM to maintain consistency across the software system. The sub-processes are visualized in Figure 5.1 and detailed as follows:

a) *CI*: Initially, developers commit their source code changes to a version control system, such as Git in our implementation. This commit triggers the CI pipeline. In our case, the CI pipeline not only initiates the build and test processes but also triggers the CIPM framework for updating aPM.

b) *Source Code Parsing:* Since our approach relies on model-based consistency preservation, it is essential to parse the recently changed source code files in order to generate corresponding source code models.

c) *Component Detection*: Software component detection through reverse engineering cannot rely solely on source code, as components are shaped by various factors beyond the code. For instance, architectural patterns, configuration files, and directory structure all can contribute to defining software components. For example, microservices components can become apparent through configuration files. Similarly, components implemented as plugins can be detected through static analysis of configuration files, which reveals plug-in interfaces, extension points, and declared dependencies. Since the current implementation of CIPM only considers the source code model in maintaining the consistency of R_{Dev}, it is essential to include a pre-processing step after parsing the recent commit to detect these components. This step incorporates automatic static analysis of architectural patterns and file/ folder structures and can be extended in future work to include static analysis of configuration files. Then, it updates the source code model with information on the detected components from the pre-processing analysis. The

predefined CPRs subsequently process this information and update the aPM accordingly with the corresponding components.

d) *State-based Propagation*: State-based propagation is a systematic approach that leverages the differences between the current and previous states of the source code model to maintain consistency and update the corresponding software models efficiently. Focusing on the deltas—i.e., the changes between two states—enables incremental updates according to the new state. This process involves several sub-steps, including detecting changes between the states, propagating the detected changes to the consistency preservation platform (Vitruvius), and applying consistency preservation rules to synchronize changes. In the following, we provide more details on the advances made in these sub-steps.

   d.1 *Changes Extraction*: To extract source code changes from the last commit, our process compares the recently parsed state of the source code model with the previous one stored in the VSUM. We use the approach of Brun and Pierantonio [BP08] to detect the model differences through three fundamental steps described in Section 2.2.7. The first step involves calculating the differences by comparing two distinct models by *matching* the related elements and *differencing* changes. In the second step, the outcome of the calculation is represented in a difference model. Our process presents the differences as changes compatible with the used consistency preservation platform (Vitruvius changes). In the last step, the represented differences are visualized in a human-readable form so that we can report the detected changes to the user.

   A novel advancement in this process lies in the matching step. The default matching algorithm in Vitruvius uses a similarity-based approach [Kol+09], which treats models as typed attribute graphs and identifies matching elements based on the aggregated similarity of their features. Although this method is commonly employed, it often fails to accurately capture the structure and hierarchy of the source code model. To overcome this limitation, we improve Vitruvius's matching capabilities by integrating new custom language-specific algorithms introduced in Section 2.2.7. These enhanced algorithms incorporate hierarchical matching, representing a significant advancement in achieving more precise matches within the source code model.

   d.2 *Change Propagation*: This process propagates the outcome of the previous step, the atomic edit changes, into the consistency preservation platform, Vitruvius. The goal is to convert the old state in VSUM into the latest source code model. Therefore, this step propagates these

changes into the source code model in VSUM to trigger the synchronization process within VSUM and update the related models. However, change propagation faces two main challenges.

The first challenge is that the Vitruvius state-based propagation technique only propagates one resource, which is not the case in the source code model. Source code models can consist of multiple resources, representing various packages, modules, or classes with interdependencies. These resources can depend on each other. This means that dependencies must be propagated into Vitruvius before the resources that depend on them, which might be difficult when cyclic dependencies exist. Our novel solution simplifies the propagation process by combining all resources into one entity.

The source code model in the VSUM of Vitruvius is consequently updated using the change sequence. This triggers the CPRs for the aPM and `IM`. Consequently, modifications that resolve the inconsistencies are generated and applied to aPM and `IM`. The second challenge involves the order in which the elements of aPM should be updated based on changes in the source code. To illustrate, the component boundaries in aPM should be updated before maintaining consistency in the behavior of the inner components. Therefore, our solution ensures the order of modifications that Vitruvius should apply to aPM. Thus, our change propagation step arranges the modification to the aPM in the following order: module modifications come first, followed by interface changes, method signatures, and method bodies changes. As far as we know, sorting modifications to allow consistency preservation between architecture models and source code is a novel process.

2. *CI-based Update of aPM*: Our update of aPM is based on CPRs between the source code model and the aPM. These CPRs are triggered as a result of source code changes that the state-based propagation occurs. For that, we build upon existing CPRs between the source code model and the aPM [Lan17], introducing new CPRs required in the CIPM context. The new CPRs include technology-specific detection of components and interfaces. Additionally, the existing CPRs are adapted to handle the addition, updating, and deletion of aPM elements. Furthermore, we extend the CPRs responsible for reconstructing abstract behavior, which is critical to CIPM's primary goal of simulating behavior and predicting performance.

3. *CI-based Instrumentation Model Update*: We propose CPRs that create appropriate instrumentation probes based on changes in the abstract behavior of the aPM. These instrumentation probes are essential for applying adaptive instrumentation, targeting the modified parts of the source code that correspond to the updated

sections in the aPM. This instrumentation enables the collection of necessary measurements for calibrating the related performance parameters in runtime.

It is important to note that beyond the automatically generated instrumentation probes by CPRs, software architects and the self-validation process can add probes into the `IM` to recalibrate their related performance parameters if they detect that these parameters are not accurate enough.

4. *CI-Based System Model Update*: The last process in CI-based update of aPM is responsible for updating the `System Model`. For that, we introduce a novel semi-automatic method for extracting the system's composition, which details how the components in the `Repository Model` are instantiated and assembled [Mon+21]. As a result, the extracted `System Model` can guide design decisions during Dev-time by enabling the application of AbPP before deployment, thereby saving both time and effort compared to manual creation.

   The extraction of `System Model` consists of two steps:

   - The first phase seeks to unify the extraction process between Dev-time and Ops-time, by using a common structure that describes the calls between services. This structure will serve as the foundation for the extraction of the `System Model` in the second step. The unified structure, known as SCG, describes the "Calls-to" links between services, along with the related resource containers that host the individual services. To extract SCG at Dev-time, we apply static analysis on the models available in the VSUM. As a result, we create the directed SCG, whose nodes are the service and its resource container. The nodes are connected by directed edges, indicating that a service on a source node calls the one in the destination node. The proposed data structure (SCG) is not dependent on the underlying programming languages or the measurements from which it is extracted, so it can be utilized in various scenarios.
   - The second step includes a process to extract the `System Model` based on SCG. Modeling the system's boundary interfaces is the first step in the suggested procedure. Subsequently, the process searches for components that offer the boundary interfaces in the `Repository Model`. The user is prompted to select the appropriate component when multiple components offer the same interface. The necessary interfaces coming from the components must be satisfied for completing the `System Model`. As a result, the process looks for all services called by the services of the provided interfaces by scanning the SCG extracted in the previous step. Similar to previously, the called service components are also identified using the `Repository Model` in order to link the necessary interfaces to the corresponding provided interfaces. Every necessary interface is satisfied iteratively until there are none left.

5. *CI-based Initialization of Measurements Model:* This process begins with creating a physical storage to store performance measurements, then updating the measurement model by adding a repository model referencing this storage and the commit identifier, as illustrated in the measurement metamodel in Figure 4.1.

   Then, the state of the VSUM is versioned in a digital twin, corresponding to the source code's state at the time of the commit. Conceptually, this versioning can be executed within Vitruvius [Ana22] or by committing the state to another version control repository.

   At runtime, the MbDevOps pipeline pipeline can create a new block in the measurement model for each set of measurements gathered from monitoring or performance tests. Sliding window techniques can be utilized to stream and aggregate measurements in blocks. The monitoring system can be configured to write these measurements into the file system, and the files can then be stored in the corresponding physical storage, defined in the measurement model. It should be noted that the implementation of [Ana22] has not been integrated into Vitruvius platform. Thus, versioning is not supported in the current implementation. Given this limitation, and in order to avoid redundant development efforts, the implementation of processing the measurements within Vitruvius platform is deferred to future work (Section 13.1.2).

6. *Calibration of Performance Parameters:* This process is triggered as a response to updating the measurement model with new measurement blocks, and it updates the performance parameters of `Repository Model` and the remaining submodels of aPM (`System Model`, `Allocation Model`, `Resource Environment Model`, and `Usage Model`).

   Unlike previous processes (1-5), this process is not directly triggered by CI, it requires applying adaptive instrumentation and executing the source code to generate the required measurements for calibration. Therefore, we do not consider it part of **C1**; instead, we visualize it in Figure 5.1 to provide context for the subsequent process. Thus, the calibration of aPM according to the measurements will be described in detail in both Chapter 6 and Chapter 7, mainly in (Section 7.3).

The concept above is implemented for Java- and Lua-based applications that are developed based on CI-pipeline within the Git repository. Two EMF-based source code models for Java and Lua are developed, with parsers and printers, while the approach remains adaptable to other version control systems or source code models. Component detection is implemented by focusing on architectures like microservices and client servers, utilizing plugins, configuration files, and specific communication protocols. Besides, state-based propagation has evolved to identify Java and Lua code changes through language-specific model-matching algorithms. The detected changes

are propagated to Vitruvius, updating the VSUM models. Consequentially, our current implementation of **C1** updates the following models in response to the CI of the source code:

1. The *source code model* in the VSUM of the Vitruvius platform for two programming languages, Java and Lua. This update is based on the first process of **C1** utilizing static analysis of source code (Section 5.1.2).
2. The *aPM* in the VSUM of the Vitruvius platform based on the second process of **C1** utilizing VitruviusCPRs. In specific, our implementation updates `Palladio Repository Model` in the VSUM (Section 5.1.3). This includes new CPRs for detecting three different technologies and the extension or direct reuse of existing CPRs [Lan17].
3. The *instrumentation model* in the VSUM of Vitruvius, based on the third process of **C1** utilizing VitruviusCPRs (Section 5.1.4). This involves the introduction of new CPRs for instrumentation purposes.
4. The *Palladio system model* based on the fourth process of **C1** (Section 5.1.5).
5. The *measurement model* is based on the fifth process of **C1**, utilizing the Git commit identifier, ensuring that performance measurements are tied to the specific version of the source code at the time of each commit. This approach enables accurate tracking and calibration of performance data relative to the exact state of the code. We implemented the calibration of the `Palladio Repository Model` (the sixth process in **C1**) at predefined time intervals as we explain in Chapter 6 and Chapter 7.

The following subsections illustrate the realization of aforementioned models that **C1** updates.

5.1.2. Realization of CI-based Source Code Model Update

The source code model is an intermediate model that serves as the basis for updating aPM ①. The input of this process is the commits from CI in a version control system (Git in our implementation) ⓐ. The purpose of using commits as input is to allow developers to continue using their preferred development platforms and version control systems while maintaining consistency through Vitruvius's delta-based consistency approach Section 2.6.3. As a result, developers are not required to use specialized editors to generate the sequence of changes necessary for Vitruvius to uphold consistency between the source code and its model. Instead, the commits can be used to provide the changes sequence, which is then processed by Vitruvius to maintain the model's synchronization automatically.

To understand the update process, we first introduce the source code models that we extend and their parsers for ⓑ in Section 5.1.2.1. We postpone the description of the component detection ⓒ, which is specific to the technologies we consider, to a later section discussing the CI update of `Repository Model` (Section 5.1.3). We describe the source code model matching applied to enable state-based propagation ⓓ in Section 5.1.2.2.

We concentrate on detailing Java source code models since our running example is a Java-based application; see Chapter 3. It should also be noted that all excerpts and examples used in the following subsections are simplified.

5.1.2.1. Source Code Models

The source code models that CIPM should be compatible with the Vitruvius platform. CIPM utilizes an EMF-based source code model because it ensures standardization and consistency through its well-defined metamodel (Ecore), which aligns with the goal of CIPM in model-driven engineering practices. Additionally, EMF's compatibility with the Vitruvius platform supports consistency preservation at Dev-time, facilitating ensuring R_{Dev}.

For this goal, we provide the following source code models for the programming language Java [Arm22; MAK23] and the programming languages Lua [AMK23].

Java Source Code Model We build upon an existing EMF-based Java metamodel [Hei+10]. Our extension enables modeling Java versions 7-15 [Arm22], incorporating new language features such as the diamond operator, lambda expressions, and modules. To understand the Java metamodel, Figure 5.2 shows a portion of this metamodel, which represents Java classes and interfaces as specializations of Java types, including their declared members like methods and fields.

For Java code parsing ⓑ, we introduce a new version of the JaMoPP parser. This parser, based on the Eclipse JDT [VSP20] (Section 2.2.6) that generates models from Java code by retrieving AST and binding information. Our parser transforms the ASTs resulting from JDT into EMF model instances. After parsing, our parser utilizes the binding information in resolving references between EMF model elements, such as those introduced by imports or inheritance. Ideally, source code is compiled before parsing so that the binding information can be incorporated to resolve all dependencies. For scenarios where the compilation is not feasible, we have introduced a recovery strategy that creates model elements for missing dependencies, enabling the generation of models with resolved references even in the absence of a complete build [MAK23]. As a result, our parser enables parsing the Java features of versions 7-15 in EMF-based

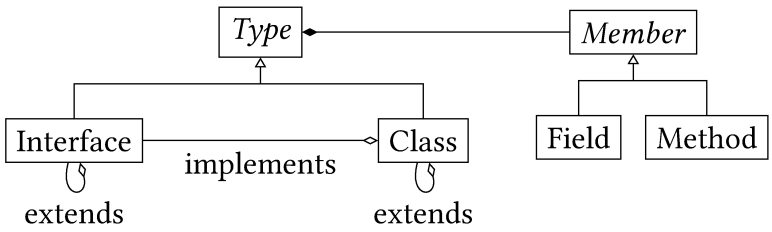


Figure 5.2: Excerpt from the Java metamodel highlighting parts to represent interfaces, classes, and their members, from [Maz+25].

format. More details on our JaMoPP extensions are available in a technical report [Arm22][1].

In addition to the parser, we developed a new version of the JaMoPP printer to output the updated Java code models in a textual format. This printer leverages the EMF-based model instances to produce code that adheres to the specifications of the extended metamodel.

Lua Source Code Model For the EMF-based metamodel, we extend an existing Xtext-based Lua grammar from the Melange project [Has]. Our extensions enhance this grammar to create a root element `Chunk` that represents the content of each Lua file. This content consists of a `Block` that serves as a container for Lua `Statements`, see Figure 5.3. Further evolution of the Lua metamodel is currently being developed in an ongoing master's thesis at our chair [Sae25].

As outlined in the foundation, Xtext provides a framework for language development, including metamodel generation, parsing, and serialization [Ecl24b], cf. Section 2.2.5.2. For Lua-code parsing ⓑ, **C1** utilizes the generated parser that translates Lua code files into EMF-models, while the printer converts them into textual form.

Source Code Model Example We illustrate the Java models through selected parts of the *Recommender* microservice code from the TeaStore [Dev24]. Listing 5.1 provides an excerpt from this code: the `IRecommender` interface defines methods for training a recommender and recommending products. Specifically, the `train` method is used to `train` the recommender system, while the `recommendProducts` method generates product recommendations based on the user's input. The `AbstractRecommender` class provides a default implementation of the `IRecommender` interface. It contains basic implementations of the `train` and `recommendProducts` methods. However, the actual recommendation logic is delegated to the abstract execute method, which must be implemented concretely by subclasses.

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

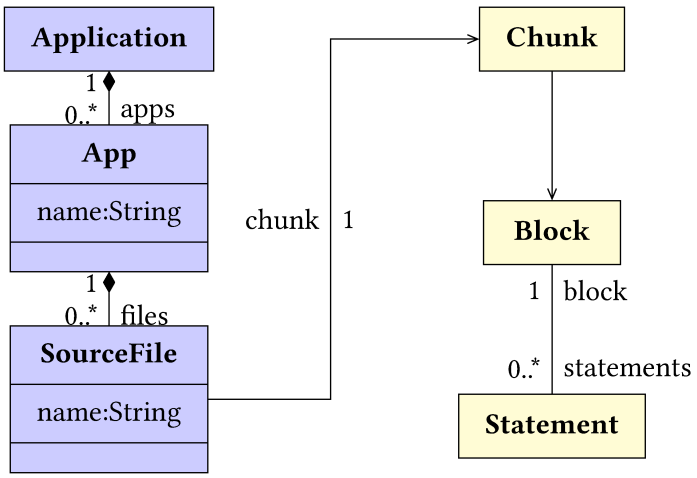


Figure 5.3: Excerpt from the Java metamodel highlighting parts to represent interfaces, classes, and their members, from [AMK23] based on [Bur23].

Parsing the code excerpt above ⓑ, results in the visual representation shown in Figure 5.4 using the extended `JaMoPP` model. The diagram illustrates how the `IRecommender` interface is modeled as an instance of the "Interface" class, consisting of two instances of the "Method" class: `train` and `recommendProducts`. These methods define the key operations that any implementing class must support. Similarly, `AbstractRecommender` is represented as an instance of the "Class" class, which implements the `IRecommender` interface. It includes the aforementioned `train` and `recommendProducts` methods, along with the execute method, which is abstract. The execute method must be implemented by subclasses, providing the concrete logic required to generate product recommendations. This structure ensures a clear separation between interface definition and the delegation of core recommendation logic to subclasses.

5.1.2.2. Model Matching Algorithm

For the fourth step of **C1**, the state-based propagation, we employ a custom language-specific matching algorithm [Kol+09] (Section 2.2.7) to detect the source code changes for Java and Lua programming languages. The implementation of this algorithm is not trivial and should be implemented for each programming language, considering its own hierarchical structure.

In the context of Java, we extend a specialized hierarchical matching algorithm, adapted from SPLevo [Kla14], and integrate it into EMF Compare to detect changes between EMF-based java models. This extension is aligned with our enhancements to the `JaMoPP` metamodel, enabling compatibility with Java versions 7 through 15. As a result, the matching algorithm accounts for the new types introduced in these versions,

```
1  public interface IRecommender {
2    public void train(List
         orderItems, List orders);
3    public List recommendProducts(
         Long userid, List
         currentItems);
4  }
5
6  public abstract class
       AbstractRecommender
       implements IRecommender {
7    public void train(List
         orderItems, List orders) {}
8    public List recommendProducts(
         Long userid, List
         currentItems) {}
9    protected abstract List execute
         (Long userid, List
         currentItems);
10 }
```

Listing 5.1: Excerpt from the Recommender code, from [Maz+25].

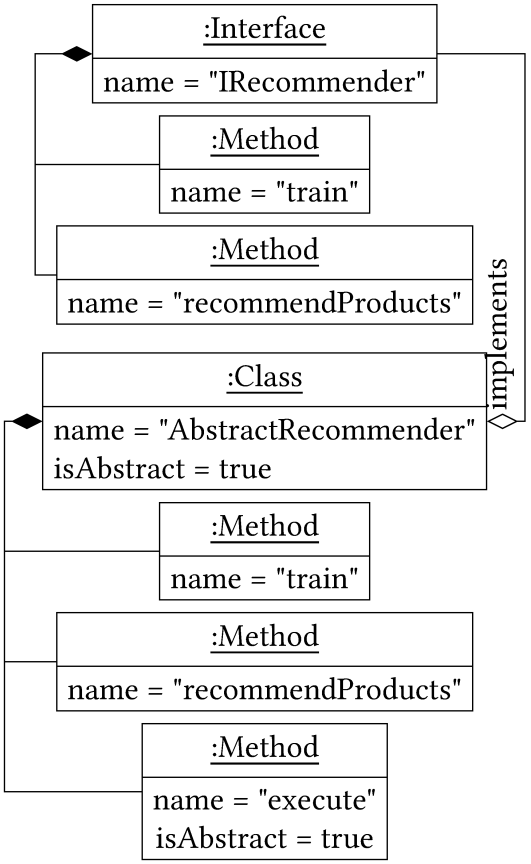


Figure 5.4: Excerpt from the Recommender Java model, from [Maz+25].

Figure 5.5: Excerpt of the source code and JaMoPP model for the Recommender component in the TeaStore case study.

while also considering the specific properties and structural details of Java to ensure accurate matching results.

Similarly, our approach is adapted for Lua models. We utilize EMF Compare to develop a specific matching algorithm tailored to Lua's unique structural characteristics. For that, we incorporate a detailed understanding of Lua's language constructs and conventions, aiming to achieve accurate element matching and efficient change detection in Lua-based software artifacts.

We acknowledge that the implementation of language-specific matching algorithms involves significant overhead. However, such custom matching is crucial for detecting changes in CIPM and other approaches for MDE. Therefore, we consider that this effort is justified, as it provides reusability across all projects utilizing the same language.

5.1.3. Realization of CI-based Repository Model Update

The realization of the second process of **C1** ② considers updating the `Palladio Repository Model` based on CPRs that are triggered due to changes in the source code model. For that, we have two groups of CPRs. One targets Java-based applications, and another targets Lua-based applications.

5.1.3.1. CPRs for Java-based applications:

Regarding Java-based applications, we extend the CPRs from the co-evolution approach that reacts to changes in `JaMoPP` and propagates them to `Palladio Repository Model`. Our extension is threefold:

1. Defining CPRs that update components and interfaces of the `Repository Model` according to the used technology and detected architecture styles.
2. Defining CPRs that reacts to updating and deleting `JaMoPP` elements.
3. Adjusting CPRs that extract the abstract behavior of the services (SEFF).

Technology-based CPRs The third process of **C1** considers additional factors like configuration files and source code structure to detect the components. Consequentially, the component detection process adjusts `JaMoPP` by adding a *module* element for each detected component. Thus, our new CPRs map module elements to the basic components. As a result, our CPRs generate a new `Palladio` basic component for each new `JaMoPP` module as illustrated in Algorithm 1.

Another example of newly defined technology-based CPRs detects REST architecture style from two technologies handling HTTP requests: JAX-RS and Jakarta Servlet. In JAX-RS, interfaces are identified based on annotations like `@Path` or `@ApplicationPath`. In Jakarta Servlet, interfaces are detected through inheritance, where classes inheriting

Algorithm 1 Pseudocode for the CPR updating the `Repository Model` after adding a new module, from [Maz+25].

```
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
```

from `HttpServlet` are modeled as interfaces overriding HTTP methods. Consequently, our CPRs create `Palladio` Interface for each `JaMoPP` class that is annotated with JAX-RS annotations or inherits `HttpServlet` as the following pseudocode of Algorithm 2 illustrates.

Algorithm 2 Pseudocode for the CPR updating the `Repository Model` after adding a new class, from [Maz+25].

```
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
```

Update and Delete CPRs CPRs that react to adding new elements are mostly reused from the coevolution approach [Lan17]. We implement, additionally, CPRs for the update and deletion of elements in `Repository Model`. Such CPRs are triggered if an element in the source code model is removed or changed, then the corresponding elements in the `Repository Model` are also removed/ adjusted. For instance, if a set of classes relating to a component is removed, the component and its interfaces are deleted. Similarly, the update CPRs react to renaming `JaMoPP` elements, as an example of update changes, by renaming the corresponding elements in `Repository Model`.

Abstract Behavior CPRs Abstract Behavior CPRs manage modifications to method bodies, such as changes or additions to statements, by reconstructing the abstract behavior model, SEFF, through reverse engineering techniques [Kro12]. Our extension to this tool extracts the mappings between source code statements and their corresponding

SEFF actions to store it in the Vitruvius correspondence model. This extension is essential for the further maintenance of the consistency at Dev-time using Vitruvius. Furthermore, these mappings are utilized in the fifth process of **C1** for `System Model` extraction at Dev-time (Section 5.1.5), and in **C2** for adaptive instrumentation (Section 5.2).

5.1.3.2. CPRs for Lua-based applications:

Similar to Java-based applications, we implement new CPRs that transform changes in the Lua model into `Palladio Repository Model`. Our CPRs include technology-based CPRs that detect industrial Lua-based sensor components and their communication. The new CPRs are proposed for updating industrial sensor applications from SICK GmbH, introduced in evaluation Section 8.3.6. In an ongoing masterwork in our chair, additional Lua-based CPRs are being defined for other architectural styles [Sae25]. Moreover, we propose new CPRs for updating the abstract behavior actions. More details are found in [Bur23].

5.1.3.3. Example of Updating the Repository Model

In this example, we illustrate how our CPRs update the `Repository Model` and add a basic component representing the `Recommender` microservice, shown in Figure 5.5. The component detection process identifies the `Recommender` based on its Maven and Docker files. Then, the component detection process updates the parsed `JaMoPP` model by adding a new module referring to the detected component. The state-based comparison detects the `Recommender` module, resulting in the ordered changes *Change*1. These changes update the `JaMoPP` model in VSUM by creating the module, adding it to the `JaMoPP` model, setting its name, and referencing the *IRecommender* interface, cf. Figure 5.6.

The addition of the module triggers the CPR described in Algorithm 1. Executing this CPR results in atomic changes *Change*2 that restore consistency by creating a new component, assigning the module name as the component name, and adding it to the `Repository Model`. As a result, the previously empty `Repository Model` is updated, and the inconsistency is resolved.

5.1.4. Realization of CI-based Instrumentation Model Update

CI-based update of the instrumentation model ③ aims to determine the required instrumentation probes based on the last commit to address the $P_{Monitoring-Overhead}$. These probes are important for the following processes in CIPM, i.e., for adaptive instrumentation and monitoring. Hence, we first propose a metamodel for `IM` that can store the required instrumentation probes, cf. Section 5.1.4.1. Secondly, we define CPRs

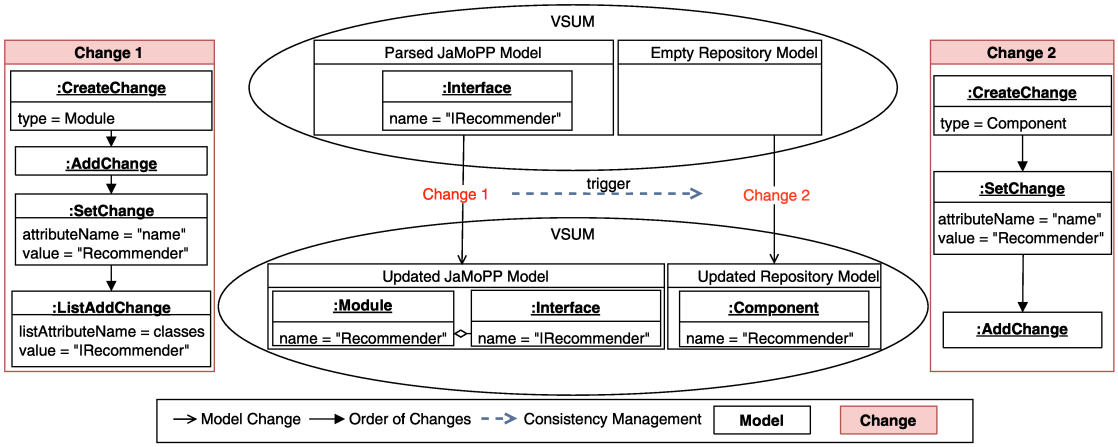


Figure 5.6: Simplified example for the CI-based update of the `Repository Model` according to change in the Java model, created based on [Maz+25].

to react to changes in the abstract behavior of `Palladio` by creating the necessary instrumentation probes and adding them to `IM`, cf. Section 5.1.4.2.

5.1.4.1. Instrumentation Model

We have designed an instrumentation metamodel based on EMF and compatible with `PCM`, which is depicted in Figure 5.7. According to this metamodel, each instance of `InstrumentationModel`, encompasses multiple instrumentation probes, i.e., `InstrumentationProbe` instances. These instrumentation probes are responsible for referring to `PCM` elements, whose performance parameters should be calibrated to instrument their corresponding source code sections, thereby enabling the collection of measurements for the calibration.

The `InstrumentationProbe` class includes a boolean attribute `active` that indicates whether the probe is currently active or not. This feature allows adaptive monitoring and reduces monitoring overhead by deactivating the instrumentation probes after calibrating the corresponding `PCM` elements.

The `InstrumentationProbe` is extended by two classes: `ServiceInstrumentationProbe` and `ActionInstrumentationProbe`. The former refers to `PCM SEFF`, `ResourceDemandingSEFF`, to instrument the corresponding service and collect the required measurements for estimating its resource demand. Since the `SEFF` consists of multiple actions, the service instrumentation probe contains sub-instrumentation probes for the actions, `ActionInstrumentationProbe`. Each action instrumentation probe refers to SEFF action, `AbstractAction`, and its type, `InstrumentationType`. The reason is that the actions have different types, and the instrumentation for each type varies. Therefore, the `ActionInstrumentationProbe` refers

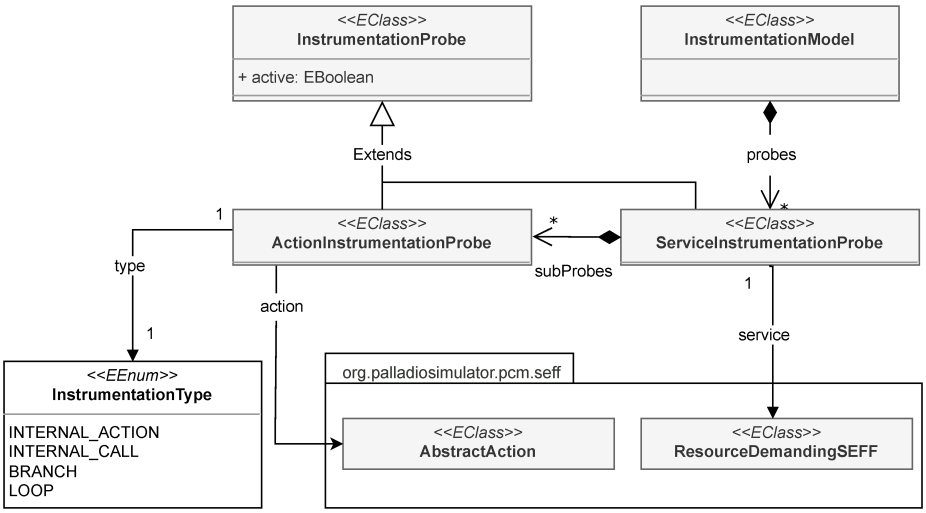


Figure 5.7: Instrumentation metamodel for providing the required measurements to calibrate PCM.

to the `InstrumentationType` enumeration, which classifies the type of instrumentation according to actions' type as `INTERNAL`, `BRANCH`, or `LOOP`.

As a result, `IM` enables monitoring of both services and their actions, supporting detailed performance tracking.

5.1.4.2. CPRs for Updating `IM`

We propose CPRs that automatically add instrumentation probes for each SEFF action that has been changed by the last commit.

The defined CPRs generate service instrumentation probes, `ServiceInstrumentationProbe`, as a reaction to updating SEFFs, whose source code statements have changed in the recent commit. Similarly, other CPRs are triggered when the actions of a SEFF are updated, create consequentially action instrumentation probes, `ActionInstrumentationProbe`, with the correct type for each changed action, and store them in `IM`. For example, an internal action probe referring to the changed internal action will be added to `IM` to provide measurements for calibrating the resource demand of the changed internal action. Similarly, loop probes and branch probes will be added to `IM` to instrument the changed loops and branches in order to calibrate their corresponding SEFF loop and branch actions.

The adaptive instrumentation utilizes the collected probes in the `IM` and the mappings in the correspondence model to generate the instrumented source code, as explained in Section 5.2.

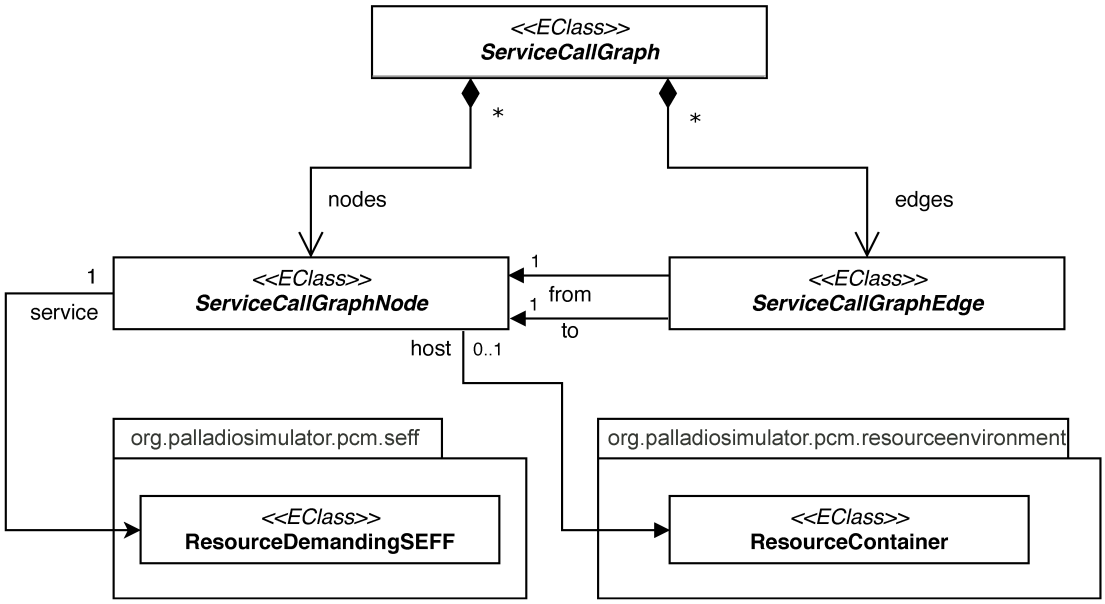


Figure 5.8: The metamodel of Service-Call-Graph from [Mon20].

5.1.5. Realization of CI-Based System Model Update

As aforementioned, we realize this process by a semi-automatic update of `Palladio System Model` ④. This enables `Palladio` simulation to provide a proactive performance assessment to support the design decision at Dev-time (e.g., proactive model-based assessment of the deployment plan).

In the following, we explain how the common data structure (SCG) is realized based on the EMF metamodel (Section 5.1.5.1). In Section 5.1.5.2, we describe how the SCG can be extracted at Dev-time. The extraction of `System Model` from the SCG, follows in Section 5.1.5.3.

5.1.5.1. Service-Call-Graph

The goal of SCG is to describe the system services' relationship and unify the structure used for system extraction at Dev-time and Ops-time. Monschein models in his master thesis [Mon20] the SCG metamodel based on the EMF metamodel [Ecl24a] to describe calls-to relationships similar to [Cal+90].

SCG metamodel is based on the directed graph [BG18]. The node represents a pair of a component's service and its resource container (hosts). However, the resource containers are not mandatory for creating SCG. They can be provided later.

The edge between two nodes indicates that the service of the source node calls the service of the destination one.

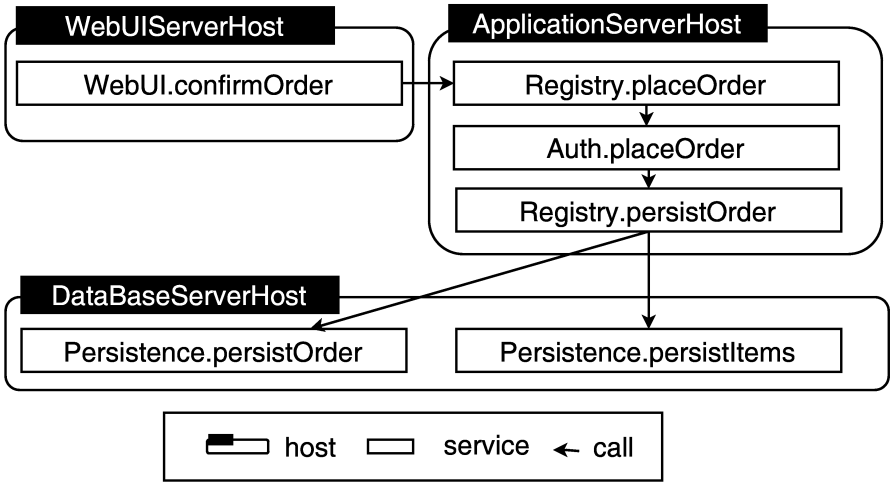


Figure 5.9: An excerpt of TeaStore SCG, based on [Mon+21].

Figure 5.9 shows a truncated simplified SCG of TeaStore. The goal is to illustrate the extraction of SCG and the `System Model` with a simple running example. This SCG simplifies the deployment and assumes that the TeaStore components are deployed on three resource containers: `WebUIServer`, `ApplicationServer` and `DataBaseServer`. According to this SCG example, the `confirmOrder` service of `WebUI` calls the `placeOrder` service of `Registry`. Similarly, Rigestry.placeOrder calls Auth.placeOrder, Auth.placeOrder calls Rigestry.persistOrder etc.

5.1.5.2. SCG Extraction at Dev-time

Three things are required to extract an SCG at Dev-time: the source code/`SCM`, the related `Repository Model` and the mapping between them. The mapping between the source code and the `Repository Model`, is provided by the CI-based update of `Repository Model` (cf. Section 5.1.3).

First, we build the source code/`SCM` to resolve the dependencies. Then, we create a method call graph by the static analysis of the source code. In some cases, e.g., inheritance or conditions, it is uncertain which call paths will be chosen at Ops-time. Therefore, extracting SCG considers all possible execution semantics. In the next step, the method call graph is traversed and transformed into a SCG based on the mapping between `SCM` and `Repository Model`, which is stored in the Vitruvius correspondence model, see Section 2.6.3. Based on the mapping, the component services are detected from the methods call graph and transformed into an SCG. At Dev-time, it is unknown where the components of the services will be hosted. Therefore, we leave this information empty.

The extraction of SCG based on the static analysis of source code is implemented and evaluated for Java [Mon20]. A static analysis at a bytecode level is executed based on [Val+10]. The analysis detects the invocation dependencies between methods to build the method call graph, which is transformed as explained into SCG.

It should be noted that in some cases, the extraction of SCG cannot be fully automatic; rather, the developer should provide additional information about the calls between services. For instance, in TeaStore, the communication between components is performed via REST interfaces. In such a case, the code only contains the HTTP addresses, and no direct method calls that can be traced. Hence, the mapping between the HTTP addresses and the REST interfaces may not be resolved solely by a code analysis. Therefore, the developer's knowledge may be required to build a meaningful SCG.

5.1.5.3. System Model Extraction from SCG

Regardless of how the SCG is extracted, the `System Model` extraction from SCG remains the same. The difference is that conflicts can occur at Dev-time due to missing information, which is not the case later at Ops-time.

In this section, we illustrate the system extraction based on the SCG, `Repository Model` and the mapping between `Repository Model` and source code. To illustrate the process, we used the simplified SCG example that we present in Figure 5.9.

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

For each system-provided interface, the `Repository Model` is searched for the components that provided it. If multiple components provide the same interface, a *"connection conflict"* occurs at Dev-time. The architect will be asked to select one of the components above. Then, assembly contexts for all selected components are created. Besides, their provided roles are linked to the system-provided interface using delegate connectors.

In our running example, `CartActions` represents the system-provided interface. Only the `WebUI` component provides this `CartActions` interface. Consequently, an instance of the `WebUI` component is created, and the `System Model` delegates its provided interface to the role `CartActions` of `WebUI`, as shown in Figure 5.10.

To complete the system model, the required roles of the added assembly contexts must be met. For that, the SCG is traversed to detect all services called by the services of the provided roles. According to Figure 5.9, the service `confirmOrder` calls `placeOrder`. Like the previous step, components that provide the called services are detected based on `Repository Model`. If multiple components provide the same required role, a *"connection conflict"* occurs at Dev-time. Then the user should resolve the connection conflict and select the right component. This is not the case for `Registry` component that provides

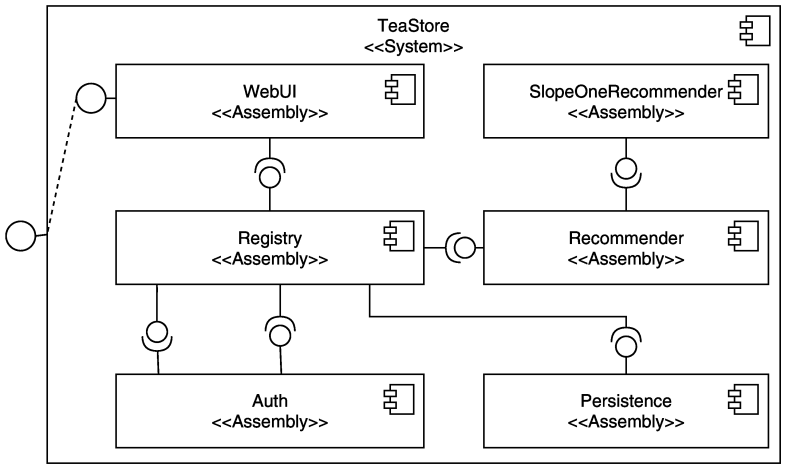


Figure 5.10: Simplified excerpt of the extracted TeaStore `System Model`.

`PlaceOrder`. If the called service cannot be detected directly from SCG like the case of REST calls, then a connection conflict also occurs and the architect is asked to select the component providing the required interface. As we will explain in Section 7.3.3, the connection conflicts do not occur at Ops-time because the SCG extracted from measurements determines which component instance is called.

In the next step, an assembly context for each selected component is created and added if no one is available in `System Model`. Otherwise, an "*assembly context conflict*" occurs. At Dev-time the architect resolves it by deciding whether to use the available assembly context or to add a new one. Again, the "*assembly context conflict*" is resolved automatically at Ops-time as we explain later in Section 7.3.3.

In our example, an instance of `Registry` component is added and no conflict at this stage occurs.

Afterward, the required roles are connected with the related provided roles. Each required role of the recently added assembly contexts is recursively satisfied by adding the required component instance or connecting to a previously added one until all required roles are satisfied. In our example, the required roles of `Registry` are satisfied by adding instances of components that provide them (`Auth`, `Persistence` and `Recommender`). Subsequently, the required role of `Auth` is satisfied, since "Auth.placeOrder" calls "Registry.persistOrder" as shown in Figure 5.9. In this case, an "*assembly context conflict*" occurs because an instance of the `Registry` component is already available. As shown in Figure 5.10, the occurred *assembly context conflict* is resolved using the existing instance of `Registry` instead of creating a new one.

In the last step, the required role of `Recommender` is met. Since there are multiple components that provide this interface, the architect is asked at Dev-time.

Following this process, the `System Model` is updated to represent the current state of the software system. As aforementioned, this process is followed by initializing the measurement model with a new repository corresponding to the current state of the software model. Consequentially, **C1** maintains the consistency between software models as defined by R_{Dev}. To complete the consistency preservation at the next stages of the SDLC. The next contribution (**C2**) injects the instrumentation probes into the source code to collect the required measurements for the calibration ⑥, and thereby maintain the remaining consistency relations that CIPM currently considers, i.e., the R_{AbPP} and R_{Ops}.

5.2. Adaptive Instrumentation

The goal of **C2** is to facilitate maintaining the consistency defined by R_{AbPP}. To achieve this, **C2** injects instrumentation probes into the source code to monitor the changed parts of the code, providing the necessary measurements for calibrating the corresponding elements in aPM.

Manual instrumentation is an expensive and error-prone process. Therefore, **C2** aims to reduce this cost, P_{Cost}, through automatic instrumentation. Additionally, full instrumentation of the source code introduces monitoring overhead, $P_{Monitoring-Overhead}$. To address this, **C2** applies adaptive instrumentation, targeting only the changed parts of the source code to minimize monitoring overhead.

Adaptive instrumentation introduces a novel approach by selectively focusing on instrumenting the changed parts through automatic detection of code changes, thereby reducing monitoring overhead ($P_{Monitoring-Overhead}$) and costs (P_{Cost}). This automated process minimizes the errors and effort associated with manual instrumentation. Furthermore, adaptive instrumentation is model-based, functioning at a high abstraction level, allowing for easy implementation across various monitoring systems and programming languages. The instrumentation captures specific performance-related information based on the detected code changes. Its design is extendable to monitor additional valuable quality attributes for CIQM.

Regarding the publications related to this contribution, **C2**, is introduced in [Maz+20] and extended in [Maz+25]. We extend the adaptive instrumentation concept by enhancing its generalizability and addressing compilation error scenarios. Our extension injects instrumentation at the model level, using EMF-Models to separate the process from implementation specifics like programming language or monitoring tools.

To illustrate the goal and process of the adaptive instrumentation, we utilize a simplified example of our running example in Section 5.2.1. The detailed process, including advances, is described in Section 5.2.2.

5.2.1. Running Example

In this example, we suppose that the `Auth.PlaceOrder` service, simplified in Listing 5.2 is newly added. According to our assumption, a new abstract behavior for this service is extracted by the second process of **C1**, the CI-based update of `Repository Model`. The abstract behavior is visualized according to our implementation using `Palladio SEFF` in Figure 5.11. However, the performance parameters of this `SEFF` should be parametrized according to measurements collected by running time. Hence, the goal of the adaptive instrumentation is to provide the required measurements by injecting an instrumentation code.

```
1  public void placeOrder(int items) {
2
3     prepareOrder();
4
5     int orderId = Registry.persistOrder();
6
7     for (int i = 0; i < items; i++) {
8        Registry.persistOrderItem(i,orderId);
9     }
10
11    finalizeOrder();
12 }
```

Listing 5.2: A simplified form of the placeOrder, ignoring the used technology "REST" and representing the internal action code as an internal call.

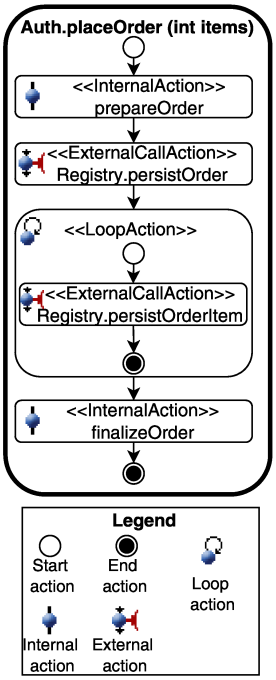


Figure 5.11: PlaceOrder SEFF.

The required information varies based on the performance parameter that needs to be calibrated. For example, estimating the resource demand of the internal action probes of placeOrder (`prepareOrder` and `finalizeOrder`) requires two probes to monitor their response times in addition to utilization monitoring. For the external calls `Registry.persistOrder` and `Registry.persistOrderItem`, two probes must monitor the return values to calibrate the data flow through external calls. For calibrating the loop action, an additional probe should capture the number of executions. Similarly, a probe should monitor which transition is selected in case of a branch action. We refer to the instrumentation for calibration of `SEFF` actions as fine-grained instrumentation.

Additionally, coarse-grained instrumentation at the service level is needed for self-validation and tracing calls to update aPM at Ops-time. Therefore, the last probe at the service level of `placeOrder` tracks response time, input parameters, caller at Ops-time, and deployment details. As a result, a simplified instrumented code is in Listing 5.3.

According to our assumption that only `placeOrder` is newly added, this implies that only this service is fine-grained instrumented as explained earlier. For the remaining services, where the source codes remain unchanged, their previously estimated PMPs stay valid. Only coarse-grained service-level instrumentation is required for self-validation and for calibrating `System Model`, `Allocation Model`, and `Resource Environment Model`.

Additionally, all probes should be capable of being deactivated to manage the monitoring overhead.

```
1  public void placeOrder(int items) {
2     /**This is a simplified pseudo code of placeOrder instrumented source code.**/
3     placeOrderExecutionData = getParametersAndTime();
4
5     tin_prepareOrder = getCurrentTime();
6     prepareOrder();
7     logInternalActionResponseTime(prepareOrder.ID,placeOrder.ID,tin_prepareOrder);
8
9     int orderId = Registry.persistOrder();
10
11
12    loopCount = 0;
13    for (int i = 0; i < items; i++) {
14       Registry.persistOrderItem(i, orderId);
15       loopCount++;
16    }
17    logLoop(loop.ID, placeOrder.ID, loopCount);
18
19    tin_finalizeOrder = getCurrentTime();
20    finalizeOrder();
21    logInternalActionResponseTime(finalizeOrder.ID, placeOrder.ID,
          tin_finalizeOrder);
22
23    logServiceResponseTime(placeOrderExecutionData, placeOrder.ID);
24 }
```

Listing 5.3: A simplified instrumentation of the placeOrder.

5.2.2. Overall Process

The adaptive instrumentation process, presented in Algorithm 3, involves the following steps:

Algorithm 3 Adaptive Instrumentation Overall Process

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

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

1. *Initialization*: A copy of the source code model is created to preserve the original state of the software source code in VSUM. As a result, the developer can continue evolving the original source code while running the instrumented version independently. This ensures that instrumentation does not interfere with ongoing development. This step corresponds to line 2 in Algorithm 3 (`codeModelCopy`).

2. *Model-based Instrumentation:* For each probe in the instrumentation model (Line 3), an instrumentation code suitable for a monitoring tool and source code model is generated and injected into the source code model as follows.

    a) *Concrete probe creation*: A concrete probe for the given source code model is created in this step (Line 4). As shown in Figure 5.12, we extend our instrumentation model to allow each probe to be implemented to be suitable

for different monitoring tools. For that, we used the bridge design pattern, which enables each probe to delegate the generation of its instrumentation code to a concrete monitoring tool, like `Kieker` (Section 2.1.3 in our design.

b) *Probe instrumentation code generation:* In this step (Line 5), we generate the instrumentation code model compatible with the type of probe, the source code model, and the monitoring tool.

c) *Instrumentation code location calculation*: In this step, the locations where the generated instrumentation code should be injected are determined. This is achieved by using the mapping between the source code and the elements of aPM, to which the probe belongs. This mapping is maintained in the Vitruvius correspondence model in our implementation.

d) *Compilation error prevention:* This step ensures that the monitoring instrumentation code does not cause issues with the control flow, such as being incorrectly placed after a return statement, which could lead to dead code or compilation errors. In Section A.3, we show an example of a source code instrumented in our evaluation. The example shows how this process of adaptive instrumentation adjusts the instrumentation code accurately to prevent compilation errors.

e) *Instrumentation code injection*: In this step, the instrumentation source code for a probe is injected into the calculated location.

3. *Dependency injection:* Line 9 ensures that the dependencies of the monitoring tool are added.

4. *Printing:* Line 10 prints the instrumented source code model to source code files, making it easier to run in a test/production environment.

After injection, the instrumented source code model is ready for deployment in the test/production environment to gather measurements for calibration. As a result, the adaptive instrumentation adaptively collects the required measurements for calibrating the performance model.

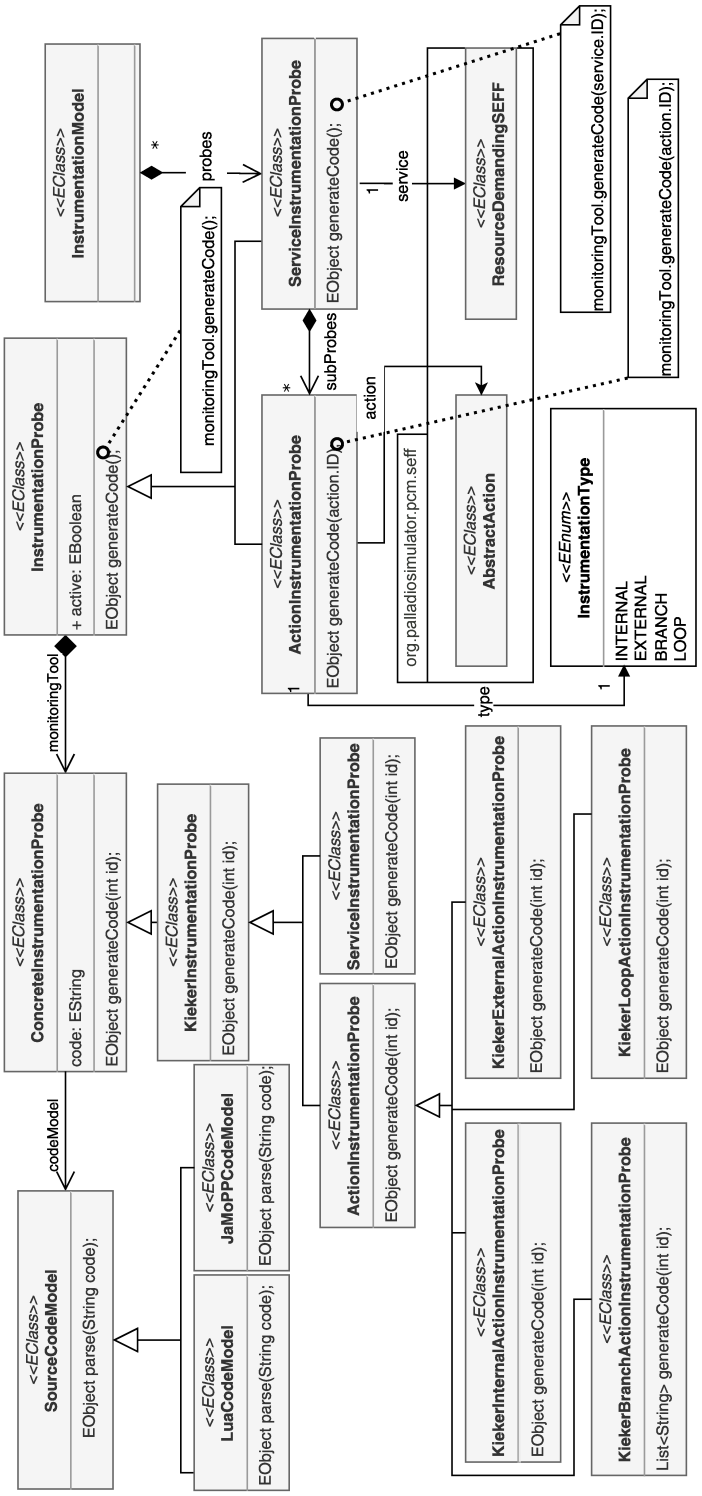


Figure 5.12: Extension of Instrumentation Model for providing instrumentation code for monitoring Tools.

5.3. Realization of the Adaptive Instrumentation

We implement **C2** using the `Kieker` monitoring tool (Section 2.1.3), which supports defining customizable probes to monitor software systems using Kieker's domain-specific record instrumentation language. This language allows for the creation of custom probes in addition to using predefined ones, such as probes for monitoring CPU utilization.

Hence, we define the following probes:

- `Service probe`: monitors input parameters for estimating parametric dependencies, caller for building SCG, current deployment for updating `Allocation Model`, and service execution time for self-validation. Regarding input parameters, we employ heuristics similar to those described in [Kro12, Chapter 5.10.4] to monitor certain properties of input parameters (e.g., the number of list elements or file size). This allows us to later investigate whether PMPs depend on these properties.
- `Internal action probe`: tracks the execution time for internal actions.
- `Loop action probe`: tracks loop iterations.
- `Branch action record`: monitors branch selection in conditional statements.

Additionally, we extend the measurement models to store the measurements provided by these probes there, as explained in the following section.

5.4. Measurement Metamodel

To process the measurements resulting from the probes in the previous section, we extend the measurement model with `Kieker` records corresponding to the defined probes, as shown in Figure 5.13.

- The service context record extends both the session context record and the host context record. Regarding the session context, this extension is necessary because our Ops-time calibration aims to capture user behavior within the observed session to keep the usage profile up-to-date. Additionally, the host context captures host information to continuously update the available resources and the allocation of the software system components on them.

  Beyond these, the service context record includes extra properties that our monitoring framework captures for the calibration of performance parameters and application of self-validation. These properties encompass:

  - The input parameters properties (e.g., type, value, number of list elements, etc.) that should be considered later as candidates for parametric dependency investigation (`parameters`).

  - The caller of this service execution (`callerExecutionID`) to learn the parametric dependency between the input parameters of both caller and callee services (`serviceID`), and calibrates the SEFF external call accordingly (`externalCallID`).
  - The allocation context that captures where the component offering this service is deployed (`hostID, hostName`).
  - The execution time of this service to be used as a reference for the self-validation process (`entryTime, exitTime`).

- Internal-action record to monitor the response time of the internal actions and use it for resource demand estimation (`entryTime, exitTime`).
- Resource utilization record to monitor the utilization (`utilization`) of a resource with the identifier `"resourceID"` in a time stamp (`timestamp`).
- Loop record to monitor the number of loop iterations (`loopIterationCount`).
- Branch record to monitor the selected branch (`executedBranchID`).

After the adaptive instrumentation (**C2**) and running the instrumented source code, the measurements are stored in the measurements records defined above and are ready to be used for the calibration, as we describe in the following chapter (Chapter 6).

5.5. Assumptions and Limits

The first contribution of CIPM has the following assumptions:

- *Source code is dominant:* Current implementation integrates the initial commit based on CPRs applied on the source code. A manually modeled performance model can be used as a basis for the Incremental Reverse Engineering (IRE) process. However, according to our current assumption, we update aPM based on the source code. We will describe the case of existing an aPM in the update process (e.g., a manually modeled aPM) in a planned future work; see Section 13.1.5.
- *Each internal action is demand-dominated by one resource:* We assume that each internal action or internal call is dominated by a single resource. According to this assumption, our adaptive instrumentation injects source code to monitor the execution time using one monitoring record. However, it is conceptually possible to inject a source code to monitor the internal action code with more than one monitoring record. For instance, the disk demand can be detected by the static analysis based ob the used libraries. This allows accurate calibration of the resource demand, see Section 13.1.12.

- *Technology Invariance:* We assume that technology is staying the same along the software SDLC. If that is not the case, the methodologist must adjust the CPRs to update the aPMs correctly.

The consistency preservation at Dev-time, including **C1** and **C2**, has the following limitations:

- *EMF Standard Limitation:* Our consistency preservation is limited to models based on EMF. Thus, applying CIPM requires a source code parser and printer compatible with the EMF-based consistency preservation platform. To evaluate the CIPM approach, we update `JaMoPP` and implement an EMFText-based parser and printer for the LUA programming language. However, future research focuses on extending support to other models, such as JavaScript Object Notation (JSON), and minimizing the initial overhead for applying CIPM in practice. See Section 13.1.9.3.

- *CPRs are Technology-based:* The predefined CPRs are currently related to the technology of the project. For instance, in the context of CIPM, we define CPRs for microservice-based and industrial IoT-based projects [Bur23]. New technologies without pre-existing CPRs necessitate initial definition and implementation. However, ongoing future work is working to generalize these CPRs, see Section 13.1.3.

- *Language-specific CPRs:* The CPRs in CIPM are language-based, requiring a distinct modeling environment for each programming language used within a system. Currently, environments are available only for Java and Lua. In systems that involve multiple languages, the CI-based strategy updates each part sequentially according to its language. Future work aims to explore better solutions through generalized CPRs to improve integration across diverse programming languages.

- *Handling Multi-language and Multi-repository Systems*: The current implementation assumes that the system is a single programming language and maintained in one repository. While CIPM can accommodate multiple languages and repositories, it suggests handling repositories and languages sequentially, utilizing the corresponding technology-specific and language-specific CPRs. Integrating these cases may pose challenges in automating this process, including human configuration during interactions in addition to generalizing CPRs would be required, which requires further research, see Section 13.1.3.

- *Static Adaptive Instrumentation*: Our adaptive instrumentation (**C2**) is based on static instrumentation, which must be injected before executing the application. Unlike dynamic instrumentation, which allows for the injection of instrumentation probes at runtime without restarting the system, this approach is inherently less flexible. However, static instrumentation should be applicable, as we assume that CIPM supports agile SDLC practices, where the source code is available and

accessible. Furthermore, the presence of monitoring probes in the source code does not inherently introduce monitoring overhead; rather, it is their execution that may lead to overhead (Section 2.1.3). Besides, our fine-grained instrumentation cannot be realized based on AOP. For instance, loops are irreversibly removed during bytecode generation, rendering them undetectable without additional measures.

Future work could explore the integration of static and dynamic instrumentation to enhance flexibility during runtime. This could involve extending the mapping between aPM and the source code to encompass the bytecode model. However, implementing such dynamic fine-grained monitoring may introduce performance overhead if numerous join points are intercepted, particularly when various join cuts, such as input parameters, are involved.

- *Limitations in Source Code-Based Updates:* The current CI-based update of software models is restricted to changes in the source code. Updates involving configuration files and runtime parameters are not covered and are defined as future work (Section 13.1.9). This limitation affects the ability to fully automate the reverse engineering of self-adaptive systems, as static analysis must extend beyond the source code to include configuration files and other settings.

- *Limitations in Static Analysis*: To detect dependency injection based on static analysis, technology-specific CPRs can scan for patterns (e.g., constructor signatures, setter methods) and specific annotations (e.g., @Autowired, @Inject). For example, Langhammer has established such rules for Google Guice [Lan17], leveraging its adherence to the JSR 330 standard. This alignment with JSR 330 suggests potential adaptability to other dependency injection frameworks. However, extracting the `System Model` and abstract behavior relies on resolving dependencies, which is not always feasible solely through static source code analysis. Frameworks like Spring, for instance, often specify dependency injection in external configuration files, not directly in the codebase. Due to CIPM's current limitation to static analysis, conflicts arise during extraction at Dev-time when users are prompted to select instances for inclusion in `System Model`. Including configuration files in the update process in future work (Section 13.1.9) may reduce user interactions.

  Regarding the `SEFF` extraction, we follow currently Langhammer's assumptions that developers avoid programming language features that instantiate classes within the services. The reason is that developers may employ reflection to instantiate classes that invoke external methods. The current implementation does not address this case. However, we define a future work that models the dependency injection within `SEFF` using a branch action conditioned on dependency types, which allows dynamic update of the `SEFF` at Ops-time, see Section 13.1.10.1.

Future work on SEFF extraction could integrate a branching action to represent dependency injection, with branch conditions guarded by the available dependency types. Adaptive instrumentation could add a specific branch probe to capture dependency types at Ops-time time, updating the SEFF dynamically during operations.

- *Limitation in Updating aPM for Self-Adaptive Systems:* Simulating self-adaptive systems requires not only providing an accurate aPM but also configuring the simulator with knowledge that defines the conditions for triggering self-adaptation actions according to simulation measurements [Bec+13]. Due to the focus on source code changes, the automatic extraction of self-adaptation logic from configuration files and run-time parameters is outside the scope of this work. Consequently, the load balancing logic for further evaluation of CIPM was manually modeled using a random distribution across available microservice instances (Section 8.3.3.1). This approach does not dynamically adapt to varying loads, which may reduce the simulation results.

5.6. Discussion

We conclude that the first contribution of CIPM, **C1**, propose a process for updating the software models according to commits in CI pipeline to maintain the defined consistency relation Dev-time (R_{Dev}).

This applies to the following models: source code model, `Repository Model`, `System Model` and `MM`. However, the update of the `System Model` and `Repository Model` at Dev-time is not fully automatic in some cases: 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`.

It should be noted that the realization of **C1** maintains the consistency between the software artifacts that are required to achieve CIPM. For CIQM considering further quality aspects, further software artifacts may be included in the consistency preservation process at Dev-time.

As a result, **C1** first addresses $P_{ArchUnderstanding}$ by providing aPM that helps understand software architectures, especially during personnel turnover or architectural evolution. Second, **C1** provides automated updates of aPM, which can be leveraged for proactive performance management rather than relying on costly, ad-hoc performance monitoring and post-development adjustments. Third, **C1** provides a tool-agnostic process that does not require special development editors nor framework, facilitating seamless integration of our CIPM approach into modern software development practices, where CI-pipelines are widely used. Consequentially, developers should not adopt

additional tools for consistency preservation, thus reducing the overhead associated with maintaining the aPM up-to-date (P_{Cost}).

Thus, the response to [RQ1], "How can we efficiently update aPMs in response to continuously evolving source code within CI pipelines?" is clarified through this contribution.

The second contribution (**C2**) answers [RQ2], "How can we efficiently observe the required information about the running software system to update aPMs accordingly without introducing significant monitoring overhead?" as follows:

C2 introduces an automatic process for injecting instrumentation probes directly into the source code. This automation reduces the manual effort and time typically associated with instrumenting the software to collect measurements. By strategically placing these instrumentation probes, the necessary measurements required for calibrating the aPMs are gathered without the need for manual intervention.

Moreover, this approach ensures that only the relevant parts of the system are instrumented, aiming to minimize the overall monitoring overhead ($P_{Monitoring-Overhead}$). This selective monitoring targets collecting data critical to maintaining the consistency relations defined by R_{AbPP} and R_{Ops}.

Furthermore, **C2** supports adaptive monitoring, which allows the system to dynamically activate or deactivate instrumentation probes based on the state of the system and the calibration requirements. This ensures that unnecessary data collection is avoided.

Consequentially, **C2** provides a fully automated, low-effort solution to observe and collect the required system information, allowing calibrating aPMs for CIPM that enable R_{AbPP}, addressing both P_{Cost} and $P_{Monitoring-Overhead}$.

Regarding **C2**, an alternative approach would be to use another way to instrument the source. For instance, it can insert probes at specific points in the program's execution flow, called join points based on the aspect-oriented programming (AOP), automatically. However, relying solely on AOP has its limitations, particularly when it comes to fine-grained monitoring. AOP is ideal for capturing method-level events but struggles with complex internal code structures, such as nested loops or conditional branches.

The realization of **C1** considers the measurements required for CIPM. However, **C2** can be extended to collect further measurements for CIQM. Additionally, the instrumentation can extended to be compatible with various monitoring tools.

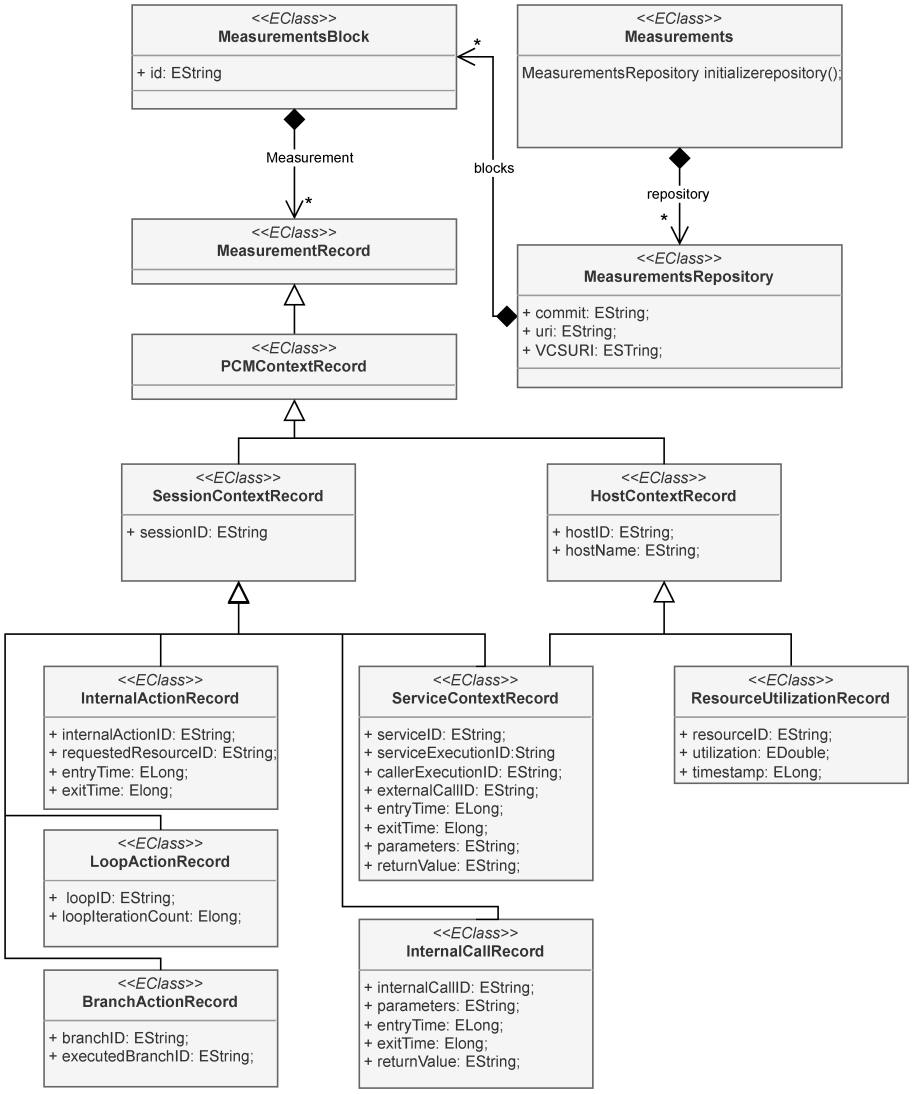


Figure 5.13: The measurements metamodel.

6. Incremental Calibration of Performance Model Parameters

In this chapter, we explain how we preserve the consistency between the performance model parameters and measurements to support design decisions through performance prediction, conforming to R_{AbPP}. Our goal is to answer the third research question:

RQ3 How can we calibrate the performance parameters with low monitoring overhead, considering parametric dependencies to support design decisions?

To answer RQ3, we propose our third contribution, **C3**, that calibrates the performance models to facilitate performance prediction at both Dev-time and Ops-time, considering the parametric dependencies and addressing $P_{Inaccuracy}$. Our calibration is based on adaptive monitoring to reduce the monitoring overhead and address $P_{Monitoring-Overhead}$.

At Dev-time, **C3** is important to enable AbPP before deployment, allowing for evaluating design alternatives without the need for expensive test-based predictions. At Ops-time, the incremental calibration seeks to improve the accuracy of the PMPs based on the real measurements. The calibration of PMPs at Dev-time is similar to that at Ops-time, the key distinction lies in the use of measurements: at Dev-time, data from the test environment is used, while at Ops-time, calibration relies on data from the production environment. Hence, the Dev-time calibration step can be skipped, and the PMPs will be calibrated for the first time at Ops-time, as detailed in Section 7.3.

The novelty of **C3** lies in its incremental calibration approach, which only calibrates the PMPs affected by the last changes. This reduces the overhead compared to traditional calibration methods, where the PMPs can be outdated according to software changes or should be recalibrated from scratch, causing monitoring overhead and ignoring manual changes, like changes to parametric dependencies. Furthermore, it incorporates self-validation mechanisms that dynamically detect inaccurate PMPs and trigger recalibration, ensuring the continuous accuracy of performance models. This dynamic, fine-grained calibration approach enables quicker adaptation to code changes in agile SDLC while maintaining high prediction accuracy with minimal monitoring overhead. To the best of our knowledge, **C3** is the first approach to update PMPs while considering both monitoring overhead and parametric dependencies.

This contribution was published in [Maz+20], where the incremental calibration process and its effectiveness in AbPP. The study also demonstrated the feasibility

of leveraging parametric dependencies to improve the accuracy of AbPP ($P_{Inaccuracy}$). Further validation of these dependencies was presented in a subsequent publication [Von+20], which confirmed their significance in enhancing predictive accuracy.

This chapter explores the process, methodologies, and considerations behind incremental calibration utilizing a running example in Section 6.1. Then, we introduce an overview of the incremental calibration process and its role in updating PMPs (Section 6.2). Next, the chapter details the specific estimation strategies for different PMPs beginning with the number of iterations for loop action in Section 6.3 followed by the selected transition of branch action in Section 6.4 and the parameters of external calls parameters in Section 6.5. The estimation of the resource demands and the parametric dependencies is in Section 6.6.

After that, we describe the adaptive optimization of the identified parametric dependencies using a genetic algorithm (Section 6.7). The chapter also includes an explanation of the self-validation process during development time (Section 6.8) and concludes with a discussion of the assumptions and limitations of this approach in (Section 6.9 and discussion in Section 6.10).

6.1. Running Example

To illustrate the process of **C3**, we assume again that the `Auth.PlaceOrder`, shown in simple form in Listing 5.2, is newly added. Then, the **C1** extracts a new abstract behavior using `Palladio SEFF`, as shown in Figure 5.11. As illustrated in Section 5.2.1, the adaptive instrumentation instruments this service with the related probes. Now, the goal of **C3** is to estimate the PMPs in relation to the impacting input data and their properties, e.g., the number of elements in a list or the size of a file [Maz+20]. Therefore, **C3** explores whether there is a relationship between the iteration number of the `loopAction` and the input parameter `cartSize`, which is indeed the case in this instance, as the loop is repeated based on the value of `cartSize`. As a result, **C3** calibrates the loop action in `SEFF` in relation to the `cartSize`.

The investigation of parametric dependencies during the incremental calibration is also extended to the `externalCall` arguments and `cartSize`.

Similarly, **C3** estimates the resource demands of the internal actions (`prepareOrder` and `finalizeOrder`) and then examines whether they depend on the input parameter `cartSize`.

Calibrating the PMPs in relation to these influential variables will enable accurate predictions, even when dealing with an unknown workload.

6.2. Overall Process

The incremental calibration process calibrates mainly the parameters of the abstract behavior, the `Palladio` SEFF in our implementation. This includes the resource demand of an `internal action`, the number of iterations in a `SEFF loop`, the branch transitions of a `SEFF branch`, and the arguments and return value of a `SEFF external call`.

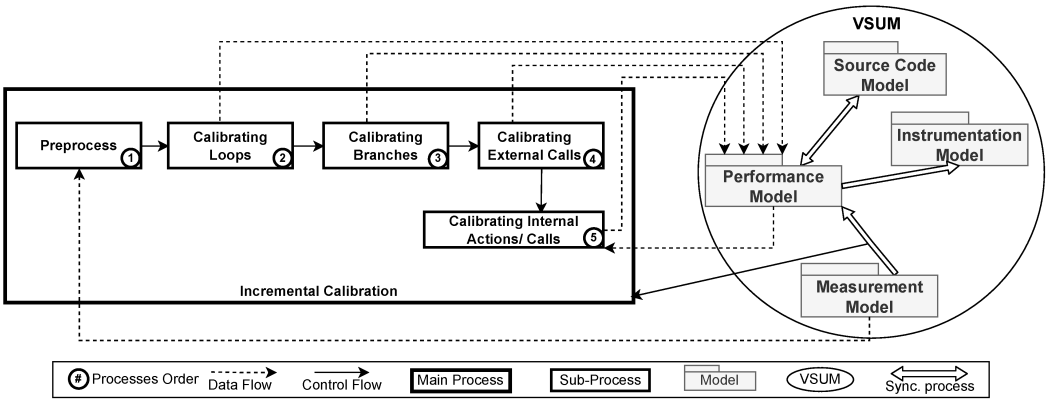


Figure 6.1: The incremental calibration process.

As illustrated in Figure 6.1, the process responds to new measurements by updating the performance parameters of the aPM, following these steps:

1. *Preprocess:* This step filters and groups the measurements into several datasets based on the types of records, such as a dataset for loop records, a dataset for internal action records, etc. Then, this process divides each dataset into two datasets: 80% training set and 20% validation set [XG18]. The measurements in the training set are then used for the calibration with parametric dependencies. The possible dependencies are provided by our instrumentation based on a heuristic similar to [Kro12, Chapter 5.10.4] and filtered by a feature selection algorithm.

2. *Calibrating Loop Actions:* Using the loop records dataset, we determine the dependencies between the measured number of iterations and the impacting parameters, primarily the input parameters, as discussed in Section 6.3. To learn these parametric dependencies, we mainly rely on numeric parameters and regression analysis, see Figure 6.2.

3. *Calibrating Branch Transitions:* Using the branch records dataset, we identify the dependencies between the measured branch transitions and their influencing parameters. The calibration process primarily employs decision tree algorithms to capture the branching logic and parametric dependencies, as described in

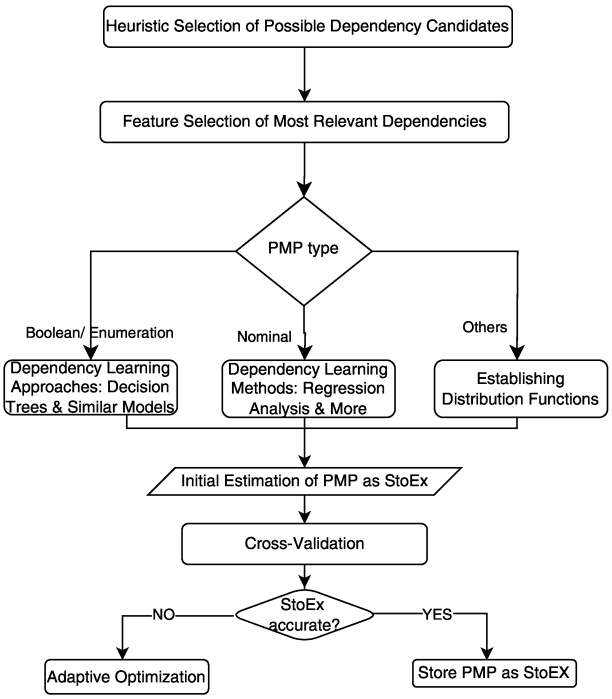


Figure 6.2: Learning the parametric dependencies.

Section 6.4 and shown in Figure 6.2. This method helps in modeling the conditions that impact the probability of taking different branches.

4. *Calibrating external call parameters and return values:* For external calls, both input parameters and return values are calibrated based on their corresponding datasets; as we explain in Section 6.5 and shown in Figure 6.2. The dependencies between the measured values and influencing factors, such as input data types and external service response times, are identified using regression analysis and machine learning techniques, as outlined in Section 2.5. As in previous steps, cross-validation is employed.

5. *Calibrating internal calls and actions:* Calibrating internal actions based on adaptive monitoring presents a challenge, as resource demand estimation may lack critical measurements that are not captured by adaptive monitoring. To address this, the available measurements are complemented with values predicted from previously estimated parameters in the aPM. This process requires traversing the control flow of the abstract behavior (SEFF) and relies on previously calibrated parameters, such as loops, branches, and external calls. Therefore, these PMPs are calibrated in the earlier steps to ensure traversal of the SEFF and estimation of missing measurements. As a result, resource demands for internal actions/calls can be estimated using the service demand law [Men+04] from queuing theory,

leveraging both available measurements and predicted values, as we explain in detail in Section 6.6.

Parametric dependencies are then identified to estimate the resource demands based on impacting factors (see Section 6.6.3).

CIPM learns the relationships between PMPs and input data using regression analysis and starts a cross-validation.

6. *Adaptive Optimization of Parametric Dependencies:* The cross-validation step utilizes the validation step to validate the identified dependencies. The optimization process is only triggered for PMPs that exhibit high cross-validation errors. The main goal of this optimization process is to enhance the accuracy of the identified PMPs by minimizing cross-validation errors. For that, **C3** utilizes a genetic algorithm aiming to reduce the cross-validation errors.

The following sections (Section 6.3-Section 6.7) provide detailed insights into steps (2-6) of the incremental calibration process for PMPs. The self-validation at Dev-time is described in Section 6.8. The assumptions and limitations follow in Section 6.9, with a discussion outlined in Section 6.10.

6.3. Calibrating Loop Actions

To estimate how the number of loop iterations depends on input parameters, we need both the loop iterations' count and the input parameters for each service call. To achieve that, we use our loop records that log the loop iterations' count, every time a loop finishes (see Section 5.2). These records refer to the service call record that contains the input parameters.

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

To estimate the dependency of the loop iteration count on input parameters, we combine the monitored loop iterations with the integer input parameter into one dataset. To do so, we filter out all non-integer parameters and take into account their integer properties, such as the number of list elements or file size. Then, we add transformations (quadratic and cubic) of parameters to the dataset to test more relations. Finally, we use regression analysis to estimate the weights of the influencing parameters. Due to the restriction that the loop iterations count is an integer number, we have to ensure that the output value is always an integer value, which is not always the case. Therefore, we have to approximate the non-integer weights or express them as a distribution of integer values. For instance, we can express the value 1.6 using a `Palladio` distribution

function of an integer variable which takes the value 1 in 40% of all cases and the value 2 in 60% of cases.

In case no relation is detected, we annotate the loop using an integer distribution function derived from the measured values.

6.4. Calibrating Branch Transitions

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

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

Our monitoring records allow us to build a dataset for each branch, which includes the branch transitions and the input parameters. To estimate the potential relations, we use the J48 decision tree of the Weka library, an implementation of the C4.5 decision tree [Qui93]. We filter out the non-significant parameters based on cross-validation. Finally, we transform the resulting tree into a boolean stochastic expression for each branch transition. If no relation is found, the resulting stochastic expression will be a boolean distribution representing the probability of selecting a branch transition.

6.5. Calibrating External Call Parameters and Return Values

This step predicts the parameters of an external call in relation to the input parameters of the calling service and their characteristics. As mentioned in Section 5.3, our instrumentation logs the input data and some of their characteristics, similar to [Kro12, Chapter 5.10.4]. These characteristics are used by learning the parametric dependencies.

For that, we first check whether each parameter of an external call is constant or identical to one of the input parameters. If that is not the case, we check whether the external call parameters depend on the characteristics of the input data. Moreover, we check the dependency on the data flow, i.e., the return value of the previous internal/external actions. A feature selection algorithm is applied to reduce the dimensionality of the parameter space [Von+20; Gro+19].

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

6.6. Calibrating Resource Demands

Existing Resource Demand Estimation (RDE) approaches typically either estimate the RDs at the service level [Spi+15], which can diminish the accuracy of AbPP, or necessitate costly fine-grained monitoring [BKK09; KKR10], resulting in high monitoring overhead that limits their applicability in Ops-time.

C3 aims to enhance the accuracy of AbPP while simultaneously reducing the monitoring overhead to address both $P_{Inaccuracy}$ and $P_{Monitoring-Overhead}$. To achieve this goal, **C3** calibrates services at a fine-grained level, taking into account the resource demands of internal actions, and uses measurements from adaptive monitoring. However, calibrating the internal actions with RD based on adaptive monitoring presents challenges because some required measurements are missed. To tackle this challenge, we propose a novel concept for RDE that utilizes adaptive instrumentation and monitoring, combined with previously estimated RDs, to enable what we refer to as an incremental RDE of internal actions.

Our incremental RDE expands upon the non-incremental approach to RDE (Section 6.6.1), making it applicable to cases involving adaptive monitoring of the modified sections of source code, as detailed in Section 6.6.2.

6.6.1. Basis: Non-Incremental Resource Demand Estimation

Brosig, Kounev, et al. [BKK09] approximate the RDs with measured response times in the case of low resource utilization, typically 20%. Otherwise, they estimate the RD of internal action i (a part of SEFF, see Section 2.4) for resource r ($D_{i,r}$) based on service demand law [Men+04] shown in Equation 6.1. In this context, $U_{i,r}$ denotes the average utilization of resource r attributed to the execution of internal action i, while C_i represents the total number of times internal action i is executed during a fixed observation period of length T:

$$D_{i,r} = \frac{U_{i,r}}{C_i/T} = \frac{U_{i,r} \cdot T}{C_i} \qquad \text{Service demand law [Men+04]} \qquad (6.1)$$

Brosig et al. measure the C_i and estimate $U_{i,r}$ by using the weighted response time ratios of the total resource utilization, as shown in Equation 6.2. Here, R_i is the average measured response time of the internal action i, and C_i indicates the number of executions during the specific time T. The expression $\sum_{j=1}^{n} R_j \cdot C_j$ represents the total weighted response time for all internal actions that stress the resource r within a specific time frame T. Each R_j denotes the average measured response time for the internal action j, while C_j signifies the number of executions of this internal action within T. The formula is expressed as:

$$U_{i,r} = U_r \cdot \frac{R_i \cdot C_i}{\sum_{j=1}^{n} R_j \cdot C_j} \tag{6.2}$$

6.6.2. Incremental Estimation of Resource Demands

Applying Brosig [BKK09] including formulation above (Equation 6.2) for estimating the RDs while applying adaptive monitoring is not applicable. The reason is that not all internal actions are monitored. Therefore, the response time R_j and execution number C_j for the not monitored internal action are not available. Therefore, our novel incremental RDE extends this approach to estimate $U_{i,r}$ and consequently $D_{i,r}$ based on the available measurements and the previously estimated RDs. To achieve this, our incremental RDE categorizes internal actions based on whether they have been modified in the source code commit preceding the incremental calibration.

1. The first category includes internal actions whose corresponding code regions have been modified in the preceding source code commit. These code regions are instrumented (cf. Section 5.2) and monitored to produce measurements for RDE. we refer to them as Monitored Internal Actions (MIAs).

2. The second category comprises internal actions whose corresponding code regions have not changed in the preceding source code commit. Monitoring unchanged source code is unnecessary, as we have already collected monitoring data for these regions in previous development iterations. Consequently, we have already estimated their RDs, and they should not be monitored. Thus, we refer to them as Not Monitored Internal Actions (NMIAs).

If we can estimate the resource utilization resulting from monitoring only the MIAs ($U_{r,MIAs}$), we will be able to apply Equation 6.2 of Brosig et al. approach in our adaptive case. Since the total utilization U_r is measurable and the utilization due to executing NMIAs can be estimated based on the old estimations of RDs, we can estimate $U_{r,MIAs}$ and estimate the RD for each internal action $i \in MIAs$ accordingly, as it will be explained in the following paragraphs.

To estimate ($U_{r,NMIAs}$), we estimate which internal actions $nmi \in NMIAs$ are processed in this interval and how many times nmi are called (C_{nmi}). For that, we analyze the service call records to determine which services are called in an observation period T and which parameters are passed. Then, our RDE traverses the control flows of the service's behaviors (i.e., their `SEFFs`) to get NMIAs and predict their RDs using the input parameters. This requires evaluating branches and loops of the control flow to decide which branch transition must be followed and how often the inner control flow of loops should be handled. Thus, our calibration calibrates the new or outdated branches and loops using the monitoring data before starting this incremental RDE, as described in

the process of our calibration (Section 6.2). Consequentially, our incremental RDE can traverse the SEFFs control flow. Consequently, we can sum up the predicted RDs for all calls of the NMIAs (C_{nmi} call) and divide the result by observation period T to estimate the $U_{r,NMIAs}$ based on the utilization law (Equation 6.1) as Equation 6.3 illustrate:

$$U_{r,NMIAs} = \frac{\sum_{nmi \in NMIAs} \sum_{k \leq C_{nmi}} D_{nmi_k,r}}{T} \tag{6.3}$$

Accordingly, we estimate the utilization due to executing the MIAs ($U_{r,MIAs}$) by calculating the difference between the measured resource utilization U_r and the estimated utilization of the non-monitored internal actions $U_{r,NMIAs}$. This relationship is expressed in Equation 6.4:

$$U_{r,MIAs} = U_r - U_{r,NMIAs} \tag{6.4}$$

Consequentially, we can estimate the utilization $U_{i,r}$ due to executing each internal action $i \in MIAs$ using the weighted response time ratios as shown in Equation 6.5, where R_i and C_i are the average response time of i and its execution count within T. Similarly, R_j is the average response time of the internal action $j \in MIAs$, and C_j is its execution frequency in T.

$$U_{i,r} = U_{r,MIAs} \cdot \frac{R_i \cdot C_i}{\sum_{j \in MIAs} R_j \cdot C_j} \tag{6.5}$$

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

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

To differentiate between the CPU demands and disk demands, we propose identifying disk-based internal actions through static analysis of the source code corresponding to the executed internal actions. Disk-based internal actions can be detected by analyzing the libraries used or by recognizing specific patterns and notations in the code that indicate disk operations. Note that we assume that each internal action is dominated by a single resource. If this is not the case, we follow the solution of Brosig et al. [BKK09] to measure the processing times of individual execution fragments so that the measured times of these fragments are dominated by a single resource.

The next step of the RDE will check whether there is a relationship between the captured input parameters and the estimated values of ($D_{i,r}$), as we describe in Section 6.6.3. In the last step, we store the RD as a stochastic formulation that may include dependencies on the input parameters.

6.6.3. Identification of Parametric Dependencies for Resource Demands

This process estimates the RDs in relation to the impacting input data and their characteristics (such as the number of elements in a list or the size of a file, environment resources). Like other PMPs, our instrumentation uses heuristics similar to [Kro12, Chapter 5.10.4]. Based on feature selection, we also reduce the dimensionality of the parameter space and use the most relevant parameters/ properties as candidates for learning the parametric dependencies. To learn the parametric dependency between the resource demand of an internal action i and parameters P, we first estimate the resource demand on resource r for each combination of the parameters ($D_{i,r}(P)$) using the proposed incremental RDE as described in Section 6.6.2. Second, we adjust the estimated RDs of an internal action using the processing rate of the resource, where it is executed, to extract the resource demand $D_i(p)$independently of the resource' processing rate.

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

$$\begin{aligned} D_i(P) = (a * p_0 + b * p_1 + \cdots + z * p_n + \\ a_1 * p_0^2 + b_1 * p_1^2 + \cdots + z_1 * p_n^2 + \\ a_2 * p_0^3 + b_2 * p_1^3 + \cdots + z_2 * p_n^3 + \\ a_3 * \sqrt{P_0} + b_3 * \sqrt{p_1} + \cdots + z_3 * \sqrt{p_n} + \mathbf{C}) \end{aligned} \quad (6.6)$$

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

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

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

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

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

6.7. Adaptive Optimization of Parametric Dependencies

As the previous section described, the incremental calibration estimates the PMPs with the parametric dependencies as a StoEx.

The optimization process is activated for PMPs with a high cross-validation error, which is measured using the Mean Squared Error (MSE) metric [UC08]. The MSE threshold is empirically determined. The goal of this optimization is to enhance the accuracy of the PMPs while minimizing the complexity of the StoEx.

The optimization process takes the estimated StoEx as input and outputs an optimized StoEx with a cross-validation error equal to or lower than the initial error. Genetic Algorithm (GA) is employed for optimization, cf. Section 2.5.3. For that, we represent mathematical functions as an AST, similar to the approach in [Kro12].

Similar to the previous learning of parametric dependencies, a feature selection algorithm (Section 2.5.4) is applied to identify a set of potential candidates for learning, focusing on the numeric characterization of the input data. This step helps reduce the search space and improves the efficiency of the optimization.

Next, GA is configured with the dataset of selected dependencies. The initial StoEx is transformed into an AST, which serves as the starting individual for the evolutionary algorithm. The use of an AST allows the representation of mathematical expressions as tree structures, facilitating operations such as crossover and mutation.

The main evolutionary loop of the GA then generates a new population through 'crossover' and 'mutation.' The 'crossover' operation randomly swaps subtrees between two parent individuals, enabling the combination of different sub-expressions to create offspring. The 'mutation' operation modifies certain genes by altering subtrees or specific values, ensuring diversity in the new population and preventing premature convergence to local optima.

These evolutionary operations allow the GA to explore various candidate solutions, iteratively improving the parametric dependency models.

In the next step, the fitness function evaluates individuals based on two criteria: prediction accuracy, measured by MSE, and the complexity of the mathematical expression, measured by the depth of the AST. The goal is to reduce both metrics simultaneously, striving for a balance between accuracy and simplicity. The fitness function defines the selection pressure, guiding the evolutionary process to choose individuals with better fitness for the next generation.

The evolution process is terminated under one of three conditions: an optimal solution is found, indicated by the best fitness value falling below a specified minimum threshold; the maximum execution time for the GA is exceeded; or a predefined number

of generations have evolved without reaching the optimal solution. These criteria ensure that the optimization process remains efficient and avoids excessive computation.

Once the evolution completes, the resulting AST is converted back into a stochastic expression, which is then transferred to `Repository Model` for further use.

After the incremental calibration finishes, the self-validation process with the test data is triggered to inform the user about the accuracy of the calibrated aPM (cf. Section 6.8).

6.8. Self-Validation at Development Time

The goal of self-validation is to evaluate the accuracy of the performance predictions related to the updated aPM. In order to determine the prediction accuracy of a model, it is necessary to have a baseline. At Dev-time, we use the validation dataset from the test environment as a baseline.

The self-validation starts the simulation using the updated aPM. Then, it compares the predicted response times of services with those measured using service records, and are available in the validation set. 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. The details of the measurements comparison and used metrics will be explained in detail in Section 7.4. The used metrics give the user feedback about the accuracy of aPM. If the model is considered accurate, developers can trust it and use it to answer What-if performance questions at Dev-time. Otherwise, the self-validation determines the inaccurate parts to be recalibrated.

For that, the tester may change the test configuration to get more representative measurements for the re-calibration aPM. Another option is to wait for the Ops-time calibration. At Ops-time, measurements based on the real usage of the system can be continuously monitored until the accuracy degree is accepted and the monitoring is deactivated. More details and explanation on the self-validation will be explained in Section 7.4.

6.9. Assumptions and Limitations

In this section, we outline the assumptions and limitations in the incremental calibration of PMPs with estimating parametric dependencies. These assumptions are critical for ensuring the accuracy of the models developed during the calibration process.

The following subsection, Section 6.9.1 details the specific assumptions we make, as well as the limitations that may affect the calibration process Section 6.9.2.

6.9.1. Assumptions

In the context of incremental calibration, we make the following assumptions:

1. **Availability of measurements**: The calibration process relies on the availability of performance data and measurements, which serve as the foundation for estimating parametric dependencies and assessing the accuracy of performance models. Therefore, we assume the existence of a basic set of automated performance tests that serve as a foundation for the calibration at Dev-time. In cases where such tests are lacking, our approach enables the integration of resource demand estimation methods through static code analysis tools like ByCounter [KKR08], as we explain in Section 13.1.3.

2. **Effectiveness of Heuristic Approaches**: Heuristic methods, such as those implemented in Weka for analyzing parametric dependencies, are assumed to yield reliable insights into the relationships between performance metrics and resource demands.

3. **Dominance of a Single Resource**: Each internal action is assumed to be dominated by a single resource. However, the concept of CIPM can enable handling such cases, as we describe in future work (cf. Section 13.1.11 and Section 13.1.12).

4. **Accuracy of Measurements**: The incremental calibration process during Dev-time relies on measurements collected from the test environment. We assume that these measurements are accurate. If the measured data are inaccurate or do not cover the domain, the calibration may yield inaccurate models for parametric performance metrics (PMPs). Therefore, we assume that the measurements are accurate enough by applying the following assumptions.

    - **Elimination of Noise**: It is assumed that there is no noise in the measurements or that any noisy data has been eliminated before analysis. Otherwise, the noise can falsify the accuracy of the performance parameters. Monitoring tools such as the used one, `Kieker` (Section 2.1.3), facilitate filtering out performance anomalies.
    - The assumption that there are no concurrent tasks that may affect the measurements, for instance, the utilization. We assume that our software is deployed in a virtual container so that we can measure the CPU.
    - **Domain Knowledge and Meaningful Data**: To obtain meaningful measurements through performance tests or benchmarking, an understanding of the domain and the availability of meaningful workloads are essential for measurement quality. Therefore, we assume that the incremental calibration relies on the availability of meaningful measurements.

Approaches that learn and apply usage patterns in performance tests or benchmarking at Dev-time can generate meaningful test data. For instance, WESSBAS extracts probabilistic workload specifications for load testing [Vög+18]. Utilizing such approaches with CIPM provides meaningful measurements for calibration at Dev-time.

- **Memory Access Impact**: A general limitation of the PCM is its inability to predict the time spent accessing different levels of memory (e.g., RAM, caches). As a result, an assumption is made that memory access times do not significantly impact performance. However, it is important to acknowledge that in memory-intensive applications, this assumption might reduce the accuracy of performance predictions.

6.9.2. Limitations

While the incremental calibration process offers a promising approach to estimating parametric dependencies, certain limitations must be acknowledged:

1. *Limited Dependency Handling*: The current CIPM implementation is limited to handling dependencies between PMPs and usage profiles in terms of service input parameters, while dependencies on global parameters, passive resources, or configuration parameters are not yet supported. However, CIPM can conceptually be extended to include learning parametric dependencies for runtime or configuration parameters.

2. *Scalability Challenges with Non-Nominal Dependencies*: Identification and characterization of dependencies between two non-nominal parameters may affect the execution time of calibration; for instance, detecting dependencies between branch transitions and string arguments requires more time compared to nominal parameters. This may result in inefficiencies when processing large-scale data or complex dependency structures. Therefore, the approach focuses on nominal data, with architects advised to manually review and refine inaccurate PMPs. Moreover, incorporating static analysis into the calibration process can aid in selecting candidates for learning parametric dependencies, which can indicate when to consider non-nominal parameters, cf. Section 13.1.11.

6.10. Discussion

The proposed incremental calibration (**C3**) addresses the third research question (RQ3) of how to effectively automate the incremental calibration of aPM to account for evolutionary, adaptive, and usage-related changes while minimizing monitoring overhead in agile software development environments.

The third contribution of CIPM proposes a solution that utilizes available measurements and the previous state of the aPM for calibration. Machine learning techniques are employed to learn parametric dependencies. The current implementation estimates resource demand based on utilization law; however, future work could explore more estimation techniques, as mentioned in [Spi+15], by adapting them to leverage available measurements and the historical state of the aPM.

Moreover, learning parametric dependencies is not restricted to the current techniques used, such as regression, decision trees, or genetic algorithms. Other methods or combinations thereof can be employed, with cross-validation identifying the most accurate models. Currently, optimization is triggered to enhance the accuracy of PMPs. Still, it can be configured to start the optimization to reduce the complexity of the resulting PMPs represented as StoEx.

Additionally, the proposed calibration employs heuristic and feature selection methods to identify potential dependencies. In future work (Section 13.1.11), static analysis during CI updates of aPM (**C1**) could provide initial estimates and dependencies that can be further examined by **C3** and updated based on measurements, similar to the operation of Ops-time calibration, which receives previous estimates, verifies them through self-validation, and updates them accordingly.

Furthermore, future efforts may expand the range of learning parametric dependencies to encompass global parameters and runtime configurations. This is crucial, as software systems may have configuration spaces that impact performance. Learning from a limited sample of configurations' measurements has shown to be effective in addressing the challenge of performance prediction in highly configurable systems [Per+21].

Regardless of the use of machine learning techniques for learning parametric dependencies and their scope, accurate incremental calibration depends on the precision of performance measurements. Therefore, we assume the elimination of noise and the use of meaningful load-producing measurements that cover the domain. Otherwise, the self-validation at Dev-time would report the low accurate PMPs, suggesting reconfiguring the performance test/ benchmark for generating more meaningful measurements or waiting for Ops-time where the calibration would be performed with real measurements from the production environment as the following chapter explains (Chapter 7).

7. Consistency Preservation at Operation Time

In this chapter, we explain how we preserve consistency between the parameterized architectural performance model and operational measurements from the production environment. The goal is to keep the architecture of aPM aligned with the current structure running in operation, which is reflected by real-time measurements, thereby ensuring the consistency relation (R_{Ops}). Additionally, this alignment ensures that performance parameters are updated with actual measurements, supporting reliable performance predictions and maintaining the consistency relation R_{AbPP}.

To address the challenges associated with preserving consistency between the parameterized architectural performance model (aPM) and measurements from the production environment, we focus in this chapter on the following research questions:

RQ4 How can we update aPMs according to adaptation changes in running software?

RQ5 How can we enhance trust in aPMs validity while reducing the required monitoring overhead?

To answer the research questions posed, we introduce two contributions (**C4** and **C5**) designed to update and validate the architectural Performance Model (aPM) in operational contexts. This approach ensures efficient consistency preservation between the aPM and operational measurements. The fourth contribution of this thesis, the Ops-time calibration (**C4**), focuses on the continuous updating of the aPM using measurements from the production environment, thereby addressing $P_{OpsInconsistency}$ and achieving consistency during the operation, which in turn addresses $P_{ArchUnderstanding}$ and enhances the understanding of the current architecture .

The fifth contribution, the self-validation (**C5**), is responsible for validating the aPM to address uncertainty ($P_{Uncertainty}$) and leverages the validation results in subsequent updates to enhance the accuracy of AbPP, thus addressing accuracy concerns ($P_{Inaccuracy}$). Additionally, **C5** manages the monitoring probes to minimize the monitoring overhead, thereby addressing monitoring overhead issues ($P_{Monitoring-Overhead}$).

The novelty of **C4** lies in its ability to continuously and automatically update various components of the aPM, including PMPs with parametric dependencies, using a customized adaptive monitoring approach. This allows performance parameters to stay

up-to-date with production environment conditions, efficiently reducing monitoring overhead.

We acknowledge that various approaches exist to retrieve different aspects of the architecture from operational measurements. However, to our knowledge, our contribution **C4** is the first that provides a pipeline updating multiple perspectives of the aPM utilizing knowledge from the last source code and monitoring focusing on the recently changed parts of the source code, while avoiding unnecessary monitoring overhead.

Additionally, as far as we know, CIPM is the first approach that addresses the challenge of aPM accuracy uncertainty through **C5**, the self-validation. **C5** provides feedback on the accuracy of aPM and its AbPP. Besides, the update process of aPM is a learning process that utilizes feedback from self-validation to improve the aPM's accuracy. This continuously updates the PMPs based on self-validation feedback, addressing parametric dependencies. This generates performance insights that can be leveraged for both software adaptation at Ops-time and development.

The initial concept of **C5**, accompanied by a preliminary evaluation, is introduced in [Maz+20]. Following its development and further detailed evaluation, both **C4** and **C5** are subsequently published in [Mon+21].

This chapter begins by presenting the overall process, introducing both the **C4** and **C5** processes and their interactions. In Section 7.2, we describe the monitoring requirements for both **C4** and **C5**.

Following this, we detail the overall process in Section 7.3, which describes the Ops-time calibration, and in Section 7.4, we provide a detailed discussion on self-validation.

We then outline the assumptions and limitations in Section 7.5, followed by a discussion in Section 7.6.

7.1. Overall Process

As we describe in the MbDevOps pipeline pipeline (Section 4.3), the main CIPM processes within Ops-time include monitoring, along with repeated Ops-time calibration (**C4**) and self-validation (**C5**), which operate within an adaptive feedback loop. In this loop, **C4** continuously refines the aPM by incorporating feedback provided by **C5**, as shown in Figure 7.1 and illustrated in the subsequent steps:

- *Monitoring:* This process gathers the measurements necessary for calibration (**C4**) and validation (**C5**) , as illustrated in Section 7.2.
- *Ops-time-Calibration:* This process updates the aPM using a transformation model based upon a tee and join pipeline architecture [Bus98]. This transformation pipeline includes multiple transformations that can access a common blackboard to write and read the data. For instance, the blackboard stores the current state

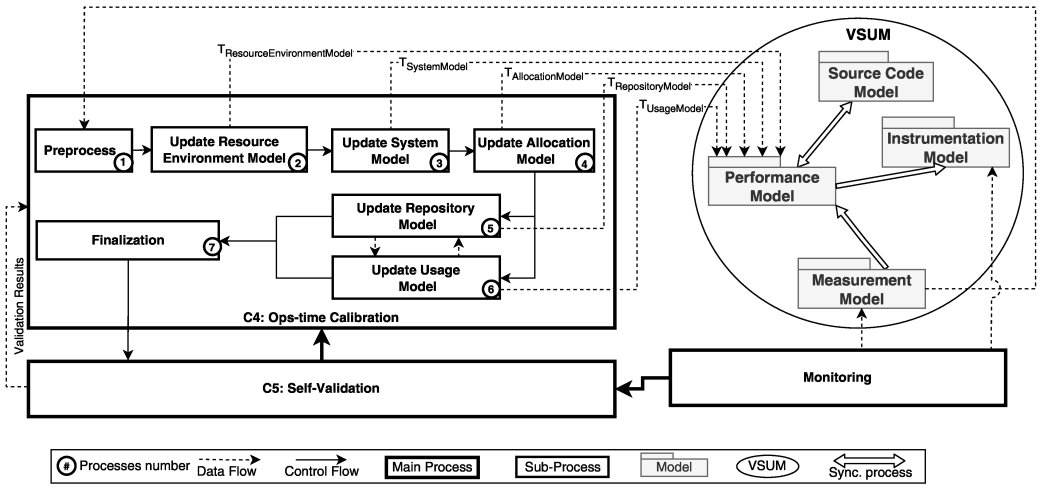


Figure 7.1: The process of Ops-time calibration and self-validation. To enhance readability, data flows between sub-processes are omitted.

of the pipeline and the results of self-validation. The transformation pipeline includes the following sub-processes:

1. **Pre-processing of Ops-time Calibration ($T_{Preprocess}$)** (Section 7.3.1): This initial step filters and structures monitoring data into service call traces, enabling system analysis, divides data into a training set for calibration and a validation set for performance assessment, store data sets into the blackboard to be accessible for remaining transformations.

2. **Ops-time Updating of Resource Environment ($T_{ResourceEnvironmentModel}$** (Section 7.3.2): This step analyzes the current Ops-time environment to detect hosts and their connections. Then, it records this information into a Runtime Environment Model (REM) stored in the blackboard, which is transformed to aPM through the CPRs defined on the Vitruvius platform.

3. **Updating the System Model ($T_{SystemModel}$)** (Section 7.3.3.2): This transformation extracts SCG and updates the system model accordingly. The established methodology ensures that the aPM reflects the current system composition.

4. **Ops-time Updating of Allocation Model ($T_{Allocation}$)** (Section 7.3.4): This step examines system deployment in parallel with the `System Model` update, utilizing the extracted SCG. Thus, it recognizes deployment and undeployment events to maintain an accurate allocation model.

5. **Ops-time Updating of Repository Model ($T_{RepositoryModel}$)** (Section 7.3.5): This process calibrates inaccurate performance metrics revealed by self-validation, utilizing monitoring data to optimize the performance metrics similarly to the development process outlined in Chapter 6.
6. **Updating Usage Model ($T_{UsageModel}$)**: This transformation updates the usage model based on data from `iObserve`, ensuring that the model accurately reflects current usage patterns and behaviors.
7. **Finalization of Ops-time Calibration ($T_{Finalize}$)** (Section 7.3.7): This final step executes self-validation, ensuring the accuracy and reliability of the calibrated models before concluding the ops-time calibration process.

- *Self-validation:* This process compares the monitoring data with simulation results to validate the accuracy of aPM (Section 7.4). The validation results are used as input to the Ops-time calibration and the basis for managing the monitoring process.

In the following sections (Section 7.2-Section 7.4), we detail the process above.

7.2. Monitoring

Successful monitoring focuses on relevant metrics, applies statistical analysis, retains historical data, and defines clear goals [Sch18], see Section 2.1.2.3. Our Monitoring is based on DevOps monitoring and aims to achieve the successful characteristics of monitoring above, cf. Section 2.1.2. First, it focuses on the performance-relevant aspects that allow calibrating the aPM. Therefore, our adaptive instrumentation (Section 5.2) injects probes (Section 5.1.4.1) to capture necessary measurements for calibrating PMPs, ensuring alignment with our measurement metamodel (Section 4.2.2). Our monitoring employs an event-based approach, where measurements are logged whenever the instrumented code is executed. Additionally, it utilizes a sampling-based strategy to capture measurements at defined intervals, such as tracking total resource utilization over time.

Second, our monitoring applies mathematical and statistical analysis techniques to aggregate and analyze the monitoring data, enabling the calibration and validation of the aPM. As we explain in the next section (Section 7.3), we employ a sliding window technique to aggregate and analyze monitoring data within fixed time intervals.

Third, CIPM recognizes the value of retaining monitoring data to uncover performance insights related to software versions. Therefore, our approach emphasizes the importance of storing measurements within our measurement model and maintaining the mapping between source code versions and their related measurements using digital twins, Section 4.2.2.

Lastly, our monitoring has a well-defined goal that concentrates on consistency preservation between software artifacts at Ops-time (R_{Ops}) and improves the accuracy of AbPP (R_{AbPP}), with minimal monitoring overhead. Hence, our adaptive monitoring plays a critical role in achieving this goal by dynamically adjusting the monitoring to collect measurements needed for consistency preservation (R_{Ops}) and accuracy improvement (R_{AbPP}). As explained in Section 7.4, self-validation manages adaptive monitoring by activating monitoring for inaccurate parts and deactivating it once an acceptable level of accuracy is achieved.

We adopt Kieker for monitoring in CIPM because it supports the definition of custom monitoring probes, enabling tailored data collection for performance metrics that align with adaptive instrumentation needs. Kieker's adaptive monitoring capabilities align with the our goal of reducing overhead by dynamically focusing on specific components or scenarios. Additionally, its ability to facilitate trace analysis, is crucial for Ops-time calibration.

7.3. Ops-time Calibration

Ops-time calibration operates via a transformation pipeline using a tee-and-join pipeline architecture [Bus98]. A potential performance issue can happen if the transformation pipeline should be triggered after each new monitoring data input. Moreover, updating the aPM based on a single measurement would not ensure an accurate reflection of architectural changes. Thus, aggregating monitoring data and processing it enables a comprehensive view of system behavior, allowing us to gain insights on the accuracy of aPM and update it. Both sampling-based and event-based approaches can be employed to collect monitoring data. The sampling-based approach involves gathering data at predetermined intervals, allowing for systematic updates to the aPM based on the collected measurements. Conversely, the event-based approach triggers data collection in response to specific events, making it efficient in scenarios where data arrives infrequently; the transformation pipeline activates once the number of collected data points exceeds a predefined threshold.

We have chosen the sampling-based approach because we want to address the case of frequent data collection and analyze the scalability of the pipeline with an increasing number of monitoring data. For our chosen method, we have implemented a sliding window technique that continuously updates the aPM based on a subset of data determined by moving a fixed-size window across the monitoring dataset, allowing for the continuous processing of the most recent data. Consequently, the transformation pipeline is triggered at specified intervals to update the aPM, taking into account configurable parameters such as the window size and trigger time. The window size is a customizable time interval that defines the duration for collecting monitoring data within the sliding window mechanism. It determines the amount of data that will be

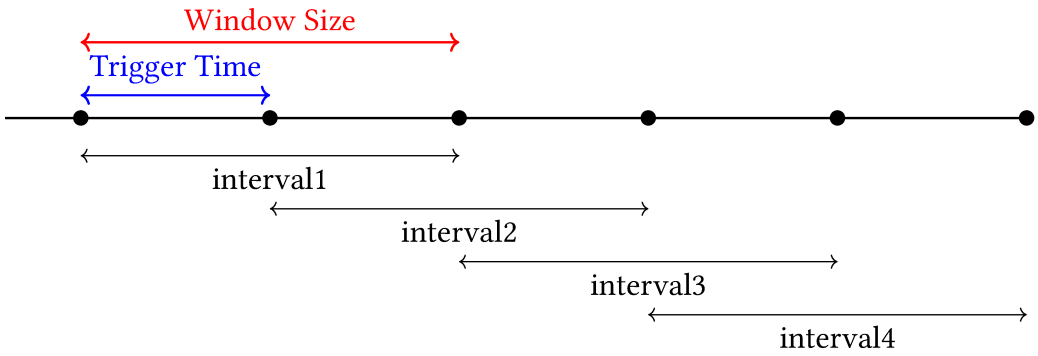


Figure 7.2: Collecting and processing monitoring data in intervals over time: window size defines interval duration, trigger time sets start and sliding interval.

included for processing in each iteration of the transformation pipeline. The trigger time specifies when the transformation pipeline starts and the amount by which the sliding window moves forward after each execution of the pipeline, see Figure 7.2.

In typical scenarios, the trigger time is set to a value smaller than the window size to ensure that all the relevant data is processed within the monitoring window. This approach enables the system to capture and analyze the necessary information effectively.

For instance, if the window size is set to 6 minutes, the monitoring data within 6 minutes is processed. Every 3 minutes (the trigger time), the window slides forward by 3 minutes, analyzing the latest measurements. This ensures continuous evaluation and captures real-time trends. By setting the trigger time smaller than the window size, CIPM processes all data more than once within each interval (2 times in this example). This approach allows for comprehensive analysis, which is particularly useful in distributed systems. In such systems, the monitoring data within a single interval may not cover the entire execution sequence or provide a complete trace of the system's behavior.

In the following, we detail the steps involved in each pipeline execution that **C4** follows to update aPM using monitoring data collected from a specific interval. The Ops-time calibration begins with the preprocessing step in Section 7.3.1, followed by updates to `Resource Environment Model` in Section 7.3.2, `System Model` in Section 7.3.3, `Allocation Model` in Section 7.3.4, `Repository Model` in Section 7.3.5 and `Usage Model` in Section 7.3.6, and concludes with the finalization step in Section 7.3.7.

7.3.1. Pre-processing of Ops-time Calibration

During the Ops-time calibration, the initial step of the transformation pipeline, denoted as $T_{PreProcess}$, involves preparing the monitoring data, ensuring its quality, filtering

out unnecessary monitoring data, and converting the relevant monitoring data into appropriate data structures like the predefined monitoring records (Figure 5.13) and monitoring traces. In the following, we detail tracing monitoring data and dividing it into training and validation sets:

- *Tracing Monitoring Data*: Since distributed systems run on different computers and communicate over the network using communication requests like HTTP, monitoring data becomes fragmented across multiple nodes. This fragmentation makes it challenging to correlate related service calls, particularly when one service call triggers another on a different machine. Therefore, an important step in the preprocessing is the construction of *request traces*, which are sequences of operations that capture the flow of requests and responses between services. Even service call traces that extend beyond system boundaries are recorded by monitoring. When a call leaves a system, the necessary information is attached so that the traces can be merged.

  The traces enable the correlation of monitoring data across different machines, ensuring that monitoring contexts, such as related service calls, remain linked. These traces are required in the following processes, like the update of `System Model` in Section 7.3.3.

  Based on `Kieker`, we implemented trace monitoring specifically for HTTP requests. Each request is tagged with a header identifying the originating service call trace. The monitoring data is subsequently bundled and sent to a backend, where the traces can be reassembled based on this header information.

- *Dividing Monitoring Data*: In this process, the monitoring data is divided into two distinct sets. The first set serves as an input for calibration and is referred to as the training set. The second set, known as the validation set, is reserved for validating the performance and accuracy of the architecture model. Typically, the monitoring data is divided into approximately 80% for the training set and the remaining 20% for the validation set. This division follows the widely used 80-20 rule, where the majority of the data is allocated for training the model, while a smaller portion is set aside for evaluating its performance [XG18]. Dividing data into separate training and validation sets helps in detecting potential overfitting issues or biases that may occur if the model is evaluated on the same data it was trained on [Yin19].

7.3.2. Ops-time Updating of Resource Environment

After preprocessing the monitoring data, the current Ops-time environment is analyzed by $T_{ResourceEnvironmentModel}$. This process analyzes both training and validation sets to detect all available resources and update the `Resource Environment Model` accordingly.

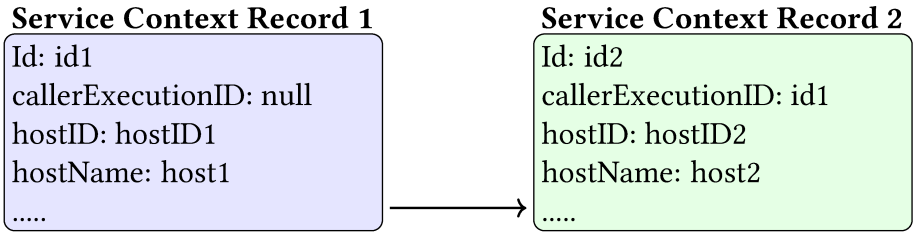


Figure 7.3: Visualization of monitoring trace connected based on the caller execution identifier.

As a proof of concept, **C4** updates the `Resource Environment Model` to represent the available resources and their connecting communication links. Future work can extend this analysis to incorporate additional performance properties, such as latency.

Available resources can be identified from service monitoring records that log the hostname, which is combined with a generated ID. Unlike the hostname, the ID remains consistent over extended execution periods. For detecting connections, we analyze the monitoring traces and link two resources whenever we observe a change in the host within a trace, as the example below demonstrates (Figure 7.3).

This example shows a monitoring trace including two service context records, where "`Service Context Record 2`" is invoked directly from "`Service Context Record 1`", as indicated by the "`callerExecutionID`" property, linking the two records. The execution of the first record occurs on "`host1`", while the second record runs on "`host2`", illustrating a multi-host environment where both "`host1`" and "`host2`" are actively engaged in this monitoring trace. This setup highlights inter-host connectivity, as "`host1`" initiates a service call that triggers an action on "`host2`". To reflect this in `Resource Environment Model`, it would be relevant to specify not only that both hosts are available but also that there is a defined connection enabling service calls between them, thereby capturing the distributed nature of this execution flow.

If the `Resource Environment Model` is available at Dev-time to reflect the available physical resources, we assume that the host names in the `Resource Environment Model` at Dev-time correspond to the physical resource names at the time of deployment. This initial naming alignment facilitates the mapping of monitoring traces to the modeled resources. Following this setup, host identifiers will be updated by the ongoing update process and used for ensuring consistency between the monitored data and the `Resource Environment Model`, considering that hostnames may be renamed during the long execution process.

For the update process, **C4** writes the Ops-time information to an intermediate model that so-called runtime environment model [Mon20, p. 53], which allows storing details about the hosts, the connections between them, and the hardware information.

We defined CPRs based on the Vitruvius platform [Kla+21] to keep our runtime environment model consistent with the `Palladio Resource Environment Model`. The advantages and the idea behind the runtime environment model are double-sided: it ensures the separation of concerns principle (Dev-time vs. Ops-time concerns), and it allows establishing a mapping between the Ops-time environment and the elements in the architecture model via the correspondence mechanism of Vitruvius.

7.3.3. Ops-time Updating of System Model

Similar to Dev-time, extracting a `System Model` at Ops-time requires two steps: first, extracting the SCG that is introduced in Section 5.1.1. The second step involves using the SCG from the first step to extract the `System Model`, similar to the procedure at Dev-time.

The next two paragraphs explain how we extract SCG from monitoring data and how we use it to update the `System Model`.

7.3.3.1. SCG Extraction at Ops-time

To construct the SCG, $T_{SystemModel}$ leverages the traces generated during preprocessing ($T_{PreProcess}$). These traces are mapped to the `Repository Model`, a relationship embedded in the source code via the adaptive instrumentation (cf. Section 5.2), whereby each service context record is associated with a unique service identifier in the `Repository Model` through the `"serviceID"` attribute. Initially, $T_{SystemModel}$ transforms these traces into service call graphs, capturing service interactions and host information, as hostnames are logged in the service context records. Subsequently, $T_{SystemModel}$ integrates all service call graphs into a unified, comprehensive SCG.

In this context, it is possible to detect which services have been called. Moreover, $T_{SystemModel}$ can attach the information about the host to the SCG nodes because this information is included in the monitoring data. Therefore, the quality of the extracted SCG is much higher compared to SCG extracted at Dev-time from the static code analysis (cf. Section 5.1.5.2).

7.3.3.2. Updating the System Model

The $T_{SystemModel}$ updates the `System Model` based on the extracted SCG and is similar to the process at Dev-time, cf. Section 5.1.5.3. The only difference is that no conflict happens during the extraction process. The reason is that the SCG at Ops-time is more accurate and includes the host information.

Each node in the SCG refers to the service provided by a particular component and the resource container where the instance of this component (assembly context) is hosted. Thus, we can derive the assembly contexts of the `System Model` from the SCG

nodes. The edges between nodes indicate assembly connectors, linking the provided role of the assembly context associated with the destination node to the required role of the assembly context associated with the source node. The $T_{SystemModel}$ processes each node, adding the associated assembly context to the `System Model` if it has not been previously included, and uses the edges to accurately establish links between these contexts.

To illustrate, the extraction of the `System Model` begins with modeling the boundary interfaces, specifically the system-provided interfaces that the system offers. To find these, $T_{SystemModel}$ identifies the entry nodes in the SCG, which lack incoming edges. We assume that these nodes exist and that there are no loops in the SCG. Then, for each entry node, the associated system-provided interface is modeled, and an assembly context with a role providing these interfaces is added, connected via delegation connectors to the system-provided interfaces.

To complete the `System Model`, $T_{SystemModel}$ recursively processes the outgoing edges from the entry nodes and the destination nodes. For each unprocessed node, $T_{SystemModel}$ adds the corresponding assembly context if it has not already been added by another node and connects the associated assembly context using assembly connectors reflecting the edges between SCG nodes. This process continues recursively across all outgoing edges until all nodes and edges have been processed. As a result, the $T_{SystemModel}$ updates the `System Model` to reflect the current system composition captured from monitoring data, ensuring the consistency relation R_{Ops}.

7.3.4. Ops-time Updating of Allocation Model

This process, $T_{AllocationModel}$, updates the `Allocation Model` based on the SCG extracted at Ops-time (cf. Section 7.3.3.1). Thus, $T_{AllocationModel}$ operates in parallel with $T_{SystemModel}$ after the SCG extraction phase. The updating process identifies allocation contexts within the SCG. The nodes specify the hosts where the components providing the services are deployed, which supports $T_{UsageModel}$ with the required information to detect both deployment and undeployment events.

If a node in the SCG shows that a service provided by a component is running on a specific host, and no corresponding allocation context exists in the `Allocation Model` to represent this, $T_{AllocationModel}$ adds a new allocation context associated with this host and component to the `Allocation Model`. Conversely, if no nodes in the SCG refer to an existing allocation context in `Allocation Model`, an undeployment event is indicated, leading $T_{AllocationModel}$ to remove this corresponding allocation context.

7.3.5. Ops-time Updating of Repository Model

The $T_{RepositoryModel}$ calibrates and updates the `Repository Model`, focusing specifically on the PMPs based on monitoring data. For calibration, $T_{RepositoryModel}$ uses the training

set generated by $T_{PreProcess}$, and it evaluates accuracy with a validation set. Additionally, $T_{RepositoryModel}$ incorporates self-validation results to perform adaptive calibration, which targets inaccurate PMPs. If certain PMPs have not yet been calibrated, they are updated similarly to the incremental calibration process at Dev-time that is described in Chapter 6.

$T_{RepositoryModel}$ employs metrics identified through self-validation to optimize the accuracy of PMPs. For optimization at Ops-time, two strategies are implemented as proofs of concept: The simple optimization uses regression analysis to adjust regression parameters based on validation results—specifically, by minimizing the error between the observed response time and the monitored one.

Optimization at Ops-time based on the genetic algorithm is the second implemented strategy, cf. Section 6.7. This optimization can be triggered if further improvement of the accuracy is necessary. This algorithm uses the previous estimations and current measurements from the production environment as input to yield a more accurate PMPs.

While our current calibration at Ops-time leverages these two optimization methods, it is designed to be extensible. Future work could incorporate additional machine learning strategies to enhance accuracy and to better learn the parametric dependencies within the system.

After optimizing the inaccurate PMPs, the update process for `Repository Model` aims to achieve optimal calibration by minimizing oscillation effects. Oscillation effects— fluctuations in accuracy arising from the dependency of AbPP accuracy on both `Repository Model` and `Usage Model` parameters— can be minimized by performing concurrent rather than sequential calibration. Accordingly, **C4** calibrates the `Repository Model` and the `Usage Model` in parallel on separate copies of the aPM stored on a blackboard. In this parallel approach, two distinct aPMs are calibrated. In one copy of the aPM, $T_{RepositoryModel}$ optimizes PMPs to update the `Repository Model`, followed by self-validation using a validation dataset. Simultaneously, in a second aPM copy, `Usage Model` is calibrated by $T_{UsageModel}$ based on a training set, then validated using the validation dataset, as detailed in the following section Section 7.3.6. After these parallel calibrations, a cross-validation process determines which of the two calibrated aPMs yields greater accuracy. This outcome establishes the optimal calibration sequence for subsequent model updates, enhancing overall simulation accuracy and avoiding the instability associated with sequential calibration. In other words, if the model obtained from executing $T_{RepositoryModel}$ demonstrates higher accuracy, the $T_{UsageModel}$ transformation is subsequently applied to this model. Conversely, if the model derived from $T_{UsageModel}$ shows better accuracy, the $T_{RepositoryModel}$ transformation is then executed.

7.3.6. Ops-time Updating of Usage Model

$T_{UsageModel}$ analyzes the monitoring data to detect patterns of user behavior and updates the `Usage Model` accordingly. Thus, our monitoring captures session information and stores it in our measurement model by associating each service context record with a session identifier, as illustrated in Section 5.4. This enables analyzing users' behaviors within the observed sessions, which is required for updating the `Usage Model`.

There are different approaches that analyze monitoring data to update the usage profile, like [Hei20; Wal+17; Vög+18]. Thus, we adapt a part of `iObserve` [Hei20] to be compatible with CIPM and integrate it into CIPM to allow consistent updates to `Usage Model`. Our adaption of `iObserve`, first, enables applying the analysis on our measurements, whose structure differs from the monitoring data used in `iObserve`. Besides, while `iObserve` employs a run-time architecture correspondence model to map source code elements to architecture components (Section 2.6.6), CIPM weaves the necessary information about aPM directly into its measurement model, cf. Section 5.4. This approach ensures that all data needed to update `Usage Model` is available in the monitoring data itself. Hence, the second modification of `iObserve` ensures that the analysis extracts the required mapping to aPM from monitoring data instead of `iObserve` run-time architecture correspondence model. After the modification above, the adapted analysis of `iObserve` is integrated into $T_{UsageModel}$.

Consequentially, $T_{UsageModel}$ processes the monitoring traces from $T_{PreProcess}$ data to identify user groups. For that, $T_{UsageModel}$ groups user sessions and clusters them according to similar patterns of user behavior, i.e., the types and count of service calls have to be similar within a user group. After that, $T_{UsageModel}$ calculates the usage intensity for each detected user group.

As mentioned in Section 7.3, $T_{UsageModel}$ updates a copy of aPM, followed by a cross-validation process to determine whether aPM calibrated with $T_{UsageModel}$ or aPM calibrated by $T_{RepositoryModel}$ yields higher accuracy. Empirically, it has been observed that the accuracy of aPM can vary based on the order in which $T_{RepositoryModel}$ and $T_{UsageModel}$ are executed. Thus, the cross-validation results can indicate the optimal calibration sequence to improve the accuracy of the final aPM.

7.3.7. Ops-time Finalization

The final step of Ops-time Calibration ensures that all parts of aPM is updated. Then, $T_{Finalize}$, executes the self-validation (Section 6.8) after updating all parts of aPM: `Resource Environment Model`, `System Model`, `Allocation Model`, `Repository Model` and `Usage Model`. The self-validation utilizes the validation data set of $T_{PreProcess}$. Based on the results and configurable criteria, the granularity of monitoring is adjusted, i.e., fine-granular monitoring can be activated/ deactivated. For instance, if the deviation matches defined criteria (e.g., distance of the means is less than 5ms), the fine-grained

monitoring for this service is deactivated. Detail on self-validation (**C5**) is given in the next section (Section 7.4). Finally, the validation results are entered as input into the next execution of the Ops-time calibration.

7.4. Self-Validation

The validation is a *confirmation in a timely manner, through automated techniques where possible, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled.* [Soc21].

In the context of CIPM, self-validation is a confirmation in a timely manner, through automated techniques and by providing objective evidence, that the updated aPM exceeds a predefined accuracy threshold. This involves checking that previous modifications to the aPM accurately reflect the current state of the software.

Self-validation (**C5**) is a continuous process that is triggered in time intervals to validate the accuracy of aPM at the Ops-time, detect the inconsistencies and resolve them by the Ops-time calibration. It provides objective evidence regarding the accuracy of the aPM and the associated monitoring overhead, using some metrics. It also aims to assess the accuracy of the aPM while managing the trade-off between enhancing its accuracy and minimizing the required overhead.

For evidence of the accuracy, **C5** utilizes the real system measurements as a reference, which are available as monitoring data. At Ops-time we use monitoring data from the production environment, while monitoring data from the test environment is used at Dev-time.

By comparing simulation data of the models with the monitoring data, it can be assessed how well the models represent the actually observed system in its current state. In case of high deviations, it is possible to intervene.

The simulation results are grouped into so-called measuring points, i.e., the points at which measurements were taken. For example, a typical measuring point is the response time of a service. The mapping of the resulting measuring points to the corresponding monitoring data is essential for comparing simulation results with real-world data. This mapping relies on the mapping between the `Repository Model` and the monitoring data, which was woven together through our adaptive instrumentation (**C2**). After the measuring points of the simulation results have been mapped to the monitoring data, we have two distributions for each measuring point. These are compared, and various metrics are calculated to assess how closely the simulation results align with the actual measured values. We use the following metrics to compare the two distributions: Wasserstein distance (WS-distance) [San15], Kolmogorov-Smirnov-test (KS-test) [Dod08] and the difference of conventional statistical measures (e.g., average and quartiles). KS-test compares two CDFs by calculating the maximal *vertical* distance, where a lower value indicates greater similarity between the predicted and

measured response times. WS-distance quantifies the minimum cost to align two CDFs through *horizontal* shifts, with lower values indicating more accurate performance predictions. More details on the metrics are in Section 2.8.3 and Section 2.8.4.

The self-validation identifies the inaccurate parts based on configurable criteria, which combine the abovementioned metrics. For instance, it determines a threshold that the metrics should exceed. According to the defined criteria, feedback is provided on the accuracy of the aPM. If the model is considered accurate, developers can trust it and use it to address What-if performance questions. Otherwise, the inaccurate parts are recalibrated by **C4**. The recalibration assesses the calculated metrics to minimize deviations and enhance the accuracy of the resulting aPM (cf. Section 7.3.5).

Additionally, the self-validation uses the calculated metrics to adjust the granularity of the monitoring. With this extension, monitoring for certain services can be deactivated if predefined criteria are met. This allows balancing the trade-off between monitoring effort and granularity. This feature is particularly important for reducing monitoring overhead, especially when the self-validation is executed during Ops-time.

Unlike performing self-validation during development, in the operational phase (Ops-time), measurements based on the real system usage can be continuously monitored until the desired accuracy is achieved, at which point the monitoring can be deactivated. Furthermore, the self-validation process can activate the fine-grained monitoring for inaccurate parts if their accuracy does not meet the predefined criteria. If these parts are not instrumented, new probes are added to the instrumentation model (`IM`) to recalibrate them following the next deployment. For instance, if there is a deviation between a service's response time and the simulated response time, fine-grained monitoring related to its abstract behavior is activated to recalibrate its performance parameters and improve its accuracy.

7.5. Assumptions and Limitations

In this section, we present the assumptions and limitations in the Ops-time calibration process. These assumptions are essential to maintaining the accuracy and applicability of the model within an operational setting. By clearly articulating these considerations, we aim to minimize potential inaccuracies resulting from environmental constraints or limitations.

The subsections below, Section 7.5.1 and Section 7.5.2, detail the primary assumptions guiding the process as well as the limitations that may impact calibration effectiveness.

7.5.1. Assumptions

In the context of Ops-time calibration, we make the following assumptions: The Ops-time calibration process shares several assumptions with incremental calibration.

First, it relies on the availability of measurements, utilizing monitoring as a foundation. Second, it presumes the effectiveness of heuristic approaches to learning parametric dependencies. Third, it assumes the accuracy of measurements and that noise is filtered out using `Kieker`.

- *Sequential Measurement Aggregation*: The measurements arrive sequentially, and we aggregate them and apply the calibration according to a defined schedule.

- *Version-Specific Measurements*: During each calibration, only measurements generated by running the current version of the system are used. Previous measurements are preserved for future analysis or comparisons between different versions' performance, but they are not considered in the current calibration process. This ensures that the calibration reflects the most up-to-date system behavior and ensures the continuous update of performance parameter models, which were previously updated by measurements from earlier versions.

- *Sufficient Monitoring Data*: Ops-time Calibration relies on the availability of accurate and sufficient monitoring data from the production environment to adjust the aPM effectively.

- *One Component Instance per one Resource Type*: We assume that only one instance of each component type can be deployed on a specific container at a time. This is necessary for the automatic update of `System Model` because we cannot reliably determine the target instance of a service call when multiple instances of the same type exist on a host, impacting the automatic update of the `System Model`, where clear connections between instances are required. Thus, a one-to-one mapping between monitoring data and component instances is essential. Although restrictive, this assumption is practical, as violations are rare, and no conflicts appeared in our validation cases.

- *Accurate Performance Metrics*: The selected performance metrics must accurately represent the resource demands associated with the internal actions being modeled, as misrepresentation may lead to inaccuracies in the estimated parametric dependencies. For instance, we assume that there are no other programs running on our CPU, and the measured utilization reflects the utilization of the software system that CIPM updates.

7.5.2. Limitation

While the Ops-time calibration process offers a promising approach to estimating parametric dependencies, certain limitations must be acknowledged:

1. The current implementation only supports HTTP requests, limiting compatibility with other communication protocols. However, this support could be extended to additional methods if needed.

2. The current approach does not calibrate the performance attributes of `Resource Environment Model`, i.e., the properties of links and containers. Comprehensive monitoring of network requests can enable accurate calculations for latency and throughput. Additionally, by measuring container performance directly, the capabilities of the hardware resources could be estimated.

3. *Data Quality*: The quality of the monitoring data can vary depending on the system's load or the precision of the measurement tools, which can impact the accuracy of the Ops-time time calibration. Thus, we select `Kieker` monitoring tool as its efficiency has been empirically validated through a comparison with two other tools. The comparison shows that `Kieker` generates slightly less overhead when processing traces [RKH21]. However, our approach can be implemented to use other monitoring tools.

4. *Monitoring Inaccurate Parts*: The self-validation process can activate probes for inaccurate parts. However, adjusting or adding instrumentation during Ops-time may introduce delays, as new probes require subsequent deployment cycles. Thus, if the inaccurate parts are not instrumented in the current version, the self-validation adds new probes to the `IM`. Adaptive instrumentation, applied in the next execution or after a new deployment, injects these probes into the source code. This restriction exists because our approach relies on source code injection, as AOP-based instrumentation does not support precise instrumentation of nested control flows, such as nested loops or branches.

5. *Focus on Backend Performance Prediction*: Although the first contribution of CIPM (**C1**) can extract the front-end interfaces, the focus of CIPM is on the backend performance prediction. Front-end performance prediction was excluded from this thesis due to its unreliability, which stems from numerous client-side factors, such as browser variations and client device differences [Wes+13]. Additionally, front-end prediction demands manual effort, specific execution models, and targeted measurements [Wes+13]. The PCM used in this work as aPM is also inherently unsuitable for front-end performance modeling.

   Future research could explore front-end performance prediction within the CIPM context. This would involve extending the `Palladio` aPM to accommodate front-end prediction or maintaining a suitable front-end performance model aligned with software artifacts.

7.6. Discussion

In this chapter, we have addressed the research questions RQ4 and RQ5, focusing on the preservation of consistency defined by R_{Ops} between the parameterized aPM and operational measurements from the production environment.

To answer RQ4, we introduced our fourth contribution, **C4**, which emphasizes the continuous and automatic updating of the aPM using real-time measurements from the production environment. This approach ensures that the aPM remains aligned with the current architecture in operation, as reflected in real-time operational measurements. By maintaining this alignment, we effectively uphold the consistency relation R_{Ops}, ensuring that performance parameters are informed by actual measurements. In comparison to existing works that extract or update the architecture from measurements at Ops-time, our approach avoids the limitations associated with probabilistic or potentially incomplete call graphs. This is achieved by constructing our SCG using the mapping between the source code model and the repository model, ensuring a more precise and comprehensive representation of the architecture. We acknowledge that future work should extend the update process of our `Resource Environment Model` to incorporate additional parameters, such as latency, and to enhance the detection of resources, including shared memory and other system-level resources. Such an extension should be feasible, as our approach already leverages the necessary monitoring data along with knowledge derived from static analysis of the source code. This combination provides a foundation for updating the `Resource Environment Model`, addressing limitations observed in the existing works discussed in the related literature.

To answer RQ5, we introduced our fifth contribution, **C5**, which focuses on self-validation for enhancing trust in the validity of the aPM while minimizing monitoring overhead. This contribution directly addresses the uncertainty inherent in the aPM by providing a systematic validation process that assesses the model's accuracy. The insights gained from self-validation feed into the update process, allowing for iterative improvements in the aPM's accuracy. Furthermore, **C5** optimizes the management of monitoring probes to reduce the overall monitoring overhead, thereby addressing the issue of excessive resource consumption during the validation process. To our knowledge, our approach is the first to employ a validation loop to improve the accuracy of the aPM while addressing multiple challenges, including balancing monitoring overhead, selecting appropriate metrics for accuracy measurement, and enabling adaptive recalibration to minimize processing time. Addressing these challenges is important for the applicability in Ops-time. Moreover, our self-validation approach enables the integration of alternative machine learning strategies or metrics to assess the accuracy of the aPM, providing flexibility to adapt to varying system requirements and validation needs.

Together, these contributions aim to facilitate an efficient pipeline for updating the aPM based on monitoring data from the production environment. The goal is not only

to ensure that the aPM reflects the current operational state but also to provide the necessary mechanisms to validate and enhance its accuracy over time. As a result, our approach facilitates AbPP at Ops-time using an up-to-date aPM, enabling the evaluation of the configuration of the system and its potential impact on performance. This capability helps in providing proactive performance management and enhances decisions making over running time.

Part III.

Validation and Discussion

8. Validation Strategy

In this chapter, we introduce the validation strategy we follow to validate the CIPM approach. Mainly, we follow Böhme and Reussner's validation levels [BR08] that are described in Section 2.7.1 to validate CIPM with measurements. The validation covers the first two levels: the validation of the accuracy of performance prediction (level 1) and the validation of the approach's applicability in terms of the required monitoring overhead and scalability (level 2). Moreover, we conduct a survey focusing on exploring the acceptable cost of adopting CIPM within the industry in terms of the required time for setting up and learning the tool, and the level of trust agile software developers can place in the CIPM approach for practical adoption (level 2).

Our validation is goal-oriented following the GQM approach, which we introduce in Section 8.1. We define the evaluation goals and classify them into evaluation levels of Böhme, and Reussner [BV08] in Section 8.1.1. Then, we introduce the related evaluation questions in Section 8.1.2. The used metrics are described in Section 8.1.3. In Section 8.2, we classify the research objects of CIPM approach and identify how we validate them. In Section 8.3, we introduce the cases that we use for the validation. The description of the implemented research objects (CIPM tool) is presented in Section 8.4.

In Section 8.5, we present an overview of the conducted experiments and their alignment with the validation goals. Subsequently, in the following chapters, we provide detailed descriptions of these experiments: Chapter 9 focuses on Experiment1 and Experiment2 that evaluate the consistency maintenance at Dev-time. Chapter 10 includes experiments that assess the accuracy of AbPP following incremental calibration. Finally, Chapter 11 includes experiments related to Ops-time.

8.1. Goal-oriented Evaluation

In our evaluation, we follow the GQM approach that is introduced by Basili et al. [Sol+02] and described in Section 2.7.4. According to this approach, we first define the main goals for the evaluation. Second, we drive the evaluation questions that can check whether the described goals are reached. Then, we apply the goal-oriented measurements to obtain the required measurements for answering the defined questions. Third, we use metrics that quantify the measured data to answer the evaluation questions.

We arrange the evaluation goals according to the validation levels of prediction models that are introduced by Böhme and Reussner [BR08]: (Level 1) validating the

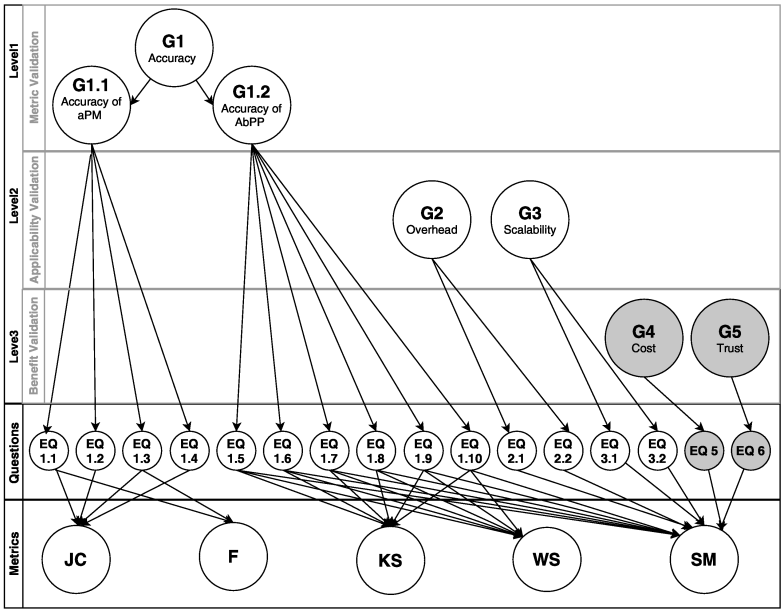


Figure 8.1: Goals, evaluation questions, and metrics employed to validate the CIPM approach. Gray-highlighted goals and their corresponding evaluation questions are explored through a survey.

predictions by comparing them with measurements, (Level 2) validating the applicability of the prediction models and (Level 3) validating the benefits gained from the prediction model in comparison to the related prediction approaches.

In Section 8.1.1, we provide more detail on the evaluation goals. In Section 8.1.2, we introduce the evaluation questions. The used metrics follow in Section 8.1.3.

8.1.1. Evaluation Goals

CIPM aims to update aPMs with high accuracy, minimize monitoring overhead, ensure scalability, and address trust concerns in the updated models.

The main validation goal is to assess the degree to which CIPM objective is achieved. This main goal can be divided into the following five validation sub-goals and arranged to validation levels of Böhme and Reussner [BR08]:

G1 **Accuracy:** The accuracy of the updated models and the associated AbPP is the main goal of this approach. Since we evaluate the accuracy by comparing the measurements with true values, we set this goal at the first level.

G2 **Overhead:** To ensure the applicability of the CIPM approach (second validation level), the required monitoring overhead should be acceptable and must not have a negative impact on the system's performance.

G3 **Scalability:** The scalability of CIPM, especially at Ops-time, is an important factor in responding as quickly as possible to changes in the system, removing the possible inconsistencies and avoiding negative impact on the system's performance. The validation of scalability is assigned to the second validation level, where it affects the applicability of CIPM approach.

G4 **Cost:** The cost associated with implementing performance predictions in agile software development using CIPM approach should be lower than that of concurrent approaches. The validation level of this goal corresponds to the third level of Böhme and Reussner's approach [BR08], as it evaluates the benefits of CIPM in comparison to alternative approaches. This goal (G4) is not validated because the empirical evaluation of the assumed benefit would be expensive: It requires conducting an experiment, where the system is implemented at least twice, ideally by the same or equivalent developers, under identical agile software development conditions. By first implementation, CIPM would be employed for predicting the performance along the development, while the other would use a competing approach. Additionally, this experiment should be repeated to thoroughly assess the impact and benefits of CIPM, which significantly increases the required cost and effort.

Therefore, we validate this goal by arguing that CIPM automation will enable performance predictions at a lower cost than architecture-based approaches (cf. Section 12.1.1), which do not provide such automatic updates of aPMs. This also applies to measurement-based approaches (cf. Section 12.1.1) that do not adopt aPMs at all and perform expensive performance tests to assess the design alternatives. Consequentially, the cost of software development would be reduced if CIPM is applied instead of APM or other approaches that manually update aPMs to apply AbPP. We are aware that additional costs will be required for adjusting the CPRs for the project. However, this cost is applied once and the CPRs will be reused for projects using the same programming language and technology.

We conducted a survey [AMK24] to ask participants about the time they are willing to invest in the setup, learning, and adoption of a new tool like CIPM. This survey seeks input from participants to inform whether the required cost for setting up CIPM, including defining suitable CPRs, is considered acceptable within their respective. Insights from results are in Section 13.1.1.

G5 **Trust:** This goal aims to assess whether the CIPM properties, mainly the operational self-validation, can increase the agile developers' trust in the updated aPMs, given their tendency to consider models as mere approximations. For that, we conducted a survey asking the experts whether CIPM features can improve their trust in the aPMs and the AbPP. However, we can argue that the operational self-validation provided by CIPM is considered one of the various

stages required for establishing trust in a model [HMY21], and providing it should improve the trustworthiness, particularly as the CIPM self-validation is based on real measurements and it informs developers how accurate the aPMs are.

As shown in Figure 8.1, the aforementioned goals are arranged to the related validation levels: the first one, "G1: Accuracy," is aligned with the first validation level of Böhme and Reussner [BR08], which is based on a comparison between predictions of aPMs and the measurements as truth values. The second and third goals ("G2: Monitoring Overhead" and "G3: Scalability") consider the applicability of CIPM approach. Therefore, both G2 and G3 belong to the second validation level.

Both G4 and G5 are explored through a survey [AMK24]. Due to time constraints, we were not able to include details on the survey's design and results in the thesis; rather, we provide the main insights gathered from the survey (Section 13.1.1). Therefore, they are colored gray.

Both the evaluation questions and metrics are described in Section 8.1.2 and Section 8.1.3, respectively.

8.1.2. Evaluation Questions

To evaluate the key aspects of CIPM approach, we define the following EQs and arrange them to the evaluation goals:

- **G1.1: Accuracy of the incrementally updated models**:
  - **EQ-1.1:** How accurately does CIPM update the `Repository Model` and its related models within VSUM according to a Git commit?
    - **EQ-1.1.1:** How accurate is the `Repository Model` that CIPM reverse-engineers by integrating an initial commit in a Git history?
    - **EQ-1.1.2:** How accurately does CIPM update the VSUM models when it aggregates multiple commits and propagates them as a single commit?
  - **EQ-1.2:** How accurately and automatically does CIPM extract the `System Model` at Dev-time?
  - **EQ-1.3:** How accurately does the adaptive instrumentation instrument the source code based on collected instrumentation probes in IM?
  - **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?
- **G1.2: Accuracy of AbPP using the updated aPMs:**
  - **Accuracy of AbPP at Dev-time:**

    - **EQ-1.5:** How accurate is the AbPP using the aPMs that are incrementally calibrated at Dev-Time?
    - **EQ-1.6:** Is the accuracy of AbPP using the incrementally calibrated aPM affected by the incremental evolution?
  - **Improving the accuracy of AbPP by parameterizing the dependencies:**
    - **EQ-1.7:** How accurately can CIPM identify the parametric dependencies, and to what extent can the estimation of them improve the accuracy of the AbPP?
    - **EQ-1.8:** To what extent can optimizing the PMPs by applying the genetic algorithm improve the accuracy of the AbPP?
  - **Accuracy of AbPP at Ops-time:**
    - **EQ-1.9:** How accurate is the AbPP using aPMs that are incrementally updated at Ops-time?
    - **EQ-1.10:**What impact do Ops-time changes have on the accuracy of the AbPP?
- **G2: Monitoring Overhead**
  - **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 CIPM approach:**
  - **EQ-3.1:** How does the CI-based update of VSUM scale with the amount of source code changes?
  - **EQ-3.2:** How does the CIPM pipeline scale with an increasing amount of monitoring records at Ops-time?
    - **EQ-3.2.1:** How does the Ops-time calibration of `Repository Model` scale with an increasing amount of monitoring records?
    - **EQ-3.2.2:** How does Ops-time calibration of both `System Model` and `Allocation Model` scale with an increasing amount of monitoring records?
    - **EQ-3.2.3:** Does the scalability of Ops-time calibration of `Usage Model` demonstrate similarities to the scalability of the method employed by `iObserve` for calibrating `Usage Model`?
    - **EQ-3.2.4:** How does the Ops-time calibration of `Resource Environment Model` scale with an increasing amount of monitoring records?
- **G4: The cost of CIPM approach**

  - **EQ-4:** How much time would the industry be willing to invest in setting up and learning a new performance prediction tool, such as CIPM?

- **G5: The trust in CIPM approach**
  - **EQ-5:** What factors influence industry professionals' trust in features of CIPM and to what extent do these factors play a role?

8.1.3. Evaluation Metrics

The metrics used for evaluation can be categorized into three main groups: The first category comprises metrics that assess the accuracy of the updated aPMs, including JC and $F1$ score in Section 8.1.3.1 and Section 8.1.3.2 respectively. The second category focuses on evaluating the accuracy of performance predictions made by these models through KS-test and WS-distance, as outlined in Section 8.1.3.3 and Section 8.1.3.4. Lastly, the third category described in Section 8.1.3.5 involves the use of statistical metrics to assess various aspects of CIPM like performance prediction accuracy, monitoring overhead, and scalability.

8.1.3.1. Jaccard Similarity Coefficient for Model Accuracy

JC is commonly used to measure the similarity between two sets by comparing their intersection to their union (Section 2.8.2). As models can be considered to be sets of elements for the purpose of evaluating accuracy [Hei20], we apply this concept in our case as well. In the context of aPM evaluation, JC quantifies how closely the updated model aligns with the original or expected model by assessing the overlap between model components, parameters, or other relevant attributes.

To determine the JC, we utilize algorithms that perform pairwise comparisons of model elements to identify equal elements between two models, A and B. These comparisons assess the equivalence of model elements based on different factors. Firstly, both elements must be of the same type. Depending on the type, further properties are then evaluated, such as:

- Named elements must have the same name.
- Certain referenced elements must match.
- Elements must be embedded in an equivalent model structure (e.g., the same position in a list).

These comparisons yield a set of equal model elements, which we consider the intersection of the models ($A \cap B$). Conversely, the union of the models ($A \cup B$) comprises the equal elements, as well as unique elements from A and B that lack an equivalent

in the other model. By calculating both the intersection and union of two models, we can directly compute the JC for both models. In the following, we explain how we calculate JC for different models, focusing on how we match the elements to calculate the intersection ($A \cap B$).

Source Code Model: As we described in Section 5.1.2.2, we used a custom language-specific matching algorithm [Kol+09] for both Java and Lua: We combine the default matching algorithm of EMF Compare [Lan19] with a hierarchical language-specific matching algorithm. For Java, we extended SPLevo [Kla14] to be compatible with Java 7-15. For Lua, we implemented a hierarchical Lua-specific matching algorithm [Bur23]. Thus, the properties and structure of SCM are considered to provide an accurate match.

Repository Model: We utilize EMF Compare [Lan19] to implement the structural similarity-based matching algorithm [Arm21]. The similarity-based algorithm defines relative weights for the features of `Repository Model` elements. For example, it considers the '`name`' of the named elements like components, interfaces, signatures, parameters, etc. Contrary to the `name` feature, the similarity-based algorithm ignores the `id` since the updated models and the reference models are created separately. Moreover, the defined similarity-based algorithm considers the structure of the model by checking the similarity of the referenced elements. For instance, for two `OperationInterface` elements, we check the similarity of the parent interface in addition to the name feature. This also applies to `OperationProvidedRole/ OperationRequiredRole` elements, where the similarity of the referenced Provided interface/ Required interface should be also checked. Another example of the weighted features is the referenced parameters and return type of an `operationSignature` element.

Regarding SEFF elements, first, we check the similarity of the referenced service. If they are considered equal, we compare the actions of the SEFFs (`InternalCallAction, ExternalCallAction, LoopAction` etc.) in addition to the sequence of these actions. For example, we match two `ExternalCallAction` elements if they call the same service and their positions in the SEFF are also similar. For more understanding of the `Repository Model` structure, see Figure A.1.

System Model: In the case of `System Model`, a static identity-based matching (cf. Section 2.2.7) is implemented [Mon20]. Because each element of the `System Model` refers to elements with a unique identifier in the `Repository Model`, it becomes possible to compare the elements of the `System Model` based on their types and some unique values of their features, e.g., identifiers of the referenced elements of the `Repository Model`.

For instance, we consider two assembly contexts identical if they are instances of the same component in `Repository Model`. In other words, two assembly contexts are identical if the IDs of their underlying referenced component are identical. Similarly, the

assembly connectors that join provided and required roles are matched if the identifiers of the reference provided and required interfaces in the Repository Model are equal.

Allocation Model: Like the System Model, elements of the Allocation Model refer to the static Repository Model, see Figure A.4. Thus, we can match elements based on the identifiers of the referenced elements. The implemented matching algorithm [Mon20] matches the allocationContext if the same components are deployed on the same resource containers. This implies that not only the types of deployed components are considered by the matching process, but also the number of deployments of each specific component type. For that, the matching algorithm checks the equality of identifiers of the encapsulated component that the assembly context refers to, and the specification of their resource containers. Therefore, we check the conformity of Allocation Model and Resource Environment Model at the same time.

Resource Environment Model: For checking the similarity of two Resource Environment Model, we implemented a similarity-based matching algorithm that matches the elements of two resource environment models based on the aggregated similarity of their features [Mon20]. For instance, the algorithm matches two resource containers if their specifications are identical, especially the processing rate and the number of CPU cores (numberOfReplicas); for more detail, see Figure A.5. Similarly, the specification of other hardware resources, like communication links, must also be identical to match the linking resources. So that, we can ensure that the simulation based on matched linking resources results in similar predictions.

8.1.3.2. F1 score for Model Accuracy

In cases where a model is updated using information from a related heterogeneous model, we can validate the updated model through an information retrieval test. This test ensures that the updated model accurately reflects all relevant information from the related model and excludes any unrelated information. Specifically, we quantify the relevance of model elements (precision) and their sensitivity (recall). These two metrics, precision and recall, are then combined into a single measure, $F1$ score (Section 2.8.1), to assess the quality of the updated model.

We use $F1$ score to validate IM and instrumented SCM. Moreover, $F1$ score is used for validating the Repository Model generated by integrating an initial commit when we have no reference model, rather than just a documented architecture.

In the following, we explain the method we use to validate the above-mentioned models using $F1$ score.

Instrumentation Model: We validate the accuracy of the IM based on the updated Repository Model. The relevant instrumentation probes in IM are either coarse-grained

points corresponding to existing SEFFs or fine-grained points corresponding to recently updated SEFF actions. The precision calculates the proportion of relevant instrumentation probes among all instrumentation probes in the updated IM. If the precision equals one, this means that all instrumentation probes are relevant. The recall measures the proportion of relevant instrumentation probes into IM among all required instrumentation probes. If the recall does not equal one, this means that some instrumentation probes are missed. In other words, IM does not include instrumentation probes for existing SEFFs or recently updated SEFF actions. Based on precision and recall, we calculate $F1$ score. If the $F1$ score value is equal to one, this means that the IM is correctly updated.

Instrumented Source Code Model: We validate the accuracy of the instrumented SCM based on the instrumentation probes in the IM. The relevant instrumentation statements that are added to the SCM are either related to coarse-grained or fine-grained instrumentation probes in IM.

The precision calculates the proportion of relevant instrumentation statements among all added instrumentation statements in the updated instrumented SCM. If all instrumentation statements are relevant and correctly added, the precision value equals one and the source code has no compiler errors. The recall measures the proportion of relevant instrumentation statements into instrumented SCM among all required instrumentation statements. If some parts of SCM are not instrumented as required, the recall is less than one, which means some instrumentation probes are missed.

In our evaluation, we distinguish between calculating these metrics for the instrumentation statements that are related to coarse-grained instrumentation probes and those related to fine-grained ones. There are two reasons for this distinction: First, the goal of validating the instrumented SCM, is the validation of the *adaptive* instrumentation, which is mainly related to the fine-grained instrumentation. The coarse-grained instrumentation is required but applied to all services (not adaptively like the fine-grained one). Second, the manual frequent evaluation of the coarse-grained instrumentation requires a big overhead since we consider long Git histories. Hence, our evaluation calculates the precision, recall and $F1$ score for the fine-grained instrumentation by manually checking the fine-grained instrumentation statements. Besides, we define an additional metric that counts the added instrumentation probes and compares the result with the count of required instrumentation statements, which is calculated based on the instrumentation probes in IM. More detail about the count-based metric is in [Arm21].

Repository Model: We validate the accuracy of the initial Repository Model based on documented architecture as a reference. The relevant elements of the Repository

`Model` are the components and interfaces that are equivalent to existing ones in the reference architecture.

The precision calculates the proportion of relevant components and interfaces in the initial `Repository Model` among all extracted components and interfaces. The recall measures the proportion of relevant components and interfaces into initial `Repository Model` among all documented components and interfaces in the reference architecture. If some components or interfaces in the reference architecture are not extracted, the value of recall is less than one.

8.1.3.3. Kolmogorov-Smirnov Test for Performance Prediction Accuracy

For evaluating AbPP, we compare the simulation results with real measurements. We predict the response time by simulating the updated aPM and comparing it to the measured response time as a (real value). We take into account the response times for multiple executions of a service and aggregate them in a CDF. Then we compare two CDFs representing the measured and predicted response time using the following metrics. First, we use the conventional statistical measures [UC08] like the average, median, etc. Second, we use the non-parametric KS-test [She07], which allows calculating the maximal vertical distance between two CDFs in a value between zero and one, see Section 2.8.3. The KS-test of identical CDFs results in zero value. Generally, the lower value the KS-test results in, the more similar is the distribution under testing. It should be noticed that the KS-test is sensitive to the shifts and shapes of distributions, which may produce undesirable false positive alerts (high values) [Huy22]. For example, if the mean of two distributions is almost identical (which we consider an accurate prediction) but the shape of the distributions is different (which we do not think is a problem for the case of performance prediction). Thus, we use the additional metrics: the Wasserstein distance and the statistical metrics, as we explain in the following sections.

8.1.3.4. Wasserstein Distance for Performance Prediction Accuracy

In our analysis, we incorporate the WS-distance [Mém11] to assess the performance prediction accuracy. The WS-distance serves as a fundamental measure for quantifying the minimum cost required to transform one CDF into another through horizontal shifts, as described in Section 2.8.4. Therefore, we employ WS-distance to quantify the degree of similarity between the CDF of response time that is obtained through simulation and the CDF of response time data obtained through measurement.

Like the KS-test, lower values of WS-distance signify a higher degree of similarity between the CDFs being compared. This means that when the WS-distance is minimized, it indicates that the two CDFs are closely aligned and share more common characteristics.

The practical implication of achieving a lower WS-distance value is that it enhances the accuracy of our AbPP. In other words, a lower Wasserstein distance suggests that our approach can more accurately estimate the performance.

8.1.3.5. Statistical Metrics

We calculate the known SMs [UC08] like mean, median, and quartiles for quantifying the accuracy or the monitoring overhead. Moreover, the absolute and relative errors were also used to calculate the error of performance prediction.

8.2. The Classification Scheme of CIPM Contributions

In this section, we classify the CIPM approach according to the classification scheme introduced by Kaplan et al. in [KWH21] and evaluated in [KK21], cf. Section 2.7.2. For that, we classify the statements of CIPM contributions that are introduced in Chapter 1 to link the statements to their evidence. Moreover, we will arrange these statements to the validation goals, questions and metrics, which are followed to provide the proof.

8.2.1. C1: Automated Consistency Maintenance at Dev-Time

The first contribution **C1** can be divided into six research objects that are listed in Table 8.1 and visualized in Figure 8.2. The main research object, "CI-based update of software models", is a Method (1.2) that aims to update `SCM`, aPM and `IM` to improve the models' "accuracy", which is classified as Property-as-such (2.1). To validate the accuracy of the models updated by this method, we conducted experiments (3.5) that update the aPMs based on mining software repositories (3.7) of four case studies (3.3) , cf. Section 9.1.

Both the designed instrumentation metamodel and the extended Java metamodel fall into the categories of Language (1.4). Although these metamodels were not subject to explicit evaluation, they have been effectively employed in the CI-based update process and self-validation, both have been subject to empirical validation. Consequently, we can assert that these metamodels serve as appropriate inputs for other contributions. Hence, we classify the Suitability of the metamodels as Property-in-relation (2.2) and the evidence supporting the suitability and the correctness of these metamodels can be categorized as argumentation (3.1).

The extension of the Java parser, including the newly added reference resolution, is Method (1.2). The main properties of this research object are the accuracy of the parsed models and the performance of the parser. The performance of the parser is related to the size of the parsed code and falls, therefore, under the category of Property-in-relation (2.2). We evaluate the functionality and execution time of the extended parser

by experiments parsing case studies [Arm22; MAK23]. The defined CPRs are classified as a model (1.3) and a Property-in-relation (2.2) that affect the accuracy of the updated models. Therefore, the evidence is based on the experiments above that validate the accuracy of the updated models and the Suitability of these CPRs.

Table 8.1: Classification of C1: consistency maintenance at Dev-time

Research Object	Statement	Evidence
(1.2) CI-based update of software models	(2.1) Accuracy	(3.5)(3.7) (3.3)
(1.5) Automated CI-based update of software models	(2.2) Scalability, Performance	(3.5)(3.7) (3.3)
(1.4) Instrumentation metamodel	(2.2) Suitability	(3.1)
(1.4) Extension of Java metamodel	(2.2) Suitability	(3.1)
(1.2) Java parser and printer	(2.1) Accuracy, (2.2) Performance	(3.5)
(1.3) Definition and extension of CPRs	(2.1) Accuracy, (2.2) Suitability	(3.5)(3.7)

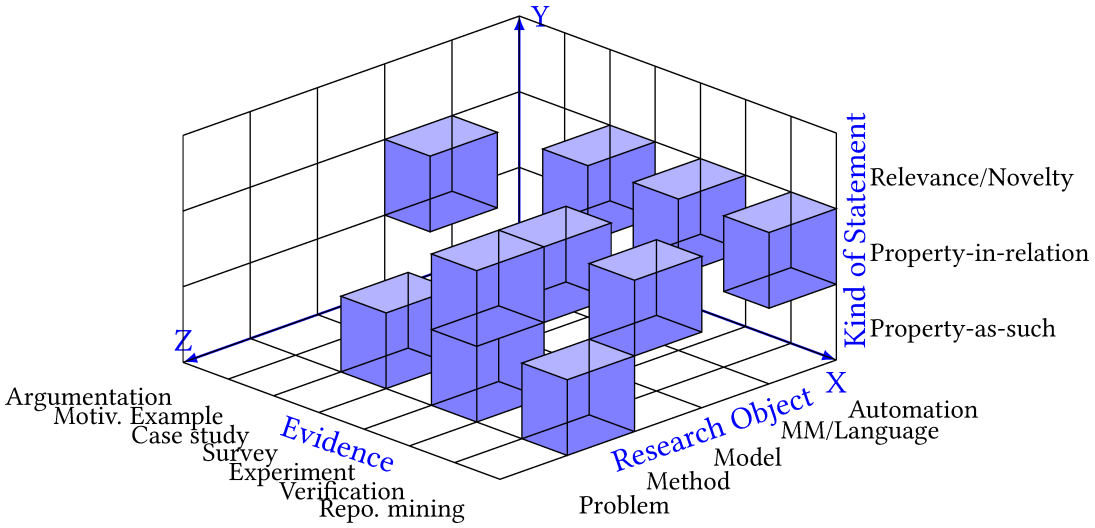


Figure 8.2: Classification of the first contribution: CI-based update of software models.

8.2.2. C2: Adaptive Instrumentation

The adaptive instrumentation is classified into two research objects: (C2.1) a process for adaptive instrumentation (Method (1.2)) and (C2.2) automation that aims to save the instrumentation overhead (Automation (1.5)), see Table 8.2 and Figure 8.3. The first research object (C2.1) aims to instrument the source code accurately and, therefore, we classify its Accuracy as a property-as-such (2.1). An additional property of this research object aims to reduce the monitoring overhead by reducing the number of monitoring probes, also classified as a property-as-relation (2.2). The evidence of these properties is

based on Experiments (3.5) conducted on three case studies that instrument the source code using real commits from software repositories (3.7). The experiments assess the accuracy and measure the extent to which adaptive instrumentation can reduce the monitoring overhead. Besides, we argue the benefits of adaptive instrumentation based on a Motivating Example (3.2).

The second research object is a Property-in-relation (2.2) that aims to reduce the cost of manual overhead required to instrument the source code adaptively according to changes in source code (commits). The evidence of (C2.2) is based on the fact that automatic instrumentation saved the required time for the instrumentation, Argumentation (1.3). Additional evidence is provided based on Experiments (3.5) that measure the required execution time to instrument the source code automatically based on mining software repositories (3.7) of two case studies. Since the cost is related to the amount and type of the changes, we classify it as a Property-in-relation (2.2).

Table 8.2: Classification of C2: automatic adaptive instrumentation

Research Object	Statement	Evidence
(1.2) Adaptive instrumentation	(2.1) Accuracy (2.2) Monitoring overhead	(3.2) (3.3) (3.5)(3.7)
(1.5) Automation of adaptive instrumentation	(2.2) Cost	(3.1) (3.3) (3.5)(3.7)

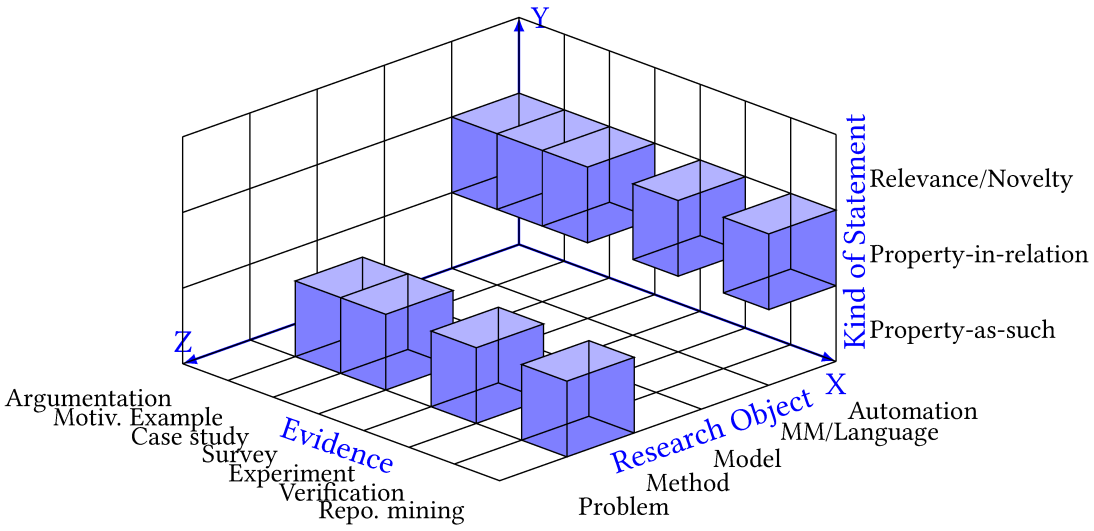


Figure 8.3: Classification of the second contribution: automatic adaptive instrumentation.

8.2.3. C3: Incremental Calibration with Parametric Dependencies

As shown in Table 8.3 and visualized in Figure 8.4, the incremental calibration of performance models consists of several research objects. First, the process itself is classified as a Method (1.2) that aims to accurately update the non-calibrated and inaccurate PMPs. The accuracy is based on the monitoring data and classified as a Property-as-relation (2.2). For this process, a metamodel for measurements is designed and used (Language (1.4)). Based on this metamodel, the incremental calibration can automatically (Automation (1.5)) provide prediction models for PMPs considering the parametric dependencies (Model (1.3)).

We use a motivating example (3.2) as evidence of the incremental calibration. Moreover, we conducted some experiments (Experiment (3.5)) on two case studies to apply incremental calibration and ensure that the changed parts of aPM are accurately updated with minimal monitoring overhead.

The Accuracy of prediction models is classified as Property-in-relation that depends on the monitoring data and Relevance because its benefit is evaluated in comparison to other calibrations. Therefore, we conducted experiments to show that parametric dependencies improve the accuracy of prediction models (Experiments (3.5)). The experiments evaluate the accuracy of prediction models for different inputs and compare them to another calibration approach that does not consider the parametric dependencies. We argue that the suitability of the measurement metamodel is indirectly evaluated through the conducted experiments and classified the evidence of the measurement metamodel as Argumentation (1.3).

The automation of incremental calibration at Dev-time (Automation (1.5)) is (Property-in-relation (2.2)) that aims to reduce cost in terms of the manual effort and the required time for calibration. For the evidence, we perform experiments that measure the required execution time for calibrating the aPMs with different input parameters (Experiments (3.5)). The execution time of the incremental calibration is not a critical property at Dev-time, it can be performed, for example, nightly. Therefore, we did not evaluate the scalability property in detail at Dev-time, as we evaluated it by the Ops-time calibration (cf. Section 8.2.4), where a long execution time can lead to actual performance issues.

Table 8.3: Classification of C3: incremental calibration.

Research Object	Statement	Evidence
(1.2) Incremental calibration process	(2.2) Accuracy, (2.3) Extrapolation-ability	(3.2) (3.3) (3.5)
(1.4) Metamodel for estimating PMPs	(2.2) Suitability	(3.1)
(1.3) Prediction models	(2.2) Accuracy, (2.3) Extrapolation-ability	(3.3) (3.5)
(1.5) Automation of incremental-calibration	(2.2) Cost	(3.5)

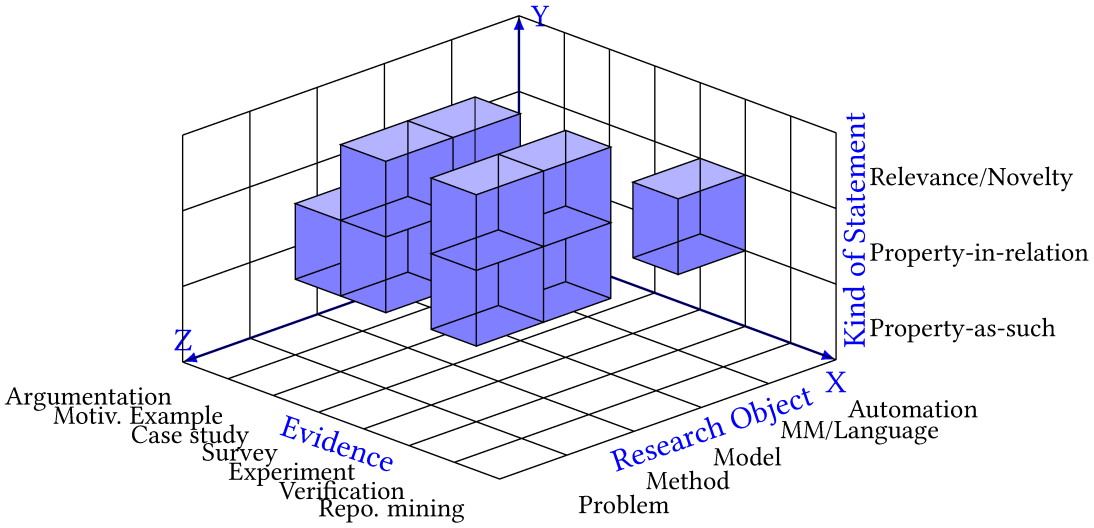


Figure 8.4: Classification of the third contribution: incremental calibration of performance models with parametric dependencies.

8.2.4. C4: Ops-Time Calibration

The automation of the Ops-time calibration (Method (1.2)) is a process that updates the aPMs continuously at Ops-time according to monitoring data (Property-in-relation (2.2)). The evidence of this process is based on experiments that measure how accurately this process can update different parts of aPMs at Ops-time (Experiments (3.5)). The experiments are performed on two case studies. Automation of Ops-time calibration (Automation (1.5)) is evaluated by investigating scalability (Property-in-relation(2.2)) by experiments measuring the execution time for different input parameters (Experiments (3.5)).

Table 8.4: Classification of C4: Ops-time calibration.

Research Object	Statement	Evidence
(1.2) Ops-time calibration process	(2.2) Accuracy	(3.2) (3.5)
(1.5) Automation of Ops-time calibration	(2.2) Scalability	(3.5)

8.2.5. C5: Self-Validation

The self-validation process (Method (1.2)) is combined with the Ops-time calibration and re-calibrates PMPs with the results of previous validation (Property-in-relation(2.2)).

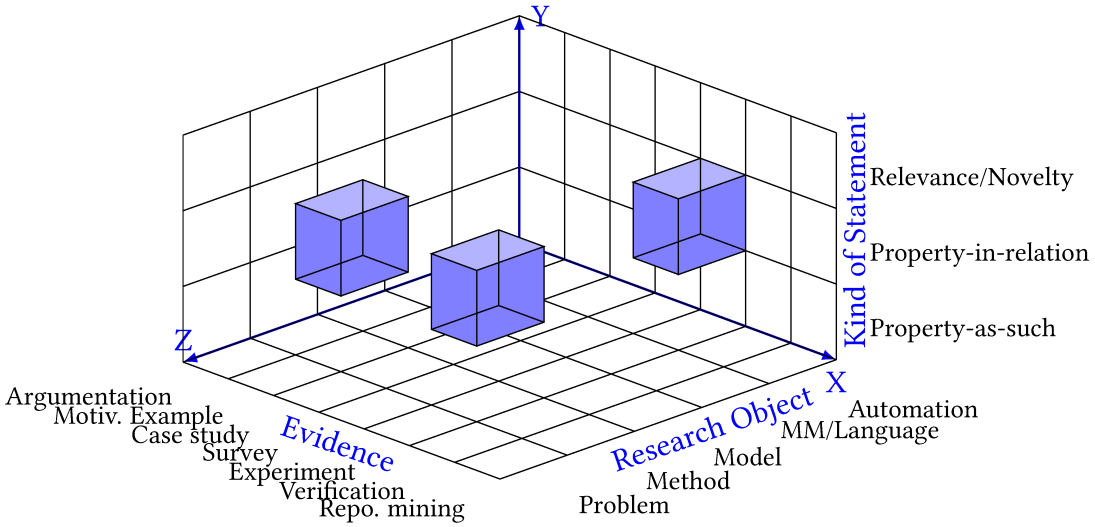


Figure 8.5: Classification of the fourth contribution: Ops-time calibration of performance models.

The evidence of the self-validation considers the accuracy of aPM including their PMPs and the reduction of monitoring overhead that the self-validation can perform.

For that, we conducted experiments (Experiments (3.5)) that recalibrated the aPMs after Ops-time changes in the worst case, where PMPs were not calibrated before. Moreover, the experiments also consider the worst case of monitoring, where all fine granular probes are activated.

Table 8.5 summarizes the classification classes of the self-calibration, whereas Figure 8.6 visualize them.

Table 8.5: Classification of C5: Self-validation.

Research Object	Statement	Evidence
(1.2) Self-validation	(2.2) Accuracy, Overhead	(3.5)

8.2.6. C6: MbDevOps Pipeline

The proposed MbDevOps pipeline, introduced in Section 4.3, explains how CIPM activities can be automated and integrated within the DevOps software development lifecycle as an example of agile software development. Therefore, we classified the pipeline as an automated process (Method (1.2), Automation (1.5)).

The proposed MbDevOps pipeline integrates the contributions above (**C1**-**C5**) to keep the software artifacts consistent. The goal is to support design decisions at each point of

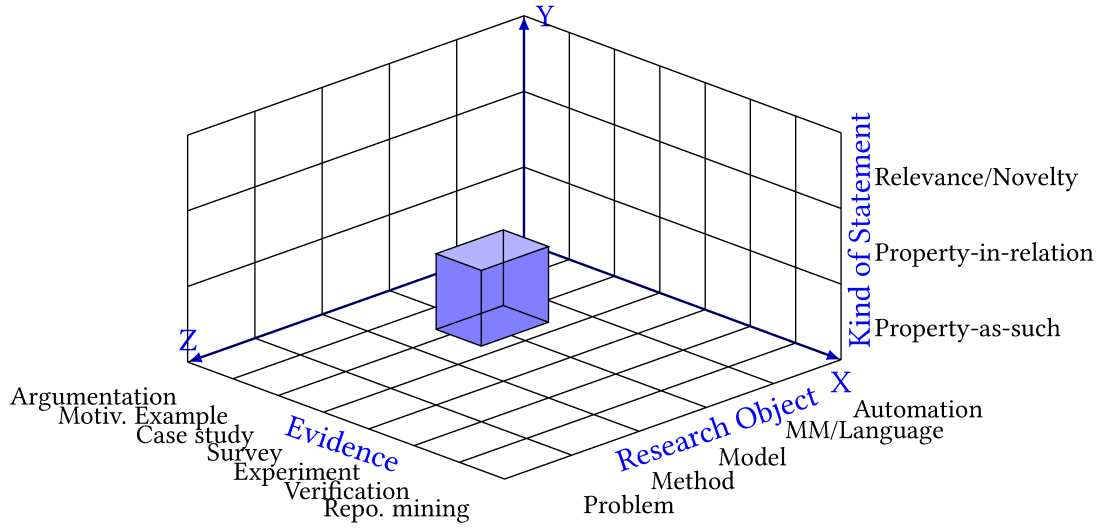


Figure 8.6: Classification of the fifth contribution: self-validation of performance models.

the software lifecycle through accurate AbPP. This can save the cost required to support such decisions through expensive performance tests. Hence, the statement "MbDevOps Pipeline extends the DevOps pipeline to provide aPM and enables cost-effective AbPP during agile software development" focuses on the usability of CIPM and its potential benefits within agile software development. Thus, the statement can be classified as Relevance (2.3) highlighting the value and novelty of the automatic update of aPM that minimizes the updating effort and monitoring overhead.

As shown in Table 8.6 and Figure 8.7, the evidence of the general applicability (in terms of the usability for agile including CI-context, the accuracy of updating models, the scalability of automated activities and the required monitoring overhead) is based on the evidence of contributions **C1**-**C4** (Case Study (3.3), Experiments (3.5) and mining software repositories (3.7)).

Additional evidence of the benefits of the MbDevOps pipeline is the Argumentation (3.1) and Motivation Example (3.2) since no empirical evidence for the cost is provided, but the benefits are just explained and argued with a motivating example. Moreover, we conducted a Survey (3.4), asking the participants about the cost, in terms of the required time, that they agree to spend for adopting and utilizing the proposed MbDevOps pipeline. The survey also inquires about participants' confidence levels in the aPM and the associated AbPP and whether the properties provided by the MbDevOps pipeline can improve the level, see Section 13.1.1. Regarding the automation of MbDevOps, it can be classified as an automation (1.5), where the scalability property (1.2) is evaluated based on experiment (3.5).

Table 8.6: Classification of C6: MbDevOps pipeline.

Research Object	Statement	Evidence
(1.2) MbDevOps pipeline	(2.3) Cost, Usability	(3.1)(3.2) (3.3) (3.4)(3.5)(3.7)
(1.5) Automation MbDevOps pipeline	(2.2) Scalability	(3.5)

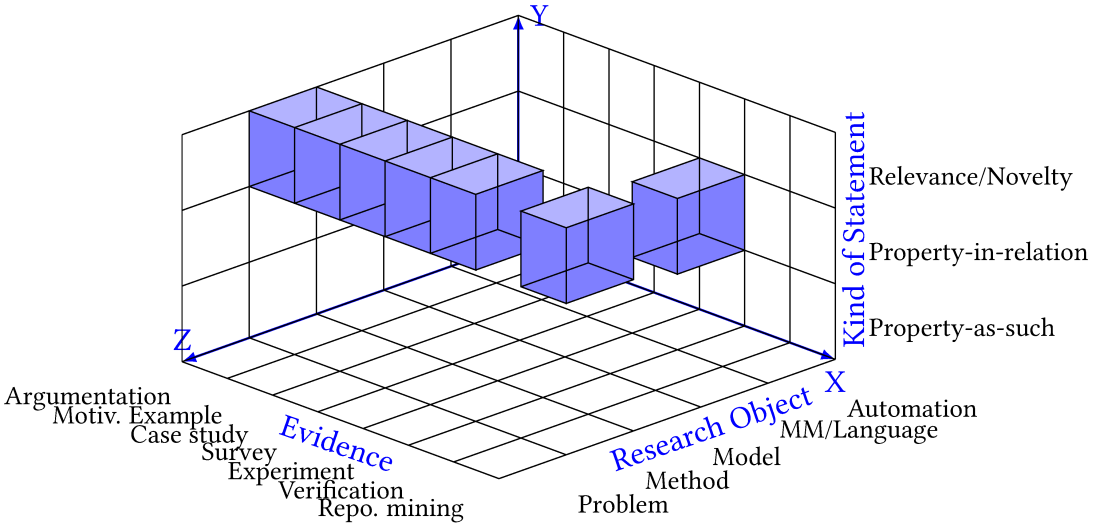


Figure 8.7: Classification of the sixth contribution: MbDevOps Pipeline.

8.2.7. Summary of Contributions Classification

In Section 8.2.7, we provide a comprehensive summary of the contributions from the CIPM approach. These contributions group their primary research objects and the types of statements they are related to. Additionally, we summarize the supporting evidence for each contribution, outline the (GQM) plan associated with them, and provide details on the related experiments. Consequentially, this table serves as a reference to gain an overview of the contributions of this thesis and how they are validated.

Table 8.7: Summary of contributions including their classification, the applied GQM plan, and validation methods.

Contrib.	Research Object	Statement	Evidence	GQM			Experiments
				G	Q	M	
C1	**(1.2) CI-based update of software models**	(2.1) Accuracy	(3.5) (3.3) (3.7)	G1.1 G3 G4	EQ-1.1 EQ-1.1.2 EQ-1.2 EQ-3.1	JC F1 SMs	**E1** **E2**
	(1.5) Automated CI-based update of software models	(2.2) Scalability, Performance	(3.5) (3.3) (3.7)				
	(1.4) Instrumentation Metamodel	(2.2) Suitability	(3.1)				
	(1.4) Extension of Java Metamodel	(2.2) Suitability	(3.1)				
	(1.2) Java parser and printer	(2.1) Accuracy (2.2) Performance	(3.5)				
	(1.3) Definition and extension of CPRs	(2.1) Accuracy (2.2) Suitability	(3.5)(3.7)				
C2	**(1.2) Adaptive instrumentation**	(2.1) Accuracy (2.2) Monitoring overhead	(3.2) (3.3) (3.5) (3.7)	G2 G4	EQ-1.3 EQ-2.1	SMs	**E1** **E3**
	(1.5) Automation of adaptive instrumentation	(2.2) Cost	(3.1) (3.3) (3.5) (3.7)				
C3	**(1.2) Incremental calibration process**	(2.2) Accuracy (2.3) Extrapolation-ability	(3.2) (3.3) (3.5)	G1.2 G4	EQ-1.5 EQ-1.6 EQ-1.7 EQ-1.8 EQ-1.9	KS WS SMs	**E3** **E4**
	(1.4) Meta-Model for estimating PMPs	(2.2) Suitability	(3.1)				
	(1.3) Prediction models	(2.2) Accuracy (2.3) Extrapolation-ability	(3.3) (3.5)				
	(1.5) Automation of incremental-calibration	(2.2) Cost	(3.5)				
C4	**(1.2) Ops-time calibration**	(2.2) Accuracy	(3.2) (3.5)	G1 G3 G4	EQ-1.4 EQ-1.10 EQ-3.2	KS WS SMs JC	**E4** **E5**
	(1.5) Automation of Ops-time calibration	(2.2) Scalability	(3.5)				
C5	**(1.2) Self-validation**	(2.2) Accuracy, Overhead	(3.5)	G1.2 G2 G4 G5	EQ-2.2 EQ-1.4 EQ-1.10	KS WS SMs	**E4**
C6	**(1.2) MbDevOps pipeline**	(2.3) Cost, Usability	(3.1)(3.2)(3.3)(3.4)(3.5)(3.7)	G1-G5	EQ1-EQ3	JC KS WS SMs	**E1** **E2** **E3** **E4** **E5**
	(1.5) Automation of MbDevOps pipeline	(2.2) Scalability	(3.5)	G3	EQ-3.1 EQ-3.2	SMs	**E1** **E5**

Research Object: Problem (1.1), Method (1.2), Model (1.3), MM/Language (1.4), Automation (1.5)
Kind of Statement: Property-as-such (2.1), Property-in-relation (2.2), Relevance/Novelty (2.3)
Evidence: Argumentation (3.1), Motiv. Example (3.2), Case Study (3.3), Survey (3.4), Experiment (3.5), Verification (3.6), Mining Software Repositories (3.7)

8.3. Case Study

In this section, we provide additional information on the process of the applied empirical case study and the selected cases. In this case study, we observe the consistency preservation phenomena that CIPM applies between software artifacts, especially the performance model.

As suggested in Section 2.7.3, we integrate two research strategies: case studies and experiments. Our research centers around observing the CIPM consistency preservation between software artifacts within the software development. To achieve this, we conduct case studies alongside experiments that isolate specific aspects of the case study to gain better control over the situation, even though it comes at the cost of some realism.

We argue that our research qualifies as a case study because it follows the points outlined in the case study definition: *A case study is (1) an empirical investigation of a case, (2) using multiple data collection methods, to study a (3) contemporary phenomenon in its (4) real-life context, and with the investigator(s) not taking an active role in the case investigated" [Woh21].*

In our research, we conducted (1) empirical investigations including a comprehensive analysis of the cases, considering (2) multiple data sources from the software system, including source code, documentation, historical data, monitoring data from benchmarks, and expert feedback.

Besides, we can consider our phenomenon of "*consistency preservation*" as a (3) contemporary phenomenon because we monitor this phenomenon throughout its contemporary execution during the development and operational phases of our cases. While we acknowledge that waiting for commits in Git history requires a longitudinal case study, we address this by concurrently observing the consistency preservation phenomenon through experiments that simulate the development process based on *real* Git history. That is because the Git history documents all the required information to simulate the development process. We also recognize that case studies should involve observing real-life usage rather than relying on synthetic workloads such as benchmarks [Has21]. However, our phenomenon preserves consistency during benchmarking or performance testing at Dev-time. Moreover, it is important to note that the older definition of case studies did not explicitly mention contemporaneity but focused on the phenomenon itself [BGM87].

In terms of the (4) real-life context, we study cases that are developed in real-life within CI-context. To achieve this, we rely on real Git history, real performance tests, and genuine monitoring data. This methodology includes industrial case studies conducted in industrial settings and open-source cases developed for performance engineering, as they all closely reflect CI-based software development practices.

Lastly, it is essential to highlight that during our investigation, we did not actively participate in the software development process itself (5). Instead, our role primarily

involves observing the phenomenon or designing experiments that aim to isolate specific aspects of reality to observe certain elements within the genuine context.

In Section 8.3.1, we describe the process of our case study. After that, we introduce our cases: TeaStore in Section 8.3.3, CoCoME in Section 8.3.4, TEAMMATES in Section 8.3.5 and industrial Lua-based cases in Section 8.3.6. Afterwards, we describe in Section 8.3.7 some artificial examples that also are considered as cases for the evaluation.

8.3.1. Case Study Process

In this section, we outline the process of our case study, which follows these five key phases: case study design, preparation for data collection, collecting evidence, analysis of collected data, and reporting, as detailed in Section 2.7.3.

8.3.1.1. Case Study Design

In DevOps practices, performance models can become outdated due to changes at Dev-time and Ops-time. This case study evaluates CIPM's effectiveness in keeping them (aPMs) up to date during DevOps activities. It also addresses the challenge of applying CIPM.

The study's objectives are defined through a GQM plan Section 8.1, focusing on three key aspects: G1 assessing the accuracy of performance models and their predictions, G2 analyzing the impact of monitoring on system performance, and G3 evaluating how well the approach scales to handle more changes.

The case study aims to answer the following main Evaluation Question (EQ): "How accurately can CIPM update performance models within the CI/CD framework, and what is the required monitoring overhead and execution time? ".

The data considered for this study comprises Git history (source code changes), performance tests, monitoring data, and relevant documentation. Due to time constraints, the study observes specific parts of the development and operation phases. Git history is instrumented, and operations are observed for specific durations using data from benchmarking.

To select Git history portions for analysis, we mainly focus on sections with significant source code changes. Available performance tests are chosen for examination. In terms of monitoring data, we employ the `Kieker` monitoring framework introduced in Section 2.1.3 for instrumentation and data collection. The documentation includes the results from evaluating the extracted aPM.

To manage costs, we utilize simulated users available in real performance tests for case operations, since the proposed incremental calibration can be used during performance tests or the operation in production environments.

8.3.1.2. Collecting Evidence

For collecting evidence, we execute experiments that apply CIPM during simulated development and benchmarking of the cases. By the experiments, we can control the setting of CIPM to achieve the objectives of our case study, such as observing the monitoring overhead in a worst-case scenario.

8.3.1.3. Analysis of Collected Data

The collected data (e.g., commits, measurements and aPM) is analyzed to assess the accuracy, monitoring overhead, and scalability. Data analysis is conducted across multiple cases. The results are individually analyzed to identify any underlying relationships. Additionally, the simulations and experiments are repeated to enhance the reliability of results and get more insights. Threats to validity are discussed after each experiment to address potential biases. . In Chapter 9 and Chapter 11, the data analysis during the conducted experiments is illustrated in more detail.

8.3.1.4. Reporting

Study results are published through research papers [Maz+25; Maz+20; Von+20; Mon+21; AMK23; MAK23; AMK24], students' work or presentations. Target audiences include researchers, the academic community, and the industrial sector. Table 8.8 provides details on research papers that are directly related to CIPM.

In addition to this thesis, relevant wiki pages[1], Git collaboration[2], and Zenodo repositories[MAM24; Maz+24] provide the necessary resources for case study design, the CIPM tool, replication package, and related materials.

8.3.2. Case Selection

The selection of cases should consider (a) realistic contexts, (b) applicability, and (c) the objective of our study.

Regarding the realistic contexts (a), ideal cases should have real-world applicability, real Git history, and available performance tests. Additionally, the cases should be relevant to realistic usage scenarios, industry, and performance benchmarking.

For the applicability, we require cases written in a programming language, whose parser and printer are available and compatible with EMF models that CIPM supports. Therefore, our selection was restricted to Java-based and Lua-based cases. Additionally, the selected cases should enable achieving the (c) objective of the study. To clarify, assessing **C1** requires a well-established Git history and proficiency in diverse languages

[1] `https://sdq.kastel.kit.edu/wiki/CIPM`
[2] `https://github.com/CIPM-tools`

Table 8.8: The case studies used for validating the contributions.

Cont.	Contribution	Case	Publication
C1	**CI-based update of software models**	**TeaStore**, Teammate & 2 sensor applications	[Maz+25; Mon+21; AMK23]
C2	**Adaptive instrumentation**	**TeaStore**, Teammate	[Maz+25; Maz+20]
C3	**Incremental calibration process**	CoCoME, **TeaStore**, artificaial example	[Maz+20]
C4	**Ops-Time calibration**	CoCoME,**TeaStore**	[Mon+21]
C5	**Self-validation**	CoCoME, **TeaStore**	[Mon+21]
C6	**MbDevOps pipeline**	non-experimental	[Maz+25]

and technologies. Consequently, we have selected specific cases, namely TeaStore, TEAMMATES, and the industrial Lua-based cases.

For a more comprehensive analysis of incremental calibration in **C3**, open-source cases including performance tests, aPMs representing the behavioral perspective, and various PMPs are required for the evaluation. Therefore, we utilized TeaStore and CoCoME, as they were employed well in other works in benchmarking and architecture-based performance prediction approaches [Hei+15; Hei+21; Kis+18; Kei+21; Kei+20].

In the context of Ops-time calibration (**C4** and **C5**), an optimal scenario involves a microservices-based application with Docker, facilitating effective manipulation of changes in Ops-time. Therefore, TeaStore is used in more experiments than CoCoME. All mentioned cases are utilized to evaluate different parts of the MbDevOps pipeline, subsequently substantiating its usability.

Table 8.8 arranges the contributions to the cases used for their evaluation.

In the following sections, we provide more detail on the cases:

8.3.3. TeaStore

The TeaStore [Kis+18] case is introduced in Chapter 3. In this section, we describe the architecture of TeaStore in more detail. Additionally, we present the TeaStore PCMs used for the evaluation.

As previously explained, TeaStore is a microservice-based open-source test application. Figure 8.8 shows the microservice-based architecture of TeaStore. TeaStore includes six Microservices. Five of them have the functionality of the TeaStore, and the Registry microservice includes the load balancer. The leading five microservices communicate with the Registry microservice. Moreover, the dashed communication lines show that both the `ImageProvider` and `Recommender` communicate with Persistence at the beginning to provide the required images for the interface and to train the recommender according to the order history data.

As mentioned in Section 8.3.2, we select TeaStore based on several factors: (a) its realistic context, including real Git history and real benchmarking, allowing us to observe the consistency preservation of CIPM; (b) its applicability, as it is implemented in Java, ensuring compatibility with EMF models supported by CIPM, and its suitability for evaluation of incremental calibration (**C3**) and Ops-time calibration (**C4**); and (c)

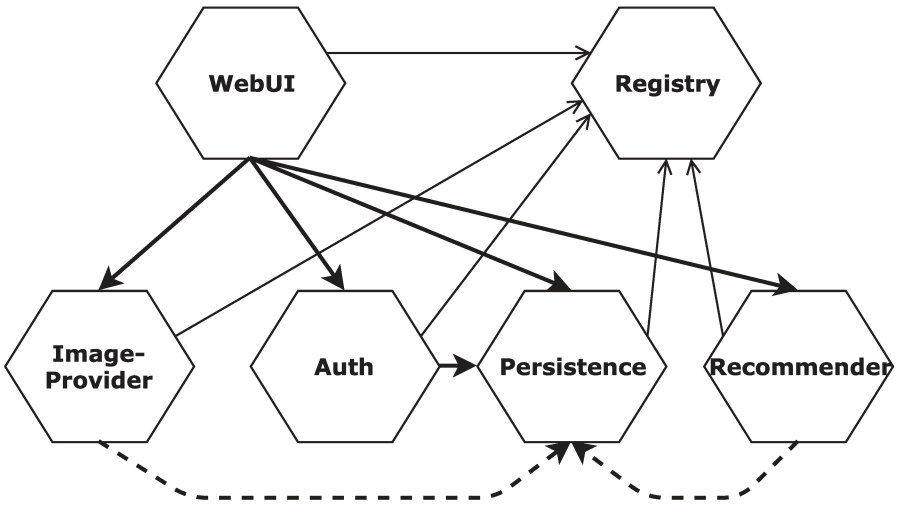


Figure 8.8: Microservice-based architecture of the TeaStore [Kis+18].

its alignment with the study's objectives, offering benchmarking, and a microservices-based architecture with Docker that facilitates effective manipulation of Ops-time changes for evaluating **C4** and **C5**. Additionally, TeaStore has been widely used in benchmarking and performance prediction studies, making it suitable for studying its architecture, manually modeling it with `Palladio` at different abstraction levels for various evaluation objectives, and using the manually modeled aPMs to validate AbPP and different parts of the MbDevOps pipeline pipeline, including evaluating the automatically extracted one.

For our evaluation, we use two architectural views of TeaStore. We model these views according to PCM viewpoint, cf. Section 2.4. The first view expresses the main logic of TeaStore without technical detail, cf., Section 8.3.3.1. The second view describes the internal behavior of `Recommender` in more detail, cf., Section 8.3.3.2. The concern of the second one is to analyze the performance of the recommendation process. Additionally, CIPM extracts a PCM representing the logical view and the architecture expressing the used technology.

In the following subsections (Section 8.3.3.1, Section 8.3.3.2), we describe the manually modeled PCMs that we use as references for our evaluation. The automatically extracted PCM will be described as a part of the experiments' results, cf. Section 9.1.6.

8.3.3.1. TeaStore PCM according to Logical View

In this section, we describe an architectural view expressing the functionality that TeaStore provides to the end user. We modeled this view utilizing PCM [Mon20]. This PCM represents the logic view that is implemented within the microservices as back-end components. Additionally, this PCM ignored technical details, like the structure of

communication between these microservices. Thus, communication between microservices is modeled with direct communication through external calls, omitting the typical indirect communication through REST interfaces.

Goal: The logical PCM of TeaStore is employed for evaluation purposes encompassing accuracy (**G1**), monitoring overhead (**G2**), and scalability (**G3**). Concerning (**G1**), experiments (Section 9.1,Section 9.2) utilize this PCM to evaluate the accuracy of the updated PCM at Dev-time. Similarly, another experiment (Section 11.1) ensures that the PCM accurately reflects simulated changes during Ops-time. To achieve this, we explicitly model the recommendation strategies in PCM to evaluate how CIPM reflects these changes in `System Model` when we simulate alterations in system composition at Ops-time.

Structure: As explained in Section 3.1, TeaStore includes six microservices. As Figure 8.9 shows, we model each microservice as a simple component: `Registry` component, `ImageProvider` component, `Auth` component, `Persistence` component, `Recommender` component, and `WebUI` component.

Different strategies are used to train the recommender and advertise products. In this model, we explicitly model the `RecommenderStrategy` interface and its various implementations. This causes a change in system composition after changing the recommendation strategy (change in the type of `RecommenderStrategy` instance), which allows us to evaluate the accuracy of updated `System Model` after structural changes in system composition at Ops-time. Hence, the following components model the recommendation strategies: `SlopeOneRecommender` component, `OrderBasedRecommender` component, `PopularityBasedRecommender` component, `PreprocessedSlopeOneRecommender` component and `DummyneRecommender` component. The running instance of the `Recommender` can be changed at Ops-time.

Abstract Behavior: During the period of the dissertation, `Palladio` Simulator lacked comprehensive support for modeling and simulating microservices-based applications, particularly regarding intra-communication via alternative technologies. As a workaround, we represented the communication between microservices by explicitly modeling external calls, without including intermediary technology in the model. Additionally, we explicitly model the `LoadBalancer` interface that includes signatures for the communication calls. `Registry` provides an instance of the called microservice, therefore, `Registry` provides the `LoadBalancer` interface in our model. The abstract behavior of the `LoadBalancer` services delegates the call to one of the existing instances of the destination microservice. For that, the modeled `SEFFs` use branches with nested transitions, including an external call to an instance of the destination microservice. The goal of branches is to distribute the call on the existing instances, similar to the

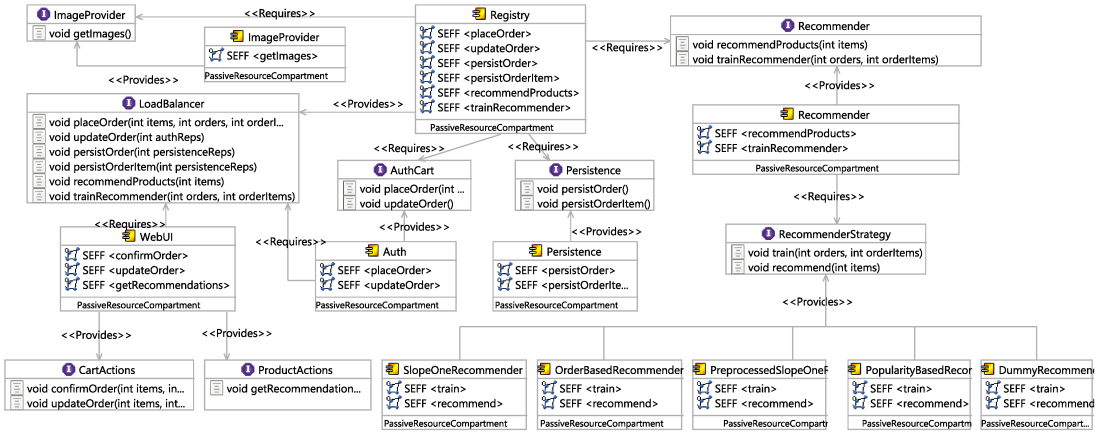


Figure 8.9: The manually modeled `Repository Model` of TeaStore.

load balancer. The branch condition of the outer branch determines the number of existing instances. The inner branches randomly select one of the transitions, including calls to existing instances. For instance, the service "placeOrder" of `Auth` is included in nested branches that approximately simulate the function of the load balancer.

The evaluation concentrates on `"ConfirmOrder"` service that the `WebUI` microservice provides. The complex functionality of this service and the inclusion of external calls to other services make it suitable for evaluating performance. The `"ConfirmOrder"` is called after the customer places his order in the shopping cart and confirms it. It then calls functionalities from other microservices (except the `ImageProvider` and the Recommender) to complete the buying process. To increase the complexity of this service and to evaluate the performance of recommender strategies, we call the train service of the `Recommender` after processing each order by `"ConfirmOrder"`. As a result, functionalities of all components except the `ImageProvider` are called in `"ConfirmOrder"`. Consequentially, performance analyzing of `"ConfirmOrder"` covers all components (except `ImageProvider`) involved in the `"ConfirmOrder"` service, cf. the modeled SEFF in Figure 8.10. To simplify the presentation of SEFF, we hide the modeling of the remaining transitions of the Branch in `Registry.placeOrder`. Moreover, we do not represent the SEFF of `Registry.persistOrder` and `Registry.persistOrderItems` since they include just an internal action.

It should be noticed that the execution time of `"ConfirmOrder"` is increasing at Ops-time. The reason is that we call the `train` service, whose execution time depends on the ordered items that are stored continuously in the database.

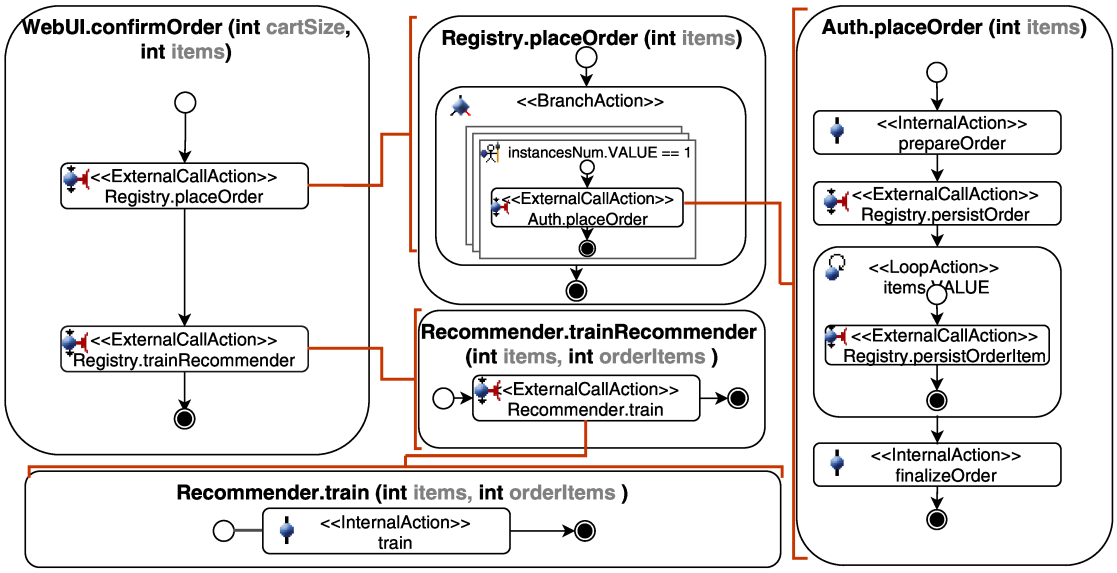


Figure 8.10: The abstract behavior of `ConfirmOrder` service using `Palladio` annotation (SEFF diagram).

8.3.3.2. TeaStore PCM according to behavioral View

We describe in this section an architectural view expressing the internal behavior of the `Recommender`. We modeled this detailed view for evaluating the incremental calibration process [Maz+20]. The following paragraphs provide more details about the goal of this view, its structure, and the detailed behavior of `Recommender`.

Goal: The goal of this view is to evaluate the performance of the recommender in more detail. Such a view allows the architect to perform bottleneck analysis. In our Validation, this view allows us to evaluate the incremental calibration of different PMPs with parametric dependencies during the incremental implementation of various recommender strategies.

Structure: The main difference between this structure and the structure of overall TeaStore PCM is that we model the `Recommender` microservice fine-granular using different basic Components like `Recommender`, `Trainer` and `Util`. The goal is to use this view for evaluating the performance of the `Recommender`. Therefore, it models the `Recommender` microservice fine-granular.

Abstract Behavior: The fine-grained modeling of `Recommender` provides more complicated abstract behavior (SEFF). This allows the investigation of the parametric dependencies for different scenarios. Figure 8.11 shows the abstract behavior of the `train`

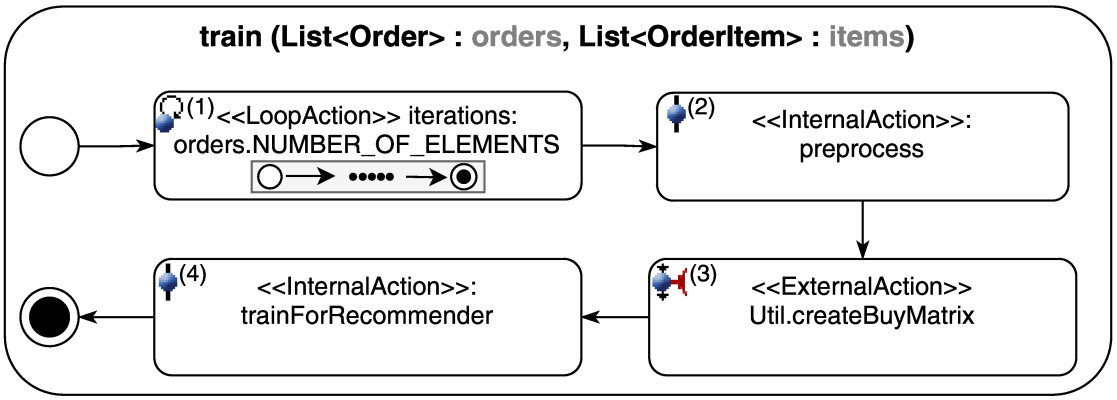


Figure 8.11: The abstract behavior of `train` service based on [Maz+20].

service. The SEFF begins with a loop that iterates according to the characterization of the `items` parameter, i.e., the number of elements in the list. The loop is followed by an internal action for preprocessing the items. Then an external call to `Util` for creating a required Matrix for training. Finally, the training process starts. This process is differentiated according to the recommendation strategy. This means that just the last internal action of `train (trainForRecommender)` is changed by implementing the recommendation strategies.

8.3.4. CoCoME

Common Component Modeling Example (CoCoME) [Her+08; Hei+15] 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.

Technology: In our evaluation, we used the cloud variant of CoCoME[3], where the enterprise server and the database are running in the cloud. This version of CoCoME is based on Java Enterprise Edition (Java EE) and uses Maven for managing sub-projects and configuring the deployment.

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

[3] see `https://github.com/CIPM-tools/cocome-cloud-jee-platform-migration`

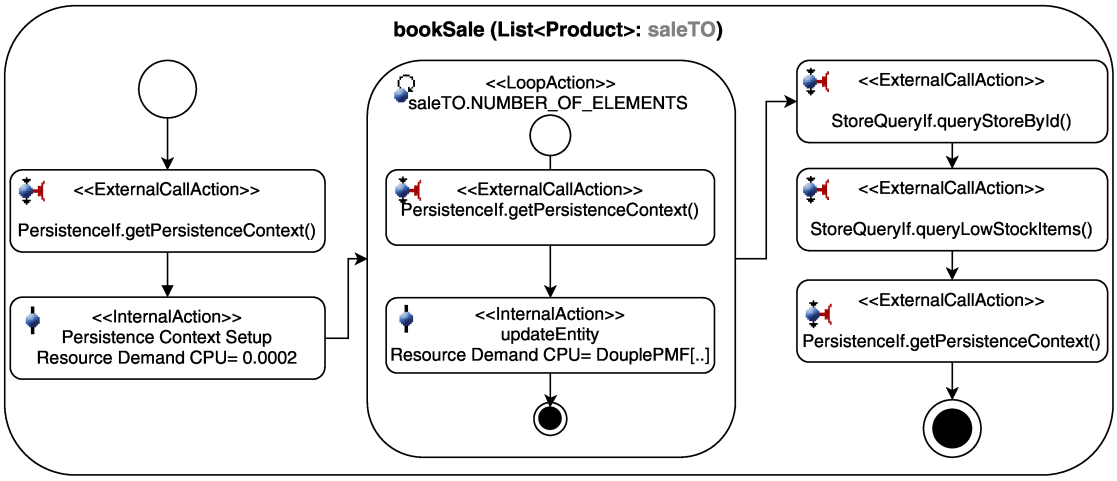


Figure 8.12: The abstract behavior of bookSale service.

evaluating the accuracy of the performance prediction (**G1**) and the required monitoring overhead (**G2**).

CoCoME is used for evaluating several approaches like iObserve [Hei20] and the proposed platform for empirical research on information system evolution [Hei+15]. 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: The CoCoME structure 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 of PCM is modeled by `Palladio` [Her+08; Hei+15]. The `Application.Store` component of the `Application` layer provides a `StoreIf` interface for storing the sales.

Abstract Behavior: In this paragraph, we describe the behavior of the main service of the `StoreIf` interface, which is called `bookSale` service. The `bookSale` service is responsible for processing a sale after a user submits his payment. As input, this service receives a list of the purchased products, including all information related to the purchase. The service consists of several internal and external actions and two loops. Figure 8.12 visualizes an excerpt of the abstract behavior of the `bookSale` service.

8.3.5. TEAMMATES

TEAMMATES is a tool for confidentially managing students' peer evaluations, instructor comments and feedback. For this goal, hundreds of universities worldwide

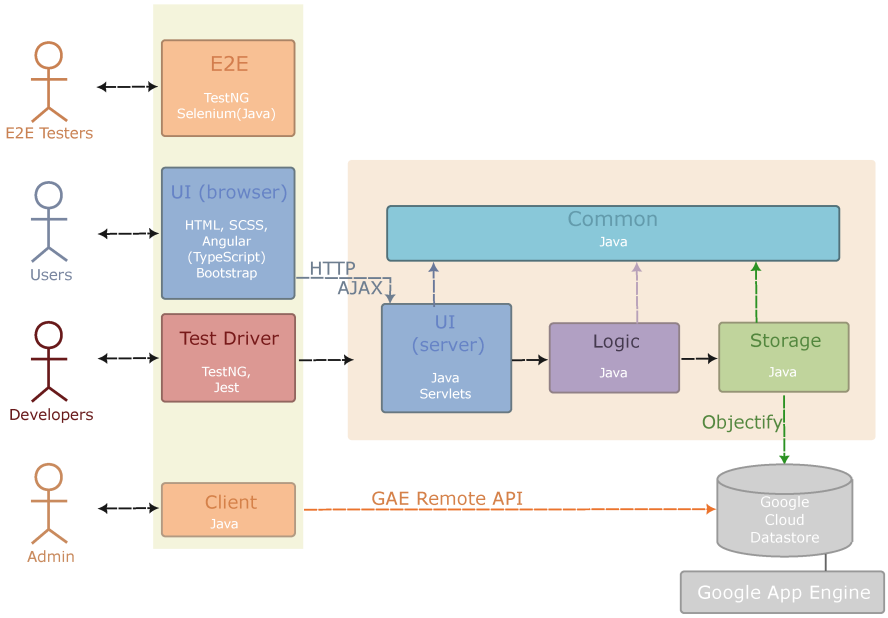


Figure 8.13: High Level Architecture of TEAMMATES [Kur24].

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 in our evaluation is to evaluate CIPM with a real tool and real Git history. Significantly, evaluating the CI-based update of aPMs requires an existing real Git history. TEAMMATES is used for evaluating the accuracy of updated models (**G1**) and the reduced instrumentation probes (**G2**). The TEAMMATES tool consists of two parts: a web-based frontend and a Java-based backend. Our evaluation covers the backend part over a Git history of 17859 real commits and changes related to 1428 files.

Structure and Technology According to the well-documented architecture [Kur24], the backend part of the TEAMMATES tool consists of the following four main components that are shown in Figure 8.13. First, `UI` represents the entry point for the application backend. `UI` is based on the RESTful controller, cf. paragraph RESTful API. It ensures the separation between the 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. For that, it uses Google Cloud Datastore. The last component `Common` includes the required common utilities.

8.3.6. Sick Sensor Applications

In this case study, we examine the applicability of CIPM approach in the context of two industrial applications from SICK [4].

Goal: The goal of using Lua-based sensor applications as a case study 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 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 the design decision through AbPP. In the following, we introduce some of these challenges:

- **Limited Sensor Node Capabilities:** Sensor nodes typically have limited processing and communication capabilities compared to commodity hardware. Therefore, optimizing and adapting sensor applications to be more performant is required to work efficiently within these limitations. For that, AbPP can assess the alternatives.

- **Relocation of Data Processing:** With the potential relocation of sensor data processing to the cloud, there is a need to evaluate how this change affects the extra-functional aspects of the application. AbPP can help in making informed decisions regarding this relocation.

- **Heterogeneous Sensor Hardware:** The heterogeneous nature of deployed sensor hardware can complicate the development and optimization of sensor applications. Since aPM represents the system structure, it allows assessing the deployment by AbPP at a low cost compared to monitoring.

- **Pre-Filtering on Sensor Nodes:** Implementing pre-filtering of sensor output on less capable sensor nodes can be beneficial, but it also poses challenges. AbPP can assist in optimizing the decision-making for pre-filtering.

- **Performance Expectations:** When using more affordable hardware for computing nodes in the network, it is crucial to determine the expected performance. aPMs can aid in predicting and managing the performance of sensor applications while considering cost-effectiveness.

[4] See `https://www.sick.com/`

Technology: The sensors by SICK, both programmable and non-programmable, can be effectively utilized to create customized solutions through Lua applications, known as `SensorApps`. SICK AppSpace, a commercial sensor application ecosystem [AG24c], enables the development of individualized `SensorApps`. These `SensorApps` are specific software applications that are developed to work with sensors. Even non-programmable sensors find a smooth integration path into the system by employing Sensor Integration Machines (SIMs) and various network protocols.

The `AppEngine` executes these `SensorApps` and operates them on programmable sensors and SIM [AG24b; AG24d]. Beyond execution, the `AppEngine` plays an essential role by providing 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 `AppEngine`. These CROWN 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.

Barcode Reader Application The BarcodeReaderApp that we use in our case study is developed from sample apps. It consists of 7 commits, involving a total of 862 lines added and 283 lines removed across one to four files.

The BarcodeReaderApp example was developed by combining existing AppSpace samples regarding the component-based architecture, and it serves as a realistic use case for the industrial Internet of Things (IoT) setting [Bur23].

The application involves capturing images using a camera module, detecting barcodes in the images, and sending the barcode information to a database via HTTP. The application is composed of four AppSpace samples from SICK, including a barcode scanner app, an HTTP client app, an HTTP server app, and a database app. 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 the 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 detail on the example is on [Bur23].

Object Classifier Application The ObjectClassifierApp under investigation represents a real-world use case involving the detection and sorting of objects from images based on color recognition. The considered developmental history spans 12 commits and encompasses a significant codebase adjustment, with 6651 lines added and 2663 lines removed across one to thirteen files. Unlike the BarcodeReaderApp, this real-world scenario offers a more complex and dynamic testing ground for CIPM. The tasks

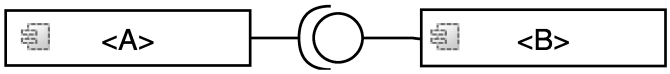


Figure 8.14: The simple structure of example1.

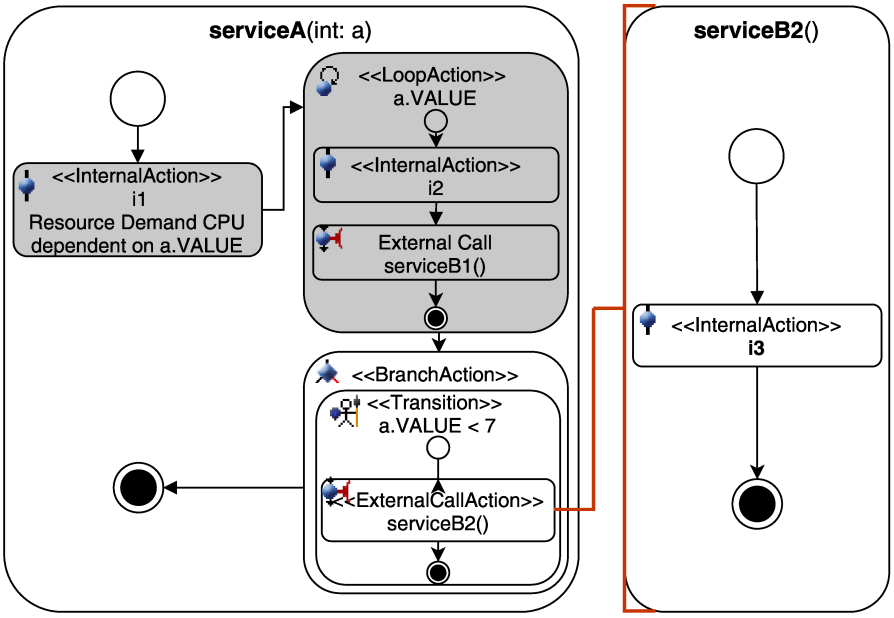


Figure 8.15: The behavior of `service1` using `Palladio` SEFF diagram.

performed by this application involve the identification and categorization of objects within images based on their color properties. More information about this real-world application can be found in [Bur23].

8.3.7. Artificial Examples

The goal of using artificial examples is to validate the approach with more cases that are not covered by the case studies. In this subsection, we introduce these examples briefly to be able to present the evaluation results that are based on these examples.

Example1: In this paragraph, we briefly introduce an example that Jägers designed during his masterwork [Jäg19]. This example includes different types of PMPs. We adjusted this example and used it to evaluate the incremental calibration. The structure of the example is simple. It consists of two components `A` and `B`, cf., Figure 8.14. Component `A` provides a `serviceA`, whereas component B provides two services: `serviceB1` and `serviceB2`. Figure 8.15 shows an abstract behavior of `serviceA` using SEFF.

Example 2: Example2 is an extension of Example1 and includes two components, A and B. However, unlike Example1, Example2 exhibits a more complex behavior. `serviceA` method of component `A` calls three services from component `B` within a nested structure that involves branching and looping. The arguments of the external service calls, branch transitions and the number of loop iterations depend on the arguments of `serviceA`. The dependencies between the arguments of service1 and the calls to component 2 are in different forms, including linear, non-linear, polynomial, logical, constant, and transcendental. More detail on the abstract behavior is available on GitHub[5] and documented in [Von+20, p. 33].

8.4. CIPM Tool

CIPM Tool [MAM24] is implemented in Java and uses the Vitruvius Platform. The tool considers the `Palladio` Component Model for performance models and the `JaMoPP Model` and `Lua Model` for source code models. The tool supports CPRs for three technologies: Microservices, Sensor Applications, and Plug-in based component applications. The current implementation of CIPM tools is compatible with applications stored and managed on Git. As a result, the tool is presently limited to Java- and Lua-based applications on Git utilizing the aforementioned technologies. However, the design of the tool is flexible, enabling developers to extend its capabilities and overcome limitations by integrating additional parsers and implementing more CPRs, as we discuss in Section 13.1.

The current implementation utilizes the `Kieker` monitoring tool for benchmarking, which facilitates trace reconstruction and processing monitoring data via custom output streams writer.

Besides, it should be noted that measurements are currently stored and processed without utilizing Vitruvius due to technical limitations. Specifically, the implementation of version control within Vitruvius [Ana22] has not been integrated into the platform. Given this limitation, and to avoid redundant development efforts, the Ops-time calibration is implemented independently of Vitruvius. In future work (Section 13.1.2), we can extend our implementation to encompass all the consistency preservation processes within Vitruvius platform.

The current tool supports the adaptive instrumentation of Java applications. However, it does not support Lua-based applications. In ongoing master's research, we are investigating the adaptive instrumentation and calibration of Lua applications.

[5] see `https://github.com/CIPM-tools/OptimizingParametricDependencies/tree/master/tests/tools.vitruv.applications.pcmjava.modelrefinement.parameters.tests/test-data/casestudy2`

Detailed information about technical aspects, tool features, and the source code can be found on GitHub [6] and is complemented by documentation and Information on reproducibility [7].

8.5. Experiments

As aforementioned in Section 8.3, we conducted experiments alongside the case study to expand our validation. We validate the CIPM approach based on five Experiments (E) that focus on validating three key aspects of our approach: the consistency preservation at Dev-time in Chapter 9, the incremental calibration in Chapter 10, and consistency preservation at Ops-time in Chapter 11. In Table 8.9, we provide a summary of these experiments along with the corresponding contribution that they validate.

Table 8.9: Summary of experiments and validation.

Validation	Experiment	Contribution
Consistency Preservation at Dev-time Chapter 9	Experiment1 (**E1**) Experiment2 (**E2**)	**C1** **C2**
Incremental Calibration Chapter 10	Experiment3 (**E3**)	**C3**
Consistency Preservation at Ops-time Chapter 11	Experiment4 (**E4**) Experiment5 (**E5**)	**C4** **C5** **C6**

The following chapters (Chapter 9, Chapter 10 and Chapter 11) describe the performed experiments. We structured our experiments into clearly defined sections that outline the objectives, methodology, results, threats to validity, and discussion for each experiment. This organization ensures clarity and thoroughness in our approach to experimental design and analysis.

[6] See `https://github.com/CIPM-tools`

[7] See `https://sdq.kastel.kit.edu/wiki/CIPM` and `https://github.com/CIPM-tools/CIPM-Pipeline/wiki`.

9. Validation of Consistency Preservation at Development Time

In this chapter, we validate the consistency preservation of the architectural performance models at Dev-time. For that, we conduct two experiments that validate the first two contributions of this thesis. The first one validates the CI-based update of aPM and the adaptive instrumentation (Section 9.1), while the second one concentrates on validating an aspect of the first contribution, i.e., the update process of the `System Model` (Section 9.2). Section 9.3 sums up the results of the experiments above.

9.1. Experiment1: CI-based Update of the Performance Model

Experiment1 (**E1**) is designed to validate the first and second contributions of this dissertation: CI-based consistency preservation at Dev-time and adaptive instrumentation during the consistency preservation of aPM at Dev-time. The goal of **E1** is described in Section 9.1.1. The scenario, in which the experiment takes place, is described in the Section 9.1.2. The experiment is executed using two cases: TeaStore and TEAMMATES. Additional two cases (Lua-based sensor applications) are considered for the first part of E1, i.e., evaluating the CI-based consistency preservation at Dev-time. **E1** on TeaStore and Lua-based sensor applications was conducted during the masterwork of Armbruster and Burgey, while **E1** on TEAMMATES and part of TeaStore was conducted for the publication referenced by [Maz+25]. In this thesis, there is some additional analysis related to **E1** on previous cases, including scalability considerations at Dev-time.

The setup of **E1** on TeaStore is described in Section 9.1.3 while Section 9.1.4 explains the setup of **E1** on TEAMMATES. The setup of Lua-based sensor applications follows in Section 9.1.5. The results obtained from the experiment are presented and analyzed in Section 9.1.6, followed by a discussion of their implications in Section 9.1.7. The limitations encountered during the experiment are addressed in Section 9.1.8 and potential threats to the validity of the experiment are discussed in Section 9.1.9.

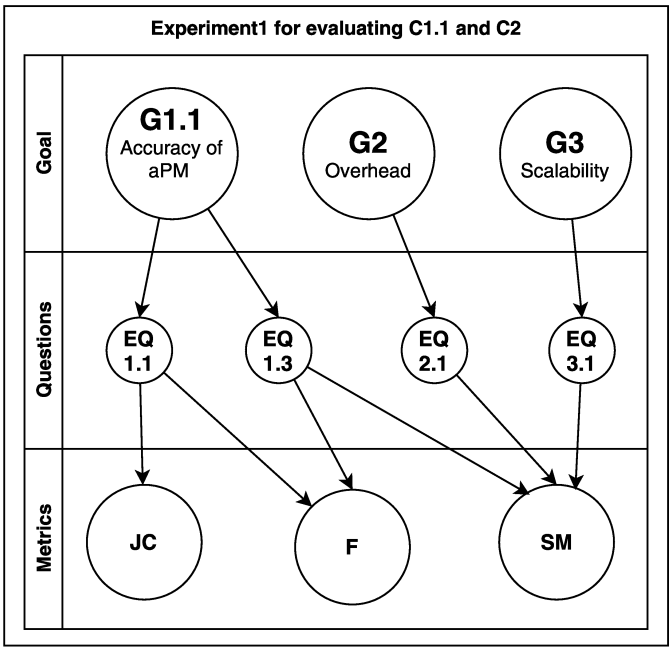


Figure 9.1: GQM plan of Experiment1.

9.1.1. Goal

The experiment **C1** validates the first two contributions of this dissertation: CI-based consistency preservation at Dev-time (**C1**) and adaptive instrumentation (**C2**), cf. Figure 9.1. **E1** addresses the following evaluation goals:

G1 The goal of **E1** is to evaluate the accuracy of the VSUM models that are updated by **C1**. Additionally, it evaluates the accuracy of the instrumented code that is generated by **C2**. For this goal, we perform **E1** to answer the related evaluation questions:
EQ-1.1: *How accurately does CIPM update the* `Repository Model` *and its related models within VSUM according to a Git commit?*
EQ-1.1.2: *How accurately does CIPM update the VSUM models when it aggregates multiple commits and propagates them as a single commit?*
EQ-1.3: *How accurately does the adaptive instrumentation instrument the source code based on collected instrumentation probes in IM?*

E1 excludes the accuracy of the updated `System Model`, since **E2** evaluates it extensively.

G2 **E1** validates the reduction of instrumentation probes that the adaptive instrumentation can apply.

To achieve this goal, **E1** addresses **EQ-2.1**: *To what extent can the adaptive instrumentation reduce the instrumentation probes?.*

G3 **E1** aims to outline the trends in scalability factors at Dev-time through answering **EQ-3.1**: *How does the CI-based update of VSUM scale with the amount of source code changes?.*

9.1.2. Scenario

We use real Git histories to update the `Repository Model` and generate the related instrumented source code, see Figure 9.2. As declared in Section 9.1.1, we evaluate then the accuracy of the resulting models and the reduction of monitoring overhead.

To ease the evaluation, we divide the Git history into intervals and update the software models based on the commits of the intervals. The commits of TeaStore and Lua-based sensor applications intervals are propagated sequentially, whereas the commits of TEAMMATES intervals are aggregated and propagated as four commits. Each commit aggregates the commits of a specific interval.

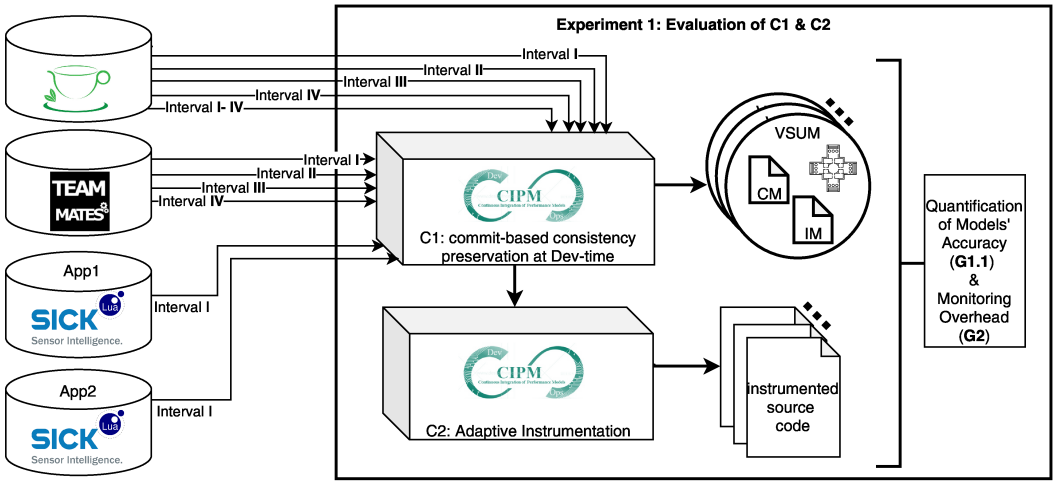


Figure 9.2: The scenario of Experiment1.

We select the initial commit in the first step and send it to the CI-based consistency preservation at Dev-time. As a result, the VSUM is updated. The VSUM of the current implementation includes three models: `SCM`, `Repository Model (RepM)` and `IM`. We evaluate the VSUM models as follows:

`SCM` We parsed the state of Git after each commit to obtain reference models that should be identical to the automatically updated `SCM` within the VSUM. We used JC to validate the accuracy of updated `SCM` as explained in Section 8.1.3.

`RepM` We validate the `Repository Models` resulting from initial commits by comparison with manually modeled ones. For that, we use JC and $F1$ score as we explained in Section 8.1.3.1.

`IM` For validating the accuracy of `IM`, we used $F1$ score to ensure that `IM` includes the required instrumentation probes that correspond to the updated parts in `Repository Model`, see Section 8.1.3.2 for more detail.

In the next step, we start the adaptive instrumentation to generate the instrumented code. Then, we validate the accuracy of the instrumented source code using $F1$ score and ensure that the instrumented source code includes no compilation errors.

Similar to the initial commit, we repeat the previous steps to update the VSUM with new commits. Each time, we validate the accuracy of VSUM models in addition to the instrumented source code, as we explained in the last two paragraphs.

We measure the execution time of each step of **E1**, i.e., the update time and the adaptive instrumentation time.

We apply **E1** on TeaStore, TEAMMATES, and Lua-based sensor applications in our evaluation. The following sections clarify the setup of this experiment in more detail.

9.1.3. Setup on TeaStore

First, we introduce Git's history covered by our evaluation and its impact on software architecture, see Section 9.1.3.1. Then we describe the actual steps for applying **E1** on this history, see Section 9.1.3.2.

9.1.3.1. TeaStore Git

We consider 181 commits of the Git history of TeaStore. The commits are between version 1.1 and version 1.3.1 of TeaStore. These commits can be split into four intervals according to the releases within the considered time period, cf. Table 9.1.

The intervals are:

I Interval I has 50 commits. The commits are between version 1.1 and version 1.2. Just 27 commits are related to Java files and affect them by adding 9553 lines and removing 7908.

II Interval II includes the commits between version 1.2 and version 1.2.1. This interval consists of 20 commits. 12 of them affect 5 Java files with 141 added lines and one removed line. Moreover, three Java files, including 123 lines in total, were added in the interval II.

III Interval III consists of eleven commits between version 1.2.1 and version 1.3. Seven commits from interval III adjust four Java files by adding 121 lines and removing 134.

Interval	Versions	Commits	Java-related Commits	Affected Java Files	Added Lines	Removed Lines	Architectural changes
I	[1.1-1.2]	50	27	144	9553	7908	5
II	[1.2-1.2.1]	20	12	8	264	1	4
III	[1.2.1-1.3]	11	7	4	121	134	0
VI	[1.3-1.3.1]	100	12	9	215	227	0

Table 9.1: TeaStore commits that are considered in the evaluation.

IV Interval IV includes 100 commits. Twelve of them are related to Java files. In overall, these 12 commits added 215 lines and removed 227.

Architectural changes: The most interesting commits are the commits impacting the `Repository Model` and causing architectural changes. In the following, we highlight the commits that impact the software architecture and describe how the `Repository Model` should change after these commits.

I The initial commit within the interval I causes significant architectural changes because there is no pre-existing architecture to update; instead, the software source code of the initial commit is reverse-engineered to extract the architecture. The IRE, as we refer to it, analyses the source code changes in 236 java files and extracts the `Repository Model` based on the predefined CPRs. As a result, we obtain a `Repository Model` representing version 1.1 of TeaStore. Successive commits in the interval (I) include five architectural-relevant changes:

I.A In the third commit of interval I, one of the servlet interfaces in the `Auth` microservice is removed.

I.B Another change in the third commit is that two classes of the Auth microservice were renamed. However, the content of the file containing these classes was also modified to be compatible with the Checkstyle configuration. This affects the matching process, which interprets the renaming as a removal of the old classes and an addition of new classes.

I.C In the tenth commit, the implementation of a service that is related to SEFF was changed by adding an if condition clause.

I.D In the eleventh commit, the implementation of another method corresponding to a SEFF was also modified. The modification deleted a statement and a loop. Additionally, the condition of an if-statement was adjusted to check a password.

I.E In the twenty-eighth commit, the visibility of two methods that are related to SEFFs was changed from public to private. According to our CPRs, the

change in the visibility of these methods was interpreted as a removal of their related SEFFs.

II Similar to interval I, propagating the first commit of interval II extracts the `Repository Model` of version 1.2 incrementally. The successive commits affect the `Repository Model` five times:

II.A In the tenth commit, an implementation of a SEFF was changed.

II.B In the eleventh commit, the change in the SEFF implementation that occurred in the tenth commit was reverted.

II.C In the thirteenth commit, a new servlet interface was added to provide control and access to log files.

II.D In the seventeenth commit, a new REST endpoint representing an interface according to our CPRs was added in `Auth` microservice to obtain readiness.

II.E In the eighteenth commit, a new REST endpoint was added in `WebUI` microservice. The added REST endpoint was identical to the one that was added in the seventeenth commit in `Auth` microservice.

III Just propagating the first commit of interval III causes changes in `Repository Model`. In other words, an IRE of version 1.2.1 is applied by integrating the first commit of interval III. Successive commits in interval III do not affect the extracted `Repository Model`.

IV Similar to interval III, integrating the first commit extracts the `Repository Model` by IRE. Successive commits in interval IV do not affect the extracted `Repository Model`.

9.1.3.2. Applying Experiment1 on TeaStore

We applied **E1** on commits of TeaStore Git and repeated it in different ways to evaluate various aspects:

Evaluating the accuracy of updated models In the first part of applying **E1** on TeaStore (**E1.1**), we started with the initial commit of interval I and updated the VSUMs models using all commits of the interval I. We repeated **E1** for the remaining intervals. Then, we executed the adaptive instrumentation after each update process.

For the evaluation of the updated VSUMs and answering **EQ-1.1**, we followed the following strategies:

`SCM'` We parsed the state of Git after each commit to obtain reference `SCM'` that should be identical to the automatically updated `SCM` within the VSUM. For that, we used

JaMoPP to parse the code state after each commit. Then, we used JC to quantify the accuracy as we explained in Section 8.1.3.1. The results are in the second column of Table 9.4.

RepM' First, we integrated version 1.3.1. The IRE resulted in the reference Repository Model ($M'_{1.3.1}$). To answer **EQ-1.1.1**, we validated the accuracy of the automatically extracted reference model by the comparison to the manually modeled Repository Model that is described in Section 8.3.3.1.

In the second step, we manually adjusted $M'_{1.3.1}$ to obtain reference models for all commits. For that, we used the architectural changes that were extracted by mining the Git history, cf. Section 9.1.3.1. To be noticed, the reference Repository Model is changed only by the found architectural changes (I.A-E, II.A-E). As a result, we obtained the distinct reference models that are shown in the fourth column of Table 9.4.

Besides, we integrated the first commit of the interval I-III based on IRE to automatically extract additional reference models (see $M_{IRE(1.2)}$, $M_{IRE(1.2.1)}$ and $M_{IRE(1.3)}$ in Table 9.5).

IM We evaluated IM using $F1$ score as we described in Section 8.1.3.1. The results are in the last column of Table 9.4.

In the second part of **E1** (**E1.2**), we aim to investigate whether aggregating the commits and propagating them together impacts the accuracy. If not, this would imply that users can choose to run CIPM either after every commit or less frequently, such as during nightly builds.

Thus, we first propagate all the commits of each interval and propagate them as one commit. For that, we integrated the first commit of interval I. Then, we propagated the first commit of the next interval (Interval II) that aggregates all changes within the first interval and this commit results in $M_{1.2_{\sum[I]}}$. Similarly, we obtain $M_{1.2.1_{\sum[II]}}$, $M_{1.3_{\sum[III]}}$ and $M_{1.3.1_{\sum[IV]}}$ by aggregating the commits of interval II, interval III and interval IV. Afterward, we assess the resulting Repository Models in comparison to both the manually established reference models from the previous phase (**E1.1**) and the automatically generated reference models by IRE for the first commit of each interval. For instance, we compare $M_{1.2_{\sum[I]}}$ with $M'_{1.2}$ and $M_{IRE(1.2)}$. The accuracy of the resulting SCM and IM are also checked.

9.1.4. Setup on TEAMMATES

Like TeaStore, we repeated **E1** with the real Git history of TEAMMATES. In this subsection, we describe the considered Git history in Section 9.1.4.1 and **E1** steps on TEAMMATES in Section 9.1.4.2.

9.1.4.1. The Git history

The evaluation covers 17.859 commits that impact 1.428 files. To ease the representing of results, we divided the considered Git into five intervals based on the following commits: commit 64842 (C_I) as the initial commit, commit 48b67 (C_{II}), commit 83f51 (C_{III}), commit f33d0 (C_{IV}), and commit ce446 (C_V). In the appendix, there is more detail on the commits and their hash code, see Table A.2. Moreover, Table 9.2 sums up the main information of the five intervals:

Interval I The interval I consists of 17832 commits. It starts with the first commit of Git history and ends with commit C_I. Integrating commit C_I aggregates the changes of the 17832 previous commits. The IRE results in the first `Repository Model` ($M_{\sum [I]}$). Within interval I, 114468 code lines were added in 709 Java files.

Interval II This interval follows interval I. It includes the commits between commit C_I and commit C_{II}. Interval II spans 3 commits with 154 added lines and 129 removed ones in 122 Java files. In this interval, the maintainer role was introduced, affecting the `Repository Model` with four architectural changes.

Interval III The third interval $[C_{II},C_{III}[$ spans two commits that affect 227 Java files by adding 3249 lines and removing 2978 ones. In this interval, the visibility of some fields was changed from public to private. Thus, public get and/ or set methods were added to allow access to the aforementioned fields. These changes affect the `Repository Model` with 244 changes.

Interval IV Two commits are in interval IV, which affected 65 Java files by adding 502 lines and removing 340 lines. In this interval, some static variables were converted into non-static. Additionally, some classes were converted to singletons.

Interval V In the last interval V, there are 20 commits adding 3457 and removing 1293 lines in 147 Java files. In this interval, JavaDoc was updated. Besides, more classes were converted to singletons.

9.1.4.2. Applying Experiment1 on TEAMMATES

First, we integrated C_I as an initial commit. The IRE results in the initial `Repository Model` ($M_{\sum [I]}$). Then, we successively propagated C_{II}, C_{III}, C_{IV} and C_V. As a result, the VSUM was updated. Consequentially, four new versions of `Repository Models` that represent the intervals II-V were also updated: ($M_{\sum [II]}$, $M_{\sum [III]}$, $M_{\sum [IV]}$ and $M_{\sum [V]}$).

For evaluating the `Repository Model` resulting from integrating the initial commit C_I (M_{C_I}), we used the extensively documented architecture of TEAMMATES[1] as a

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

Interval	Range	Commits	Affected Java Files	Added Lines	Removed Lines	Architectural changes
I	$C_I]$	17832	709	114468	0	IRE
II	$]C_I$-$C_{II}]$	3	122	154	129	4
III	$]C_{II}$,$C_{III}[$	2	227	3249	2978	244
IV	$]C_{III}$, $C_{IV}]$	2	65	502	340	27
V	$]$ C_{IV},$C_V]$	20	147	3457	1293	176

Table 9.2: TEAMMATES commits that are considered in the evaluation.

reference and $F1$ score as a metric, see Section 8.1.3. We obtained reference models for intervals II-V by adjusting the initial `Repository Model` M_{C_I} based on the architectural changes within intervals II-V. As a result, we obtained the following reference models for intervals II-V: $M'_{C_{II}}$, $M'_{C_{III}}$, $M'_{C_{IV}}$ and M'_{C_V}. Besides, we integrated each of C_{II}, C_{III}, C_{IV} and C_V as an initial commit to automatically extract additional reference models. This resulted in four reference `Repository Models` that represent interval II-V: $M_{\text{IRE}(C_{II})}$, $M_{\text{IRE}(C_{III})}$, $M_{\text{IRE}(C_{IV})}$ and $M_{\text{IRE}(C_V)}$.

Additionally, the accuracy of the resulting `SCM` and `IM` were evaluated similarly to the way that was followed by applying **E1** on TeaStore.

9.1.5. Setup on Lua-based Applications

E1 is applied to two scenarios within an industrial context: BarcodeReaderApp and ObjectClassifierApp. The input was the Lua-based source code. In the following, we provide an overview of the considered Git repositories (Section 9.1.5.1) and the specific **E1** steps applied to Lua-based sensor applications (Section 9.1.5.2).

9.1.5.1. Git Repository History

As shown in Table 9.3, the Git repository of BarcodeReaderApp contains a total of 7 commits, impacting 13 files. The changes in Lua files are 862 added lines and 283 removed ones. In the ObjectClassifierApp, we consider 12 commits affecting 52 files. The cumulative changes involve the addition of 6651 lines and the removal of 2663 lines. For more details on the commits, please refer to the appendix (Section A.5).

Table 9.3: Commits of Lua-based sensor applications that are considered in the evaluation.

Case	Commits	Affected Files	Added Lines	Removed Lines
BarcodeReaderApp	7	13	862	283
ObjectClassifierApp	12	52	6651	2663

9.1.5.2. Applying Experiment1 on Lua-based sensor applications

We start the process with the first commit of the BarcodeReaderApp. Then, we sequentially update the VSUMs models by incorporating the following 6 commits consecutively. Similarly, we repeat **E1** for ObjectClassifierApp by integrating the first commit and updating the VSUM with the remaining 11 commits consequentially. The resulting models within VSUM are compared with reference models as illustrated in Section 9.1.2. For evaluating the `Repository Models` resulting from integrating the first commit (M_1), we used manually created reference models that are assessed by experts. The updated `Repository Models` are evaluated by comparing them to reference models that are automatically reverse-engineered by IRE. Additionally, a manual verification of the accuracy of reference models is performed, considering the initially created reference models (M_1') and the changes summarized in Table A.3 and Table A.4.

9.1.6. Results

In this section, we present the results of **E1** based on TeaStore, TEAMMATES and Lua-based sensor applications.

TeaStore: *The accuracy of IRE:* We compared $M_{1.3.1}$ to the manually modeled PCM. In this comparison, we found 6 true positive components representing the 6 microservices. There was 1 false positive component representing the "rabbitmq" component, which represents a plugin for messaging used by all microservices and ignored by the manually modeled PCM. Additionally, we did not find any false negative cases. Consequently, the precision for the component was approximately 0.857, indicating that the extracted model is more detailed. The recall is 1.0, meaning $M_{1.3.1}$ includes all manually modeled components. The F-score is approximately 0.923.

The evaluation of interfaces resulted 0.372 for the $F1$ score because the precision value was 0.229, indicating that about 22.9% of the extracted interfaces matched those in the reference model. The remaining interfaces represent the communication interfaces, which are, in our case, RESTful technology. The recall was perfect at 1.0, implying that all reference model interfaces were successfully identified. It should be noted that our matching process ignored naming conventions. For instance, 'AuthCartRest' was mapped to 'AuthCart' despite differences in naming.

Hence, our model includes additional communication interfaces because we define our CPRs to detect the overall architecture including the microservices and the technology used by microservices to communicate with each other. These communications were ignored by the manually modeled reference models due to the limitation of the `Palladio` tool in simulating asynchronous communication, where these communications were modeled by external calls for enabling the simulation, see Section 8.3.3.1.

Consequentially, we can conclude that the resulting `Repository Model` ($M_{1.3.1}$) accurately represents the overall architecture of TeaStore web application.

The accuracy of updated models: Table 9.4 sums up excerpts of the results of applying **E1** on TeaStore by sequential propagation of commits, i.e., **E1.1**. To notice, the table shows the results of propagating commits related to new releases or commits impacting the architecture. The results of evaluating the VSUM after propagating the remaining commits confirmed the accuracy of VSUM and are available in [Arm21]. However, the results in [Arm21] in slightly different outcomes compared to those documented in the replication package of **E1** results [Maz+24], where an optimization of the `JaMoPP` parser is applied [MAK23]. This optimization effectively would reduce the execution time and enable the propagation of commits despite compilation errors.

Regarding the `SCM`, the value of JC is always 1 except in the case of IRE of initial commits, where a failure in calculating JC resulted in the value 0.999997. That is because of an incorrect mismatch case reported by the similarity checker. The two methods in `SCM` are identical except that their parameter dimensions are different, i.e., the parameter type of the first one is `byte` whereas `byte[]` of the second one. The current similarity checker did not succeed in matching these methods in VSUM with their corresponding ones in the reference model because of an incorrect check of the dimensions of the parameter. Therefore, it reported an addition and removal of an array dimension, which only affected the automatically calculated value of JC. However, this mismatch does not mean that the `SCM` in VSUM is inaccurate because the manual check disproved that. Because of lack of time, we planned to extend the similarity checker in future work.

Regarding the accuracy of the `Repository Models` that are updated incrementally, the results in the fifth column of Table 9.4 confirmed the accuracy of the `Repository Models`. Besides, the last column confirmed the accuracy of the last model in the VSUM, i.e., the `IM`.

The last three rows of Table 9.4 shows the average execution time of IRE, update execution time, and the required time for instrumenting the source code adaptively.

Table 9.5 sums up the results of **E1.2**. Similar to **E1.1**, we observed that the IRE for the first commit resulted in a JC value less than one due to the reasons mentioned above. However, propagating the subsequent commits resulted JC values of one were obtained, indicating the accuracy of the updated `SCM`.

The sixth column shows the comparison results between the updated `Repository Models` and the manually updated ones. The comparison results between the updated `Repository Models` and the automatically extracted reference `Repository Models` are in the seventh column. The JC values in both columns were one, which confirms the accuracy of the updated `Repository Models`. In addition, the value of one for the $F1$ score metric in the eighth column serves as confirmation of the accuracy of the `IM`.

Table 9.4: Excerpt of Experiment1.1 results based on [Arm21]. Experiment1.1 applies sequential update of TeaStore VSUM in four intervals.

	JC(SCM, SCM')	updated RepM	Reference RepM	JC (M,M')	F(IM,RepM)
I	0.99997	$M_{IRE(1.2)}$	$M'_{1.2}$	1	1
	1	$M_{I.A-B}$	$M'_{I.A-B}$	1	1
	1	$M_{I.C}$	$M'_{I.C}$	1	1
	1	$M_{I.D}$	$M'_{I.D}$	1	1
	1	$M_{1.E}$	$M'_{1.E}$	1	1
II	0.99997	$M_{IRE(1.2.1)}$	$M'_{1.2.1}$	1	1
	1	$M_{II.A}$	$M'_{II.A}$	1	1
	1	$M_{II.B}$	$M'_{II.B}$	1	1
	1	$M_{II.C}$	$M'_{II.C}$	1	1
	1	$M_{II.D}$	$M'_{II.D}$	1	1
	1	$M_{II.E}$	$M'_{II.E}$	1	1
III	0.99997	$M_{IRE(1.3)}$	$M'_{1.3}$	1	1
IV	0.99997	$M_{IRE(1.3.1)}$	$M'_{1.3.1}$	1	1
Average execution time of IRE =26.5					
Update time average=3.8					
Instrumentation time average=0.7					

Table 9.5: The results of Experiment1.2 based on [Arm21]. Experiment1.2 updates the VSUM of TeaStore considering the *aggregated* commits of the intervals I-IV.

	JC(SCM, SCM')	updated RepM	Reference RepM		JC (M,M')	JC (M,M_{IRE})	F(IM,RepM)
			M'	M_{IRE}			
I	0.99997	$M_{1.2_{\sum[I]}}$	$M'_{1.2}$	$M_{IRE(1.2)}$	1	1	1
II	1	$M_{1.2.1_{\sum[II]}}$	$M'_{1.2.1}$	$M_{IRE(1.2.1)}$	1	1	1
II	1	$M_{1.3_{\sum[III]}}$	$M'_{1.3}$	$M_{IRE(1.3)}$	1	1	1
IV	1	$M_{1.3.1_{\sum[IV]}}$	$M'_{1.3.1}$,	$M_{IRE(1.3.1)}$	1	1	1
Execution time of IRE =36,6							
Update time average=5.1							
Instrumentation time average=0.5							

The reduction of instrumentation probes: Regarding the instrumentation probes, Table 9.6 sums up how much the adaptive instrumentation reduced the instrumentation probes. Overall, there was a substantial decrease ranging from 60.5% to 80.6% across the four intervals. Regarding the fine-grained instrumentation probes, the reduction was more, reaching levels of 85.2% to 100%. The 100% indicates that no fine-grained instrumentation was performed. This corresponds to the commits in the third and fourth intervals, during which no architectural changes were applied.

Table 9.6: The reduction of the instrumentation probes that the adaptive instrumentation achieved during applying Experiment1 on TeaStore, based on [Arm21].

Interval	Overall reduction	Fine-grained reduction
I	60.5%- 67.7%	85.2%-95.5%
II	67.8%- 68.1%	96.3%-97.6%
III	80.6%	100%
IV	80.6%	100%

The relation between update times and changes: We investigated the factors influencing execution time, specifically examining the number of affected lines (added, modified, or deleted lines) and the number of changes executed by Vitruvius (Figure 9.3). Our analysis confirms the impact of these factors on execution time. It is notable that execution time is not only determined by the total number of changes in a commit; the types of these changes also play a crucial role. Execution time varies according to the executed CPRs, where different CPRs may exhibit distinct execution times.

The overall execution time is relatively high compared to other cases. Specifically, the execution time of the IRE, representing the worst scenario with a substantial number of changes, is approximately 26.5 minutes on average. In contrast, the average update time is around 5.1 minutes. This is attributed to parsing the source code with all dependencies and resolving them to obtain a detailed model, whereby the model complexity directly impacts the time required for matching and change detection. More details are in Section 9.1.7.

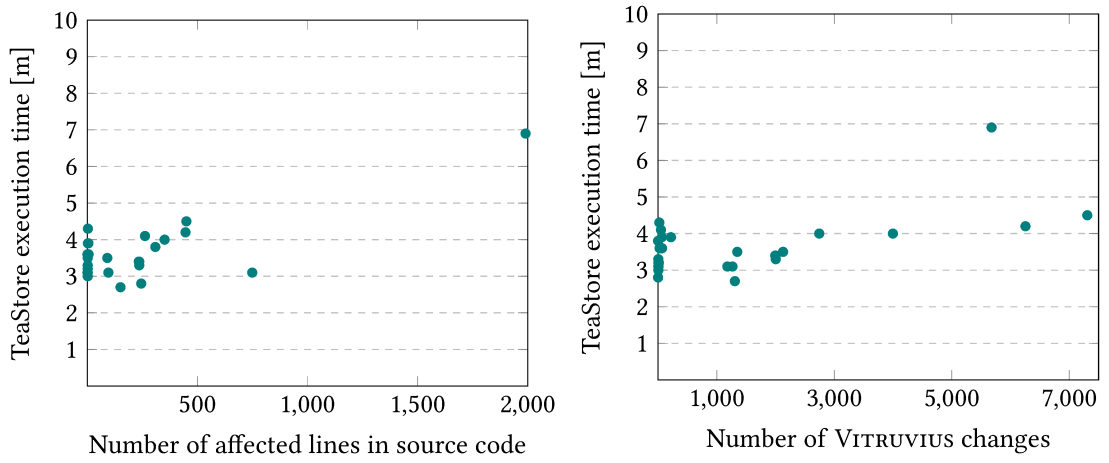


Figure 9.3: On the left side, the relation between the number of affected lines in TeaStore and the update time. On the right side, the relation between the number of Vitruvius changes and the update time.

Table 9.7: The result of applying Experiment1 to update the VSUM of TEAMMATES considering the aggregated commits of the intervals I-V, based on [Maz+25].

	JC(SCM, SCM')	updated RepM	Reference RepM		JC (M,M')	JC (M,M_{IRE})	F(IM,RepM)
			M'	M_{IRE}			
I	1	$M_{\sum[I]}$	M'_{C_I}	$M_{IRE(C_I)}$	1	1	1
II	1	$M_{\sum[II]}$	$M'_{C_{II}}$	$M_{IRE(C_{II})}$	1	1	1
II	1	$M_{\sum[III]}$	$M'_{C_{III}}$	$M_{IRE(C_{III})}$	1	1	1
IV	1	$M_{\sum[IV]}$	$M'_{C_{IV}}$,	$M_{IRE(C_{IV})}$	1	1	1
V	1	$M_{\sum[V]}$	M'_{C_V},	$M_{IRE(C_V)}$	1	1	1
Execution time of IRE =9.39							
Update time average=2.05							
Instrumentation time average=0.45							

TEAMMATES *The accuracy of IRE:* We assessed the accuracy of the $M_{\sum[I]}$ resulting from the IRE of the C_I with the documented architecture of TEAMMATES. We found that the components and interfaces were accurately detected, resulting in the value 1 for the $F1$ score Ṫhis indicates that the defined CPRs were effective in detecting the architecture of teammates.

The accuracy of updated models: We assessed the accuracy of models within VSUM that CI-based consistency preservation at Dev-time updated during **E1**. The outcomes of the accuracy evaluation for TEAMMATES are illustrated in Table 9.7. The results show that the models within VSUM were accurately updated through propagating C_{II} to C_V. The JC values were consistently 1 for Java SCM. Additionally, the sixth and seventh columns of Table 9.7 confirmed the accuracy of the updated Repository Models. Besides, the $F1$ score values were 1 for the IM, suggesting an accurate update of the required instrumentation probes.

The reduction of instrumentation probes: The reduction of instrumentation probes, which the adaptive instrumentation led to, is presented in Table 9.8. The findings show coarse-grained reductions ranging from 55.4% to 60.7% and fine-grained reductions from 72.8% to 99.5%.

Table 9.8: The reduction of the instrumentation probes that the adaptive instrumentation achieved by applying Experiment1 on TEAMMATES, based on [Maz+25].

Interval	Overall reduction	Fine-grained reduction
I	0	0
II	55.4%	88.9%
III	46%	72.8%
IV	62.8%	99.5%
V	60.7%	96.4%

The Relation between Update Times and Changes: Similar to TeaStore, the analysis of execution time of TEAMMATES follows the same structure, where the left side analyzes the impact of affected lines, while the right side examines Vitruvius changes. The examination of the data indicated that the execution times are comparatively lower than those observed in TeaStore, a phenomenon attributed to the use of the recovery strategy for resolving dependencies [MAK23]. This approach promises a significant time-saving effect, particularly in parsing, dependency resolution and changes detection.

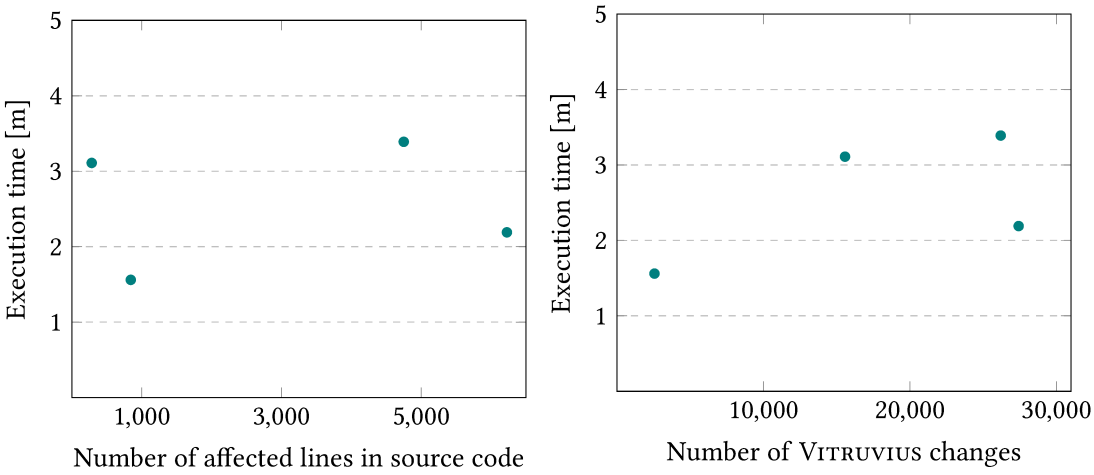


Figure 9.4: On the left side, the relation between the number of affected lines in TEAMMATES and the update time. On the right side, the relation between the number of Vitruvius changes and the update time.

Lua-based sensor applications *The accuracy of updated models:* The results of **E1** on BarcodeReaderApp reveal accurate updates of all models (`SCM`, `IM` and `RepM`) through the propagation of all commits, as shown in Table 9.9. The JC values were consistently 1 for both Lua `SCM` and `RepM`. Besides, the F-score values were 1 for the `IM` across all commits. However, this is not the case for ObjectClassifierApp. Despite correctly updating the VSUM models with the first six commits of ObjectClassifierApp, the accuracy of the Lua model for the subsequent 6 commits was slightly below 100%, as detailed in Table 9.10. These six commits resulted in accurate updates for almost all elements of the Lua model (at least 94.7%). The reason is that minor elements in the Lua model were out of order within the VSUM, causing the matching algorithm to struggle to match these elements. This impacted the subsequent update process of the remaining models (`RepM`and `IM`) and subsequent commits. Regarding `RepM`, the JC value approaches 0.992 for commits 7 to 10 and 0.9472 for commits 11 and 12. Regarding the `IM`, it was updated accurately with an exception by the 11th commit where only one instrumentation probe failed to be activated as intended, resulting in a 99% accuracy.

Commits	$JC(SCM, SCM')$	$JC(RepM, RepM')$	$F(IM, IM')$
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1
5	1	1	1
6	1	1	1
7	1	1	1
Execution time of IRE = 0.269 seconds			
Update time average = 0.49 seconds			

Table 9.9: Results of Experiment1 on Lua-based BarcodeReaderApp, based on [Bur23].

Commits	$JC(SCM, SCM')$	$JC(RepM, RepM')$	$F(IM, IM')$
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1
5	1	1	1
6	1	1	1
7	0.99	0.99	1
8	0.99	0.99	1
9	0.99	0.99	1
10	0.99	0.99	1
11	0.94	0.96	0.99
12	0.94	1	1
Execution time of IRE = 2.092 seconds			
Update time average = 3.93 seconds			

Table 9.10: Results of Experiment1 on Lua-based ObjectClassifierApp, based on [Bur23].

The issue with the Lua matching algorithm that arose during the execution of **E1** on ObjectClassifierApp remains unresolved. This is due to the lack of access to the closed-source ObjectClassifierApp following the conclusion of the collaboration with Sick GmbH during Burgey's master thesis [Bur23]. Ongoing work involves a detailed examination of the implemented hierarchical matching for Lua models based on EMF Compare, aiming to investigate and evaluate its ability to accurately match out-of-order elements based on additional Lua-based cases. It is crucial to emphasize that the matching issue is attributed to implementing the hierarchical matching algorithm and is not inherent to the CIPM concept itself.

The reduction of instrumentation probes: As shown in Table 9.11, in BarcodeReaderApp, the adaptive instrumentation reveals variability in reduction metrics across commits. Commit 7 achieves the maximum overall reduction at 68.8%, indicating a significant optimization in instrumentation probes. In contrast, commit 3, which removes some functionality from `DatabaseAPI`, shows the minimum overall reduction of 27.3% due to the need for fine-grained monitoring to calibrate the new functionality.

Concerning fine-grained reduction, commit 6 and commit 7 stand out with the maximum value of 100%, representing a fully optimized state. These commits involve the removal of some server calls from `DatabaseAPI`, affecting no PMPs, eliminating the need for fine-grained instrumentation.

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 9.11: The reduction of the instrumentation probes that the adaptive instrumentation achieved during Experiment1 on BarcodeReaderApp, based on [Bur23].

Shifting to ObjectClassifierApp, commit 5 achieves the maximum total reduction of instrumentation probes at 36.28%, with no fine-grained instrumentation, leading to a 100% reduction in fine-grained instrumentation probes, see Table 9.12. This is attributed to the fact that commit 5 fixed minor bugs without impacting PMPs, thus requiring no fine-grained instrumentation. In contrast, commit 11 displays the minimum total reduction of instrumentation probes at 12.62%, indicating a less optimized state compared to commit 5. This is due to the fact that commit 11 affecting approximately 5000 source code lines through refactoring and the addition of new features. Consequently, fine-grained instrumentation was crucial to ensure the quality of PMPs related to the altered source code.

The relation between update times and changes: Similar to TeaStore and TEAMMATES, we analyzed the relationships between update times and the number of affected source code lines on one side and Vitruvius changes on the other side. Both Figure 9.5 and Figure 9.6 presented diagrams provide insights into the relationships for BarcodeReaderApp and ObjectClassifierApp.

For the BarcodeReaderApp, the execution time was relatively too short. The correlation between the number of affected lines in the source code and the execution time. On the right side, a similar analysis is conducted, but with a focus on the number of Vitruvius changes.

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

Table 9.12: The reduction of the instrumentation probes that the adaptive instrumentation achieved during Experiment1 on ObjectClassifierApp, based on [Bur23].

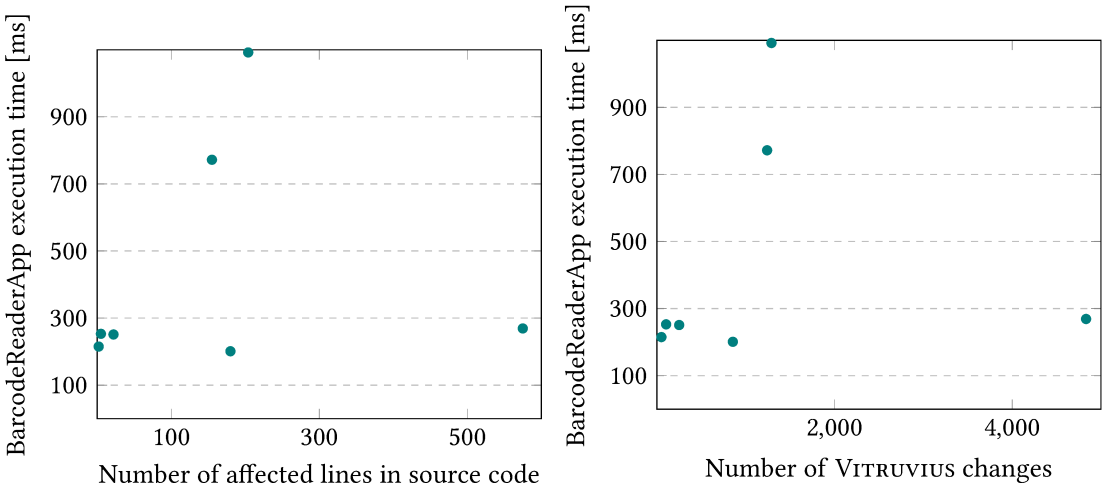


Figure 9.5: On the left side, the relation between the number of affected lines in BarcodeReaderApp and the update time. On the right side, the relation between the number of Vitruvius changes and the update time.

Moving to the ObjectClassifierApp, the left-side diagrams replicate the structure of the BarcodeReaderApp analysis, investigating the impact of affected lines on update time. The right-side diagrams, again mirroring the BarcodeReaderApp counterpart, assess the influence of Vitruvius changes on update time for ObjectClassifierApp.

The diagrams above provide insights into the performance characteristics of scalability at Dev-time for the Lua-based applications. These results indicate that the scalability met acceptable standards for realistic scenarios, as discussed further in Section 9.1.7.

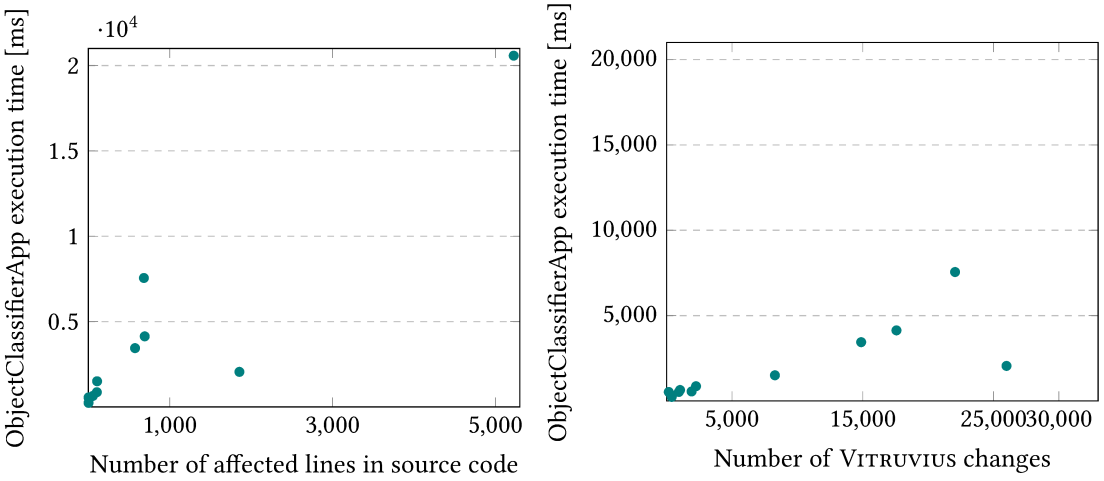


Figure 9.6: On the left side, the relation between the number of affected lines in ObjectClassifierApp and the update time. On the right side, the relation between the number of Vitruvius changes and the update time.

9.1.7. Discussion

Answering EQ-1.1 Considering the results from **E1** across different cases (TeaStore, TEAMMATES, BarcodeReaderApp and ObjectClassifierApp), it was observed that the VSUM models were accurately updated in most cases. However, minor discrepancies were noted in **E1** on ObjectClassifierApp due to implementation issues in the Lua matching algorithm, resulting in a decrease in the JC metric to no less than 0.94%. Further, issues stemming from updating the `SCM` of the ObjectClassifierApp affected the accuracy of the remaining models in VSUM, achieving a minimum of 0.96 for JC when evaluating `Repository Model` and 0.99% for $F1$ score when evaluating `IM`. However, promising results were obtained from the other cases.

Consequently, the results suggest that the proposed CI-based strategy effectively updates the VSUM models without necessitating the use of a specialized editor or platform for source code development. However, future efforts should focus on refining the matching algorithm implementation to address specific cases more effectively.

Answering EQ-1.1.1 To answer **EQ-1.1.1** we compared $M_{IRE(1.3.1)}$ with the manually modeled `Repository Model`. We found that defined CPR extracted the components correctly. In other words, both $M_{IRE(1.3.1)}$ and $M'_{1.3.1}$ model the microservices as basic components. Moreover, $M_{IRE(1.3.1)}$ includes the interfaces within $M'_{1.3.1}$ in addition to other communication interfaces, e.g., the RESTful interfaces. However, the additional technical details in $M_{IRE(1.3.1)}$ do not affect the fact that $M_{IRE(1.3.1)}$ modeled the relevant structure of the manually modeled `Repository Model`. However, it should be mentioned that the IRE was not a fully automatic process. In some cases, the architect was asked

to make a decision. For example, the case of the plug-in with a POM file can be considered a microservice candidate. The absence of a docker file prevented the CPRs from classifying this case as a basic component. The architect was once asked to decide whether it was a component or not. The decision was stored for the next propagation.

Regarding TEAMMATES, the evaluation also confirmed the accuracy of the $M_{IRE(C_I)}$. The comparison with the documentation showed that the components and interfaces were correctly modeled by IRE. This applies also to Lua-based sensor applications.

Consequently, we affirm **EQ-1.1.1** by stating that the defined CPRs successfully constructed architectures for various project types employing different technologies. These CPRs could be reused for projects utilizing similar programming languages and technologies. Additionally, it is noteworthy that the extraction of `Repository Model`, in some cases, may be not entirely automated, requiring user input for decision-making. The made decisions, however, are recorded and utilized for subsequent commit propagation

Answering EQ-1.1.2 We evaluated the accuracy of VSUM by propagating aggregated commits of TeaStore and compared the updated models to the reference models, see the sixth and seventh columns of Table 9.5. The JC values were always equal to one, which confirmed the accuracy.

Based on the results mentioned in this paragraph, we can answer **EQ-1.1.2** as follows: There are no differences between the models resulting from propagating aggregated commits ($M_{\sum I_x}$) and the reference models (M_x). This means that the architect can decide when to update the aPM, by disabling the update process by some commits and enabling it by others. However, it should be noted that the `IM` will differ. The required instrumentation probes for calibrating $M_{\sum I_x}$ will be equal to or bigger than M_x. That is related to the architectural changes within the propagated aggregated commits.

Answering EQ-1.3 The evaluation of the adaptive instrumented source code as explained in Section 8.1.3.2, confirmed the accuracy: The number of added instrumentation statements belonged to the acceptable range. All the fine-grained instrumentation probes were correctly added. The $F1$ score resulting from the manual evaluation of the fine-grained instrumentation was equal to one. Besides no compilation errors were found by the instrumented source code. For more details on evaluating the adaptive instrumentation, see [Arm21].

Answering EQ-2.1 According to evaluation results from **E1** on TeaStore and TEAMMATES, we found that the adaptive instrumentation can save at least 12.62% of the required fine-monitoring overhead and between 26.21% and 100% of the overall instrumentation probes, see Table 9.6, Table 9.8, Table 9.11 and Table 9.12.

Answering EQ-3.1 In **E1** we empirically identified factors impacting execution time for our four cases. The key factors influencing execution times include the parser, commit size, and changes within the commit. The IRE for the initial commit, representing a worst-case scenario, shows variations based on commit size and the parser technique and configuration. Notably, the execution time difference in the Incremental Reverse Engineering between the java-based application, TeaStore including 236 files with 26636 added lines and TEAMMATES including 709 files with 114468 added lines, is 27.21 minutes, attributed to the parsing approach and a recovery strategy that we used for TEAMMATES [MAK23]. The reduced execution times underscore the efficacy of employing a recovery strategy in TEAMMATES, emphasizing its positive impact on scalability. It is essential to acknowledge this optimized approach as a noteworthy factor influencing the observed execution time. The recovery strategy used in TEAMMATES is a significant step towards an incremental parsing strategy that parses only modified files for optimizing the execution time.

Despite parsing the entire source code by the following commits, the average execution time for propagating commits is 4.24 minutes for TeaStore and 2.05 minutes for TEAMMATES, which is not critical for development. In the worst case, the execution of MbDevOps during development can be nightly scheduled.

For Lua-based applications, the execution time is significantly reduced (1.181 seconds for IRE compared to 2.7 seconds for updates). This reduction can be attributed to the efficiency of the Lua parser and the smaller size of the application. For instance, the IRE of BarcodeReaderApp includes 4 files and 575 added lines and IRE of ObjectClassifierApp includes 7 files with 1855 added lines). The analysis underscores the impact of both the quantity and characteristics of changes on the overall execution time.

In summary, the scalability analysis suggests a proportional impact of changes on execution time in TeaStore, TEAMMATES, and Lua-based sensor applications. The proposed incremental parsing in Java-based applications is expected to further reduce execution time. Overall, the results can affirm the applicability of the CIPM approach, since the measured execution times are not critical for Dev-time.

9.1.8. Limitation

It is important to recognize that the current implementation of **E1** has a specific limitation: it is applicable only to applications that are stored on Git. The ability to update aPMs using **E1** is restricted if the applications are not situated on this platform. Nevertheless, this limitation opens up the opportunity for future work, which could expand the applicability of **E1** by incorporating other version control systems, such as Mercuria [2] or Perforce [3] thereby extending its utility.

[2] see `https://www.mercurial-scm.org/`.

[3] see `https://www.perforce.com/`.

9.1.9. Threats of Validity

The threats to validity for case study research [Woh+12] can be divided into four dimensions: *internal validity*, *external validity*, *construct validity* and *conclusion validity*.

Internal Validity A threat to validity is the selection of reference models and metrics, which can distort the evaluation results.

Regarding TeaStore, the primary reference `Repository Model` $M'_{1.3.1}$, which underwent manual modeling and subsequent review by other researchers [Mon20], was employed by Armbruster [Arm21] for executing **E1**. This approach aimed to mitigate the subjectivity inherent in the evaluation of the resulting `Repository Models`.

Regarding TEAMMATES, we do not have a reference PCM to evaluate the initial `Repository Model` (M'_{C_I}). However, we used the well-documented architecture that the developers of TEAMMATES documented. The reference models for intervals II-V of TEAMMATES Git were built based on M_{C_I} and the models resulting from the IRE of C_{II}-C_V ($M_{IRE(C_{II})}$, $M_{IRE(C_{III})}$, $M_{IRE(C_{IV})}$ and $M_{IRE(C_V)}$). We calculated the diffs between the `Repository Models` for each of intervals II-V and compared them manually to the diffs between the commits C_{II}-C_V in order to build ($M'_{C_{II}}$, $M'_{C_{III}}$, $M'_{C_{IV}}$ and M'_{C_V}). The reason was to reduce the effort and errors by manually building reference models in the case of many architectural changes.

By Lua-based sensor applications, reference models were manually modeled by Burgey [Bur23] and reviewed by experts from Sick GmbH and me.

Regarding the used metrics, JC has been used in related work [Hei20] for evaluating PCM accuracy and $F1$ score is used for information retrieval problem [Chr10].

External Validity: 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 17859 commits of the real application TEAMMATES, 181 of the case study TeaStore and 19 commits from the Lua-based sensor applications. The IRE of the initial commits utilize most of adding CPRs. Propagating the following commits covers most of the remaining CPRs, for instance propagating commits C_{II}-C_V from TEAMMATES Git covers 41% of the remaining CPRs.

Construct Validity: In the evaluation, we rely on the predefined CPRs that are project-specific rules. For that, we used project-specific CPRs in our evaluation, e.g., detecting the microservice components for TeaStore. However, the used CPRs were evaluated as mentioned above and may not be suitable for other projects. The $M_{IRE(1.3.1)}$ was evaluated based on the manually modeled `Repository Model`. Wrong modeling can affect the following results. Therefore, $M'_{1.3.1}$ was reviewed by three researchers from our chair and Würzburg University. The model was suitable for the performance prediction goal, considering the microservice architecture and the capabilities of the `Palladio`

simulator. However, the manual models ignored the asynchronous communication and modeled the communication between microservices by external calls. This point was considered when evaluating $M_{IRE(1.3.1)}$. Additionally, we consider modeling the asynchronous communication by the remaining reference models.

Conclusion Validity: To avoid a researcher's subjectivity by evaluating the accuracy of initial `Repository Models`, two researchers checked the resulting `Repository Models` compared to reference models provided by another researcher of developers. The evaluation results of instrumentation overhead and the accuracy of both `IM` and `SCM` are easy to understand and interpret.

9.2. Experiment2: System Model Update at Development Time

Experiment2 (**E2**) evaluates the extraction of a `System Model` at Dev-time that we explained in Section 5.1.5. In the following subsections, we describe the goal of **E2** in Section 9.2.1 and **E2** scenario in Section 9.2.2. The setup of **E2** follows in Section 9.2.3. We present the results in Section 9.2.4 and discuss them in Section 9.2.5. The limitations and threats of validity follow in Section 9.2.6 and Section 9.2.7.

The Section 9.3 summarizes both **E1** and **E2**.

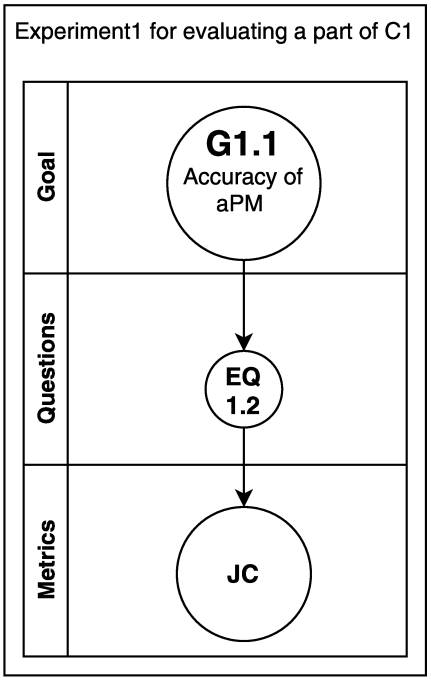


Figure 9.7: GQM plan of Experiment2.

9.2.1. Goal

The experiment **E2** validates the second part of the first contribution, i.e., the extraction of `System Model` at Dev-time. The goal is to evaluate the accuracy of the extracted `System Model`.

As Figure 9.7 shows, **E2** addresses just one evaluation question (**EQ-1.2**): "*How accurately and automatically does CIPM extract the* `System Model` *at Dev-time?*".

The accuracy of the updated `System Model` is quantified by JC as explained in Section 8.1.3.1.

9.2.2. Scenario

The input of **E2** is the source code of two case studies: CoCoME and TeaStore. The additional input is the correspondence model of VSUM, which includes the mapping between the `Repository Model` and source code.

For extracting `System Model`, the SCG is first built. Then, the `System Model` is extracted based on `System Model`. Since the extraction of `System Model` at Dev-time is semi-automatic, the architect is asked to resolve the potential conflicts.

In addition to `System Model`, the experiment results in the number of all found conflicts and how many are resolved by the architect.

Finally, the resulting `System Model` is compared to a reference model representing the actual system composition to evaluate the accuracy. The JC is used for qualifying the accuracy. Figure 9.8 simplifies **E2** scenario.

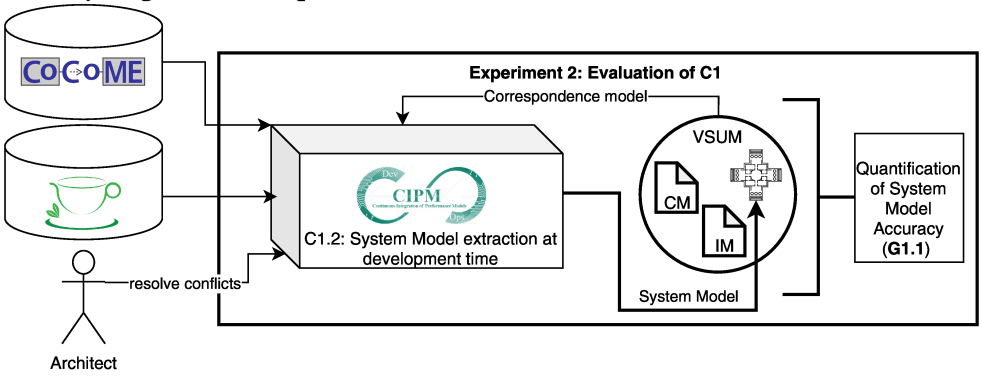


Figure 9.8: The scenario of Experiment2.

9.2.3. Setup

We used the cloud-based implementation of CoCoME[4] and version 1.3.1 of TeaStore in **E2**. Then, we used the corresponding available `Repository Models`[5] and integrated them into VSUM based on the linking integration strategy [Leo+15]. The resulting corresponding models were used as input for **E2**.

[4] `https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest`

[5] The repository model of CoCoME is available on `https://github.com/CIPM-tools/Incremental-Calibration-Pipeline/tree/master/evaluation/evaluation-automation-platform/casestudys/cocome/pcm` and the repository model of TeaStore is on `https://github.com/CIPM-tools/Incremental-Calibration-Pipeline/tree/master/modelrefinement.parameters.root/modelrefinement.parameters.casestudy.teastore/casestudy-data/pcm`

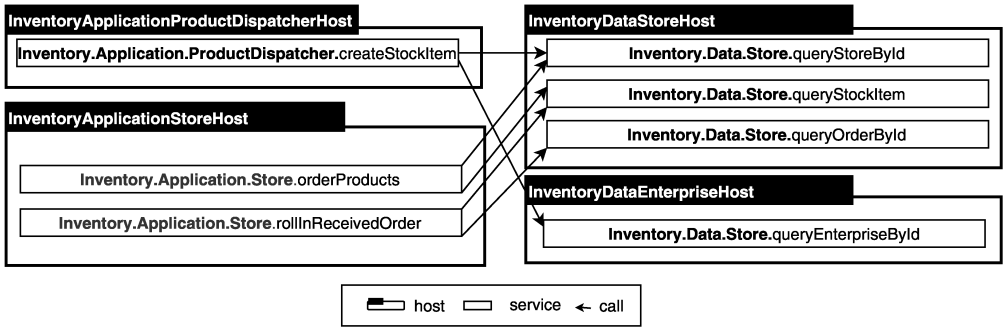


Figure 9.9: Excerpt of the extracted CoCoME SCG, adjusted from [Mon20].

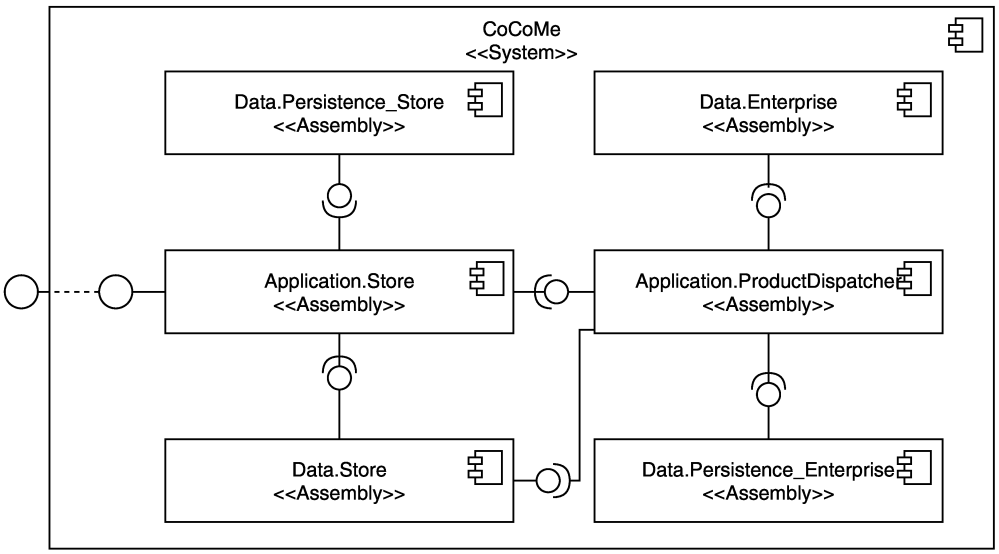


Figure 9.10: Simplified excerpt of the extracted CoCoME `System Model`.

9.2.4. Results

This section presents the results of **E2** based on CoCoME and TeaStore.

CoCoME Figure 9.9 shows an excerpt of the resulting SCG, which was used to extract the `System Model`. The extraction of CoCoME `System Model` resulted in 5 conflicts. Three of them were resolved automatically by the predefined decisions based on the static analysis. However, the architect had to resolve the remaining two conflicts. An example of the conflict can be deduced from Figure 9.10: Both of `Application.Store` and `Application.ProductDispatcher` require `Data.Store`, which raises an assembly conflict. The architect was asked whether a common instance of `Data.Store` assembly context had to be used or not. The comparison with the reference model after the semi-automatic extraction of `System Model`, resulted in the value 1 for JC, see the

Table 9.13: Results for deriving the System Model at Dev-time

Examples	JC(M_{sys},M'_{sys})	Model Elements	Conflicts	Ratio of automatically extracted elements
CoCoME	1.0	16	2	68.75%
TeaStore	1.0	18	5	72.3%

first row of Table 9.13. The Table also shows the number of resulting System Model elements, i.e., Operation Provided Role, Delegations, Assembly Contexts and Assembly Connectors. Based on the number of System Model elements and the number of the automatically extracted elements (without conflicts), we determine how automatically the System Model is extracted at Dev-time. Hence, 68.75% of CoCoME System Model elements was automatically extracted.

TeaStore The extraction of System Model caused five conflicts. None of them were resolved automatically. The reason was that the static analysis of the source code could not track the destination of the REST calls. The resulting System Model after resolving the conflicts is presented in Figure 5.10. As shown in the second row of Table 9.13, the JC was equal to one. Additionally, 72.3 of the System Model elements were extracted automatically.

9.2.5. Discussion

The results of **E2** that are summed in Table 9.13 confirmed the accuracy of the System Model. Hence, we can answer **EQ-1.2** as follows: CIPM can support the architect by extracting an accurate System Model. The evaluation manifests that at least 68.75% of the System Model elements were automatically extracted. The architect had to make a design decision to complete the extraction of the remaining elements.

Consequentially, the architect can use the updating System Model to perform AbPP before executing the source code in the production environment.

9.2.6. Limitation

E2 is restricted to quantifying the accuracy of the resulting System Model using a reference model that is built in agreement with the manually made decision. Currently, **E2** cannot judge the accuracy of the resulting model using an equivalent reference model. For instance, the comparison between the resulting System Model of TeaStore (shown in Figure 5.10) and an equivalent one including two instances of Registry does not result in the value 1 for JC.

9.2.7. Threats of Validity

The threats to the validity of **E2** are four dimensions according to [Woh+12]:

- *Internal validity*: During the execution of **E2** the resulting conflicts were resolved manually. **E2** results are strongly dependent on the decision made. If the person makes a wrong decision, the resulting JC will not equal 1.
- *External validity*: The selection of examples impact the results. However, CoCoME and TeaStore are widely used in research fields. Since TeaStore uses REST calls, more conflicts occur when extracting `System Model`. Therefore, it is an example of an application that uses asynchronous communications.
- *Construct validity*: The reference `System Model` can be created in different ways that are considered correct. The reference model must be constructed in the same way that the architect follows to resolve the conflicts. Otherwise, the JC will result in a value lower than 1. Besides, the evaluation results rely on the person who resolved the conflicts. The conflict resolution was applied by Monschein [Mon20], who performed **E2**. However, the results were reviewed by me to avoid the researcher's subjectivity.
- *Conclusion validity*: As aforementioned in threats to construct validity, the manual made decisions for resolving the conflicts and the constructed reference model was reviewed again to avoid the researcher's subjectivity by interpreting the results.

9.3. Summary

According to the results of **E1** and **E2**, we conclude that CIPM can accurately update the software models at Dev-time. This applies to the following models: `Source Code Model (SCM)`, `Repository Model`, `Instrumentation Model (IM)`, instrumented `Source Code Model (SCM)` and `System Model`. However, the update of `System Model` and `Repository Model` at the Dev-time is not fully automatic: the architect can be asked in some cases to confirm the detected components of the `Repository Model` or to decide whether to create or reuse available component instances for the `System Model`.

Moreover, **E1** confirmed that the adaptive instrumentation could reduce the instrumentation probes according to our case studies from 12.26% to 80.6%.

10. Validation of Incremental Calibration

This chapter presents the validation of the incremental calibration of the performance models. The validation is performed based on an experiment (Section 10.1) performed, mainly alongside two case studies (CoCoME and TeaStore). The findings are summed up in Section 10.2.

10.1. Experiment3: Incremental Calibration of Performance Model

Experiment3 (**E3**) evaluates incremental calibration by applying scenarios of incremental evolution and calibrating the updated models accordingly. We used two case studies (TeaStore and CoCoME) and two artificial examples for this experiment. In the following, we introduce the goal of Experiment3 and its scenario in Section 10.1.1 and Section 10.1.2. Then we describe how we design Experiment3 for the case studies in sections 10.1.3 to 10.1.6. The results, discussion and limitations of Experiment3 follow in sections 10.1.7 to 10.1.9. Finally, we discuss the threats to validity in section 10.1.10. A summary of the findings of **E3** will be provided in Section 10.2.

10.1.1. Goal

The experiment **E3** aims to validate the third contribution of CIPM approach, which is incremental calibration with parametric dependencies (**C3**). Furthermore, **E3** evaluates the applicability of the second contribution: "adaptive instrumentation".

As shown in Figure 10.1, the main objectives of **E3** are to evaluate the accuracy of performance prediction (**G1**) after incremental calibration (**C3**). Moreover, we measure the reduction of monitoring points according to the adaptive instrumentation for both TeaStore and CoCoME (**G2**).

E3 addresses five evaluation questions to achieve these objectives, including evaluating the accuracy of performance prediction using incrementally calibrated performance models (**EQ-1.5**), assessing the stability of accuracy over incremental evolution (**EQ-1.6**), identifying and estimating parametric dependencies (**EQ-1.7**), optimizing parametric model parameters using a genetic algorithm (**EQ-1.8**), and examining the impact of adaptive instrumentation on monitoring points reduction (**EQ-2.1**).

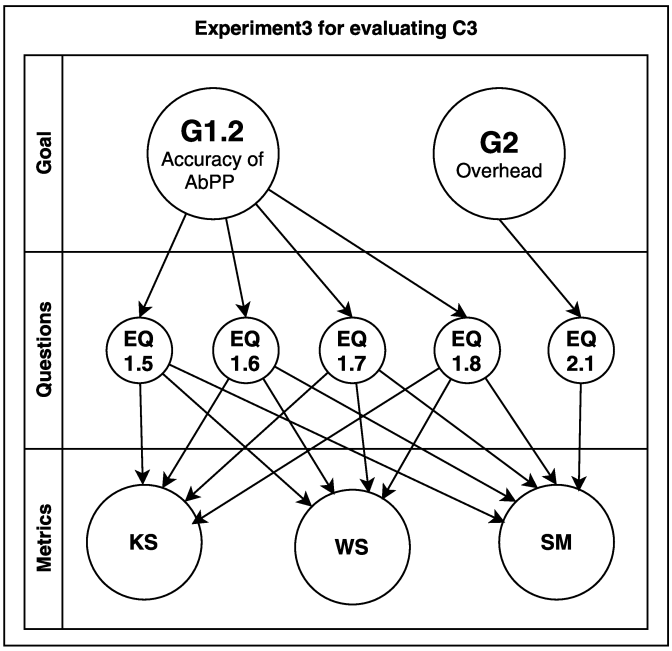


Figure 10.1: GQM plan of Experiment3.

10.1.2. Scenario

Experiment3 expects two inputs: the source code and the related PCM. Then, we construct an evolution scenario and instrument the source code accordingly. For the adaptive monitoring, we execute the source code using an input load. After that, we incrementally calibrate PCM according to the resulting measurements.

To evaluate the accuracy of the calibrated performance models, we follow three strategies that we explain in detail in the following subsections: We validate the prediction models using the measurements as a real state as we explain in Section 10.1.2.1. For more evaluation, we validate the prediction power for unseen states, see Section 10.1.2.2. To show the advantage of the parametric dependencies, we compare the accuracy of our calibration with the accuracy of another calibration approach; see Section 10.1.2.3.

10.1.2.1. Evaluation of Performance Prediction

For evaluating the resulting PCM, we apply the measurements-based approach of Böhme and Reussner [BR08], which is briefly presented in Figure 10.2. For that, we follow the following steps: We use an input load and measure the performance by monitoring the source code for a specific time (e.g., 60 minutes). The monitoring is applied at the service level without fine-grained instrumentation to prevent monitoring overhead from falsifying the evaluation results.

Then, we predict the performance of the input load by simulating the incrementally calibrated `PCM`. For the simulation, we configure the `Usage Model` with the same load and perform the simulations 100 times to get more representative results. Results from a single execution may not represent the stochastic distributions well.

Finally, we use the metrics in Figure 10.1 to quantify the difference between the measured and predicted performance.

10.1.2.2. Evaluation of performance prediction for unseen states

We predict the performance for an unseen load, i.e., a load different from that used for the calibration. For that, we adjust the `Usage Model` to predict the unseen state. Like the evaluation in the previous section, we repeat the simulation 100 times to obtain a representative prediction. Then, we measure the performance for the unseen load by monitoring the source code with coarse-grained instrumentation probes. Similar to the first evaluation scenario in Section 10.1.2.1, we compare the measured and predicted performance to quantify the accuracy.

10.1.2.3. Evaluation of parametric dependencies impact

The evaluation of the parametric dependencies is identical to the evaluation of performance prediction for the unseen state, c.f, Section 10.1.2.2. The only difference is that we repeat the evaluation steps to evaluate the accuracy of another PCM, which is non-incrementally calibrated by another approach. Then we compare the prediction accuracy of the incrementally calibrated PCM and non-incrementally calibrated one. Our comparison considers the prediction power for the current state and the unseen state. We apply **E3** on Example1, Example2, TeaStore and CoCoME.

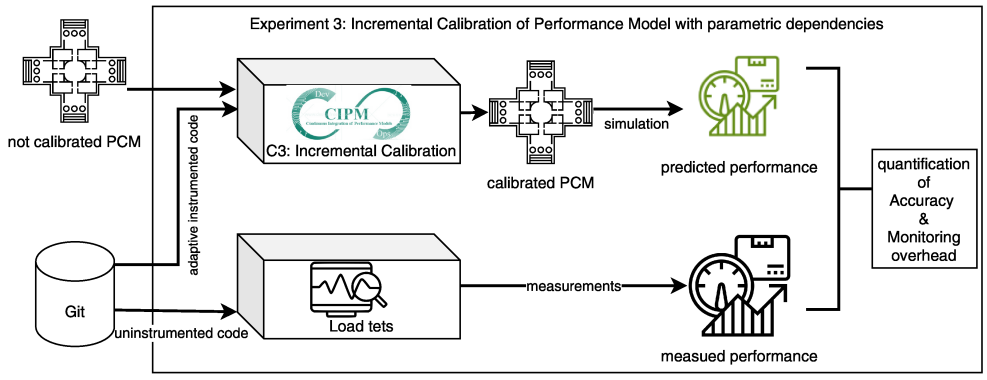


Figure 10.2: The scenario of Experiment3.

Table 10.1: The setup of Experiment3 on Example1: $M1$ represents the PCM of the source code from the first iteration, and $M2$ represents the PCM from the second one.

Iteration	Instrumentation	Load	`Repository Model`	Measurement
First iteration	full	L1	$M1_{non-inc(L1)}$	L1
	full	L2	$M1_{non-inc(L2)}$	L2
Second iteration	adaptive	L1	$M2_{inc(L1)}$	L1,L2,L3
	adaptive	L2	$M2_{inc(L2)}$	L2,L3,L3
	full	L1	$M2_{non-inc(L1)}$	L1,L2,L3
	full	L2	$M2_{non-inc(L2)}$	L2,L3,L3

10.1.3. Setup on Example1

We use Example1 to evaluate the accuracy of the incremental calibration considering different PMPs like branches. Besides, the impact of the selected load and adaptive instrumentation on the accuracy are studied according to this example.

Evolution scenario: We supposed that the example was evolved in two iterations: The highlighted part of `serviceA` shown in Figure 8.15 was evolved by the first iteration. The next iteration evolved the remaining part according to our assumption.

Setup: We instrument the source code of the first iteration fully. By the second iteration, adaptive instrumentation and full instrumentation are performed. The reason is to compare the calibration accuracy based on fully instrumented and adaptively instrumented source code.

We perform load tests based on the instrumented code and two loads: The first has a population of 1 user and no thinking time (L1). The second one has a population of five concurrent users and a think time of 5 ms. The load test is executed on a computer with one CPU core of an Intel(R) Core(TM) i7-4720HQ. The load driver calls `serviceA` 500 times with a think time after each service execution and with a random value for the input parameter a. The random value is between zero and ten. In this experiment, we deactivate the adaptive monitoring. In other words, all instrumentation probes remain active while collecting the measurements.

The resulting measurements are used as input for incremental calibration. Because we want to validate the impact of the selected load on the accuracy of the estimated PMPs, we calibrate the related `Repository Model` twice: In the first time, we use the measurements resulting from load tests on the related source code using L1. The second time, we calibrate another instance of `Repository Model` using the measurements we obtain by applying L2 on the load test.

The incremental calibration results in two `Repository Models` representing the first iteration and four PCMs representing the second iteration, cf., Table 10.1.

We apply both evaluation scenarios mentioned in Section 10.1.2: we apply the load L1 and L2 on the source code of both iterations to measure the performance. In the second step, we measure unforeseen states that we have not used before by the calibration. For that, we use Load L3 (a load driver with a population of 10 and a think time of 5 ms), Load L3' (a load driver with a population of 10 and a think time of 500 ms) and L2' (a load driver with a population of 5 and a think time of 500 ms). We start the simulation to predict the performance for all aforementioned loads. Then, we compare the simulated response times with the measured ones.

10.1.4. Setup on Example2

The objective of using Example2 is to assess the effectiveness of using the genetic algorithm to improve the precision of the incremental calibration. This is particularly relevant since Example2 involves non-linear correlations between the input data and the performance parameters.

Evolution scenario: We suppose that `serviceA` was recently added. According to our assumption, we calibrated it incrementally.

Setup: To generate the necessary calibration data, we first employ fine-granular monitoring. The `serviceA` is executed 500 times with ten concurrent users, using randomly generated arguments for the input parameters x, y, and b. The x parameter is generated from the set $[0..9]$, y from the set $[1, ..10]$, and b from [0,1]. The monitoring process, with the described setup, lasts approximately 20 minutes.

Using the monitoring data obtained, we calibrate three different `PCMs`: one with the distribution functions of the estimated performance model parameters calibrated manually, one with our incremental calibration without incremental optimization, and one with the incremental optimization of complex dependencies. Next, we evaluate the resulting `PCMs` as described in Section 10.1.2.3.

For that, we conduct simulations using the three `PCMs` to predict the response time of `serviceA` for an unforeseen usage profile, where the x parameter is a random integer from the set $[0..19]$, y parameter is a random integer from the set $[1..20]$, and b is a random boolean. We repeat the simulation 50 times for each aPM to ensure the reliability of the results.

To evaluate the accuracy of our simulation results, we utilize a coarse-grained monitoring approach to measure the response time of `serviceA` for the unseen usage profile. Finally, we compare the simulation results with the actual monitoring data using accuracy metrics.

10.1.5. Setup on CoCoME

We use CoCoME to evaluate the accuracy of the incremental calibration in comparison to other calibrations of CoCoME. The evaluation of the accuracy is also based on real monitoring as we explained in Section 10.1.2.1. However, the prediction power against real measurements is also calculated for two reference models. The first one is the initial model of CoCoME that is manually calibrated [Her+08]. Such manual calibration may not be accurate enough. Therefore, we evaluate the accuracy of incremental calibration against a reference model in addition to the initial one. The second reference model is calibrated using the mean value of the resource demand estimated by the utilization law and our estimation of the parametric dependencies.

Evolution scenario: We suppose that the `BookSale` is newly evolved and integrated into the source code. According to this assumption, we adaptively instrument the `BookSale` and monitor it. Then we incrementally calibrate it using the measurement resulting from monitoring. In the last step, we evaluate the accuracy of the performance prediction following the steps in Section 10.1.2.1.

Setup For monitoring, we use JMeter [ASB17] to configure the usage. Besides, we deploy the case study on Docker to avoid the side effect of concurrent applications on CPU utilization during the monitoring. For the simulation, we use PCM-Headless[1] to provide a REST interface to execute the `Palladio` simulation on Docker. Then we follow the evaluation scenario described in Section 10.1.2.1. More details on the **E3** configuration and the reproducibility of the results are documented on GitHub[2].

10.1.6. Setup on TeaStore

First, we use the behavioral TeaStore PCM described in Section 8.3.3.2 to evaluate the fine-grained behavior of PCM after the incremental calibration. Additionally, we adjusted the source code to train the `Recommender` after placing an order. The reason is that we want to evaluate the impact of orders' number on the training time since `train` service expects the past orders and the associated ordered items as input. During the operation, the number of orders in the database grows. Consequentially, the training time of the `Recommender` is also increased during the operation.

Evolution scenario: According to behavioral TeaStore PCM (Section 8.3.3.2), `Recommender` has a service that calculates the recommendations for a certain shopping cart ('`recommend`') and a service that derives information from the previous orders

[1] see `https://github.com/CIPM-tools/PCM-Headless`.
[2] see `https://github.com/CIPM-tools/Incremental-Calibration-Pipeline`.

('train'). As we explained in Section 3.2, there are four recommendation strategies and their related training process. In our evolution scenario, we assume that these implementations are incrementally evolved and tested over four evolution iterations. Besides the recommendation strategy is replaced over the next six evolution iterations. Consequentially, train is incrementally evolved to train the data for each recommendation strategy. Changing the recommendation strategy does not impact the preprocessing of training data, i.e., the source code corresponding to the first three actions of the train SEFF shown in Figure 8.11. Just the source code corresponding to the last Internal action 'trainForRecommender' is changed according to the recommendation strategy. Hence, the resource demand for this internal action is incrementally reestimated ten times responding to the evolution of the recommendation strategy.

The impact of parametric dependencies: We also suppose that the developer adds an enumeration parameter to enable the selection of one of the recommendation strategies. This change impacts the resource demand of trainForRecommender, which will now depend on the value of the enumeration parameter. Specifically, the resource demand will vary depending on the implementation strategy chosen.

Setup To evaluate the evolution scenario, we instrument the source code of the 'trainForRecommender' internal action. Then, we fine-granular monitor it for 15 minutes using JMeter [ASB17] [3]. The resulting measurement was to calibrate the model with the RD. To obtain reference measurements, we coarse-granularly monitor the 'train' service for another 15 minutes. In the next step, we change the implementation of the 'trainForRecommender' to be compatible with the second implementation of Recommender. To achieve this, we instrument the changed source code and calibrate the corresponding internal action with the resulting measurements. Like the first step, we monitor the train to obtain measurements representing the real state. We repeat the same procedure for the third and fourth implementations of Recommender. After that, we randomly replace the recommendation strategy with another one and reestimate the resource demand of 'trainForRecommender'. We replace the strategy six more times to calculate the accuracy for six additional iterations. During the evaluation of the incremental evolution, the database fills over time with more products. This increases the response time of the train service, which is related to the number of products in the database. After each of the ten iterations, we evaluate the accuracy following the steps in Section 10.1.2.1.

To assess the parametric dependencies as described in Section 10.1.2.3, we adjust the source code by adding an enumeration parameter that indicates which 'Recommender' is being used, and we expand the SEFF of train by incorporating the enumeration parameter. Then, we fine-granular monitor the behavior of train with a load of 20

[3] The used load and additional technical setup are on https://github.com/CIPM-tools/Incremental-Calibration-Pipeline.

concurrent users (L1). Subsequently, we use the obtained monitoring data to calibrate two models: one with parametric dependencies using our incremental calibration approach ($M_{inc(L1)}$), and one without parametric dependencies using the observed value distributions ($M_{CDF(L1)}$). To evaluate the accuracy of the resulting PCMs, we determine the response time of `train` by applying L1 on a coarse-granular instrumented source code. Then, we use the monitoring data resulting from this step to compute the accuracy metrics.

To evaluate the accuracy of performance predictions for unseen states, we predict the performance of `train` at a higher load of 40 concurrent users (L2). We establish a baseline by measuring the performance with a coarse-granular monitoring approach using the unseen load (L2). Finally, we compare the predicted performance for the unseen state L2 with the actual measurements (monitoring data from L2). It should be noted that the increased load results in longer response times for the `recommend` service, as the database populates more quickly and more orders need to be processed by the `Recommender`.

10.1.7. Results

This section provides an overview of the outcomes of **E3**, which is based on Example1, Example2, CoCoME, and TeaStore.

Example1 Regarding the parametric dependencies, CIPM was able to detect them correctly, such as the relation between the branch transition and A.VALUE. Regarding the accuracy of AbPP, the results of **E3** using the artificial example Example1 are summarized in Table 10.1. The table distinguishes between the evaluation using a known state and an unforeseen state. The second column indicates the load applied during the load test for measuring a specific state, which is also applied on `Usage Model` for simulating the same state. The third column displays the model used for the simulation, as well as the load used for calibrating it (*inc* indicates that adaptive instrumentation was applied on the source code used for generating measurements for calibration, whereas *non-inc* indicates that the source code was fully instrumented). The remaining columns show the resulting metrics.

The results of applying **E3** on Example1 suggest that measurements from lower loads can achieve more accurate calibration of performance models. This is likely due to the long wait time caused by the higher load, which may reduce the accuracy of the estimated resource demands. Additionally, the use of adaptive instrumentation may improve the accuracy of the models according to Example1, since it reduces the monitoring overhead. The accuracy of the model representing the first iteration is higher than the second one without adaptive instrumentation. The reason is that the monitoring overhead is related to the source code size. Generally, the accuracy is increased when the monitoring overhead is reduced.

Table 10.2: Results of Experiment3 on Example1, based on data from [Jäg19]. $M1$ is the PCM related to the first iteration, $M2$ is the PCM related to the second one.

	load	Model	Measured Mean	Simulated Mean	Absolute Error	Relative Error	KS	WS
Current states	L1	$M1_{non-inc(L1)}$	0.081388 s	0.080876 s	-0.000512 s	-0.63 %	0.109	0.004
		$M2_{inc(L1)}$	0.152795 s	0.157824 s	0.005029 s	3.29 %	0.177	0.005
		$M2_{non-inc(L1)}$	0.152795 s	0.160873 s	0.008078 s	5.29 %	0.233	0.008
	L2	$M1_{non-inc(L2)}$	0.284481 s	0.385267 s	0.100786 s	35.43 %	0.204	0.101
		$M2_{inc(L2)}$	0.661022 s	0.770102 s	0.109080 s	16.50 %	0.323	0.109
		$M2_{non-inc(L2)}$	0.661022 s	0.762943 s	0.101921 s	15.42 %	0.319	0.102
AbPP for unseen states	L3	$M1_{non-inc(L2)}$	0.549842 s	0.765495 s	0.215652 s	39.22 %	0.218	0.216
		$M2_{inc(L2)}$	1.302726 s	1.544499 s	0.241773 s	18.56 %	0.331	0.242
		$M2_{non-inc(L2)}$	1.302726 s	1.531873 s	0.229147 s	17.59 %	0.329	0.229
	L2,	$M1_{non-inc(L1)}$	0.107496 s	0.134979 s	0.027483 s	25.57 %	0.103	0.028
		$M2_{inc(L1)}$	0.315652 s	0.391847 s	0.076195 s	24.14 %	0.250	0.076
		$M2_{non-inc(L1)}$	0.315652 s	0.393105 s	0.077453 s	24.54 %	0.263	0.077
	L3,	$M1_{non-inc(L2)}$	0.227739 s	0.324683 s	0.096944 s	42.57 %	0.146	0.097
		$M2_{inc(L2)}$	0.871090 s	1.044384 s	0.173294 s	19.89 %	0.249	0.173
		$M2_{non-inc(L2)}$	0.871090 s	1.060309 s	0.189218 s	21.72 %	0.292	0.189

Example2 Table 10.3 compares the measured response time with the predicted response time using the non-parameterized, parametrized, and optimized models. The results are based on 100 simulations and are summarized using five statistical measures: minimum, first quartile, median, third quartile, and maximum. The measured response time had an average value of 0.825, with a range between 0.009 and 2.589. The non-parametrized and parametrized models had similar average predicted response times of 3.045 and 3.078, respectively, with a wider range of errors compared to the measured response times. In contrast, the optimized model had a significantly lower average of the predicted response time of 1.199, with a smaller range of errors compared to the other models. This suggests that the optimization of the estimated parametric dependencies improves the accuracy of the performance model.

Table 10.3: Comparison of non-parameterized, parametrized and optimized models' predictive accuracy with monitoring data from [Von+20].

Distribution	Min	Q1	Q2	Q3	Max	Avg
Monitoring	0.009	0.217	0.59	1.318	2.589	0.825
Non-parameterized aPM	**0.021**	1.576	2.857	4.369	7.151	3.045
Parametrized aPM	0.025	1.643	2.904	4.546	7.082	3.078
Optimized aPM	0.111	**0.676**	**1.222**	**1.676**	**2.232**	**1.199**

Table 10.4 presents the resulting performance metrics for the three models above. Based on the results, the optimized model outperformed both parametrized and non-parametrized models in terms of both KS-test and WS-distance. The optimized model had significantly lower KS-test statistics and WS-distance, indicated by the lowest values in all the performance metrics. This suggests that the optimized model had a better fit with the measured data compared to the other.

Table 10.4: Comparison of the predictive accuracy of non-parameterized, parametrized, and optimized model of Example2 using KS-test and WS-distance. Aggregated results based on 100 simulations from [Von+20].

Metric	Non-parameterized	Parameterized	Optimized
KS Q1	0.585	0.581	**0.278**
KS Avg.	0.595	0.592	**0.293**
KS Q3	0.608	0.601	**0.306**
WS Q1	1.527	1.535	**0.292**
WS Avg.	1.568	1.56	**0.313**
WS Q3	1.609	1.591	**0.339**

Overall, the performance metrics suggest that optimization can significantly improve the predictive accuracy of the performance models, especially in the case of existing non-linear dependencies.

CoCoME The graph in Figure 10.3 displays the CDF diagrams for the measured response time of 'Booksale' and the simulated response time, providing insight into their similarities and differences. The simulated response time distribution closely matches the measured response time distribution in the first three quartiles but deviates in the last quartile. The reason could be that the execution time of ca. 5% of the data is high due to the garbage collection, which `Palladio` does not model or consider by the simulation.

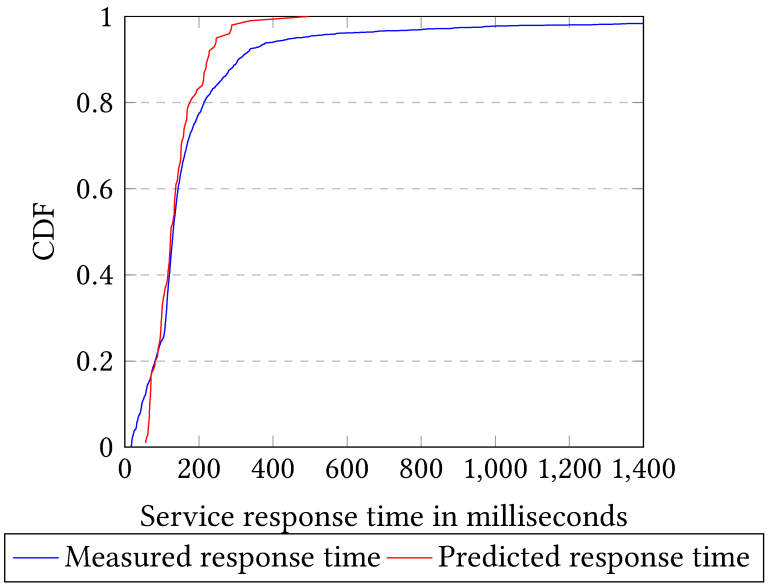


Figure 10.3: CDFs for the response time of "bookSale" service.

During a garbage collection cycle, the Java virtual machine pauses the execution of the application to identify and remove unreferenced objects from memory. This pause time can vary depending on the garbage collection algorithm being used and the size of the heap. If the heap size is too large, the garbage collection cycles may take longer and cause more significant pauses, resulting in a long execution time. Such long execution times can be considered in our case as outliers since the simulator does not model the garbage collection. Therefore, we studied the accuracy of the calibrated model twice: the first time, we compared the simulation results with all monitoring data. This provided an understanding of the model's overall accuracy, taking into account all possible scenarios, including outliers. The second time, we compared the predicted performance with the monitoring data after the elimination of the outliers. This provided insights into the model's accuracy in more typical scenarios, excluding extreme cases that cannot be simulated. To identify the anomalies in the monitoring data set, we used the 95th percentile as our threshold. Any values above this threshold are considered outliers and removed. A detailed comparison of the results of a single simulation and the monitoring data without outliers is in Table 10.5.

Table 10.5: Quartiles for the probability distributions of a single simulation and the monitoring for the "bookSale" service ignoring the outlier, based on [Maz+20].

Distribution	Min	Q1	Q2	Q3	Max	Avg
Monitoring	31ms	101ms	130ms	187ms	461ms	143ms
Simulation	56ms	97.7ms	133.5ms	176.5ms	496ms	148ms

As described in setup (Section 10.1.5), we evaluated the accuracy of incremental calibration against the original `PCM` of CoCoME [Her+08], and a reference one, which was calibrated based on utilization law. As shown in Table 10.6, the first two columns list the accuracy metrics of the initial model and a reference model, respectively. The third and fourth columns show the accuracy metrics of the incrementally calibrated models obtained in two different ways: one using a single simulation compared to all monitoring data, and the other sums up the accuracy metrics resulting from the comparison of 100 simulations results to the monitoring data without outliers. The results state that incremental calibration outperformed the two reference models in terms of prediction accuracy. Looking at the table, we can see that both metrics improved significantly by the incremental calibration, especially after the elimination of outliers.

Table 10.6: Comparison of the accuracy metrics (KS-test and WS-distance) for CoCoME's "bookSale" with two reference models based on the comparison to monitoring data with and without outliers.

	Initial Model	Reference Model	Incremental Calibration	
			One Simulation	100 Simulation without outlier
KS-test	0.6673	0.7212	**0.3156**	**0.1248- 0.1774**
WS-distance	118.1	228.1	**79.8**	**14,58- 41,24**

Additionally, the left chart in Figure 10.4 demonstrates that the KS-test values resulting from 100 simulations in comparison to monitoring data without outliers did not exceed **0.2**. Moreover, the WS-distance values among 100 simulations ranged from **14,58** to **41,24**, as depicted in the right section of Figure 10.4.

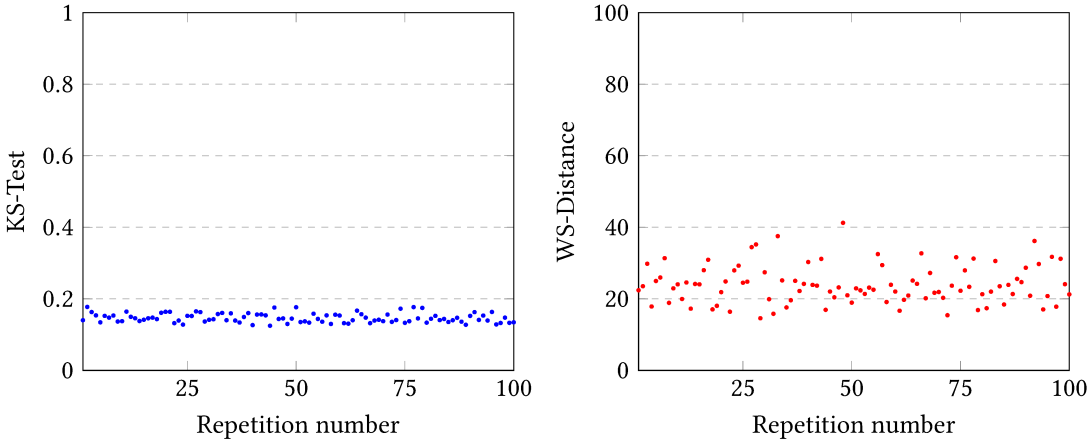


Figure 10.4: The KS-test result and the absolute WS-distance between the monitoring data and the simulation results (100 repetitions) for CoCoME's "bookSale" service.

In general, the results of **E3** on CoCoME confirm the accuracy of the incremental calibration.

Monitoring overhead: Regarding the monitoring overhead, the study found that the average monitoring overhead for the "bookSale" service was **1.31ms** for fine-grained monitoring and **252μs** for coarse-grained monitoring. Besides, the fine-grained monitoring overhead scales with the complexity of the service, while the coarse-grained monitoring has almost no influence on the execution times. Since the execution times of the services are significantly higher than the overhead of around 1*ms*, it had no drastic impact on the performance.

TeaStore The statistical summary of the KS-test and WS-distance results obtained from the comparison between the real measurements and 100 simulation outcomes are presented in Table 10.7 and Table 10.8, respectively. Each row represents an iteration after changing the recommender, and each column represents a statistical measure for the metric results, including minimum, first quartile (Q1), third quartile (Q3), maximum, and average. The results were aggregated over 100 simulation repetitions. The summary row presents the average values of the minimum, Q1, Q3, maximum, and average calculated across all iterations.

These results demonstrate the accuracy of the models across the different evolution stages, as the maximum values remain within acceptable limits.

Table 10.7: KS-tests results conducted over the iterative evolution of TeaStore.

Iteration	min	Q1	Q3	max	avg
1	0.195	0.226	0.283	0.290	0.246
2	0.183	0.220	0.274	0.287	0.239
3	0.106	0.130	0.168	0.173	0.143
4	0.128	0.136	0.168	0.184	0.150
5	0.083	0.087	0.134	0.160	0.114
6	0.139	0.144	0.184	0.192	0.162
7	0.194	0.211	0.281	0.324	0.246
8	0.153	0.199	0.238	0.277	0.217
9	0.144	0.183	0.207	0.233	0.193
10	0.179	0.199	0.234	0.257	0.217
Summary	**0.149**	**0.172**	**0.221**	**0.245**	**0.190**

Table 10.8: WS-distance results conducted over iterative evolution of TeaStore.

Iteration	min	Q1	Q3	max	avg
1	0.24	0.26	0.30	0.32	0.28
2	14.96	15.07	22.77	24.63	18.70
3	18.79	24.55	27.93	29.93	26.03
4	0.02	0.08	0.11	0.13	0.09
5	5.92	8.20	11.41	12.21	9.56
6	0.04	0.05	0.13	0.23	0.10
7	11.07	12.21	15.13	19.83	14.11
8	22.40	28.96	33.72	39.46	31.25
9	0.13	0.20	0.23	0.45	0.23
10	0.22	0.24	0.28	0.37	0.27
Summary	**7.24**	**9.48**	**12.88**	**15.53**	**11.78**

The summary row for the KS-test values in Table 10.7 shows that the minimum KS-test value is 0.083, and the maximum is 0.324, with an average of 0.191. These results indicate that there is not significant difference between the KS-test values across the ten iterations of the system's evolution. This suggests that the evolutionary process did not result in substantial changes in the prediction accuracy of the incrementally calibrated aPM.

It is worth noting that the variability observed between the simulation results and real measurements is due to the differences in resource demands associated with the various implementation strategies employed. As shown in Figure 10.5 on the left side, the KS-test values remained low throughout the evolution stages. However, there were discrepancies in the WS-distance results shown on the right side of Figure 10.5. The reason is that some recommendation strategies were incompletely implemented. As a

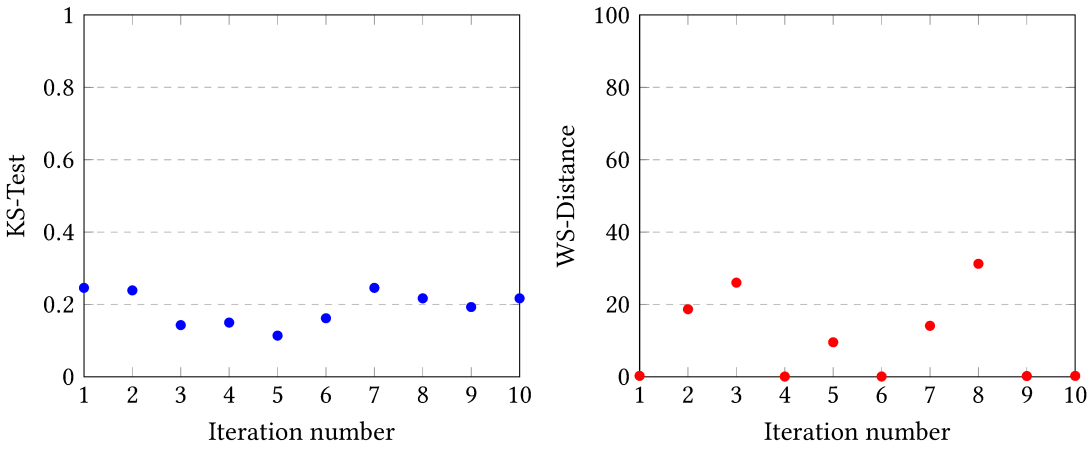


Figure 10.5: The average values of KS-test for the iterative evolution of TeaStore are presented on the left side, while the average values of WS-distance are shown on the right side.

result, their resource demands were too low, rendering their runtimes almost static and, in most cases, less than 10 milliseconds, thereby making the obtained WS-distance low, indicating high accuracy.

The impact of parametric dependencies: The findings of the second part of **E3** on TeaStore, confirmed that our calibration approach effectively identified the parametric dependencies. For instance, the estimated resource demand of `trainForRecommender` was associated with the type of recommendation implementation. Additionally, the resource demand of `trainForRecommender` for some recommendations depended on the number of orders and items in the database.

To assess the impact of the parametric dependencies, we evaluated the accuracy of the parameterized model and the non-parameterized model in predicting performance under varying loads. The results presented in Table 10.9 demonstrate that both the parameterized and non-parameterized models perform well in accurately predicting performance for a known load of 20 users. However, when it comes to predicting performance for an unseen load of 40 users, the accuracy of the non-parametric model decreases significantly compared to the parameterized model. Therefore, it can be concluded that the inclusion of parametric dependencies in the model leads to improved predictive accuracy for unseen loads. In other words, the parameterized model provides a more robust and accurate prediction of system performance under varying loads, which can help in decision-making and optimizing system performance.

Monitoring overhead: Regarding TeaStore's Recommender.train service, we measured the monitoring overhead on average to be **1.81ms** with fine-grained monitoring and **88μs** with coarse-grained monitoring. Similarly to the CoCoME case, we observe that the coarse-grained monitoring has a negligible impact on the execution times of the

Table 10.9: Comparison of the predictive accuracy of the parameterized (PM) and non-parameterized (NPM) models, aggregated over 100 simulation repetitions. The table shows the KS-test and WS-distance for the known load of 20 users and the unseen load of 40 users, based on [Maz+20].

Accuracy metrics	Known load of 20 users		Unseen load of 40 users.	
	PM	NPM	PM	NPM
KS Q1	0.1024	**0.1023**	**0.1378**	0.2352
KS Min.	0.0822	**0.0724**	**0.0975**	0.1526
KS Avg.	0.1267	**0.1239**	**0.1609**	0.2575
KS Q3	**0.1431**	0.1441	**0.1810**	0.2834
KS Max.	**0.2297**	0.2504	**0.2680**	0.3348
WS Min.	7.7451	**6.4162**	22.4151	**20.8585**
WS Q1	15.4911	**11.2484**	**33.0235**	43.5959
WS Avg.	19.1764	**16.2669**	**39.6138**	51.3531
WS Q3	22.2654	**18.2179**	**46.8370**	59.9197
WS Max.	**38.7047**	47.1203	**54.9709**	79.2013

service. However, the fine-grained monitoring overhead for this service is higher than the one observed for the "bookSale" (**1.31ms**) service. However, the impact of the monitoring overhead on the overall performance of the train service is not significant because the execution times of the service are much higher due to the increasing amount of training data in the database.

The evolution scenario demonstrates that the adaptive instrumentation can significantly reduce the required monitoring points. Instrumenting just the changed internal action, `trainForRecommender` instead of instrumenting all internal actions, saved at least **75%** of the fine-grained monitoring points.

10.1.8. Discussion

In summary, the evaluation results of performance prediction using incrementally calibrated performance models can answer the predefined evaluation question as follows:

Answering EQ-1.5 For the question **EQ-1.5** "*How accurate is the AbPP using the aPMs that are incrementally calibrated at Dev-Time?*", the evidence suggested that incrementally calibrated performance models were effective in predicting system performance under varying loads and conditions. This conclusion was based on the resulting accuracy metrics, which remained within acceptable levels across all tested cases. However, the accuracy of these results is related to the representativeness of the measurements used by the calibration, which will be discussed in more detail in Section 10.1.10. An

additional finding was that the use of measurements from lower loads consisting of one user led to more accurate calibration of performance models. This was demonstrated through the application of **E3** on Example1. When there are multiple concurrent requests to the CPU, longer waiting times can be resulted by the CPU. These waiting times form part of the response times used as the basis for estimating resource demand (as described in Section 6.6.2), which may affect the resource demand estimation. This issue is less when only a single user is making requests, leading to more accurate resource demand estimations.

Answering EQ-1.7 Regarding **EQ-1.7** "*How accurately can CIPM identify the parametric dependencies, and to what extent can the estimation of them improve the accuracy of the AbPP?*", CIPM was able to detect the dependencies. The identification of parametric dependencies has a positive impact on the accuracy of performance models. The parameterized models provided more accurate predictions of system performance under unseen states, as evidenced by Example1, Example2, CoCoME, and TeaStore.

Answering EQ-1.8 Considering **EQ-1.8** "*To what extent can optimizing the PMPs by applying the genetic algorithm improve the accuracy of the AbPP?*," the adaptive optimization of parametric model parameters has played a positive role in improving the predictive accuracy of performance models. This improvement is especially evident in scenarios involving non-linear dependencies, a conclusion supported by the outcomes observed in **E3** on Example2.

Answering EQ-2.1 Concerning **EQ-2.1** " *To what extent can the adaptive instrumentation reduce the instrumentation probes?*", the application of **E3** demonstrated that adaptive instrumentation can reduce the monitoring points, as seen in the evolution scenario on TeaStore. Furthermore, the reduction in monitoring overhead improved the accuracy of the performance models. This was evident from the higher accuracy of the performance model from the first iteration compared to that from the second iteration without adaptive instrumentation, as observed in Example1. This can be attributed to the fact that high monitoring overhead increases CPU utilization. When estimating CPU utilization for monitored internal actions, it is important to account for the utilization caused by monitoring overhead. A greater monitoring overhead can, therefore, have a negative impact on the accuracy of the estimated resource demands, making the reduction of this overhead crucial for improving model accuracy.

Answering EQ-1.6 "*Is the accuracy of AbPP using the incrementally calibrated aPM affected by the incremental evolution?*" In response to the question **EQ-1.6** "*Is the accuracy of AbPP using the incrementally calibrated aPM affected by the incremental evolution?*", the evidence indicated that the incrementally calibrated performance models consistently

maintained their accuracy throughout the evolutionary process. This was evident in the TeaStore results where the values of the accuracy metrics consistently stayed within acceptable limits, indicating that the incremental evolution did not bring about significant changes in the prediction accuracy of the incrementally calibrated aPMs. This finding, corroborated by the TeaStore results, underscores the stability of these models, affirming their dependability for continuous application in performance prediction throughout system evolution.

10.1.9. Limitation

During **E3** the required adaptive instrumentation was applied manually. This was because the Git history for the assumed evolution scenarios was not available, and the CI-based strategy had not been fully implemented at the time of evaluation. Despite this, the manual evaluation still utilized the defined measurements metamodel, which is also employed by the automatic model-based instrumentation. This ensured a comparable outcome, regardless of whether the adaptive instrumentation was applied manually or automatically. Therefore, the results of **E3** are expected to be consistent and unaffected by the method of instrumentation application. In addition, the accuracy of the automatic adaptive instrumentation was evaluated in Example1.

10.1.10. Threats of Validity

Similar to Example1 and **E2**, in **E3** we also need to discuss the threats to the validity of the experiment. The validity of **E3** can be evaluated based on the categories of threats, as defined by [Woh+12]:

- External validity: Similar to Example1 and Example2, the selection of cases presents a potential threat to validity. This is because the results obtained from these cases may not be broadly representative. To mitigate this threat, we selected CoCoME and TeaStore, in addition to two artificial examples. Both CoCoME and TeaStore have been extensively used in various research fields, including performance prediction [Reu+19; Kei+21; Gro+19; Kis+18]. By employing a diverse set of case studies, we aimed to minimize the risk of non-representative results and enhance the external validity of our findings.
- Construct validity: There are two threats to construct validity:
- Metric selection: The selection of metrics used within the evaluation is another potential threat to validity. To assess distribution comparisons, we utilized the WS-distance, the KS-test, and conventional statistical measures. These metrics, which have been employed in related studies [Hei+17; Eis23; MP10], were chosen due to their proven effectiveness. By combining these metrics, we sought to

minimize the risk of a skewed interpretation of the evaluation results. The JC, another metric utilized in this study, has also been validated in related work [Hei20], thereby further enhancing the reliability of our findings.

Data generation: In **E3**, we relied on data obtained directly from system execution. However, when using data derived directly from system execution, the quality of monitoring events is crucial. To ensure high-quality monitoring data, we use the `Kieker` framework, enhanced with extensions implemented in previous projects [HWH12b; Maz+20]. These extensions help to ensure the accuracy and reliability of the monitoring data, thereby reducing potential sources of error.

- Conclusion validity: To minimize personal bias and subjectivity when interpreting the resulting accuracy metrics, we evaluated the performance predictions not only against the measurements but also against predictions resulting from an alternative calibration method. To ensure a fair comparison, we used the same simulator (`Palladio`) across all calibration methods. Additionally, we used the same measurement data set for the calibration process and a separate, distinct data set for validating the models calibrated differently.

10.2. Summary

To summarize, **E3** confirmed the accuracy of incrementally calibrated performance models across system evolution at Dev-time.

Moreover, the results showed the capability of incremental calibration in identifying the parametric dependencies. This ability enhanced prediction accuracy, particularly in scenarios with non-linear dependencies or previously unobserved system states.

Besides the confirmed accuracy of performance predictions derived from incrementally calibrated aPMs, **E3** also demonstrated the impact of adaptive instrumentation on monitoring overhead. The findings indicate that the use of adaptive instrumentation can reduce the monitoring overhead and also concurrently improve the accuracy of the aPMs.

11. Validation of Consistency Preservation at Operation Time

In this chapter, we describe two experiments related to the Ops-time. The first one validates the Ops-time calibration and self-validation (Section 11.1), while the second experiment validates the scalability at Ops-time (Section 11.2). The experiments' results are summed up in Section 11.3.

11.1. Experiment 4: Updating Performance Models during Operation

This experiment aims to evaluate the process of updating aPM at Ops-time. In other words, Experiment4 (**E4**) evaluates the fourth and fifth contributions of this dissertation: Ops-time calibration and self-validation. **E4** was conducted in the context of [Mon20] and was repeated with a similar configuration for [Mon+21].

In the following sections, we introduce the goal of **E4** and its scenario in Section 11.1.1 and Section 11.1.2. Then, we describe how we designed **E4** for the case studies CoCoME and TeaStore in Section 11.1.5 and Section 11.1.6. The results and discussion of **E4** are presented in Section 11.1.7 and Section 11.1.8.

11.1.1. Goal

Experiment4 aims to assess Ops-time calibration (**C4**), and the self-validation (**C5**) of CIPM approach. For that, **E4** has several concrete goals as illustrated in Figure 11.1. First, it aims to assess the accuracy of aPMs (G1.1) and their performance predictions (G1.2) at Ops-time. Second, **E4** aims to measure the required monitoring overhead at Ops-time (**G2**). To achieve these objectives, **E4** addresses three evaluation questions: The first question (**EQ-1.4**) evaluates the accuracy of aPMs updated by **C4** and **C5** after system adoptions. The second question (**EQ-1.9**) focuses on evaluating the accuracy of performance prediction using updated aPMs. Lastly, the third question (**EQ-2.2**) examines whether adaptive monitoring can reduce the overall monitoring overhead at Ops-time.

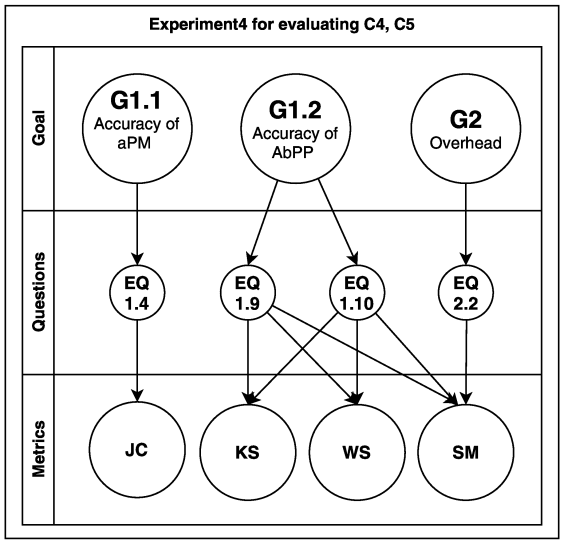


Figure 11.1: GQM plan of Experiment4.

11.1.2. Scenario

E4 utilizes a fully instrumented source code of the system and its corresponding uncalibrated aPM as input to perform the validation under worst-case conditions, see Figure 11.2. **E4** operates and monitors the system while keeping the aPM up-to-date using the CIPM approach: The self-validation (**C5**) manages monitoring overhead by activating or deactivating monitoring points, and the Ops-time calibration (**C4**) continuously updates and calibrates the aPM based on collected monitoring data. This enables the first evaluation scenario: evaluating the accuracy of performance predictions along the Ops-time (G1.2).

In the second evaluation scenario, **E4** simulates adoption scenarios of the running system. For that, the scenario generator applies one of predefined change scenarios, such as altering deployment, resource environments, or system composition. These changes during the operation trigger CIPM to update aPM accordingly. This allows **E4** to evaluate the accuracy of the different parts of aPM after adopting the running system (G1.1) and ensure that the performance prediction remains accurate (G1.2).

During the two evaluation scenarios, **E4** measures the monitoring overhead for G2. In the following, we provide a detailed description of the aforementioned evaluation scenarios in Section 11.1.3 and Section 11.1.4 respectively.

11.1.3. Evaluation of Performance Prediction

E4 evaluates the accuracy of AbPP along the operation. For that, **E4** stores the aPM and the related monitoring data at different points in time. Then, **E4** runs the simulation to evaluate the accuracy of aPM at the considered points of time. Similar to

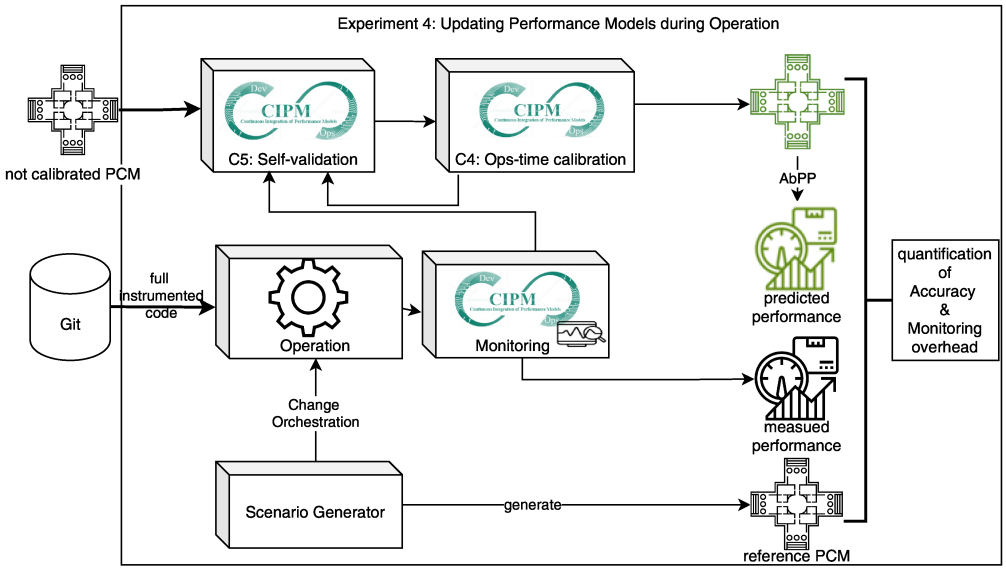


Figure 11.2: The scenario of Experiment4.

E3 (Section 10.1.2.1), **E4** compares the simulation results with the monitoring data to assess the accuracy. For that, **E4** follows two approaches: Forward prediction involves comparing simulation results with monitoring data collected by an additional run of the system, which assesses how well the updated aPM predicts future scenarios. Conversely, backward prediction involves assessing how well the aPM reproduces past observations by comparing simulation results with monitoring data collected chronologically to the update process. As shown in Figure 11.1, the evaluation employs the same accuracy metrics as used by **E3**: KS-test, WS-distance and SMs.

11.1.4. Evaluation of Model Accuracy

To evaluate the accuracy of aPM that CIPM updates after system adoptions, **E4** compares the updated aPM with a generated reference aPM. For that, **E4** describes a set of predefined change scenarios that can be executed. The scenario generator shown in Figure 11.2 can select a specific number of these scenarios. To apply a specific scenario, the scenario generator utilizes the change orchestration component to adjust the running system based on the selected changes. Besides, the scenario generator is aware of the executed change and can also determine how the aPM should be modified after applying this scenario. Hence, the scenario generator adjusts a reference model to represent the running system after applying the selected scenario.

The monitoring of CIPM is designed to detect the changes at Ops-time, which allows the Ops-time calibration to update the aPM accordingly.

Finally, **E4** compares the updated aPM to the reference model to detect any deviations. For that, **E4** uses JC to quantify the accuracy of aPM as described in Section 8.1.3.1. In our implementation, the JC for the following parts of PCM is calculated: `System Model`, `Allocation Model` and `Resource Environment Model`. The following paragraph describes the change scenarios defined in the change generator.

Changes Scenarios The change orchestration component simulates the changes occurring during Ops-time. For that, it interacts with the test suite responsible for conducting load tests on the application. By adjusting these tests, it can simulate various changes in usage and load.

In addition to test adjustments, the change orchestration component utilizes Docker[1], a containerization platform, to manipulate the virtual resource environment. Docker enables the creation, management, and isolation of containers that encapsulate specific components of the system. Consequentially, the change orchestration component can dynamically modify the deployment, composition and environment of the system.

By coordinating test adjustments and Docker-based container management, **E4** can generate the following changes to evaluate the accuracy of PCM parts:

- Changes affecting the `Resource Environment Model`: To modify the resource environment, the change orchestration component of **E4** starts up a new container or shuts down an existing one. Before shutting down a container, it ensures that no component is running on it.
- Changes affecting the `Allocation Model`: The change orchestration component of **E4** impacts the allocation by the migration of components. In this case, a microservice is removed from its current Docker container and deployed on another existing container.
- Changes affecting `System Model` and `Allocation Model`: Replication, de-replication, and replacement of microservice instances during the Ops-time impact both the system composition and the allocation. Replication involves creating additional instances of a component; for example, creating a new instance of a microservice similar to what the load balancer does when it distributes the workload and improves scalability. De-replication, on the other hand, involves removing a replicated instance.

  In addition to replication and de-replication, the instances of a microservice can be replaced with different implementations or versions during the Ops-time process. Changing the Recommender implementation of TeaStore at Ops-time is an example of replacing a microservice (component) instance, which directly impacts the system composition.

[1] See `https://www.docker.com/`

- Changes affecting `Usage Model`: To change the usage behavior, the change orchestration component utilizes different load test scenarios. An example of TeaStore is varying the numbers of the purchased products, which directly impacts system performance. Besides, the workload can be changed by modifying the number of users in the load test. In TeaStore example, workloads of 20, 30, or 40 users are applied, which affects the database fill-up and service response time.

In the following sections, we describe the experiment setup for CoCoME and TeaStore. Additional details are available in [Mon20]. It is important to note that the setup details and results differ slightly when **E4** is repeated with a similar configuration, as outlined in [Mon+21; Maz+25].

11.1.5. Setup on CoCoME

The input for **E4** consists of the manually modeled PCM of CoCoME and a fully instrumented copy of the cloud variant of CoCoME. The instrumentation of CoCoME is done semi-automatically using Monschein's approach [Mon20]. The monitoring is performed using Jmeter, which applies a load test to CoCoME on the same computer. The monitoring setup includes a sliding window for 15 minutes and a sliding window trigger of 5 minutes.

The monitoring data are accessed from the calibration process, which runs on a separate computer. The self-validation process evaluates the accuracy of the calibrated aPM using the `Palladio` simulation, which is executed on a third computer. The self-validation is configured to consider the model to be accurate when the KS-test value is below 0.2 and the WS-distance value is less than 20. After the calibration, we evaluate the forward prediction. We apply a new load test to evaluate the resulting aPM with unseen monitoring data. Additionally, we evaluate the accuracy of aPM along **E4** to demonstrate how it would be affected throughout the experiment, where more monitoring data is collected and used for calibrating aPM.

The experiment lasts for 3 hours. **E4** is also repeated 25 times, to obtain more meaningful data. We aggregate the results of these repetitions using the median metric of SMs. Since the accuracy of different perspectives of CoCoME PCM is investigated in another work [Hei20], **E4** on CoCoME case concentrates only on evaluating the performance prediction (Section 11.1.3). More detailed information on the configuration of **E4** for CoCoME can be found in [Mon20, P. 84].

11.1.6. Setup on TeaStore

The input of **E4** is the coarse-grained PCM of TeaStore (cf. Section 8.3.3.1) in addition to the related source code. Similar to CoCoME, the setup of **E4** is also distributed across three computers: one for application and monitoring, one for calibration, and one for

simulation. However, TeaStore as a microservices application is further distributed across different Docker containers. Each microservice of TeaStore runs within its container. The change orchestration component can apply modifications to the TeaStore system running on these virtualized containers. The applied dynamic adjustments can affect the composition, deployment, and operating environment of the system, cf. Section 11.1.4. Hence, the **E4** on TeaStore is carried out in two stages: the first without system adjustments and the second with adjustments applied during system operation.

In the first stage, **E4** is executed for two hours and repeated 25 times. At this stage, the change orchestration does not apply any change. The backward performance prediction is continuously evaluated using monitoring data gathered during the experiment. However, in the second stage of **E4**, the evaluation of both forward and backward prediction is performed.

The second stage of **E4** assesses how aPM is updated and adjusted in response to changes during operations. Therefore, the scenario generator, as depicted in Figure 11.2, simulates frequent changes at a rate of one change every 5 minutes. To capture these changes, monitoring data is processed more frequently using smaller sliding windows, compared to CoCoME, with a sliding window size of approximately 5 minutes and a trigger interval of 3.

The experiment execution lasts about 150 minutes and is repeated 10 times. To be noted, this stage is also executed with a similar configuration and more repetitions (50 times) during the master's work of Monschein [Mon20].

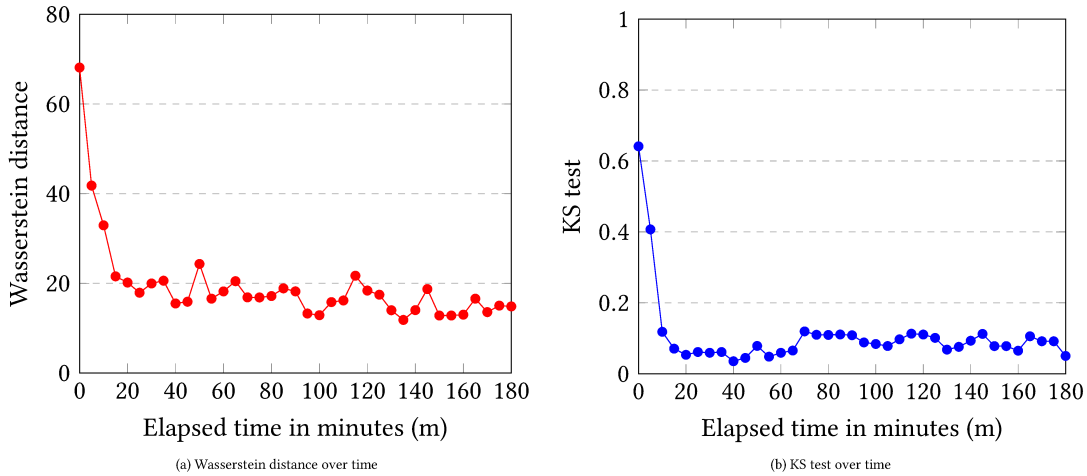
To assess the accuracy of performance prediction, the second stage of **E4** on TeaStore utilizes monitoring data collected up the evaluation time point for evaluating backward prediction. Additionally, monitoring data that are collected after the experiment are used to evaluate forward prediction for future unseen states.

E4 also analyses the overall monitoring overhead over Ops-time in the worst-case scenario, where the source code is fully instrumented. Monitoring overhead is calculated every 5 minutes by summing up the overhead from the previous 5 minutes. It is important to consider that certain parts of the monitoring, such as observing resource utilization, are independent of the monitoring granularity.

11.1.7. Results

This section presents the results of **E4** based on CoCoME and TeaStore.

Results on CoCoME The result of **E4** on CoCoME regarding the accuracy of performance prediction was quantified using WS-distance KS-test and SMs. The evaluation of the AbPP accuracy along the update process resulted in the charts shown in Figure 11.3. Each of them shows the measured accuracy using one of the accuracy metrics, plotted against the elapsed time throughout the experiment.



(a) Wasserstein distance over time

(b) KS test over time

(c) Distance between conventional statistical measures over time

Figure 11.3: The results of Experiment4 on the CoCoME based on [Mon20].

On the chart (A) of Figure 11.3, the initial measured WS-distance value is relatively high, indicating a larger discrepancy between the predicted and observed performance because the aPM was not calibrated. However, as time progresses, the WS-distance value decreases, indicating an improvement in the accuracy of performance prediction.

Around 15 minutes, the WS-distance value reaches a level close to 20, which indicates a relatively accurate performance prediction at that time. Subsequently, the WS-distance values remain below 20 with minor fluctuations, indicating a sustained level of accuracy in the performance prediction.

A similar observation is shown in (Figure 11.3 (b)). The KS-test values show a significant drop around the tenth minute, indicating a close match between the predicted and observed performance distributions. Subsequently, the KS-test test values remain consistently low, suggesting accurate performance predictions throughout the experiment. This demonstrates the reliability of **E4** on CoCoME in accurately predicting the system's performance. The values of conventional statistical measures in (Figure 11.3 (c)) also confirm the improvement in accuracy.

The improvement in accuracy over time can be explained by the accumulation of more data and the continuous learning from monitoring data. As the pipeline receives more data during the operation of the system, it gains a better understanding of the performance characteristics of the application. Additionally, the repository transformation mechanism uses this data to refine and adjust the performance models, leading to more accurate predictions.

The comparison between CDF of the predicted response time of the bookSale service from a random simulation of the updated aPM and the CDF of the measured response time of bookSale is presented in Table 11.1 and visually represented in Figure 11.4. It can be inferred that the distributions are close to each other, indicating the accuracy of AbPP. The KS-test and WS-distance values further validate this accuracy, with a KS-test value of 0.076 and a WS-distance distance of 12.38. These findings affirm the accuracy of the Ops-time calibration and self-validation process.

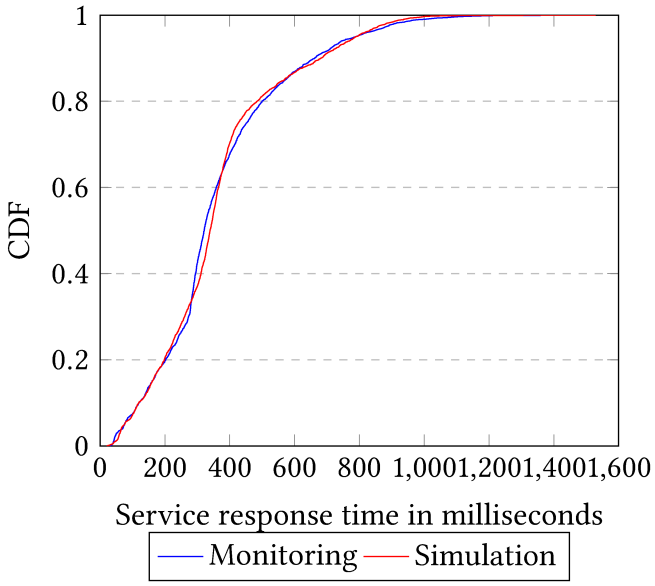


Figure 11.4: CDF of response time of “bookSale” service, from [Mon20].

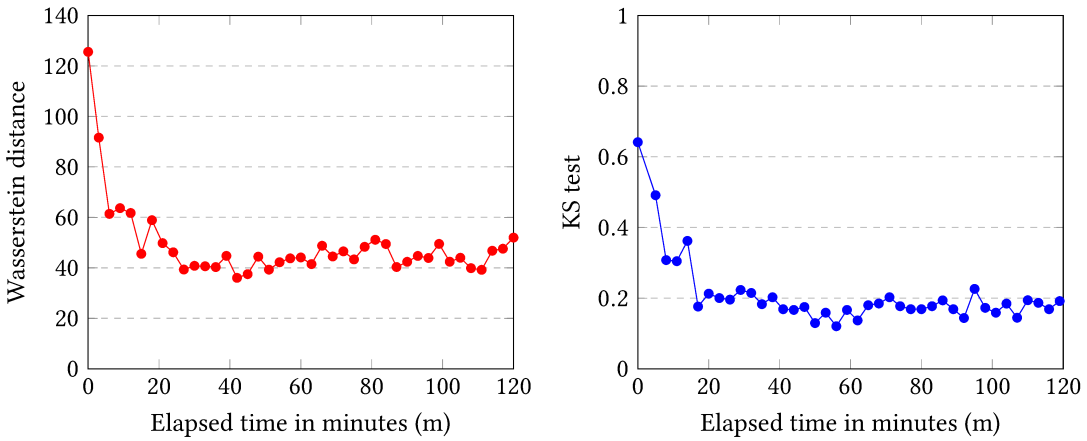
Table 11.1: Quartiles for the randomly selected probability distributions of a single simulation and the monitoring for the "bookSale" service, from [Mon20].

Distribution	Min	Q1	Q2	Q3	Max	Avg
Monitoring	29ms	240ms	322ms	453ms	1521ms	364ms
Simulation	21ms	229ms	339ms	427ms	1529ms	363ms

Monitoring Overhead: During **E4**, the self-validation manages monitoring overhead by activating monitoring points for inaccurate PMPs and deactivating them after accurate calibration. In the case of CoCoME, all fine-grained monitoring points were deactivated after 25 minutes, corresponding to five executions of the pipeline since the start of **E4**. This resulted in approximately a 29.23% reduction in monitoring points. This decrease in the number of monitoring records reduces the load on the pipeline as well as monitoring overhead.

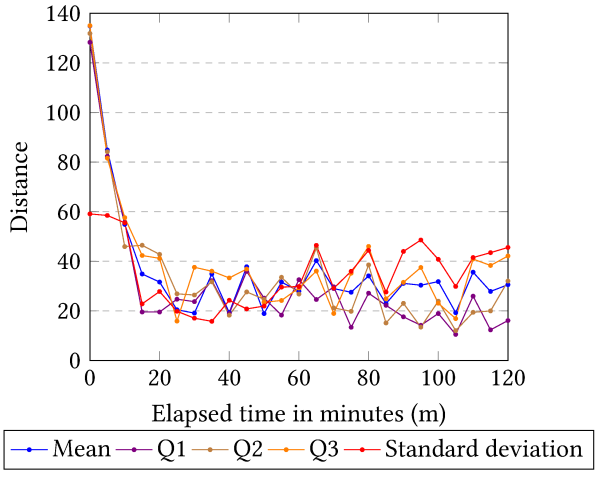
Results on TeaStore The results of the first stage of **E4** confirm the finding of **E4** on CoCoME. The Ops-time calibration and self-validation were able to improve the accuracy of the performance prediction over time. Figure 11.5 shows the improvement in the accuracy metrics over the execution time of Experiment4.

The results of the **E4** second stage demonstrate the accuracy of the updated PCM after applying adjustments during system operation, see Table 11.2. The minimum resulting JC is used as a measure of the models’ accuracy. The results are grouped according to the types of applied changes, including (de-)/allocation, (de-)/replication, migration, system composition, and workload.



(a) Wasserstein distance over time

(b) KS test over time



(c) Distance between conventional statistical measures over time

Figure 11.5: The results of the first stage of Experiment4 on the TeaStore based on [Mon20].

The results show that for all change types, the models achieved a minimum JC of 1.0 for the `System Model`, `Allocation Model` and `Resource Environment Model`. This indicates that CIPM captured the simulated changes and updated the models accordingly. The resulting JC of 1.0 confirms that the updated models successfully reflect the changes introduced by the adaptation scenarios.

Table 11.2: Model accuracy during the second stage of Experiment4 that simulates adaptation scenarios on TeaStore.

Change Type	**Minimum Jaccard Index**		
	System	Allocation	Resource Environment
(De-)/Allocation	1	1.0	1.0
(De-)/Replication	1.0	1.0	1.0
Migration	1.0	1.0	1.0
System Composition	1.0	1.0	1.0
Workload	1.0	1.0	1.0

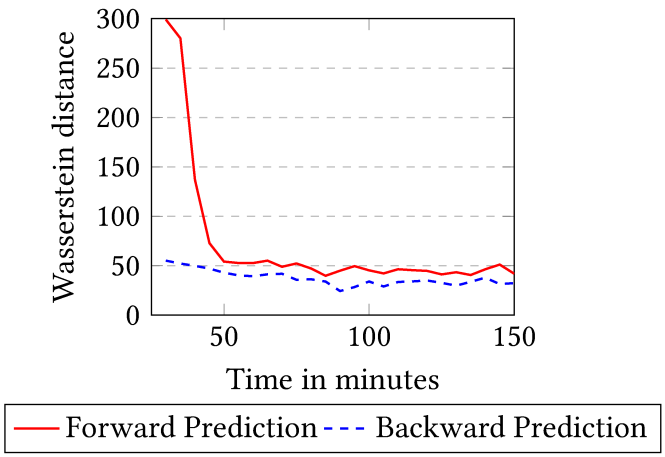


Figure 11.6: Median Wasserstein distance of the forward and backward prediction for a model derived at a given point in time - regarding the response times of TeaStores *confirmOrder* service, from [Mon+21].

In the second stage of Experiment4, the accuracy of performance prediction is also evaluated as described in Section 11.1.3. Figure 11.6 shows the median WS-distance over time for both the forward and backward predictions of the response times of `confirmOrder`.

During the initial stages of evaluation, the resulting WS-distance values are high, along with a noticeable difference between forward and backward predictions. This can be attributed to the limited amount of monitoring data available during the model's calibration. The accuracy of predicting future scenarios (forward prediction) is more significantly affected by this limitation compared to predicting past ones. However, as the evaluation progresses and more monitoring data becomes available, the model

gradually improves its accuracy. This is reflected in a decrease in the WS-distance value for forward prediction, bringing it closer to the values of the backward prediction. Then, it stabilizes at a level close to the backward prediction.

Furthermore, the results indicate that the models successfully capture the parametric dependency between response time and the number of orders in the database. The PMPs in **E4** are appropriately calibrated without the genetic algorithm optimization since it is explored in more detail in **E3**, as illustrated in Section 10.1.4.

Table 11.3: Aggregated accuracy metrics related to the second step of Experiment4 on TeaStore [Mon+21].

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

The validity of the observations from Figure 11.6 can be further supported by considering additional metrics like KS-test and SMs, see Table 11.3. In forward prediction, the KS-test values vary between 0.125 and 0.168, with a mean of 0.199 and a standard deviation of 0.131. In backward prediction, these values are generally lower, ranging from 0.098 to 0.121, with a mean of 0.114 and a smaller standard deviation of 0.031. These findings align with the observations made from the WS-distance analysis.

The mean distance between predicted and actual response times is approximately 61.573 ms for forward prediction and approximately 23.329 ms for backward prediction. These values provide insights into the accuracy of predictions relative to the average response time of the "confirmOrder" service, which was approximately 1000 ms.

It is important to note that the initial high values at the experiment's start significantly impact the average and standard deviation of the forward prediction metrics.

In summary, all the metrics demonstrate that the updated models capture the observed behavior. Furthermore, updating the models based on monitoring data, allows them to predict the performance of unobserved system states accurately.

Monitoring Overhead: The analysis of monitoring overhead is closely tied to the period, within which we observe the monitoring overhead (5 minutes in our case). Given that **E4** is repeated several times for more reliable results, we compute the median of the resulting monitoring overhead by each execution to mitigate outliers and measurement errors. We consider the second part of **E4** on TeaStore since it is more complicated due to the simulated Ops-time changes.

In Figure 11.7, the median monitoring overhead over time is shown when considering five-minute intervals. The graph illustrates the elapsed time in minutes on the x-axis and the monitoring overhead in seconds on the y-axis.

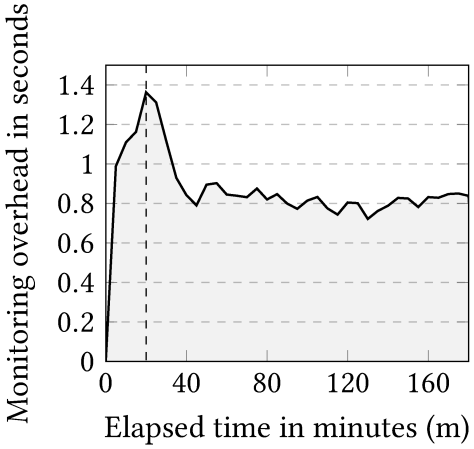


Figure 11.7: Median of the arising monitoring overhead over time when considering five minute intervals, adjusted from [Mon+21].

Initially, the monitoring overhead started to gradually increase due to the activation of the fine-grained monitoring for calibration of the `Repository Model`. At approximately the 20-minute, indicated by the dashed line, the monitoring overhead was maximal, where all monitoring points were activated. Then, a significant change in the monitoring overhead appears. It started with an overall downward trend with some fluctuations. The reduction in monitoring overhead from a maximum value of 1.362s to an average of 0.833s can be attributed to the deactivation of fine-grained monitoring points that the self-validation achieved, which leads to a more efficient monitoring process. This reduction represents approximately 38.86% decrease in monitoring overhead.

It is important to consider that the monitoring overhead includes monitoring independent of the granularity, such as monitoring resource utilization. Therefore, even though the fine-grained monitoring was deactivated, the monitoring overhead is still persistent but at a lower level and stabilizes over time.

An analysis of monitoring points during **E4** on TeaStore with a similar configuration [Mon20] also indicates a reduction of approximately 25.34% in monitoring points, starting after 20 minutes of **E4**. This finding aligns with the previously discussed results.

Additional analysis of monitoring overhead for the "confirmOrder" service revealed an absolute overhead of 2.8551 milliseconds for fine-grained monitoring and 949.59 microseconds for coarse-grained monitoring. Considering an average response time of 500 milliseconds for the "confirmOrder" service, the relative overhead was found to be 0.571% for fine-grained monitoring and 0.19% for coarse-grained monitoring, indicating the proportion of monitoring overhead to the total execution time of the service. It

can be concluded that neither fine-grained nor coarse-grained monitoring significantly impacts the service's response time. Comparing the two monitoring types reveals that coarse-grained monitoring results in up to 66.741% less overhead.

11.1.8. Discussion

Answering EQ-1.4 Regarding **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?*", the results confirm the proficiency of Ops-time calibration in precisely updating aPMs in response to actual system changes. As a result, aPMs represent the current state of the running software system. Additionally, the accuracy in updating the parts of the aPM (`System Model`, `Resource Environment Model`, and `Allocation Model`) contributes to the reliability in predicting system performance within dynamic contexts. It should be noted that the detection and the application of changes are related to the window size and trigger time of the pipeline. The consistency relation at Ops-time (R_{Ops}) is tolerant with the trigger time.

Answering EQ-1.9 In this paragraph, we provide an answer to **EQ-1.9** "*How accurate is the AbPP using aPMs that are incrementally updated at Ops-time?*". The self-validation mechanism proves its effectiveness by successfully detecting inaccuracies in the aPM and identifying inaccurate PMPs. These inaccuracies are then recalibrated through Ops-time calibration, ensuring that the aPMs accurately represent the observed behavior.

It is important to note that, during **E4**, the aPMs were uncalibrated. Additionally, only optimization based on linear regression was performed by the Ops-time calibration, and the genetic algorithm was not triggered. However, the results suggest that even without incremental calibration at Dev-time nor optimization with the genetic algorithm at Ops-time, the accuracy of the AbPP is increased over time by learning from more monitoring data and stabilizes at an acceptable level.

The use of the genetic algorithm, particularly when aPMs are calibrated at Dev-time and updated at Ops-time, is anticipated to result in even more accurate performance predictions. Unfortunately, due to time constraints, we were unable to explore the trade-offs between the accuracy achievable by triggering the genetic algorithm at Ops-time and the impact of its computational overhead on the scalability of the calibration process. However, this aspect represents an opportunity for further investigation and refinement in future work.

Answering EQ-1.10 The second part of **E4** on TeaStore provides insights for **EQ-1.10** "*What impact do Ops-time changes have on the accuracy of the AbPP?*". The aggregated accuracy metrics in Table 11.3 illustrate that the calibrated models were able to

predict the performance with a degree of accuracy close to calibrating the aPM without changes at Ops-time. However, the later calibration achieves an acceptable accuracy degree earlier than those calibrated with monitoring data collected during various types of changes at Ops-time. The improvement in the accuracy is seen after twenty minutes (Figure 11.5), whereas fifty minutes (Figure 11.6) were required for model calibration when changes were applied during Ops-time.

Considering the median value of experiment repetitions in Table 11.3 does not fully depict the impact of changes on accuracy. Monschein empirically studied this impact in his masterwork [Mon20] by calculating the loss of accuracy before and after changes. His findings suggest that only changes in load and usage can reduce accuracy compared to other changes. This could be attributed to the simulation results being closely tied to the applied load and usage, as reflected in `Usage Model`. Such changes in workload and usage can push the system into an overload situation. This is a common challenge across approaches, as accurately predicting the performance behavior of modern systems is extremely difficult due to numerous influencing factors like the operating system, scheduling strategy, and hardware. The accuracy of AbPP will be improved in the next calibration slides, where more monitoring data is processed and models are accordingly updated.

Answering EQ-2.2 Regarding **EQ-2.2** "*To what extent does the adaptive monitoring at Ops-time help to reduce the monitoring overhead?*", the results indicate that approximately a quarter of the monitoring points were deactivated within a maximum of 25 minutes of executing **E4** on CoCoME and TeaStore.

The evaluation of monitoring overhead during system adoption also underscores the success of self-validation in efficiently reducing overall monitoring overhead. Despite fully instrumented source code, the adaptive monitoring approach significantly minimizes the overhead and deactivates all fine-grained points after ca. 50 minutes, reducing about 39% of the monitoring overhead.

11.1.9. Threats of Validity

Similar to previous experiments, we consider the potential threats to the validity of Experiment4. Again, we examine the threat categories presented by [Woh+12]:

- External validity: Like previous experiments, we selected two cases (CoCoME and TeaStore) used in similar research contexts [Reu+19; Kei+21; Gro+19; Kis+18]. Indeed, there is still a threat that these cases do not fully represent the whole expected scenario.
- Construct validity: Similar to **E3**, **E4** faces construct validity threats such as mono-method bias and validity threats to used data, see Section 10.1.10. We

addressed metric selection by using a combination of WS-distance, KS-test, and SMsto prevent misinterpretation. During **E4**, we used monitoring data from system execution. Thus, there is a validity threat to the accuracy of monitoring data. External factors, such as the operational environment and concurrent processes running in parallel, can introduce noise and affect the accuracy of measurements. Therefore, Docker images were used to isolate the execution, monitoring, and simulation environments. This approach helps minimize interference from external processes and provides a controlled environment for monitoring and data collection.

We acknowledge that the simplified modeling of the load balancer may reduce the accuracy of performance predictions. This simplification arises because we do not model the self-adaptive system with sufficient fidelity. An accurate model should account for the dynamic nature of self-adaptive systems, which typically involve complex interactions and conditions under which adaptation occurs. However, relying on a manually defined load-balancing logic can potentially lead to deviations between predicted and actual system performance.

- Conclusion validity: To mitigate potential personal bias and subjectivity in interpreting accuracy metrics, we evaluated performance predictions against measurements and compared them with predictions derived from an alternate calibration method. A consistent simulator (`Palladio`) was used across all calibration techniques, and the same measurement dataset was used for calibration, while a separate and distinct dataset was used for validating differently calibrated models.

  Besides, the experiment was repeated several times to obtain more meaningful results and avoid potential biases or anomalies associated with a single trial.

11.2. Experiment 5: Updating Performance Models with Increasing Monitoring Data

This experiment assesses the scalability of Ops-time calibration with self-validation at Ops-time. Synthetic monitoring data is generated and utilized as input for each specific type of Ops-time calibration. **E5** was conducted in the context of [Mon20; Maz+25].

In the following sections, we introduce the objectives and scenario of **E5** in Section 11.2.1 and Section 11.2.2. Next, we describe the design of **E5** for each type of the Ops-time calibration in Section 11.2.3, Section 11.2.4, Section 11.2.5, and Section 11.2.6. Subsequently, we present the results, engage in a discussion, and highlight limitations of **E5** in Section 11.2.7, Section 11.2.8, and Section 11.2.9 respectively. Finally, we address potential threats to the validity of our findings in Section 11.2.10.

Finally, we provide a summary of the outcomes of both **E4** and **E5** in Section 11.3.2.

11.2.1. Goal

The goal of **E5** is to evaluate the applicability of CIPM in terms of scalability at Ops-time (**G3**), see Figure 11.8. Particularly, we analyze how Ops-time calibration and self-validation work when dealing with more and more monitoring data. This study is important because we want to make sure that calibration during operation does not affect the running system, even when processing a significant amount of monitoring data. Being able to handle a growing amount of monitoring records is key for making CIPM applicable to real-world situations.

So, **E5** delves deeply into how scalable CIPM is at Ops-time, checking its ability to calibrate different parts of aPM using increasingly generated synthetic monitoring data. As a result, **E5** can provide answers for the scalability questions, addressing the impact of escalating amounts of monitoring data on the calibration processes of `Repository Model` (**EQ-3.2.1**), `System Model` and `Allocation Model` (**EQ-3.2.2**), `Usage Model` (**EQ-3.2.3**), and `Resource Environment Model` (**EQ-3.2.4**).

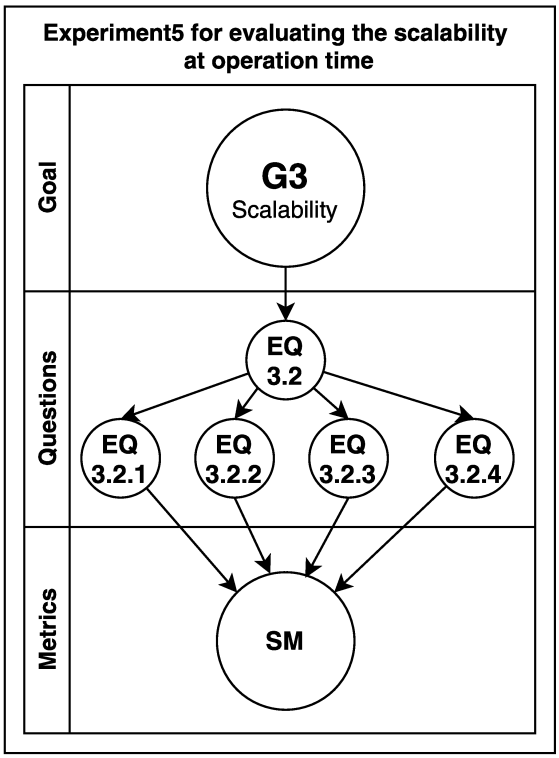


Figure 11.8: GQM plan of Experiment5.

11.2.2. Scenario

E5 involves analyzing the factors that impact the execution time of each transformation pipeline within the Ops-time calibration. Subsequently, we configure a scenario generator responsible for generating uncalibrated performance models (PCMs) and the corresponding synthetic monitoring data for each scenario. The goal is to analyze the execution time in the worst-case scenarios. Consequentially, the generator provides PCMs and the input monitoring data for the required worst-case scenarios. The monitoring data is then employed to calibrate PCMs across the transformation pipelines. **E5** measures the required execution time and analyzes the relation between the execution time and influencing factors. **E5** is iteratively replicated for analyzing the scalability by calibrating different aspects of PCM while considering a different range of influencing factors.

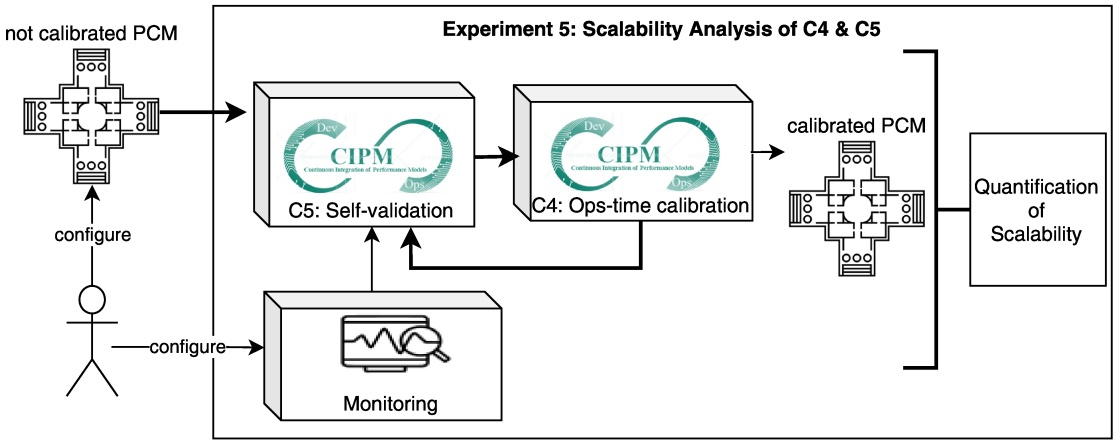


Figure 11.9: The scenario of Experiment5.

Moreover, the experiment is conducted multiple times for each aspect to enhance the reliability of the results and avoid measurement errors and outliers. Then the mean value of the results is calculated. An approach of 20 repetitions is employed [Mon20].

In the upcoming sections, we provide an in-depth exploration of the **E5** design, focusing on the analysis of the scalability of all transformation pipelines. For example, in Section 11.2.3, we assess the scalability of calibrating `Repository Model`, while in Section 11.2.4, we delve into the scalability aspects related to calibrating the system and allocation models. Similarly, Section 11.2.5 investigates scalability in calibrating usage models, and Section 11.2.6 concentrates on the environmental models.

11.2.3. Setup for Repository Model

The scalability of the repository model transformation has been analyzed following the aforementioned scenario.

Analyzing the influencing factors: The $T_{RepositoryModel}$ analyzes the simulation results of the self-validation by calculating the accuracy metrics for PMPs and subsequently calibrates the inaccurate PMPs. Calculating accuracy metrics has a negligible execution time in comparison to the calibration. For calculating the metrics, the analysis iterates the simulation results once and takes not more than one second to calculate them, even if the simulation is configured with excessive simulation times and measuring points. Therefore, our analysis of the scalability of $T_{RepositoryModel}$ concentrates on the calibration time.

Two main factors affect the execution time of calibrating the `Repository Model`. The first factor is the number of PMPs that have to be calibrated. The second one is the amount of available monitoring data used for the calibration process.

The worst-case scenarios: For the scalability analysis of $T_{RepositoryModel}$, we built two worst-case scenarios. In the first one, the number of internal actions, as an example of PMPs, is increasing. In the second scenario, the number of measurement data points is used for calibrating a single internal action.

11.2.4. Setup for System Model and Allocation Model

Both $T_{SystemModel}$ and $T_{AllocationModel}$ are executed concurrently to calibrate `System Model` and `Allocation Model`. Hence, we analyze their scalability together, considering the impact of increasing monitoring data.

Analyzing the influencing factors: Regarding `System Model`, To comprehend the scalability of both `System Model` and `Allocation Model`, we concentrate on two crucial influencing factors. In the case of `System Model`, the number of changes in the system composition plays an important role. Conversely, for `Allocation Model`, the number of changes in deployments affects the execution time. The design of the scalability analysis is closely related to these factors.

The worst-case scenarios: The scalability analysis delves into worst-case scenarios where changes in both system composition and deployments are concurrently considered. For that, we determine the number of changes, populating one half with deployment changes and the other half with changes to the system composition. These change scenarios, varying in size, are generated to evaluate the scalability impact. Both $T_{SystemModel}$ and $T_{AllocationModel}$ are applied to these scenarios, providing insights into the CIPM scalability under different conditions.

11.2.5. Setup for Usage Model

The $T_{UsageModel}$ is derived from `iObserve` and adapted to our monitoring data structure. Heinrich thoroughly analyzes the scalability of $T_{UsageModel}$ in the context of `iObserve` [Hei20]. Our analysis aims to assure the scalability of $T_{UsageModel}$ in the context of CIPM.

Analyzing the Influencing Factors: The usage of the system is directly correlated with both the number of users that interact with the system and the frequency of service calls initiated by each user. Therefore, $T_{UsageModel}$ is impacted by these two factors (Number of users and number of service calls).

The Worst-Case Scenarios: We consider two different cases. First, an increasing number of users, all of them triggering exactly one service call. Second, an increasing number of service calls are triggered by a single user.

11.2.6. Setup for Environment Model

$T_{ResourceEnvironmentModel}$ focuses on detecting changes in the available hosts and network connections within them.

Analyzing the Influencing Factors: The execution time of $T_{ResourceEnvironmentModel}$ is mainly related to the number of hosts to be detected and the connections within `Resource Environment Model`.

The $T_{ResourceEnvironmentModel}$ utilizes CPRs within VITRUVIUS framework to update the resource containers in the `Resource Environment Model` and link them with each other. The execution time of the $T_{ResourceEnvironmentModel}$ is primarily influenced by the VITRUVIUS CPRs, which is triggered by changes in hosts or changes in the connections between hosts.

The Worst-Case Scenarios: We explore two scenarios to analyze the scalability of $T_{ResourceEnvironmentModel}$. In the first scenario, we increase the number of new hosts with a sparse mesh configuration, where each new host has only one network connection. The second scenario triggers CPRs, updating more connections between hosts in addition to updating hosts. For that, we increase the number of new hosts with a fully meshed configuration, where each new host is connected to every other host.

11.2.7. Results

The following sections provide an overview of the outcomes from **E5** considering $T_{RepositoryModel}$ (Section 11.2.7.1), both $T_{SystemModel}$ and $T_{AllocationModel}$ (Section 11.2.7.2), $T_{UsageModel}$ (Section 11.2.7.3), and $T_{ResourceEnvironmentModel}$ (Section 11.2.7.4).

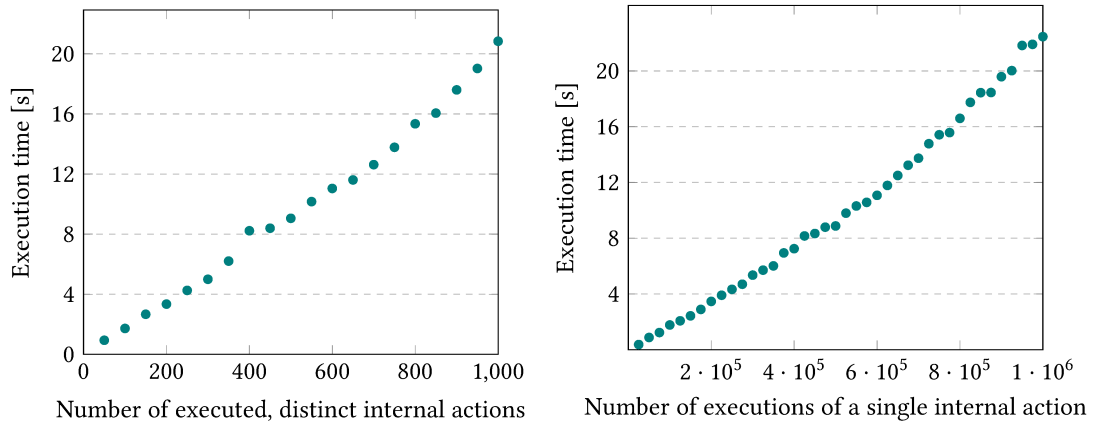
11.2.7.1. Repository Model

In Figure 11.10, we present scalability analysis results for updates to the `Repository Model`. The analysis covers $T_{RepositoryModel}$ with varying numbers of internal actions to be calibrated (illustrated in the left chart (a)), as well as scalability concerning $T_{RepositoryModel}$ with an increasing amount of monitoring data used for calibrating a single internal action (shown in the right chart (b)).

In both scenarios, the execution time remains below 30s even for large models and the transformation seems to linearly scale with increasing impacting parameters: with a high number of internal actions (a) and when increasing the data points per internal action (b).

11.2.7.2. System Model and Allocation Model

The chart in Figure 11.11 illustrates the cumulated execution time of two transformations ($T_{SystemModel}$ and $T_{AllocationModel}$) while the number of changes in the system composition and allocation increases.



(a) Scalability of $T_{RepositoryModel}$ with an increasing number of internal actions

(b) Scalability of $T_{RepositoryModel}$ with an increasing measurement data points for calibrating a single internal action

Figure 11.10: Exploration of the scalability of the $T_{RepositoryModel}$ that updates `Repository Model`, based on results from [Maz+25].

As depicted in the Figure 11.11, there is a gradual increase in execution time as the number of changes rises. However, beyond approximately 500 changes, the slope becomes steeper, indicating a higher rate of increase in execution time. Notably, even with 1200 changes, the execution time remains below 4 seconds. This observation suggests that both transformations exhibit acceptable scalability for realistic scenarios.

Figure 11.11 shows the cumulated execution time of both transformations for an increasing number of changes.

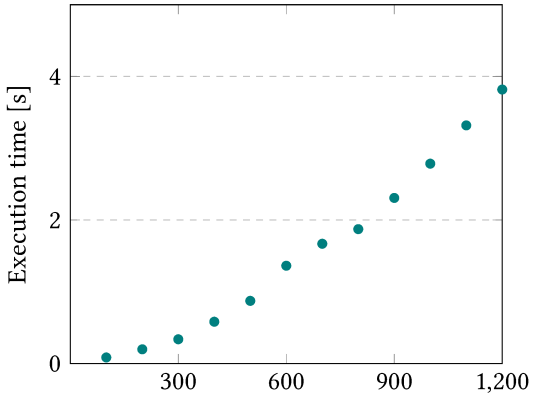


Figure 11.11: Execution times of $T_{SystemModel}$ and $T_{AllocationModel}$ with an increasing number of changes in the system composition [Maz+25].

The chart shows that the execution times scale approximately linearly, with a slightly lower slope at the beginning compared to a higher but stable slope from about 500

changes onwards. Even for a total number of 1200 changes the execution time is lower than 4 seconds. Because such several changes between two pipeline executions probably never occur in practice, it can be concluded that both transformations scale well.

11.2.7.3. Usage Model

Figure 11.12 presents the scalability analysis for both scenarios. Chart (a) illustrates the increase in execution times corresponding to the rising number of users, while chart (b) shows the growth corresponding to an increasing number of service calls initiated by a single user.

The analysis of the first scenario indicates that the execution time scales almost linearly with an increasing number of users, a conclusion consistent with the findings from `iObserve`'s scalability analysis [Hei20]. Similarly, consistent results were obtained when analyzing the execution time for an increasing number of service calls initiated by a single user.

In scenarios with 100 or fewer initiated service calls, the execution time demonstrates sublinear growth. Nevertheless, once this threshold is surpassed, a superlinear increase occurs, marked by a significant growth in execution time beyond 10,000 initiated service calls. Notably, within this superlinear range, loop detection emerges as the primary influencer of execution time, as illustrated in [Hei20].

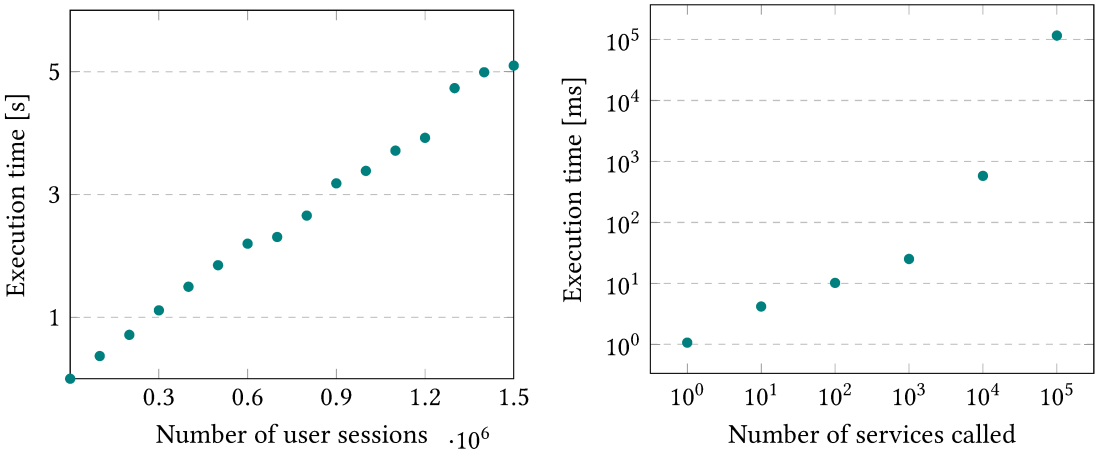
While the extreme increase in execution time with a high number of triggered service calls may seem concerning, it is important to note that such user behavior is unlikely in practical scenarios. In real-world usage scenarios, users would not initiate such a huge number of service calls. The results were also consistent with `iObserve` [Hei20].

11.2.7.4. Environment Model

The scalability analysis of updating the `Resource Environment Model` is summarized as follows:

The chart (a) in Figure 11.13 demonstrates the scalability of $T_{ResourceEnvironmentModel}$ within sparse meshed Ops-time environments. The execution time seems to scale almost linearly by adding up to 180 new hosts. It can be demonstrated that within our testing environment, a single execution of $T_{ResourceEnvironmentModel}$ for a practical scenario, involving the addition of 20 new hosts, took around 2 seconds.

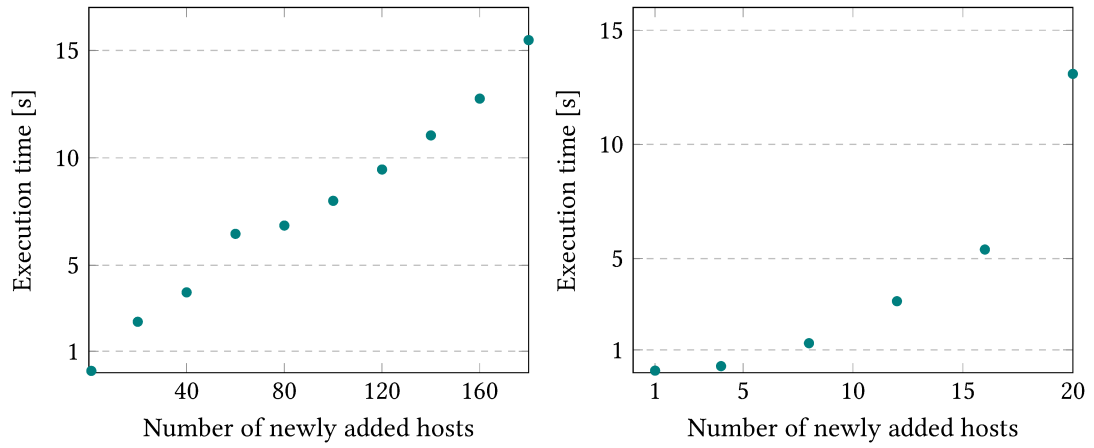
In contrast, chart (b) in Figure 11.13 illustrates that the $T_{ResourceEnvironmentModel}$ scales almost exponentially when incorporating fully meshed hosts. This is primarily due to updating the connections between hosts, which occurs sequentially one by one. Nevertheless, it can still be concluded that the transformation offers acceptable execution times for most use cases. In practical scenarios, encountering more than 10 new fully meshed hosts between two executions of the pipeline would be rare.



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

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

Figure 11.12: Scalability analysis of the $T_{UsageModel}$ that updates `Usage Model`, based on results from [Maz+25].



(a) Scalability of the $T_{ResourceEnvironmentModel}$ with an increasing number of newly added hosts; each new host has only one connection to another host

(b) Scalability of the $T_{ResourceEnvironmentModel}$ with an increasing number of newly added hosts; each new host has a connection to all other hosts (fully meshed)

Figure 11.13: Scalability analysis of the $T_{ResourceEnvironmentModel}$ that updates `Resource Environment Model`, based on results from [Maz+25].

11.2.8. Discussion

Answering EQ-3.2.1 Drawing from the findings outlined in Section 11.2.7.1, we can answer **EQ-3.2.1** "*How does the Ops-time calibration of* `Repository Model` *scale with an*

increasing amount of monitoring records?". It can be inferred that the transformation sustains linear scalability even in worst-case scenarios, where either more PMPs are calibrated or additional monitoring data is employed for calibration.

Answering EQ-3.2.2 The findings in Section 11.2.7.2 suggest the following answer for **EQ-3.2.2**"*How does Ops-time calibration of both* `System Model` *and* `Allocation Model` *scale with an increasing amount of monitoring records?*". Both $T_{SystemModel}$ and $T_{AllocationModel}$ transformations scale effectively with increasing amounts of monitoring data. The analysis revealed a nearly linear scalability pattern, indicating that the execution time of these transformations increases proportionally with the number of changes in `System Model` and `Allocation Model`. This highlights the suitability for updating these models while maintaining acceptable performance levels (less than 4 seconds for 1200 changes).

Answering EQ-3.2.3 In this paragraph, we answer **EQ-3.2.3** "*Does the scalability of Ops-time calibration of* `Usage Model` *demonstrate similarities to the scalability of the method employed by* `iObserve` *for calibrating* `Usage Model`*?*", Based on the results presented in Section 11.2.7.3, it can be concluded that the findings of our scalability analysis align with those of `iObserve`. Therefore, it can also be inferred that the execution times of the $T_{UsageModel}$ within CIPM are appropriate for the real-case scenarios.

Answering EQ-3.2.4 Regarding **EQ-3.2.4** "*How does the Ops-time calibration of* `Resource Environment Model` *scale with an increasing amount of monitoring records?*", the findings from Section 11.2.7.3 indicate that the scalability analysis of the `Resource Environment Model` transformation demonstrates appropriate execution time scaling for realistic scenarios.

11.2.9. Limitation

The evaluation of $T_{RepositoryModel}$ scalability is restricted to the regression-based optimization. However, the genetic algorithm can also be employed. In such a case, the genetic algorithm can be configured to ensure that its execution time does not exceed a predefined threshold, preventing any negative impact on scalability. It is important to know that there is a tradeoff between the execution time of a genetic algorithm and the accuracy of the resulting `Repository Model`. Therefore, the scalability analysis in the case of using the genetic algorithm has to be evaluated in future work.

E5 is restricted to calibrating internal actions as an example of PMPs, as we assumed that the calibration of internal actions takes a longer time than the other PMPs. Future evaluation can analyze the scalability of $T_{RepositoryModel}$ considering the other PMPs.

The scalability analysis excludes the simulation time during $T_{Self-Validation}$ since it does not depend on monitoring data but is predominantly related to the simulation tool and its configuration. Our observations indicate that, in the worst-case scenario of executing **E4** on our cases, self-validation does not exceed 15 seconds [Mon20]. Given the customary time intervals between CIPM pipeline executions at Ops-time (spanning several minutes), self-validation is unlikely to cause any bottlenecks, ensuring timely processing of monitoring data.

Notably, **E5** utilized the headless `Palladio` simulator developed within the scope of this dissertation [2] to expedite simulation execution. Unfortunately, using the faster simulation engine, SimuLizar [BBM13], could not be applied due to an implementation issue. Consequently, SimuCom [Bro+15], identified as deprecated in the context of the `Palladio` simulator due to performance concerns, had to be employed. Transitioning to a more efficient simulation tool like SimuLizar NG [KS19], which addresses extensibility and scalability issues in simulation execution, particularly for self-adaptive software architectures, is anticipated to reduce the required simulation time.

11.2.10. Threats of Validity

In the context of **E5**, validity threats may arise from two folds:

- **Internal validity:** Internal validity threats may arise from confounding variables that could influence the observed scalability of transformation pipelines. For instance, variations in hardware configurations or software environments across experimental runs could introduce bias in the results.

- **Construct validity:** Regarding the data generation, we relied on synthetically generated monitoring data to conduct controlled experiments focusing on specific transformations of the pipeline. The experimental scenarios do not adequately represent real-world conditions and the sampled scenarios do not cover a diverse range of system configurations and workload characteristics. This may pose a threat to validity. The use of synthetic monitoring data instead of real-world data may limit the applicability of the findings to actual operational environments. However, in [Maz+20; Mon20], we addressed this concern by measuring the execution time of pipeline transformations with increasing monitoring data for CoCoME and TeaStore. This empirical analysis guided us in conducting **E5** and analyzing the influencing factors.

[2] See `https://github.com/CIPM-tools/PCM-Headless/`

11.3. Summary

In this section, we summarize the validation results regarding consistency preservation at Ops-time. This validation encompasses both the update process of aPMs (**E4**) and the scalability analysis, focusing on the time required for these operations (**E5**). Below, we provide a summary of the conducted experiments.

11.3.1. Summery of Experiment4

In **E4**, the focus was on assessing the adaptability and accuracy of aPMs in dynamic operational environments. Key findings and outcomes of **E4** include:

Adaptation to Changes: **E4** successfully demonstrated the ability of aPMs to adapt and fine-tune themselves in response to frequent changes in the system, even without incremental calibration at the development stage.

Prediction Accuracy: The aPMs' accuracy improved over time as they learned from more data, stabilizing at an acceptable level. This highlights the potential of aPMs for predicting system performance in dynamic contexts.

Monitoring Overhead: **E4** also showed that adaptive monitoring strategies could reduce monitoring overhead, even when the source code was fully instrumented.

In summary, **E4** revealed the capacity of Ops-time calibration and self-validation to enhance aPMs, facilitating performance prediction in dynamic environments. This enhancement can be further improved with incremental calibration at Dev-time. Moreover, **E4** refers to the potential benefits of employing genetic algorithms for additional optimization at Ops-time, which is an area for future exploration.

11.3.2. Summery of Experiment5

Based on the findings from the evaluations, as presented in equations **EQ-3.2.1**, **EQ-3.2.2**, **EQ-3.2.4**, and **EQ-3.2.3**, it can be inferred that all transformations demonstrate appropriate scalability across various scenarios, what positively answers **EQ-3.2**. Despite potentially encountering worst-case scenarios, the observed execution time remains within acceptable bounds, particularly considering that the CIPM pipeline is typically activated at intervals of minutes, such as every 3 minutes. Therefore, it seems that the CIPM pipeline effectively scales to real-world situations, highlighting its practicality and effectively addressing scalability concerns, which answers **EQ-3.2**.

Part IV.

Reflection

12. Related Work

In this chapter, we discuss research areas related to CIPM, structuring the related work into six sections. In Section 12.1, we discuss methods for evaluating system performance within agile development. Then, in Section 12.2, we explore modeling during agile development. In Section 12.3, we discuss concepts for maintaining up-to-date performance models amidst frequent changes in agile development. Section 12.4 includes approaches for automatic instrumentation. In Section 12.5, we investigate approaches for estimating resource demands. Finally, in Section 12.6, we examine approaches for estimating dependencies among performance parameters and their influencing factors.

12.1. Performance Assessment in Agile Development

This section reviews work related to performance assessment in agile development. In Section 12.1.1, we explore techniques and tools for monitoring and managing application performance in real-time. Then, in Section 12.1.2, we focus on methodologies and frameworks for testing system performance within agile processes. Finally, in Section 12.1.3, we discuss approaches for integrating performance evaluation into DevOps workflows, emphasizing continuous assessment and improvement.

12.1.1. Application Performance Monitoring and Management

Since Application Performance Monitoring is widely used, especially in the DevOps pipeline, an industrial survey indicates an industrial survey highlights that APM tools are considered the most important tools for DevOps [Bez+19a]. Application performance monitoring tools monitor applications and analyze the measurements to ensure that applications and services comply with service level agreement (SLA).

While APM is often used interchangeably with application performance monitoring, APM encompasses more than monitoring metrics like response times, error rates, and resource utilization [IBM24]. APM integrates data analysis to detect trends and bottlenecks, automated troubleshooting, and optimization tools to enhance app efficiency.

Hence, Kowall and Cappelli [KC12] emphasize the importance of monitoring during the rapid evolution in the market. They assess the commercial APM tools like AppDynamics [App24], BMC Software [BMC24], CA Technologies [CA24], Compuware

[Com24], HP [Hew24], IBM [IBM24], Microsoft etc. According to Kowall and Cappelli [KC12], there are five main functionalities. Below are these functionalities, enhanced with examples and possible approaches:

- Monitoring end-user experience: The tools monitor applications by instrumenting the script that the end-user executes and sending back the measured performance metrics to the servers. The goal is to assess the quality attributes on the client side, e.g., availability, latency etc. For instance, [ML03] measures user-perceived latency at websites, using a server-side solution to allow server-side optimization and prioritization based on actual observed performance.

- Discovering the application architecture and topology: For that, the monitoring data is analyzed to detect the components, the interaction between them, and their deployment; see Section 12.3.2 for some examples. This step is followed by a graph-based visualization of the discovered architecture, topology, or workflow of the application. Software landscape visualizations are mostly found in APM tools [Heg+17] and often use a flat graph-based representation of nodes, applications, and their communication. Fittkau et al. [FKH17] introduce ExplorViz, a tool that provides a hierarchical and multi-level visualization approach for solving system comprehension tasks for large software landscapes and individual software applications. ExplorViz, generates visualizations from monitoring traces, offering an understanding of the system composition.

- User-defined profiling transactions: The goal is to detect the business transaction that is responsible for performance issues. This requires mapping between business transaction and their implementation to trace events among executing transactions. An example of this instance is the approach of Kiciman et al. [KF05]. This approach is triggered by performance issues, such as high response times or errors. It proposes a diagnosis, which involves inspecting the transaction's call tree, similar to profiling, to find the transactions causing the performance issues. These are usually identified due to high response times, frequent calls, or exceptions.

- Application component deep dive: This requires fine-grained monitoring of an application component to cover all call trees. The call trees include data flows, internal calls, and third-party calls. The goal is to enrich the call trees with performance measurements like response times and resource demands. For example, an analysis of Alibaba's large-scale microservices in production environments examines call trees to represent the topology of graph dependency, which can be used to analyze the real-time performance of online services [Luo+22].

- Analytics: This functionality applies statistical analysis or data mining techniques on the monitoring data resulting from the above-mentioned functionalities. The

goal is to detect the performance issues and define actionable performance management items. For instance, approaches such as those by [EH11; Wan+18; Zha+21; Hou+21; CSF20] detect anomalies to identify unusual patterns in system performance—such as unexpected spikes in response times or error rates—which support actionable performance management strategies, including locating root causes of software failures and preventing failures through early fault identification and correction.

Tarek et al. [Ahm+16] study the effectiveness of four APM approaches: New Relic [New24], App Dynamics [App24], Dynatrace [Dyn24], and the open-source tool Pinpoint [Pin24]. Their studies concentrate on the effectiveness of detecting performance regressions.

CIPM distinguishes itself from related architecture performance monitoring approaches by enabling proactive performance management while also enhancing the understanding of performance models.

Regarding APM tools that facilitate architecture discovery through dynamic analysis, as discussed later in Section 12.3.2, these approaches are limited to identifying only the parts of the system that are actively executed and monitored. They struggle to detect unexecuted or less frequently used components because they do not consider the extraction of comprehensive performance models. Ignoring the source code and its influence on the accuracy of extracted aPMs can reduce the accuracy of performance predictions. This limitation arises because such approaches often rely on approximations or probabilistic call graphs to infer system behavior.

12.1.2. Performance Testing

Performance tests focus on assessing how a system performs under specific, real-world scenarios or varying conditions [WFP07a; Bon14]. Jiang et al. [JH15] surveys approaches for load testing, highlighting the importance of designing appropriate loads, executing the tests effectively, and analyzing results to verify system functionality and stability under load. For instance, there are approaches

Creating a workload is one of the primary challenges in performance testing [Abb+12]. [Vög+18; Tru+18] for performing load tests based on operational workload scenarios to validate under realistic load conditions. Schulz et al. [SHW19] focus on reducing the manual effort required to update load tests due to changing production. Avritzer et al. [Avr+20] propose using operational profiles to automatically generate load tests that assess scalability pass/ fail criteria for microservice deployment configurations.

Similarly, CIPM utilizes `iObserve` to extract the usage profile and usage intensity from the operation environment to enable AbPP under realistic conditions. Using aPM for the performance prediction can reduce the cost required for intensive performance testing, which can take several hours in some cases [Bru+15; Fer21].

To reduce the cost of performance testing on the entire system, some approaches suggest using unit testing. For instance, Heger et al. [HHF13] present an approach that automatically detects and identifies the root cause of performance regressions during software development using performance-aware unit tests and revision history. Although the approach of Heger et al. provides automatic continuous performance tests, their approach is limited for performance regression. Evaluating design alternatives based on performance tests will still be expensive.

Grambow et al. [Gra+22] use optimized microbenchmark suites to detect application performance changes, since the extensive benchmarking studies usually take several hours, which is problematic when examining dozens of daily code changes in detail.

In this paragraph, we introduce examples of approaches for reducing the cost of tests through automating unit tests/ or declarative DSL. However, such an approach focuses only on automating performance tests and is still expensive compared to simulating performance using model-based approaches.

In contrast to the approaches above, CIPM provides the aPM, not only for performance prediction but also for giving an overview of the architecture for supporting design decisions.

12.1.3. Performance Assessment during DevOps

Bezemer et al. [Bez+19a] conducted an industrial survey to examine how performance is addressed in DevOps practices. The survey results reveal that while CI is widely implemented, CD and continuous provisioning are much less common. Moreover, 88% of participants reported not using models for performance management, despite nearly 70% expressing interest in adopting such models. A key challenge identified is the complexity of performance engineering tools and methodologies, which impedes their adoption. The authors recommend that performance engineering approaches should prioritize lightweight solutions that seamlessly integrate into existing DevOps pipelines. CIPM addresses the challenges and recommendations outlined by Bezemer et al. By employing CI-based updates to maintain an up-to-date aPM, CIPM significantly aims to reduce the complexity associated with traditional performance modeling approaches by automating the update process and enabling simulation associated with the visualization of the results. Besides, the design integrates adaptive monitoring and automated calibration within the CI pipeline, ensuring seamless compatibility with DevOps practices and tools.

Waller et al. [WEH15] introduce an approach for integrating automated performance benchmarks into CI environments to support DevOps practices. Their approach aims to detect performance regressions early through regression benchmarks to improve the release cycles and reduce deployment risks. Their case studies demonstrate the seamless integration of benchmarks within CI pipelines to enhance software quality. Similarly, Vincenzo Ferme integrates performance testing into DevOps through a declarative

DSL for selecting and configuring the tests and a framework automating tests across the continuous SDLC [Fer21]. However, although his approach reduces the cost of executing performance tests, it is still expensive to evaluate the design alternatives compared to running tests using model-based approaches. Besides, in contrast to the approaches in this paragraph, CIPM provides an aPM that goes beyond performance prediction to provide architectural insights, bridging the gap between performance analysis and design understanding. While automation in these approaches lowers the cost of performance assessment, evaluating design alternatives remains costly compared to model-based methods.

12.2. Modeling during Agile Development

Mognon et al. [MS16] provide a systematic literature review on the integration of modeling practices within Agile Software Development. They explore how modeling supports design and communication in agile methodologies, especially for complex systems and distributed teams. Findings reveal the use of early-stage modeling for requirements and architecture, and ongoing exploration of formal methods and MDD, though their effectiveness remains inconclusive. In their literature review, they do not provide information about how to maintain these models in the later stages, instead focusing on the importance of models for complex and distributed systems and the modeling that occurs in the early stages of the SDLC. However, other related works refer to the challenges of maintaining models up to date [Tai+23; Can+18; Kas+21; FRS16; WA20].

Lunesu et al. [Lun+21] present a new approach using software process simulation modeling to model key risk factors in agile development, such as project duration and the number of implemented issues. Based on simulation, they quantitatively evaluate risks and provide insights for project managers. CIPM addresses another quality property by focusing on updating the architecture for performance prediction, whereas the risk models have a specific role and do not reflect the software architecture.

Chan et al. [CZ01] provide an object-oriented, modular method for modeling agile manufacturing systems, enabling reconfiguration and reusable modeling capabilities. They address the need for developing agile manufacturing systems in response to the dynamic and competitive global manufacturing market. The goal is to establish a modeling process suitable for agile manufacturing. However, as mentioned in the paper, the modeling tool is currently under development based on the presented methodology. To reduce the effort of manual maintainability of models, they propose a modular and object-oriented framework that allows for rapid reconfiguration and reuse of predefined modeling objects. In contrast, CIPM focuses on automatically updating performance models within software development, ensuring minimal manual intervention and continuous alignment with evolving software artifacts.

García-García et al. [Gar+20] conducted a systematic literature review on software process simulation modeling, emphasizing the value of simulation for decision support by enabling cost-effective comparison of design alternatives. They highlight that in the last two decades, there has been a rise in the use of simulation techniques such as agent-based simulation in software engineering. The study was not just exclusive to approaches used for agile and does not discuss the challenges of simulating models, such as the validity of used models and how to keep them consistent. AbPP in CIPM utilizes simulation, as suggested by García-García et al. [Gar+20], by simulating the aPM to enable accurate performance predictions and support decision-making.

Ghane [Gha14] utilize a simulation model to improve software quality control and quality assurance by simulating future activities. For this purpose, the model evaluates input variables such as time, resource, cost, and scope, and uses historical data to model the error rate of activity estimates. However, unlike CIPM, their simulation cannot answer quality questions related to the software architecture since they do not model it.

Kasauli et al. [Kas+21] describe requirements engineering challenges and practices in large-scale agile system development, presenting a multiple-case study with seven companies. The study reports their challenges and proposed practices, including suggestions from case companies and frameworks. One of the challenges identified is the need for modeling in an agile manner, which can supplement user stories and test cases to describe overall system functionality. The aPM that CIPM provides can provide the required overview of the overall system architecture and services.

12.3. Consistency Management Approaches in Software Development

One of the practical challenges in systems engineering is the degradation of artifacts, particularly architecture models and requirements [Woh+19]. Therefore, maintaining an up-to-date architectural model is essential to ensure continuous alignment and coherence between evolving requirements and architectural decisions.

Numerous approaches are dedicated to achieving automatic consistency among software models, as explained in Chapter 1. These approaches fall into two primary categories: A1 involves generating up-to-date artifacts through a batch process (denoted as $\checkmark_B$ in Table 12.1), exemplified by reverse engineering methods. On the other hand, A2 follows an incremental approach, checking consistency issues to be eliminated (marked as $\checkmark_{inc}$ in Table 12.1).

Categorically, both A1 and A2 are further classified into three subcategories based on the phase of consistency maintenance: Dev-time (Section 12.3.1), Ops-time (Section 12.3.2), or a combination of both (Section 12.3.3). The organization and classification

Table 12.1: An overview of the related work.

scope	Consistency Management Approaches	aPM					PMPs	AbPP	Parametric dependencies	Self-Validation
		Repository	System	Allocation	Res. Env.	Usage				
Dev-time	Co-evolution [Lan17, p. 35]	✓$_{inc}$					✓$_{B}$	✓		
	Co-evolution: Mbeddr [Voe+12], Focus [DM01]	✓$_{inc}$								
	Consistency checker [KP07], [Buc+13], [Det12],[LMÁ09]	✓$_{inc}$								
	Extract [Lan+16]	✓$_{B}$				✓$_{B}$		✓		
	ROMANTIC-RCA [AAM10], CAESAR [Ibr+23], HYGAR [Ely+23]	✓$_{B}$								
	SoMoX+Beagle [KKR10; Kro12]	✓$_{B}$					✓$_{B}$	✓	✓	
Ops-time	Brosig et al. [BHK11]	✓$_{B}$	✓$_{B}$	✓$_{B}$		✓$_{B}$	✓$_{B}$	✓		
	PMX [Wal+17], Weber et al. [WWH]	✓$_{B}$	✓$_{B}$	✓$_{B}$			✓$_{B}$	✓		
	Brunnert et al. [BVK13]	✓$_{B}$	✓$_{B}$	✓$_{B}$	✓$_{B}$		✓$_{B}$	✓		
	SLAstic [Hoo14a; van+09], SLAstic.SIM [MHH11]	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{B}$	✓		
	iObserve [Hei20]	✓$_{inc}$		✓$_{inc}$		✓$_{inc}$		✓		
	Approach for CPE by [Cor+22]		✓$_{inc}$		✓$_{inc}$					
Hybrid	EjbMoX [Lan17, p. 140]	✓$_{B}$						✓		
	Konersmann´s approach [Kon18; KH16]	✓$_{B}$		✓						
	PRISMA [Spi+19]	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{B}$	✓	✓	
	CIPM	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓$_{inc}$	✓	✓	✓

of these approaches into subcategories are summarized in Table 12.1, with A1 falling under the ✓$_{B}$ label and A2 under ✓$_{inc}$. This facilitates a systematic overview of the existing works and their alignment with the designated subcategories.

12.3.1. Consistency Management at Dev-time

As shown in Table 12.1, numerous approaches focus on consistency maintenance at Dev-time, where they either extract an architecture model (A1) or maintain an existing one (A2):

A1 Reverse engineering at Dev-time (A) can be based on static analysis techniques, such as clustering, pattern detection, and model-driven approaches [GIM13], and may be further enhanced with dynamic analysis based on monitoring from a test environment [Pér12; Kro12]. A systematic literature review on various model-driven reverse engineering solutions is found in [RFZ17], focusing on the models used and the transformations applied.

Examples of reverse engineering approaches at Dev-time (A1) that primarily rely on static analysis of source code include ROMANTIC-RCA [AAM10], CAESAR [Ibr+23], HYGAR [Ely+23], SoMoX [Bec+10] and Extract [Lan+16]. ROMANTIC-RCA [AAM10] extracts component-based architecture from an object-oriented system using relational concept analysis. Similarly, Ibrahim et al. [Ibr+23] employs an automated, knowledge-based solution to recover software architecture by clustering modules based on context and enhancing cohesion through a code distance model, supported by visualizations at multiple abstraction levels. Elyasi et al. [Ely+23] propose a hybrid genetic algorithm for software architecture recovery, incorporating a greedy heuristic with a genetic algorithm for clustering software

modules. The retrieved approaches by [AAM10; Ibr+23; Ely+23] support system understanding but do not support AbPP.

SoMoX [Kro12] utilizes various strategies for detecting the architectural elements based on heuristics to extract `Palladio Repository Model`. To be noted, we reuse the `SEFF` extraction of SoMoX in our CPRs as it is adapted by Langhammer [Lan17]. The Extract [Lan+16] approach also uses source code to recover static and use case-based views of a system (`Repository Model` and `Usage Model` of `Palladio`) based on static analysis of source code and unit tests.

Reverse engineering approaches based on static and dynamic analysis can be classified under approaches that maintain consistency at Dev-time, if they rely on monitoring data obtained from execution tests conducted during the Dev-time phase. Thus, these approaches do not extract the execution view or update the architecture based on monitoring data from Ops-time. For example, Krogmann extended SoMoX with a calibration of PMPs [KKR10] based on the dynamic analysis (Beagle approach [Kro12]). However, Krogmann's approach requires high monitoring overhead, which restricts the collection of monitoring data from the production environment rather than from the test environment. Therefore, we assign Krogmann's approach to the Dev-time approaches. Similarly, Pérez-Castillo [Pér12] proposes a semi-automatic, model-based reverse engineering approach to extract business processes from legacy systems. It combines static and dynamic analysis to extract knowledge and utilizes business patterns to discover business processes. Unlike Beagle and CIPM, the approach of Pérez-Castillo does not support performance prediction.

A shortcoming of A1 approaches is that they ignore the possible manual optimization of the extracted model in the next extraction.

A2 The incremental consistency maintenance at Dev-time (A2) encompasses approaches that aim to (a) minimize architectural erosion through consistency-checking techniques, (b) prevent it using Co-evolution approaches, or (c) repair it through targeted resolution methods [SB12]. A comprehensive summary of these approaches is provided in [SB12]. In the following, we provide examples of these approaches:

a Torres et al. [TBS20] present a systematic literature review of cross-domain model consistency checking by model management tools, summarizing that 40% of the tools are providing consistency checking across models from different domains. Moreover, Knodel Jens et al. compare the approaches that minimize architecture erosion by detecting architectural violations at Dev-time [KP07]. JITTAC tool [Buc+13] also detects the inconsistencies between architecture models and source code, but does not eliminate them automatically. Archimetrix [Det12] detects also the most relevant deficiencies

through continuous architecture reconstruction based on reverse engineering.

b Examples of A2 approaches that prevent the inconsistency between source code and architecture model at Dev-time are the co-evolution approaches, e.g., the mbeddr approach [Voe+12] and Langhammer's approach [Lan17]. The mbeddr approach uses a single underlying model for implementing, testing, and verifying system artifacts like component-based architecture. Similarly, the approach of Langhammer [Lan17] uses a virtual single underlying model to allow the co-evolution of PCM and source code. The Focus approach [DM01] avoids the inconsistencies by recovering the architecture and using it as a basis for the evolution of the object-oriented applications.

c Lucas et al. [LMÁ09] 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 that addresses the detected limitations and resolves the inconsistencies based on model transformation techniques. In [Maz+18], we propose an approach for detecting the inconsistency between architectural automotive models and resolving them based on Vitruvius, to be able to generate compatible source code from these models.

The main limitation of the consistency management at Dev-time is that the provided models are mostly considered as system documentation and should be enriched with PMPs if the AbPP is to be supported. Therefore, some of these approaches are extended to allow AbPP. For example, Langhammer [Lan17] calibrated the Co-evolved model with an approximation of resource demand (response times) to show that it can be used for AbPP. Similarly, Krogman extends SoMoX with Beagle, considering the parametric dependencies and enabling AbPP. In general, the calibration of the whole project after each adjustment in the models causes monitoring overhead and ignores possible manual adjustments of PMPs, which our approach overcomes by incremental calibration. Moreover, the consistency management approaches at Dev-time ignore the impact of adaptations at Ops-time on the accuracy of aPMs. Thus, they fail to update the current execution view. Although Extract can derive a static usage profile (`Usage Model`) through a static analysis of unit tests source code, this `Usage Model` does not accurately reflect the current `Usage Model` in operation.

12.3.2. Consistency Management at Ops-time

The approaches that maintain the consistency at Ops-time are based mainly on the dynamic analysis of monitoring events. For example, the approach of Brosig et al. [BHK11], PMX [Wal+17], and [BVK13] are reverse engineering approaches (A1) that

extract parts of the PalladioaPM based on dynamic analysis to allow AbPP. Weber et al. [WWH] examine the integration of performability-model extraction and prediction in CI/CD pipelines, building upon the PMX. They use monitoring infrastructure to collect traces and metrics, such as utilization and network capacity. Traces reveal the control and data flow within applications, and when combined with metrics, they enable the derivation of system models.

Furthermore, the SLAstic approach [Hoo14a; van+09; MHH11] extracts aPM and can detect some changes at Ops-time, such as migrations, and reflect them in the model. Hoorn also utilizes Palladio for simulation-based analysis and online performance prediction. Thus, they applies model to model transformation to transform SLAstic models to PCM. Regarding PMPs, fixed values for RDs and loop iterations are used.

iObserve [Hei20], described in Section 2.6.6, can also respond to changes in deployment and usage by updating the related parts in PCM. As aforementioned in Section 7.3, we build upon iObserve to update Usage Model and Allocation Model through CIPM. Other approaches that also extract/ update performance models at Ops-time are summarized in [SZ16].

Similar to iObserve, Cortellessa et al. [Cor+22] propose a model-driven approach that utilizes traceability relationships between monitoring data and architectural models (UML profiles with UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (MARTE)) to identify design alternatives for addressing performance issues. 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 source code to update the UML model accordingly.

The drawbacks of the Ops-time approaches are that a continuously high monitoring effort is required to extract/ update models. Moreover, they cannot model system's parts that have not been called and consequentially not covered by the monitoring. Besides, the source code changes are ignored and these approaches do not validate the accuracy of the models.

12.3.3. Hybrid Approaches

The scope of hybrid approaches spans Dev-time and Ops-time. For example, Langhammer introduces also a reverse engineering tool (EjbMox) [Lan17, P. 140] that extracts the behavior of the underlying Enterprise JavaBeans (EJB) source code, by analyzing it at Dev-time and calibrating it based on the dynamic analysis at Ops-time. EjbMox from Langhammer extracts Palladio Repository Model based on EJB-specific CPRs and calibrates it using a distribution function of the measured response times to approximate the resource demands of SEFF internal actions. However, it ignores parametric dependencies and is restricted to EJB-based projects.

Konersmann's approach integrates information about the architecture model into the code via annotations [Kon18]. This enables the dynamic generation of an architecture model from the source code via transformations [Kon18]. Moreover, the approach of Konersmann synchronizes allocation models with running software [KH16], with a limited scope of the consistency preservation compared to CIPM.

Spinner, Grohmann, et al. [Spi+19] propose PRISMA, an agent-based reference architecture for online model learning in virtualized environments, enabling the automatic extraction of the aPM. The architecture extends traditional virtual appliances by incorporating agents to monitor applications and build submodels, which are later merged into a Descartes Modeling Language (DML) model. Although PRISMA primarily updates the aPM at Ops-time, we consider it a hybrid approach, as it employs static analysis on newly deployed systems to identify EJB components and interfaces, using dependency injection analysis or bytecode analysis followed by dynamic analysis. This reflects the impact of Dev-time changes that have been deployed; however, the extracted `Repository Model` does not include undeployed components. Moreover, PRISMA calibrates control flow and resource demand as constant values, utilizing LibReDE without modeling parametric dependencies, as Spinner, Grohmann, et al. consider the learning of parametric dependencies an optional role that is not implemented. Furthermore, nested loops or recursive functions cannot be accurately extracted in the abstract behavior. A general constraint of dynamic analysis is that it only traces paths that are executed during runtime under specific scenarios, potentially overlooking unexplored execution paths.

12.4. Instrumentation Approaches

The related approaches estimate the resource demands either based on coarse-grained monitoring data [Spi+15; Spi+14] or fine-grained data [BKK09; Wil+15]. The latter approaches give a higher accuracy by estimating PMPs but have a downside effect because of the overhead of instrumentation and monitoring. Our approach reduces the overhead through automatic adaptive instrumentation and monitoring.

Chittimalli et al. [CS12] propose a unified programming language model (UPLM) as an intermediate representation for source-level instrumentation, enabling language-agnostic instrumentation across multiple programming languages. Each language parser transforms the source code into a UPLM-based intermediate representation. The tool supports statement, condition, branch, and method-level instrumentation and allows extensibility through a specification file defining AST nodes and probes. The probe inserter adds the probes, and the printer generates the instrumented source code.

AIM [WSH15] provides adaptable instrumentation of the services of the Application Under Test (AUT) with a two-agent architecture: the Event Agent for low-level JVM event hooking and the main one for bytecode instrumentation and monitoring processes.

AIM's real-time adaptability, which allows modification of instrumentation instructions at runtime, sets it apart from static approaches like Kieker, which is utilized in our approach. Although, AIM reduces the required monitoring overhead to get accurate measurements for estimating the resource demands, AIM lacks Kieker's advanced adaptive monitoring capabilities, such as regular expression matching and wildcard support.

Moreover, the integration of AIM with Kieker has been explored in a bachelor's thesis to enable dynamic Bytecode instrumentation by Kieker. This approach (BCI) reduces monitoring overhead compared to Kieker's traditional method, but comes at the cost of increased turnaround time. Furthermore, reliability drops due to a rise in lost transactions. Additionally, a freshly instrumented class operates in interpreter mode, requiring a transition period of approximately a few thousand executions for Just-In-Time (JIT) optimization after class instrumentation, which limits the effectiveness of dynamic BCI and restricts its applicability during Ops-time.

Contrary to the abovementioned instrumentation approaches, the adaptive instrumentation of CIPM (**C2**) automatically detects what parts of the source code with which kind of instrumentation probes should be instrumented, enabling automatically oriented model-based instrumentation for updating inaccurate performance parameters. The measurements are then collected from test or production environments to be used by the calibration.

Like CIPM, Kiciman et al. propose a platform (AjaxScope) for the instrumentation of JavaScript code of web applications [KL10] to allow performance analysis and usability evaluation. Based on coarse-grained monitoring, AjaxScope identifies where the source code runs slowly and instruments it to find the cause of the slowness. For that, AjaxScope leverages instant redeployability in web applications to dynamically serve new code versions. The AjaxScope prototype dynamically rewrites JavaScript-based applications sent to users' browsers to enable monitoring of performance. However, AjaxScope is limited to instrumenting the front-end of JavaScript-bsaed web applications.

12.5. Resource Demand Estimation Approaches

The related approaches estimate the resource demands either based on coarse-grained monitoring data [Spi+15; Spi+14] or fine-grained data [BKK09; Wil+15]. The latter approaches give a higher accuracy by estimating PMPs but have a downside effect because of the overhead of instrumentation and monitoring. Our approach reduces the overhead through automatic adaptive instrumentation and monitoring.

[Urg+07] scheduling-aware estimation approaches Linear regression and maximum likelihood estimators are used to estimate demands in multi-threaded applications, response times, or queue-length samples.

Spinner et al. [Spi+15] propose a classification scheme for resource demand estimation approaches and evaluate their accuracy using various factors. Their work focuses primarily on statistical approaches to resource demand estimation that rely on coarse-grained system measurements, such as CPU utilization and end-to-end response times.

There are several statistical methods for resource demand estimation, each with distinct characteristics: *Linear regression* [RV95; Pac+08; PPC13; RV98; Kra+09; ZCS07] is a widely adopted statistical method to establish relationships between workload classes and resource demands. However, it exhibits sensitivity to outliers, which can negatively impact the precision of demand estimations.

Kalman filters [KTZ09; Zhe+05] offer a dynamic estimation approach, continuously adapting to changes in resource demands over time, which makes them suitable for systems with varying workloads.

Machine learning techniques employ algorithms to estimate resource demand models based on their relationships with workload classes. Techniques such as clusterwise linear regression, independent component analysis, and support vector machines are used to identify patterns and dependencies between workload characteristics and resource demands [CDS10; CS14; Sha+08]. These approaches can capture complex relationships and adapt to non-linear behaviors in the data. For instance, the approach of Cremonesi et al. [CDS10] clusters observations and estimates resource demand for each cluster. However, they often require data across diverse system configurations for effective training, posing challenges in generalization. Hu et al. [HD18] propose an agile method for estimating changed resource demand based on elastic clusters for energy saving. Based on estimation at the service level, their approach determines when to scale and thus reduces the energy wasted on powering unused resources. For that, they use knowledge from previous resource demands, based on past workloads, to estimate resource demand for the new incoming workload by learning the dependencies to workload features. This estimation supports resource scaling, reducing scaling delay and energy waste. To improve estimation accuracy, they propose a two-layer neural network method that utilizes multiple linear regression processes as neurons [Hu+21].

Optimization techniques [Zha+02; Men08; KZT10] aim to minimize objective functions, such as prediction error for response times and utilization, to derive resource demands, which can improve accuracy but may involve computational overhead.

The *Maximum likelihood* method estimates demands by maximizing the probability of observing the measured response times [Kra+09; PPC13]. While this approach is known for high accuracy, it is computationally intensive and may be less practical for large datasets. *Gibbs sampling* applies Bayesian inference to estimate resource demands in complex queuing networks [SJ11; WC13], offering a probabilistic approach that accommodates parameter uncertainty. Demand estimation techniques that provide confidence intervals, such as in [Kal+12], offer insights into estimation uncertainty,

helping to quantify the reliability of resource demand estimates. The approaches above are based on coarse-grained measurements, unlike CIPM, which utilizes an incremental and adaptive approach to estimate resource demands incrementally at a fine-grained level to improve the accuracy of the estimation. Besides, CIPM continuously refines the estimated resource demands based on new measurements, dynamically calibrating only the affected parameters, thus improving accuracy over time. Additionally, CIPM employs machine learning algorithms to learn parametric dependencies, optimizing the model according to new measurements.

[Spi+14] provides a library for resource demand estimation, aiming to assist performance engineers during performance model construction. The library allows for the comparison of the estimation accuracy of different approaches in a given context, helping to select the most optimal method for the task at hand. In their work, the output is an average value for resource demand, without parametric dependencies or distribution functions. We build upon their implementation and provide an incremental approach to resource demand estimation (RDE), considering adaptivity where the output is a stochastic expression with parametric dependencies.

Self-adaptive resource demand estimation framework, (SARDE), is proposed by Grohmann et al. [Gro+21] to employ an ensemble of estimation techniques. Their approach continuously tunes, selects, and executes an ensemble of resource demand estimation approaches to adapt to changes in the environment. The framework applies continuous online optimization to reduce model error and improve the accuracy of resource demand estimates. Their resulting estimation is a mean value changing over the operation time. In contrast, CIPM considers the parametric dependencies and optimizes the estimated stochastic expression at Ops-time.

Willnecker et al. [Wil+15] compared academic and industry monitoring approaches, alongside a library integrating six resource demand estimation techniques [Spi+14]. They evaluated the techniques at both the system entry point and component operation levels. The results showed that estimation techniques perform better at the system entry point level. In contrast, direct measurements are better at the component level. Direct measurement techniques and adaptive estimation techniques can accurately determine resource demands, but the choice depends on available tools, measurement overhead, and granularity requirements

12.6. Parametric Dependencies

Krogmann (SoMoX+Beagle) [Kro12], instrument the source code fine-granularly and monitors the number of executed byte-code instructions to estimate the resource demand and the method parameters to learn the parametric dependencies.

Curtois et al. [CW00] characterize the parametric dependencies using regression splines to model resource demand measurements.

Ackermann et al. [Ack+18] introduce an approach that combines techniques for identifying and characterizing parametric dependencies based solely on run-time monitoring data.

Simon Eismann's research, as described in his dissertation [Eis23], focuses on improving simulation time for white-box performance models in serverless platforms. His approach combines black-box and queuing models and is built upon the Descartes modeling tool [Hub+17]. Additionally, he introduces a method to integrate empirically observed parametric dependencies into architectural performance models, enhancing their accuracy and realism.

Contrary to the approaches mentioned above, CIPM enables the characterization of parametric dependencies through an *incremental* update and calibration of the aPM during both Dev-time and Ops-time phases.

In joint work with Grohmann et al. [Gro+19], we analyzed the parametric dependencies in performance models at the level of Ops-time. Three feature selection techniques were evaluated, representing the primary groups: a filter method, an embedded method, and a wrapper method. The evaluation highlighted the superior performance of the filter technique over the other methods. In CIPM, we also apply the feature selection to reduce the dimensionality of the parameter space considered by detecting the parametric dependencies [Von+20].

13. Outlook

This chapter presents a forward-looking view of the research and its potential directions. In Section 13.1, we outline potential areas for future research and development related to CIPM, discussing unresolved challenges and new opportunities for enhancing the approach. In Section 13.2, we summarize the contribution and findings of this research.

13.1. Future Work

Future work for CIPM includes several advancements. First, the investigation of cost and trust in CIPM is discussed in Section 13.1.1. Next, utilizing version controlling within CIPM is discussed in Section 13.1.2, followed by the generalization of consistency preservation rules for CIPM, which is described in Section 13.1.3. The possible improvement of the scalability of MbDevOps pipeline pipeline is introduced in Section 13.1.4.

Another direction is enhancing CIPM with round-trip engineering capabilities, detailed in Section 13.1.5. Supporting performance analysis for automobile software systems is described in Section 13.1.6. Additionally, the use of a version control system within CIPM is described in Section 13.1.7. Then, we describe applying adaptive performance testing within MbDevOps pipeline pipeline in Section 13.1.8.

An important future work includes expanding the consistency maintenance range of CIPM, as covered in Section 13.1.9, and enhancing the abstract behavior, as addressed in Section 13.1.10. Enhancing dependency resolution for dynamic updates of SEFF is explored in Section 13.1.10.1. Further, there is a focus on combining static and dynamic analysis for incremental calibration, as detailed in Section 13.1.11. Finally, multi-resource demand calibration for internal actions is discussed in Section 13.1.12.

13.1.1. Investigation the Cost and Trust in CIPM

Our next step is to finalize the analysis of the conducted survey, which investigates the challenges in performance management, potential trust in CIPM, and the potential costs associated with adopting CIPM.

The survey design follows guidelines from Linåker et al. [Lin+15] and Ralph et al. [Ral+21], to improve the quality criteria and essential attributes for questionnaire surveys. The target population consists of professionals involved in software development, representing various roles to gain diverse perspectives. A non-probabilistic

sampling method was used, with two recruitment rounds, including both personal contact outreach and survey platform participation. The survey received a total of 72 responses by November 1, 2024.

Regarding Cost, we investigate the factors considered when introducing a new performance prediction tool, e.g., CIPM, like the cost implications of learning and adopting CIPM in the industry. Specifically, we asked for the time required for setting up and maintaining CIPM, including the process of defining or adjusting CPRs. The effort involved in these tasks is expressed in terms of work time, providing an understanding of the acceptable cost of adopting CIPM in industrial settings.

Regarding insights on trust, we asked the participants about the level of trust and confidence that organizations have in AbPP and further asked whether the properties of CIPM can enhance this trust. The properties of CIPM that are provided for increasing trust include the availability of metrics regarding the accuracy of the prediction results, the calibration of the performance model based on monitoring data, and the validation of prediction results with monitoring data. The operational validation provided by CIPM is considered one of the various stages required for establishing trust in a model [HMY21]. Thus, we want to investigate its impact on trust.

Beyond cost and trust assessment, the survey explores challenges in performance management to detect areas that require improvement. We inquire about any limitations or desired features in performance management.

The initial results of our survey [AMK24] provide insights into the questions regarding the cost of the CIPM approach. For **EQ-4**, which explores how much time the industry would be willing to invest in setting up and learning a new performance prediction tool like CIPM, participants, including project managers and teams members indicate a preference for spending two to five working days for learning and setting up a new tool. Regarding the adoption of the programming languages and technologies, the time ranges between one and four weeks, with two tendencies. These findings show that the time participants are willing to spend learning and adopting the tool aligns with our ideal setup, learning, and adoption time for CIPM.

Regarding **EQ-5**, which investigates the factors influencing industry professionals' trust in CIPM 's features, the results suggest that all three properties of the CIPM approach significantly contribute to improving trust in performance predictions. The combination of these features is expected to further increase trust.

The survey setup and results are discussed in detail in our preprint [AMK24]. Collecting more responses and further analysis of the data, is the next step to uncover additional findings and to publish these results in a peer-reviewed conference.

13.1.2. Version Controlling within CIPM

The goal of this future work is to apply version control system (VCS) principles to VSUM, including software models, ensuring its versions are controlled and consistent with the

software version controlling system. This includes maintaining these mappings using digital twins, ensuring that VSUM mirrors the same versions as the version control system, as described in Section 5.1.1.

To ensure that VSUM includes all related software models, this future work includes also implementing the configuration for committing changes (measurements in our case) from the Vitruvius view into the related model (measurement model) at time intervals to move the implementation of the window sliding approach into Vitruvius, allowing us to cover all consistency preservation processes in a common place (VSUM) within Vitruvius platform.

Combining the application of VCS to VSUM through digital twins will reduce the effort required to update the aPM if the user decides to roll back changes or check out old source code changes. In this case, the digital twins will apply the same commands as the developer to ensure that VSUM, along with the corresponding aPM, is retrieved. This also includes ensuring that the new monitoring data is written to the measurement repository related to this version.

13.1.3. Generalizing Consistency Preservation Rules for CIPM

To enhance the CIPM approach, ongoing work focuses on generalizing CPRs and refining the definition process. Currently, CIPM uses technology-specific language-specific CPRs. Handling the source code based on multiple technologies and/ or programming languages consequently may be costly because the used CPRs for updating aPM should be changed according to the programming language and technology used by the changed source code.

Ongoing master work [Wen24] addresses the challenges regarding the use of different technologies or changing technology, focusing on the following points:

- **Generalization of CPRs**: Develop general CPRs applicable across various projects and technologies to reduce the need for frequent updates.
- **Standardized CPR Definition Process**: Create a standardized process for extracting, creating, and documenting CPRs, adaptable to different technologies and contexts.
- **Template Design for CPRs**: Design user-friendly templates for defining and documenting general CPRs to streamline the implementation process.

The previous process should be extended to study the applicability of defining language-independent CPRs that update aPM based on source code changes extracted from different programming languages.

13.1.4. Improvement of Scalability

Future work can focus on evaluating the scalability of the optimization approach using a genetic algorithm. This involves addressing the trade-off between the accuracy of aPMs and the computational overhead introduced by the configuration of the GA. As the complexity of the GA settings increases, so does the overhead, which could hinder scalability in large amounts of monitoring data. The works aim to allow dynamic optimization of the GA parameters, such as population size, mutation rate, and crossover techniques, based on the amount of monitoring data to improve the efficiency of maintaining the accuracy of the models without affecting the scalability.

Moreover, hybrid optimization techniques, such as combining GA with simulated annealing or particle swarm optimization, will be explored to enhance scalability and adaptability, ensuring the solution remains feasible even for more extensive systems. This will enable better management of the trade-offs between optimization effectiveness and computational cost in real-world scenarios.

13.1.5. Enhancing CIPM with Round-Trip Engineering

CIPM currently lacks forward engineering capabilities, which limits its efficiency and adaptability in dynamic development environments. To address this gap, we propose enabling round-trip engineering to support techniques commonly applied in domains such as the automotive industry (Section 13.1.6).

The first step in starting with CIPM, when an existing aPM is available, is to integrate legacy models into Vitruvius platform. Building on our previous work in the automotive domain [Maz+18; Maz16], where we integrated multiple models into the Vitruvius consistency preservation platform, this approach can be adapted for software system models. The process includes checking for consistency between the manual aPM and the source code, integrating them to link the corresponding elements, and resolving possible inconsistencies.

Evolve the aPM: After ensuring consistency between the aPM and the source code, the next step is to evolve the aPM. This involves updating the aPM to reflect changes in the software architecture and performance requirements. By evolving a copy of aPM, CIPM can support forward engineering through generating required source code and learning from the manually modeled aPM by further consistency preservation, to anticipate future performance challenges.

Generate Source Code Stubs Using CPRs: Once the aPM has been evolved, CIPM will use the predefined CPRs, similar to the Coevolution approach [Lan17], to generate source code stubs. These stubs serve as a starting point for the development of new features or the enhancement of existing ones. Developers can further evolve these stubs and integrate them into CI pipeline, ensuring that the system's architecture remains consistent with the aPM.

Update the aPM in VSUM: Based on CPRs and knowledge from the manually evolved aPM, the aPM will be updated in the VSUM, considering both the CPRs and any manual adjustments made by developers. This ensures that the aPM reflects the latest state of the system, including any manual enhancements. By incorporating knowledge from manual updates, CIPM ensures that the CI-based strategy not only automates updates but also leverages human expertise where available. This step should be thoroughly implemented and evaluated to confirm its effectiveness in maintaining the accuracy and relevance of the aPM throughout the software lifecycle.

13.1.6. Performance Analysis for Automobile Applications

We aim to ensure consistency between automotive system models using CIPM, as the automotive software adopts continuous deployment practices. Continuous deployment is carried out in short development cycles to improve safety, correct failures, and introduce new features. Consequently, CIPM aims to provide quick feedback on performance after such changes.

Models used in the automotive application, such as OpenTOSCA [1], Autosar [2], AMALTHEA [3] or ADOM [Maz16], need to be integrated into Vitruvius framework. The effort to integrate the models above is expected to be manageable due to the compatibility of these EMF-based models with the Vitruvius framework [Maz+18]. This integration also involves defining mappings between these models and the corresponding source code and monitoring data, establishing consistency preservation rules, and ensuring seamless data flow between the models and runtime data.

However, conceptual improvements in the management and analysis of large-scale monitoring data are required. The challenge lies in developing mechanisms to effectively handle such data, which will require innovative approaches to storage, processing, and analysis, as explored in the context of the SofDCar [4] project. This management must be adaptable to handle the complexities introduced by multiple software variants, ensuring efficient data management across diverse configurations and deployments [Stü+24]. The main required improvements of CIPM for supporting automotive systems are described in Section 13.1.2 and Section 13.1.5.

13.1.7. Support Diverse Version Control Systems

Future work could explore whether the integration of additional version control systems requires conceptual adjustments to the CIPM pipeline, or whether it can be achieved through purely technical extension. The current limitation of the CIPM implementation

[1] see https://www.opentosca.org/
[2] see https://www.autosar.org/
[3] see https://https://itea4.org/project/amalthea.html/
[4] see https://sofdcar.de

to applications stored on Git presents significant opportunities for future advancements. A promising direction for such developments is to implemnt the functionality of the CIPM to seamlessly integrate with other version control systems and to find out whether a conceptual extension of CIPM is required. These can include, but are not limited to, systems such as Mercurial[5] or Perforce[6].

Such an expansion would allow a wider range of applications to benefit from the performance prediction capabilities of CIPM. Incorporating these additional version control systems would make the CIPM implementation applicable in a wider variety of software development environments, thereby enhancing its utility and impact.

Additionally, this would not only allow CIPM to be more versatile in its application but would also contribute to a more comprehensive understanding of how to generalize the process of interpreting development-time changes, such as source code modifications, from different version control systems. Future research should address whether these improvements require conceptual changes to the current CIPM framework or whether they can be achieved solely through implementation efforts.

13.1.8. Adaptive Performance Testing for MbDevOps Pipeline

The CIPM incremental calibration of aPM at Dev-time requires executing benchmarking/ selective performance tests.

Adaptive performance testing for the MbDevOps pipeline pipeline requires applying minimum performance tests, as extensive testing is time-consuming and not ideal for DevOps practices (cf. Section 2.1.4). This can be achieved by focusing the testing efforts on the components impacted by the changes, based on the architectural knowledge captured from the VSUM. By identifying the relevant components, the necessary performance tests can be automatically generated and executed.

Declarative performance engineering [Wal+16] offers a promising solution for this purpose, allowing the definition of tests through the use of domain-specific languages at a high level of abstraction, independent of complex infrastructure. The core idea behind declarative performance engineering is to address user concerns related to software performance, emphasizing the decoupling of user concerns from solution approaches, the automation of performance queries, and the concealment of complexity. This approach reduces the effort required for performance testing.

Tools and frameworks like Ferme's declarative domain-specific language and model-driven framework [Fer21] enable explicit declaration of test goals, improving quality in continuous software development. Integration of these approaches with the MbDevOps pipeline pipeline can help in configuring and executing minimum performance tests at Dev-time. Such oriented performance testing can be achieved by utilizing CIPM's

[5] see `https://www.mercurial-scm.org/`

[6] see `https://www.perforce.com/`

knowledge of changed source code to automatically configure the required tests for the changed service through model-based transformation between the instrumentation model and written tests. This enables the application of performance tests suitable for specific changes in the code.

Besides, utilizing realistic workloads like [Vög+18; Tru+18] and automated performance unit test [HHF13] can improve the accuracy of the resulting measurements and the cost of the performance testing.

13.1.9. Expanding the Consistency Maintenance Range of CIPM

We aim to enhance CIPM capabilities by extending the current CI-based update process. This expansion involves considering documentation (Section 13.1.9.1) and configuration files within a Git repository (Section 13.1.9.2) in addition to considering more technologies (Section 13.1.9.3).

13.1.9.1. Considering the Informal Documentation

The initial focus is on leveraging documentation files as input to augment the automation level of CIPM during continuous integration. This involves integrating our approach with Fuchß's method [Fuc21], which processes sketches and natural language to extract insights about architectures. Fuchß's approach takes textual and structural inputs, such as UML diagrams, and produces JSON files containing architectural insights. In our future work, we propose extending Fuchß's approach to handle documentation files from commits, including those in Git-Wiki or readme files. The extracted results can then be systematically stored in VSUM, necessitating the definition of a metamodel for the JSON files. This resulting model will empower our consistency rules, enabling more accurate detection of architectural components during continuous integration.

13.1.9.2. Considering the Configuration Files for Enhanced Architecture Detection

We aim to extend CIPM's knowledge base by processing additional configuration files, such as eXtensible Markup Language (XML) and YAML, to enhance the update process of aPMs. This will improve static behavior modeling by incorporating more information about dependency injection (Section 13.1.10.1) and refine the modeled resource environment through improved host architecture and component detection, leveraging the structure and metadata found in XML configurations.

Previous works [Lan17; Kir+24] have employed configuration files to capture architectural knowledge and address part of the desired enhancement scope. Future work can include different configuration files in the consistency preservation process, through parsing and processing them in VSUM. This strategy allows CIPM to automatically

update aPMs based on various knowledge resources, enabling extracting information about the deployment, resources, and dependencies.

13.1.9.3. Enhancing CIPM's Support for Technologies

The current CIPM implementation supports two programming languages, including Java and Lua. Additionally, CIPM detects the architecture based on source code structure and the technologies used, including microservices, REST technologies, and industrial IoT. However, modern software systems are increasingly complex, often involving the use of multiple programming languages and technologies during development.

Looking ahead, a crucial area of focus is to investigate how CIPM can be enhanced to support a wider variety of languages and technologies. This expansion could significantly increase the applicability and impact of CIPM, given the diverse and evolving nature of software development practices today. Potential areas of exploration include how to adapt CIPM to accommodate different programming paradigms and architectural styles, as well as how to handle the performance considerations associated with various web technologies.

In fact, a new Ph.D. work in our group is already tackling some of these challenges. The work aims to determine whether conceptual changes to CIPM will be required to support a broader range of programming languages and technologies. Another goal of this work is to investigate the potential benefits of using a general metamodel for programming languages. This could potentially reduce the initial overhead required to adopt CIPM for performance prediction in the industrial context.

13.1.10. Enhancing the Abstract Behavior

This section focuses on enhancing abstract behavior through improved dependency resolution for dynamic SEFF updates (Section 13.1.10.1), detection of event-based systems (Section 13.1.10.2), and calibration of synchronous communication to align system behavior with evolving architectural models (Section 13.1.10.3).

13.1.10.1. Enhancing Dependency Resolution for Dynamic Update of SEFF

The goal is to enhance `SEFF` extraction to represent dependency injection through branch actions that vary by dependency type. CPRs can detect the dependencies' patterns like constructor signatures, setter methods, and specific annotations such as '@Autowired' and '@Inject'. These annotations and patterns signal where dependencies are injected and are modeled as conditional branches based on the dependency type, which is subsequently passed as a usage argument. For example, Langhammer has established such rules for Google Guice, leveraging its adherence to the JSR 330 standard, suggesting adaptability to similar dependency injection frameworks [Lan17].

To calibrate the branch at Dev-time, the static analysis should be extended to configuration files. For instance, dependencies injected through XML files in Spring, YAML configuration in Micronaut, and JSON configuration in Node.js-based frameworks could be analyzed. At Ops-time time, calibration requires a custom instrumentation probe. When the CPRs recognize the addition of a branch for dependency injection, they must insert a custom probe that logs the dependency type at runtime. This allows Ops-time calibration to dynamically update `SEFF`, ensuring it reflects the currently active instance.

13.1.10.2. Detecting Event-based Systems

Event-based systems are widely used across various domains [HSB09]. According to [Bro+13], these systems consist of four essential components: source, sink, transmission system, and event. The source emits meaningful events that affect the system. The sink processes incoming events and reacts to them. The transmission system facilitates communication between the source and sink, allowing the source to invoke event processing or the sink to actively request new events for processing [Sac10]. The event represents a significant change in state as defined by Chandy [Cha06], which triggers communication and processing between the source and sink.

Ratherfeld [Rat13] extends the `Palladio` metamodel to support event-based communication within architectural frameworks. Additionally, Singh et al. [SK24] provide automated reverse engineering of message-oriented middleware through static analysis of source code.

Future work will focus on enhancing the static analysis capabilities of CIPM to detect the transmission system and apply custom instrumentation for monitoring. Following this, the dynamic analysis extension of CIPM will aim to determine the specific communication patterns between the source and sink. Furthermore, this dynamic analysis will calibrate the transmission system to facilitate the simulation of event-based systems. These advancements are intended to improve performance prediction accuracy and provide deeper insights into the behavior of event-driven architectures.

13.1.10.3. Calibrating Synchronous Communication

CIPM 's static analysis can detect RESTful communication patterns at Dev-time, identifying REST endpoints, HTTP methods, and request-response. Custom instrumentation can be added to capture these interactions and monitor data flows across components. Following this, a dynamic analysis extension of CIPM can focus on calibrating REST-based interactions at Ops-time, thereby updating the performance model with observed runtime behaviors. This calibration will enable accurate simulation and prediction of REST interactions within self-aware systems, enhancing adaptability and providing actionable insights into the system's behavior under varying workloads and interaction patterns.

13.1.11. Combining Static and Dynamic Analysis for Incremental Calibration

To enhance the accuracy and efficiency of performance model parameter estimation, a combined approach using both static and dynamic analysis can be integrated into CIPM. This extension allows for the detection of direct parametric dependencies, such as external or internal calls that utilize input parameters, and provides an initial estimation of resource demands for internal actions during the CI-update of the aPM. This is especially beneficial in cases where no automated benchmarks are available during development.

The static analysis can be applied to the source code associated with internal actions, aiming to detect low-resource-demand actions, such as internal actions, with just a simple assignment statement. This can be detected during the `SEFF` extraction process, where the source code statements are assigned to the related internal actions. In this step, the statements related to an internal action can be analyzed to estimate resource demand statically. For instance, approaches similar to [KKR08] can be used to estimate resource demands before runtime monitoring.

Since adaptive instrumentation involves adding probes to monitor the changed source code, combining an estimation based on the static analysis can reduce monitoring overhead. If the self-validation process finds the accuracy of some parameter estimates acceptable, it can keep certain probes inactive, thus minimizing the monitoring overhead. Additionally, the static analysis step can identify the primary resource type associated with an internal action, such as CPU or disk demand, which can inform subsequent calibration steps.

13.1.12. Multi-Resource Demand Calibration for Internal Actions

In the current approach, it is assumed that each internal action's resource demand is dominated by a single resource. Consequently, only one aggregated record is used to monitor the resource demands of an internal action. This assumption simplifies the calibration process but fails to capture cases where an internal action has significant demands on multiple resources, such as both CPU and disk. When demands are spread across several resources, aggregating them into a single measure may lead to inaccurate performance predictions, as the contributions of individual resources are not explicitly accounted for.

However, it is conceptually possible that CIPM considers the different resources to increase the accuracy of the calibration. Thus, we propose a solution similar to the approach of Brosig et al. [BKK09] that measures the processing times of individual execution fragments, ensuring that the measured times are dominated by a single resource. To automate such a solution, we propose utilizing the static analysis, similar

to what is explained in Section 13.1.11, to detect the various execution fragments, enabling instrumenting them and monitoring of individual execution fragments.

The static analysis of the source code associated with internal actions during **C1** can identify potential multi-resource demands and help determine whether an action involves significant usage of more than one resource type. The analysis can detect the demand for disk resources through the use of some libraries or patterns referring to disk demands. If these patterns are used, the CI-based update of aPM can update the instrumentation model, by adding more than one probe related to this internal action, referring that there are multi-resource demands.

The adaptive instrumentation (**C2**) can then generate an instrumentation code using separate records for each resource type. The source code for instrumentation will combine the internal action identifier with the resource type, ensuring that each monitored record corresponds to a specific resource.

The calibration process (**C3**) can then handle these separate records, accurately estimating the resource demand for each type.

Finally, the accuracy of the multi-resource calibration can be validated and refined as needed, aiming to improve the accuracy of predicted performance with observed measurements.

13.2. Conclusions

Considering the software architecture model increases the understandability as well as the productivity of software development [OEW17]. Moreover, applying AbPP promises proactive detection of performance problems by simulation instead of the expensive measurement-based performance prediction.

This thesis addresses the research question, "How can a descriptive aPM be accurately updated and automatically validated with minimal overhead within CI/CD pipelines?" by proposing the Continuous Integration of architectural Performance Models (CIPM) approach, which integrates automated consistency preservation, incremental calibration, and validation mechanisms into the CI/CD pipeline.

For that, the approach defines the main consistency relations that should be preserved to enable accurate architecture-based performance prediction.

At Dev-time, the CI-based strategy (**C1**) extracts source code changes from commits and updates the structure of aPM, including abstract behavior and the system composition.

The adaptive instrumentation (**C2**) targets changed source code parts by inserting custom predefined probes to collect measurements, enabling calibration of the affected parameters of the aPM.

To allow the simulation of aPM, our calibration (**C3**) estimates the parametrized PMPs incrementally and uses a novel incremental resource demand estimation based

on adaptive monitoring. The calibration identifies the parametric dependencies and optimizes them based on the genetic algorithm.

In addition to PMPs, the Ops-time calibration (**C4**) observes the adaptive changes and updates the affected parts of aPM accordingly. This applies to changes in deployment, resource environment, usage, and even system composition. The proposed self-validation (**C5**) continuously analyses the accuracy of the AbPP. The results of self-validation are used to manage the monitoring and calibrate aPM at Ops-time.

Besides, we describe in this thesis how to integrate CIPM approach, including the contributions above, in an automated MbDevOps pipeline pipeline (**C6**), ensuring that the aPM remains up-to-date throughout DevOps SDLC.

For our validation, we adopted a measurement-based approach to validate our approach at two levels. In the first level, we assessed the accuracy of the updated aPMs and the accuracy of the associated AbPP by comparing performance predictions with actual measurements. At the second level, we focused on evaluating the applicability of CIPM during DevOps development. Thus, we assessed the required monitoring overhead and scalability. The validation utilized three Java-based cases (TeaStore, CoCoME, and TEAMMATES), and two industrial Lua-based sensor applications from SICK AG [AG24a].

We conducted two experiments to validate CIPM at Dev-time (**E1** and **E2**) by analyzing real software repositories containing approximately 18,000 history commits. Besides, we validated CIPM at Ops-time by conducting three empirical experiments (**E3**, **E4**, and **E5**).

Our findings indicate that CIPM can keep the aPM up-to-date and accurately estimate performance parameters, thereby enabling accurate performance predictions. Additionally, calibrating performance parameters while considering the parametric dependencies can significantly enhance the predictive accuracy of AbPP. Besides, CIPM's adaptive instrumentation reduced the number of required probes by 12.6% to 69%, depending on the commit changes affecting PMPs. Adaptive monitoring further reduces the overhead by up to approximately 40%. Lastly, CIPM demonstrates reasonable execution times and scalability with an increasing number of model elements and monitoring data, confirming its practicality for real-world applications.

Despite the promising capabilities of CIPM, challenges remain in adapting CIPM to more complex, heterogeneous environments involving varying technologies and programming languages. In this thesis, we describe future work to extend the adaptability of CIPM to ensure its effective application across diverse contexts and technologies.

List of Acronym

AbPP	Architecture-based Performance Prediction	i
AbPV	Architektur-basierte Performance-Vorhersage	iii
AOP	aspect-oriented programming	19
API	application programming interface	30
aPM	architectural Performance Model	i
APM	Application Performance Management	4
AST	Abstract Syntax Tree	26
AUT	Application Under Test	271
JDT	Java Development Tool	26
JIT	Just-In-Time	272
CD	continuous deployment	ii
CDF	Cumulative Distribution Function	50
CDFs	Cumulative Distribution Functions	xvii
CI	continuous integration	ii
CIPM	Continuous Integration of architectural Performance Models	i
CIQM	Continuous Integration of architectural Quality Models	61
CPR	Consistency Preservation Rule	42
CoCoME	Common Component Modeling Example	178
PMP	Performance Model Parameter	5
CPU	Central Processing Unit	19
Dev-time	Development time	xx
Ops-time	Operation time	3
DML	Descartes Modeling Language	271
DSL	domain-specific language	22
E	Experiment	16
IoT	Internet of Things	182
EJB	Enterprise JavaBeans	270
EMF	Eclipse Modeling Framework	24
EQ	Evaluation Question	171

SCM Source Code Model . . . 64
RepM Repository Model . . . 189
GA Genetic Algorithm . . . 125
GQM Goal Question Metric . . . 44
HTTP Hypertext Transfer Protocol . . . 30
Institute of Electrical and Electronics Engineers
IM Instrumentation Model . . . 72
IRL Instrumentation Record Language . . . 21
IRE Incremental Reverse Engineering . . . 108
JaMoPP Java Model Parser and Printer . . . 27
JSON JavaScript Object Notation . . . 109
JC Jaccard similarity Coefficient . . . 49
KS-test Kolmogorov-Smirnov-test . . . xvii
MARTE UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems . . . 270
MbDevOps Model-based DevOps . . . 62
MDD Model-Driven Development . . . 22
MDE Model-Driven Engineering . . . 22
MDSD Model-Driven Software Development . . . 22
MIA Monitored Internal Action . . . 122
MM Measurements Model . . . 64
MOF Meta Object Facility . . . 24
NMIA Not Monitored Internal Action . . . 122
PCM Palladio Component Model . . . xvi
PMP Performance Model Parameter . . . 5
RD Resource Demand . . . 32
RDE Resource Demand Estimation . . . 121
RepM Repository Model . . . 189
REST Representational State Transfer . . . 30
MSE Mean Squared Error . . . 125
SCM Source Code Model . . . 64

SCG Service-Call-Graph xvi
SEFF Service Effect Specification 31
SIM Sensor Integration Machine 182
SLA service level agreement 261
SDLC Software Development Life Cycle 7
SM Statistical Metric 67
SPE Software Performance Engineering 4
StoEx stochastic expression 35
UML Unified Modeling Language 24
VCS version control system 278
VSUM Virtual Single Underlying Model xix
WS-distance Wasserstein distance xvii
XML eXtensible Markup Language 283

Bibliography

The titles in this bibliography are hyperlinks pointing to sources such as DOIs or other online resources.

[AAM10] Alae-Eddine El Hamdouni, Abdelhak-Djamel Seriai, and Marianne Huchard. "Component-based Architecture Recovery from Object Oriented Systems via Relational Concept Analysis". In: 2010, pp. 259–270.

[Abb+12] F. Abbors, T. Ahmad, D. Truscan, and I. Porres. "MBPeT: a model-based performance testing tool". In: *2012 Fourth International Conference on Advances in System Testing and Validation Lifecycle*. sn. 2012, p. 18.

[AC08] M. Antkiewicz and K. Czarnecki. **"Design Space of Heterogeneous Synchronization"**. In: *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*. 2008, pp. 3–46.

[ACB17] M. Arjovsky, S. Chintala, and L. Bottou. "Wasserstein generative adversarial networks". In: *International conference on machine learning*. PMLR. 2017, pp. 214–223.

[Ack+18] V. Ackermann, J. Grohmann, S. Eismann, and S. Kounev. "Black-box Learning of Parametric Dependencies for Performance Models". In: *Proceedings of 13th Workshop on Models@run.time (MRT), co-located with MODELS 2018* (Oct. 14, 2018). 2018.

[Ahm+16] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang. "Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report". In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 2016, pp. 1–12.

[Ala09] A. Alakeel. "A Guide to Dynamic Load Balancing in Distributed Computer Systems". In: *International Journal of Computer Science and Network Security (IJCSNS)* 10 (2009).

[AM14] Amritesh and S. C. Misra. **"Conceptual modeling for knowledge management to support agile software development"**. In: *Cambridge University Press* 29.4 (2014), pp. 496–511.

[AMK23] M. Armbruster, M. Mazkatli, and A. Koziolek. “Recovering Missing Dependencies in Java Models”. In: *Softwaretechnik-Trends Band 43, Heft 4*. 2023, pp. 38–40.

[AMK24] M. Armbruster, M. Mazkatli, and A. Koziolek. ***Trust and Costs for Evolving Architectural Performance Models: A Survey***. Tech. rep. To be submitted to ICSA25, A preprint is available at KIT library. KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology, 2024.

[Arm21] M. Armbruster. “Commit-Based Continuous Integration of Performance Models”. Supervised by Manar Mazkatli. Master Thesis. Karlsruher Institut für Technologie, 14, 2021.

[Arm22] M. Armbruster. ***Parsing and Printing Java 7-15 by Extending an Existing Metamodel***. Tech. rep. Karlsruhe Institut for Technology, 2022.

[ASB17] R. Abbas, Z. Sultan, and S. Bhatti. **“Comparative Study of Load Testing Tools: Apache JMeter, HP LoadRunner, Microsoft Visual Studio (TFS), Siege”**. In: *Sukkur IBA Journal of Computing and Mathematical Sciences* 1.2 (2017), pp. 102–108.

[Avr+20] A. Avritzer, V. Ferme, A. Janes, B. Russo, A. v. Hoorn, H. Schulz, D. Menasché, and V. Rufino. **“Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests”**. In: *Journal of Systems and Software* 165 (1, 2020), p. 110564.

[BBM03] F. Budinsky, S. A. Brodsky, and E. Merks. ***Eclipse Modeling Framework***. Pearson Education, 2003.

[BBM13] M. Becker, S. Becker, and J. Meyer. “SimuLizar: Design-Time Modelling and Performance Analysis of Self-Adaptive Systems”. In: *Proceedings of Software Engineering 2013*. Vol. P-213. 2013, pp. 71–84.

[BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach. “The Goal Question Metric Approach”. In: *Encyclopedia of Software Engineering - 2 Volume Set*. 1994, pp. 528–532.

[Bec+10] S. Becker, M. Hauck, M. Trifu, K. Krogmann, and J. Kofron. “Reverse Engineering Component Models for Quality Predictions”. In: *Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track*. IEEE. 2010, pp. 199–202.

[Bec+13] S. Becker, R. Mirandola, L. Happe, and C. Trubiani. “Towards a methodology driven by dependencies of quality attributes for QoS-based analysis”. In: *Proceedings of the 4th Joint ACM/SPEC International Conference on Performance Engineering (ICPE ’13), Work-In-Progress Track*. 2013.

[Beh+20] W. Behutiye, P. Karhapää, L. López, X. B. Illa, S. Martínez-Fernández, A. M. Vollmer, P. Rodríguez, X. Franch, and M. Oivo. **"Management of quality requirements in agile and rapid software development: A systematic mapping study"**. In: *Inf. Softw. Technol.* 123 (2020), p. 106225.

[Bet16] L. Bettini. ***Implementing Domain Specific Languages with Xtext and Xtend - Second Edition***. 2nd. Packt Publishing, 2016.

[Bez+19a] C.-P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker. "How is Performance Addressed in DevOps?" In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 4, 2019, pp. 45–50.

[Bez+19b] C. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker. "How is Performance Addressed in DevOps?" In: *Proceedings of the 2019 ACM/SPEC Int. Conference on Performance Engineering*. 2019, pp. 45–50.

[Béz05] J. Bézivin. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188.

[BG18] J. Bang-Jensen and G. Gutin, eds. ***Classes of directed graphs***. 1st ed. 2018. Springer, 2018.

[BGM87] I. Benbasat, D. K. Goldstein, and M. Mead. **"The Case Research Strategy in Studies of Information Systems"**. In: *MIS Quarterly* 11.3 (1987), pp. 369–386.

[BHK11] F. Brosig, N. Huber, and S. Kounev. "Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. 2011, pp. 183–192.

[Bie16] M. Biehl. *RESTful Api Design*. Vol. 3. API-University Press, 2016.

[BKK09] F. Brosig, S. Kounev, and K. Krogmann. "Automated Extraction of Palladio Component Models from Running Enterprise Java Applications". In: *Proceedings of the 1st International Workshop on Run-time mOdels for Self-managing Systems and Applications (ROSSA 2009). In conjunction with the Fourth International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2009)*. 2009, 10:1–10:10.

[BKR09] S. Becker, H. Koziolek, and R. Reussner. **"The Palladio component model for model-driven performance prediction"**. In: *The Journal of Systems and Software* 82 (2009), pp. 3–22.

[BMK16] E. Burger, V. Mittelbach, and A. Koziolek. "View-based and Model-driven Outage Management for the Smart Grid". In: *Proceedings of the 11th Workshop on Models@run.time co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. 2016.

[Bon14] A. B. Bondi. ***Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice***. 1st. Addison-Wesley Professional, 2014.

[BP08] C. Brun and A. Pierantonio. "Model differences in the eclipse modeling framework". In: *UPGRADE, The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34.

[BR08] R. Böhme and R. Reussner. **"Validation of Predictions with Measurements"**. In: *Dependability Metrics*. Vol. 4909. 2008. Chap. 3, pp. 14–18.

[Bri+98] L. Briand, S. Carrière, R. Kazman, and J. Wüst. **"COMPARE: A Comprehensive Framework for Architecture Evaluation"**. In: (1998), pp. 48–49.

[Bro+13] F. Brosig, F. Gorsler, N. Huber, and S. Kounev. "Evaluating Approaches for Performance Prediction in Virtualized Environments". (Short Paper). In: *Proceedings of the IEEE 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2013)*. 2013.

[Bro+15] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev. **"Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-based Architectures"**. In: *Software Engineering, IEEE Transactions on* 41.2 (2015), pp. 157–175.

[Bru+15] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. R. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert. ***Performance-oriented DevOps: A Research Agenda***. Tech. rep. SPEC-RG-2015-01. SPEC Research Group - DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), 2015.

[Buc+13] J. Buckley, S. Mooney, J. Rosik, and N. Ali. "JITTAC: A Just-in-Time tool for architectural consistency". In: *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 1291–1294.

[Bur23] L. Burgey. "Continuous Integration of Performance Models for Lua-Based IIoT Applications". Supervised by Manar Mazkatli and Martin Armbruster. Master Thesis. Karlsruher Institut für Technologie, 15, 2023.

[Bus98] F. Buschmann. ***Pattern-orientierte Software-Architektur: ein Pattern-System***. Addison-Wesley, 1998.

[BV08] B. W. Boehm and R. Valerdi. "Achievements and Challenges in COCOMO-Based Software Resource Estimation". In: *IEEE Software* 25.5 (2008), pp. 74–83.

[BVK13] A. Brunnert, C. Vögele, and H. Krcmar. "Automatic performance model generation for java enterprise edition (ee) applications". In: *Computer Performance Engineering*. Springer. 2013, pp. 74–88.

[BWZ15] L. Bass, I. Weber, and L. Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[Cal+90] D. Callahan, A. Carle, M. Hall, and K. Kennedy. **"Constructing the procedure call multigraph"**. In: *IEEE Transactions on Software Engineering* 16.4 (1990), pp. 483–487.

[Can+18] M. Canat, N. Catala, A. Jourkovski, S. Petrov, M. Wellme, and R. Lagerström. **"Enterprise Architecture and Agile Development: Friends or Foes?"** In: *2018 IEEE 22nd International Enterprise Distributed Object Computing Workshop (EDOCW)* (2018), pp. 176–183.

[CDM20] A. Capizzi, S. Distefano, and M. Mazzara. "From DevOps to DevDataOps: Data Management in DevOps Processes". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. 2020, pp. 52–62.

[CDS10] P. Cremonesi, K. Dhyani, and A. Sansottera. "Service Time Estimation with a Refinement Enhanced Hybrid Clustering Algorithm". In: *Analytical and Stochastic Modeling Techniques and Applications*. 2010, pp. 291–305.

[Chr10] H. S. Christopher D. Manning Prabhakar Raghavan. **"Introduction to Information Retrieval"**. In: *Natural Language Engineering* 16.1 (2010), pp. 100–103.

[Cor+22] V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci. **"A model-driven approach for continuous performance engineering in microservice-based systems"**. In: *Journal of Systems and Software* 183 (2022), p. 111084.

[Cou+17] N. Courty, R. Flamary, A. Habrard, and A. Rakotomamonjy. "Joint distribution optimal transportation for domain adaptation". In: *Advances in neural information processing systems* 30 (2017).

[CS12] P. K. Chittimalli and V. Shah. "GEMS: A Generic Model Based Source Code Instrumentation Framework". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, pp. 909–914.

[CS14] P. Cremonesi and A. Sansottera. **"Indirect estimation of service demands in the presence of structural changes"**. In: *Performance Evaluation* 73 (1, 2014), pp. 18–40.

[CSF20] J. Á. Cid-Fuentes, C. Szabo, and K. Falkner. **"Adaptive Performance Anomaly Detection in Distributed Systems Using Online SVMs"**. In: *IEEE Transactions on Dependable and Secure Computing* 17 (2020), pp. 928–941.

[Cut11] M. Cuturi. "Fast computation of wasserstein distances and statistical learning". In: *Advances in Neural Information Processing Systems*. 2011, pp. 2292–2300.

[CW00] M. Courtois and M. Woodside. "Using Regression Splines for Software Performance Analysis". In: *Proceedings of the 2nd Int. Workshop on Software and Performance*. 2000.

[CZ01] F. Chan and J. Zhang. **"Modelling for agile manufacturing systems"**. In: *International Journal of Production Research* 39 (2001), pp. 2313–2332.

[Det12] M. von Detten. "Archimetrix: A Tool for Deficiency-Aware Software Architecture Reconstruction". In: *WCRE 2012*. 2012.

[Dis+11] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. "From state-to delta-based bidirectional model transformations: The symmetric case". In: *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings 14*. Springer. 2011, pp. 304–318.

[DM01] L. Ding and N. Medvidovic. "Focus: a light-weight, incremental approach to software architecture recovery and evolution". In: *Proceedings Working IEEE/IFIP Conference on Software Architecture*. 2001, pp. 191–200.

[Dod08] Y. Dodge. "Kolmogorov–Smirnov Test". In: *The Concise Encyclopedia of Statistics*. 2008, pp. 283–287.

[DR18] G. K. Dziugaite and D. M. Roy. "Computing Nonparametric Entropy Rates with Applications to Privacy". In: *arXiv preprint arXiv:1807.08350* (2018).

[DS98] N. R. Draper and H. Smith. *Applied regression analysis*. Vol. 326. John Wiley & Sons, 1998.

[EH11] J. Ehlers and W. Hasselbring. **"A Self-adaptive Monitoring Framework for Component-Based Software Systems"**. In: (2011), pp. 278–286.

[Eis+20] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn. "Microservices: A Performance Tester's Dream or Nightmare?" In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 138–149.

[Eis23] S. Eismann. “Performance Engineering of Serverless Applications and Platforms”. doctoralthesis. Universität Würzburg, 2023.

[EKR02] H.-F. Eckey, R. Kosfeld, and M. Rengers. ***Multivariate Statistik***. 2002.

[Ely+23] M. Elyasi, M. E. Simitcioğlu, A. Saydemir, A. Ekici, O. Ö. Özener, and H. Sözer. **“Genetic algorithms and heuristics hybridized for software architecture recovery”**. In: *Automated Software Engineering* 30.2 (26, 2023), p. 19.

[Fer21] V. Ferme. “Declarative Performance Testing Automation: Automating Performance Testing for the DevOps Era”. PhD. USI, 2021.

[FF09] O. F. F. Filho and M. A. G. V. Ferreira. “Semantic Web Services: A RESTful Approach”. In: *IADIS International Conference WWWInternet 2009*. 2009, pp. 169–180.

[FIE99] R. FIELDING. **“Hypertext transfer protocol-HTTP/1.1. IETF RFC 2616”**. In: *http://www.ietf.org/rfc/rfc2616.txt* (1999).

[FKH17] F. Fittkau, A. Krause, and W. Hasselbring. **“Software landscape and application visualization for system comprehension with ExplorViz”**. In: *Information and Software Technology* 87 (2017), pp. 259–277.

[FRS16] W. R. Fitriani, P. Rahayu, and D. I. Sensuse. **“Challenges in agile software development: A systematic literature review”**. In: *2016 International Conference on Advanced Computer Science and Information Systems (ICACSIS)* (2016), pp. 155–164.

[Fuc21] D. Fuchß. “Sketches and Natural Language in Agile Modeling”. In: *15th European Conference on Software Architecture - Companion (ECSA-C 2021), Virtual online (originally: Växjö, Sweden), September, 13-17, 2021. Ed.: R. Heinrich*. 15th European Conference on Software Architecture. Vol. 2978. ISSN: 1613-0073. 2021, Paper–ID: 94.

[Gar+20] J. García-García, J. Enríquez, M. Ruiz, C. Arévalo, and A. Jiménez-Ramírez. **“Software Process Simulation Modeling: Systematic literature review”**. In: *Computer Standards & Interfaces* 70 (2020), p. 103425.

[Gen+18] A. Genevay, M. Cuturi, G. Peyré, and F. Bach. “Learning Generative Models with Sinkhorn Divergences”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 4604–4614.

[Gha14] K. Ghane. “A model and system for applying Lean Six sigma to agile software development using hybrid simulation”. In: *2014 IEEE International Technology Management Conference*. 2014, pp. 1–4.

[GIM13] J. Garcia, I. Ivkovic, and N. Medvidovic. "A comparative analysis of software architecture recovery techniques". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 486–496.

[Gol11] T. Goldschmidt. "View-based textual modelling". PhD thesis. 2011. 376 pp.

[Gra+22] M. Grambow, D. Kovalev, C. Laaber, P. Leitner, and D. Bermbach. **"Using Microbenchmark Suites to Detect Application Performance Changes"**. In: *IEEE Transactions on Cloud Computing* 11 (2022), pp. 2575–2590.

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

[Gro+21] J. Grohmann, S. Eismann, A. Bauer, S. Spinner, J. Blum, N. Herbst, and S. Kounev. **"SARDE"**. In: *ACM Transactions on Autonomous and Adaptive Systems* 15.2 (2021), pp. 1–31.

[Gui05] G. Guizzardi. "Ontological Foundations for Structural Conceptual Models". PhD thesis. 2005.

[Has18] W. Hasselbring. "Software Architecture: Past, Present, Future". In: *The Essence of Software Engineering*. 2018, pp. 169–184.

[Has21] W. Hasselbring. "Benchmarking as Empirical Standard in Software Engineering Research". In: *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*. 2021, pp. 365–372.

[HD18] C. Hu and Y. Deng. **"Fast resource scaling in elastic clusters with an agile method for demand estimation"**. In: *Sustain. Comput. Informatics Syst.* 19 (2018), pp. 165–173.

[Heg+17] C. Heger, A. van Hoorn, M. Mann, and D. Okanović. "Application Performance Management: State of the Art and Challenges for the Future". In: *Proceedings of the 8th ACM/SPEC on Intl. Conference on Performance Engineering*. 2017, pp. 429–432.

[Hei+10] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. **"Closing the Gap between Modelling and Java"**. In: *Software Language Engineering*. Vol. 5969. 2010, pp. 374–383.

[Hei+15] R. Heinrich, S. Gärtner, T. Hesse, T. Ruhroth, R. Reussner, K. Schneider, B. Paech, and J. Jürjens. **"The CoCoME Platform: A Research Note on Empirical Studies in Information System Evolution"**. In: *Int. Journal of Software Engineering and Knowledge Engineering* 25.09&10 (2015), pp. 1715–1720.

[Hei+17] R. Heinrich, P. Merkle, J. Henss, and B. Paech. **"Integrating business process simulation and information system simulation for performance prediction"**. In: *Software & Systems Modeling* 16.1 (2017), pp. 257–277.

[Hei+21] R. Heinrich, F. Durán, C. L. Talcott, and S. Z. (eds.) *Composing Model-Based Analysis Tools*. Ed. by R. Heinrich, F. Durán, C. L. Talcott, and S. Zschaler. in press, to appear. Springer, 2021.

[Hei14] Heinrich, Robert and Schmieders, Eric and Jung, Reiner and Rostami, Kiana and Metzger, Andreas and Hasselbring, Wilhelm and Reussner, Ralf and Pohl, Klaus. "Integrating run-time observations and design component models for cloud system analysis". In: *9th Int'l Workshop on Models@run.time* (2014), pp. 41–46.

[Hei16] R. Heinrich. **"Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications"**. In: *ACM SIGMETRICS Performance Evaluation Review* 43.4 (2016), pp. 13–22.

[Hei20] R. Heinrich. **"Architectural runtime models for integrating runtime observations and component-based models"**. In: *Journal of Systems and Software* 169 (2020).

[Her+08] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. "CoCoME - The Common Component Modeling Example". In: *The Common Component Modeling Example: Comparing Software Component Models*. 2008, pp. 16–53.

[Hew24] Hewlett Packard Enterprise Development LP. ***HPE Insight Control Performance Management User Guide***. Accessed on: DATE. 2024.

[HHF13] C. Heger, J. Happe, and R. Farahbod. **"Automated root cause isolation of performance regressions during software development"**. In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2013), pp. 27–38.

[HKP11] R. Heinrich, A. Kappe, and B. Paech. "Modeling Quality Information within Business Process Models". In: *Proceedings of the 4th SQMB Workshop, TUM-I1104*. 2011, pp. 4–13.

[HMY21] A. Harper, N. Mustafee, and M. Yearworth. **"Facets of trust in simulation studies"**. In: *European Journal of Operational Research* 289.1 (16, 2021), pp. 197–213.

[Hoo14a] A. van Hoorn. ***Model-Driven Online Capacity Management for Component-Based Software Systems***. 2014/6. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, Kiel University, 2014.

[Hoo14b] A. van Hoorn. "Model-Driven Online Capacity Management for Component-Based Software Systems". PhD thesis. Technische Fakultät, Christian-Albrechts-Universität zu Kiel, 2014.

[Hou+21] B. Hou, C. Hou, T. Zhou, Z. Cai, and F. Liu. **"Detection and Characterization of Network Anomalies in Large-Scale RTT Time Series"**. In: *IEEE Transactions on Network and Service Management* 18 (2021), pp. 793–806.

[HSB09] A. Hinze, K. Sachs, and A. Buchmann. "Event-based applications and enabling technologies". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. 2009, 1:1–1:15.

[Hu+21] C. Hu, Y. Deng, L. Yang, and Y. Zhao. **"Estimating the Resource Demand in Power-Aware Clusters by Regressing a Linearly Dependent Relation"**. In: *IEEE Transactions on Sustainable Computing* 6 (2021), pp. 385–397.

[Hub+17] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bähr. "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language". In: *IEEE Transactions on Software Engineering (TSE)* 43.5 (2017), pp. 432–452.

[Hub14] N. Huber. "Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures". PhD thesis. Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2014.

[Huy22] C. Huyen. ***DESIGNING MACHINE LEARNING SYSTEMS: An iterative process for production-ready applications***. First edition. O'REILLY MEDIA, INC, USA, 2022.

[HWH12a] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. 2012, pp. 247–248.

[HWH12b] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd ACM/SPEC Intl. Conference on Performance Engineering*. 2012, pp. 247–248.

[Ibr+23] K. Ibrahim, H. Hassan, K. T. Wassif, and S. Makady. **"Context-Aware Expert for Software Architecture Recovery (CAESAR): An automated approach for recovering software architectures"**. In: *Journal of King Saud University - Computer and Information Sciences* 35.8 (2023), p. 101706.

[ISO11] ISO/IEC/IEEE 42010:2011(E). ***Systems and software engineering – Architecture description***. International Organization for Standardization, Geneva, Switzerland, 2011, pp. 1–46.

[Jäg19] J.-P. Jägers. "Iterative Performance Model Parameter Estimation Considering Parametric Dependencies". Master Thesis. Karlsruhe Institute of technology, 13, 2019.

[Jam+13] G. James, D. Witten, T. Hastie, R. Tibshirani, et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[JH15] Z. Jiang and A. Hassan. **"A Survey on Load Testing of Large-Scale Software Systems"**. In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1091–1118.

[JHS13] R. Jung, R. Heinrich, and E. Schmieders. "Model-driven instrumentation with Kieker and Palladio to forecast dynamic applications". In: *Symposium on Software Performance*. Vol. 1083. 2013, pp. 99–108.

[Joy+00] B. Joy, G. Steele, J. Gosling, and G. Bracha. ***The Java Language Specification, Third Edition***. Addison-Wesley Reading, 2000.

[Kal+12] A. Kalbasi, D. Krishnamurthy, J. Rolia, and S. Dawson. **"DEC: Service demand estimation with confidence"**. In: *IEEE Transactions on Software Engineering* 38.3 (2012), pp. 561–578.

[Kap+21a] A. Kaplan, M. Rapp, S. Ananieva, H. Hajiabadi, R. Kühn, M. Mazkatli, L. Schmid, and S. Singh. "Poster: Towards Managing and Organizing Research Activities". In: *8th ACM Celebration of Women in Computing: womENcourage™ 2021, virtual (coordinated from Prague, Czech Republic), Sepetember 22-24, 2021*. 2021.

[Kas+21] R. Kasauli, E. Knauss, J. Horkoff, G. Liebel, and F. G. de Oliveira Neto. **"Requirements engineering challenges and practices in large-scale agile system development"**. In: *Journal of Systems and Software* 172 (2021), p. 110851.

[KC12] J. Kowall and W. Cappelli. “Magic quadrant for application performance monitoring”. In: *Gartner Research ID G* 232180 (2012).

[Kei+20] J. Keim, A. Kaplan, A. Koziolek, and M. Mirakhorli. ***Using BERT for the Detection of Architectural Tactics in Code***. Tech. rep. 2. Karlsruhe Institute of Technology (KIT), 2020.

[Kei+21] J. Keim, S. Schulz, D. Fuchss, C. Kocher, J. Speit, and A. Koziolek. “Tracelink Recovery for Software Architecture Documentation”. In: *Software Architecture*. 2021, pp. 101–116.

[KF05] E. Kıcıman and A. Fox. **“Detecting application-level failures in component-based Internet services”**. In: *IEEE Transactions on Neural Networks* 16 (2005), pp. 1027–1041.

[KH16] M. Konersmann and J. Holschbach. **“Automatic Synchronization of Allocation Models with Running Software”**. In: *Softwaretechnik-Trends* 36.4 (2016).

[Kir+24] Y. R. Kirschner, M. Gstür, T. Sağlam, S. Weber, and A. Koziolek. ***Retriever: A View-Based Approach to Reverse Engineering Software Architecture Models***. Tech. rep. 43.31.01; LK 01. Elsevier B.V., 2024. 36 pp.

[Kis+18] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev. “TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research”. In: *2018 IEEE 26th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2018, pp. 223–236.

[KK21] A. Kaplan and J. Keim. “Towards an Automated Classification Approach for Software Engineering Research”. In: *Evaluation and Assessment in Software Engineering*. 2021, pp. 347–352.

[KKR08] M. Kuperberg, M. Krogmann, and R. Reussner. “ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations”. In: *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Budapest, Hungary, 5th April 2008 (ETAPS 2008, 11th European Joint Conferences on Theory and Practice of Software)*. 2008.

[KKR10] K. Krogmann, M. Kuperberg, and R. Reussner. **“Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction”**. In: *IEEE Transactions on Software Engineering* 36.6 (2010), pp. 865–877.

[KL10] E. Kiciman and B. Livshits. **“AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications”**. In: *ACM Trans. Web* 4.4 (2010).

[Kla+21] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner. **"Enabling consistency in view-based system development – The Vitruvius approach"**. In: *Journal of Systems and Software* 171 (2021).

[Kla14] B. Klatt. "Consolidation of Customized Product Copies into Software Product Lines". PhD thesis. Karlsruhe Institute of Technology (KIT), 2014.

[Kla21] H. Klare. "Building Transformation Networks for Consistent Evolution of Interrelated Models". PhD thesis. Karlsruhe Institute of Technology (KIT), 2021. 428 pp.

[Kol+09] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. "Different models for model matching: An analysis of approaches to support model differencing". In: *2009 ICSE Workshop on Comparison and Versioning of Software Models*. 2009, pp. 1–6.

[Kon18] M. Konersmann. "Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code". en. PhD thesis. 2018.

[Koz16] H. Koziolek. **"Modeling Quality"**. In: *Modeling and simulating software architectures: the Palladio approach*. 2016.

[KP07] Knodel Jens and Popescu Daniel. "A Comparison of Static Architecture Compliance Checking Approaches". In: *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*. 2007, p. 12.

[Kra+09] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. "Estimating service resource consumption from response time measurements". In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. 20, 2009, pp. 1–10.

[Kra17] M. E. Kramer. "Specification Languages for Preserving Consistency between Models of Different Languages". PhD thesis. Karlsruhe Institute of Technology (KIT), 2017, pp. 11–19. 278 pp.

[Kro12] K. Krogmann. ***Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis***. Vol. 4. KIT Scientific Publishing, 2012.

[Kru95] P. Kruchten. **"The 4+1 View Model of architecture"**. In: *IEEE Software* 12.6 (1995), pp. 42–50.

[KS19] S. D. Krach and M. Scheerer. "SimuLizar NG: An extensible event-oriented simulation engine for self-adaptive software architectures". In: *Softwaretechnik-Trends Band 39, Heft 3*. 2019, pp. 43–45.

[KTZ09] D. Kumar, A. Tantawi, and L. Zhang. "Real-time performance modeling for adaptive software systems". In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. 2009, pp. 1–10.

[Küh+23] T. Kühn, D. Fuchß, S. Corallo, L. König, E. Burger, J. Keim, M. Mazkatli, T. Sağlam, F. Reiche, A. Koziolek, and R. Reussner. ***A Formalized Classification Schema for Model Consistency***. Tech. rep. 1. Karlsruher Institut für Technologie (KIT), 2023. 37 pp.

[KWH21] A. Kaplan, M. Walter, and R. Heinrich. "A Classification for Managing Software Engineering Knowledge". In: *Evaluation and Assessment in Software Engineering*. 2021, pp. 340–346.

[KZT10] D. Kumar, L. Zhang, and A. Tantawi. "Enhanced Inferencing: Estimation of a Workload Dependent Performance Model". In: 2010.

[Lan+16] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. "Automated extraction of rich software models from limited system information". In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE. 2016.

[Lan17] M. Langhammer. "Automated Coevolution of Source Code and Software Architecture Models". PhD thesis. Karlsruhe Institute of Technology (KIT), 2017. 259 pp.

[Leo+15] S. Leonhardt, B. Hettwer, J. Hoor, and M. Langhammer. "Integration of Existing Software Artifacts into a View- and Change-Driven Development Approach". In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. 2015, pp. 17–24.

[Lin+15] J. Linåker, S. M. Sulaman, R. Maiani de Mello, and M. Höst. **"Guidelines for Conducting Surveys in Software Engineering."** In: *ELLIIT: the Linköping-Lund initiative on IT and mobile communication* (2015).

[LK15] M. Langhammer and K. Krogmann. "A Co-evolution Approach for Source Code and Component-based Architecture Models". In: *17. Workshop Software-Reengineering und-Evolution*. Vol. 4. 2015.

[LMÁ09] F. J. Lucas, F. Molina, and J. Álvarez. **"A systematic review of UML model consistency management"**. In: *Inf. Softw. Technol.* 51 (2009), pp. 1631–1645.

[Lun+21] M. I. Lunesu, R. Tonelli, L. Marchesi, and M. Marchesi. **"Assessing the Risk of Software Development in Agile Methodologies Using Simulation"**. In: *IEEE Access* 9 (2021), pp. 134240–134258.

[Luo+22] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C. Xu. **"An In-Depth Study of Microservice Call Graph and Runtime Performance"**. In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 3901–3914.

[MAK23] M. Mazkatli, M. Armbruster, and A. Koziolek. "Towards Continuous Integration of Performance Models for Lua-Based Sensor Applications". In: *Softwaretechnik-Trends Band 43, Heft 4*. 2023, pp. 41–43.

[MAM24] M. Mazkatli, M. Armbruster, and D. Monschein. ***Continuous Integration of Architectural Performance Models***. Version 0.1.0. 2024.

[Maz+17] M. Mazkatli, E. Burger, A. Koziolek, and R. H. Reussner. "Automotive Systems Modelling with Vitruvius". In: *INFORMATIK 2017* (Chemnitz). Vol. P-275. 2017, pp. 1487–1498.

[Maz+18] M. Mazkatli, E. Burger, J. Quante, and A. Koziolek. "Integrating semantically-related Legacy Models in Vitruvius". In: *Proceedings of Modelling in Software Engineering co-located with the 40th International Conference on Software Engineering* (Gothernburg, Sweden). 2018, pp. 41–48.

[Maz+20] M. Mazkatli, D. Monschein, J. Grohmann, and A. Koziolek. "Incremental Calibration of Architectural Performance Models with Parametric Dependencies". In: *IEEE International Conference on Software Architecture (ICSA 2020)*. Acceptance rate (Full Paper): 18%, **Best-Paper-Award Nomination**. 2020, pp. 23–34.

[Maz+24] M. Mazkatli, D. Monschein, M. Armbruster, R. Heinrich, and A. Koziolek. ***Replication Package for "Continuous Integration of Architectural Performance Models with Parametric Dependencies – The CIPM Approach"***. Version 1.0.0. 2024.

[Maz+25] M. Mazkatli, D. Monschein, M. Armbruster, R. Heinrich, and A. Koziolek. **"Continuous Integration of Architectural Performance Models with Parametric Dependencies – The CIPM Approach"**. In: *Automated Software Engineering Journal* (2025). A preprint is also available in KITopen: `https://doi.org/10.5445/IR/1000151086`.

[Maz16] M. Mazkatli. "Consistency Preservation in the Development Process of Automotive Software". MA thesis. Karlsruhe Institute of Technology (KIT), 2016.

[Mém11] F. Mémoli. **"Gromov-Wasserstein Distances and the Metric Approach to Object Matching."** In: *Foundations of Computational Mathematics* 11.4 (2011), pp. 417–487.

[Men+04] D. A. Menasce, V. A. Almeida, L. W. Dowdy, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[Men08] D. A. Menasce. "Computing Missing Service Demand Parameters for Performance Models." In: *Int. CMG Conference*. 2008, pp. 241–248.

[MHH11] R. von Massow, A. van Hoorn, and W. Hasselbring. "Performance Simulation of Runtime Reconfigurable Component-Based Software Architectures". In: *Software Architecture*. 2011, pp. 43–58.

[MK18] M. Mazkatli and A. Koziolek. "Continuous Integration of Performance Model". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 153–158.

[ML03] M. Marshak and H. Levy. **"Evaluating web user perceived latency using server side measurements"**. In: *Comput. Commun.* 26 (2003), pp. 872–887.

[Mon+21] D. Monschein, M. Mazkatli, R. Heinrich, and A. Koziolek. "Enabling Consistency between Software Artefacts for Software Adaption and Evolution". In: *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. 2021, pp. 1–12.

[Mon20] D. Monschein. "Enabling Consistency between Software Artefacts for Software Adaption and Evolution". Supervised by Manar Mazkatli and Robert Heinrich. Master Thesis. Karlsruher Institut für Technologie, 4, 2020.

[MP10] A. Mellit and A. M. Pavan. **"A 24-h forecast of solar irradiance using artificial neural network: Application for performance prediction of a grid-connected PV plant at Trieste, Italy"**. In: *Solar Energy* 84.5 (2010), pp. 807–821.

[MS16] F. Mognon and P. C. Stadzisz. **"Modeling in Agile Software Development: A Systematic Literature Review"**. In: (2016), pp. 50–59.

[NF11] J. C. Navón and F. Fernandez. **"The Essence of REST Architectural Style"**. In: (2011), pp. 21–33.

[OEW17] T. Olsson, M. Ericsson, and A. Wingkvist. "Motivation and Impact of Modeling Erosion Using Static Architecture Conformance Checking". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 204–209.

[OMG15] OMG. ***MOF 2.5 Core Specification (formal/2015-06-05).*** Object Management Group (OMG), 2015.

[Our+18] R. Ouriques, K. Wnuk, T. Gorschek, and R. Berntsson-Svensson. **"Knowledge Management Strategies and Processes in Agile Software Development: A Systematic Literature Review"**. In: *ArXiv* abs/1807.04962 (2018).

[Pac+08] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi. **"CPU demand for web serving: Measurement analysis and dynamic estimation"**. In: *Performance Evaluation* 65.6 (1, 2008), pp. 531–553.

[Per+21] J. A. Pereira, M. Acher, H. Martin, J.-M. Jézéquel, G. Botterweck, and A. Ventresque. **"Learning software configuration spaces: A systematic literature review"**. In: *Journal of Systems and Software* 182 (2021), p. 111044.

[Pér12] R. Pérez-Castillo. "MARBLE: Modernization approach for recovering business processes from legacy information systems". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 671–676.

[PPC13] J. F. Pérez, S. Pacheco-Sanchez, and G. Casale. "An Offline Demand Estimation Method for Multi-threaded Applications". In: *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. 14, 2013, pp. 21–30.

[PW10] O. Pele and M. Werman. "A fast algorithm for the exact solution of the Wasserstein distance". In: *International Journal of Computer Vision* 86.3 (2010), pp. 205–221.

[Qui93] J. R. Quinlan. ***C4.5: Programs for Machine Learning***. Morgan Kaufmann Publishers Inc., 1993.

[Rat13] C. Rathfelder. "Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation". PhD thesis. 2013. 358 pp.

[Reu+16] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. ***Modeling and Simulating Software Architectures – The Palladio Approach***. MIT Press, 2016. 408 pp.

[Reu+19] R. Reussner, M. Goedicke, W. Hasselbring, B. Vogel-Heuser, J. Keim, and L. Märtin. ***Managed Software Evolution***. Springer, Cham, 2019.

[RFZ17] C. Raibulet, F. A. Fontana, and M. Zanoni. **"Model-Driven Reverse Engineering Approaches: A Systematic Literature Review"**. In: *IEEE Access* 5 (2017), pp. 14516–14542.

[RKH21] D. G. Reichelt, S. Kühne, and W. Hasselbring. “Overhead Comparison of OpenTelemetry, inspectIT and Kieker”. In: *Symposium on Software Performance 2021*. 2021.

[RKH23] D. Reichelt, S. Kühne, and W. Hasselbring. **“Towards Solving the Challenge of Minimal Overhead Monitoring”**. In: *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering* (2023).

[RRM13] F. J. B. Ruiz, Ó. S. Ramón, and J. G. Molina. “Definition of processes for MDE-based migrations”. In: *Proceedings of the Third Workshop on Process-Based Approaches for Model-Driven Engineering*. 2013.

[RTG00] Y. Rubner, C. Tomasi, and L. J. Guibas. “The earth mover’s distance as a metric for image retrieval”. In: *International journal of computer vision* 40.2 (2000), pp. 99–121.

[Run+12] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[RV95] J. Rolia and V. Vetland. “Parameter estimation for performance models of distributed application systems”. In: *Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*. 1995, p. 54.

[RV98] J. Rolia and V. Vetland. “Correlating resource demand information with ARM data for application services”. In: Cited by: 46; All Open Access, Bronze Open Access. 1998, pp. 219–230.

[Sac10] K. Sachs. “Performance Modeling and Benchmarking of Event-Based Systems”. PhD thesis. University of Darmstadt, 2010.

[Sae25] J. Saenz. “Continuous Integration of Architectural Performance Models for Lua Applications”. Supervised by Martin Armbruster and Manar Mazkatli. MA thesis. Karlsruhe Institute of Technology (KIT), 2025.

[San15] F. Santambrogio. ***Optimal Transport for Applied Mathematicians: Calculus of Variations, PDEs, and Modeling***. Springer International Publishing, 2015.

[Sas+07] Y. Sasaki et al. “The truth of the F-measure”. In: *Teach tutor mater* 1.5 (2007), pp. 1–5.

[SB12] L. de Silva and D. Balasubramaniam. **“Controlling software architecture erosion: A survey”**. In: *Journal of Systems and Software* 85.1 (2012), pp. 132–151.

[Sch18] T. Schlossnagle. **“Monitoring in a DevOps World”**. In: *Commun. ACM* 61.3 (2018), pp. 58–61.

[Sel03] B. Selić. **“The pragmatics of model-driven development”**. In: *IEEE Computer Society* 20.5 (2003), pp. 19–25.

[SG22] M. Segovia and J. García. **"Design, Modeling and Implementation of Digital Twins"**. In: *Sensors (Basel, Switzerland)* 22 (2022).

[Sha+08] A. B. Sharma, R. Bhagwan, M. Choudhury, L. Golubchik, R. Govindan, and G. M. Voelker. **"Automatic request categorization in internet services"**. In: *SIGMETRICS Perform. Eval. Rev.* 36.2 (2008), pp. 16–25.

[She07] D. J. Sheskin. ***Handbook of Parametric and Nonparametric Statistical Procedures***. 4th ed. Chapman and Hall/CRC, 2007.

[SHW19] H. Schulz, A. van Hoorn, and A. Wert. **"Reducing the maintenance effort for parameterization of representative load tests using annotations"**. In: *Software Testing, Verification and Reliability* 30 (16, 2019).

[SJ11] C. Sutton and M. Jordan. **"Bayesian inference for queueing networks and modeling of internet services"**. In: *Annals of Applied Statistics* 5.1 (2011), pp. 254–282.

[SK24] S. Singh and A. Koziolek. "Automated Reverse Engineering for MoM-Based Microservices (ARE4MOM) Using Static Analysis". In: *2024 IEEE 21st International Conference on Software Architecture (ICSA), Hyderabad, 4th-8th June 2024*. IEEE International Conference on Software Architecture. ICSA 2024 (Hyderabad, Indien, June 4–8, 2024). 46.23.01; LK 01. 2024, pp. 12–22.

[Soc21] I. C. Society. ***IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment***. 2021.

[Sol+02] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach. **"Goal Question Metric (GQM) Approach"**. In: *Encyclopedia of Software Engineering*. 2002.

[Spi+14] S. Spinner, G. Casale, X. Zhu, and S. Kounev. "LibReDE: A Library for Resource Demand Estimation". In: *Proceedings of the 5th ACM/SPEC Int. Conference on Performance Engineering*. 2014.

[Spi+15] S. Spinner, G. Casale, F. Brosig, and S. Kounev. "Evaluating approaches to resource demand estimation". In: *Performance Evaluation* 92 (2015).

[Spi+19] S. Spinner, J. Grohmann, S. Eismann, and S. Kounev. "Online model learning for self-aware computing infrastructures". In: *Journal of Systems and Software* 147 (2019), pp. 1–16.

[Sta73] H. Stachowiak. ***Allgemeine Modelltheorie***. Springer-Verlag, 1973.

[Ste20] P. Stevens. **"Maintaining consistency in networks of models: bidirectional transformations in the large"**. In: *Software and Systems Modeling* 19.1 (1, 2020), pp. 39–65.

[Stü+24] J. Stümpfle, L. Ludmann, D. Fuchß, H. Guissouma, L. Seidel, A. Weigl, R. Fehrenbacher, M. Kodetzki, T. Pett, M. Mazkatli, J. Bachmeier, S. Weber, J. Henle, A. Schmid, and A. Salah. *VESKO: Variant-Aware Development of Safety-Critical Over-the-Air Updates Demonstrator Platform.* Tech. rep. A contribution to the Software-Defined Car funding project; To be submitted to the Stuttgarter Symposium on 13.12.24. University of Stuttgart, Karlsruhe Institute of Technology (KIT), FZI Research Center for Information Technology, Vector Informatik GmbH, Research Institute for Automotive Engineering and Powertrain Systems Stuttgart (FKFS), 2024.

[SV06] T. Stahl and M. Völter. ***Model-driven software development : technology, engineering, management***. Ed. by J. Bettin, K. Czarnecki, and B. von Stockfleth. John Wiley & Sons, 2006.

[SW03] C. U. Smith and L. G. Williams. ***Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software***. Addison Wesley Longman Publishing Co., Inc., 2003.

[SZ16] M. Szvetits and U. Zdun. **"Systematic Literature Review of the Objectives, Techniques, Kinds, and Architectures of Models at Runtime"**. In: *Softw. Syst. Model.* 15.1 (2016), pp. 31–69.

[Tai+23] D. Taibi, Y. Cai, I. Weber, M. Mirakhorli, M. W. Godfrey, J. T. Stough, and P. Pelliccione. "Continuous Alignment Between Software Architecture Design and Development in CI/CD Pipelines". In: *Software Architecture: Research Roadmaps from the Community*. 2023, pp. 69–86.

[TBS20] W. Torres, M. Brand, and A. Serebrenik. **"A systematic literature review of cross-domain model consistency checking by model management tools"**. In: *Software and Systems Modeling* 20 (2020), pp. 897–916.

[TMD09] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. ***Software Architecture: Foundations, Theory, and Practice***. Wiley, 2009.

[Tru+18] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche. **"Exploiting load testing and profiling for Performance Antipattern Detection"**. In: *Information and Software Technology* 95 (1, 2018), pp. 329–345.

[Tru+19] C. Trubiani, P. Jamshidi, J. Cito, W. Shang, Z. M. Jiang, and M. Borg. **"Performance Issues? Hey DevOps, Mind the Uncertainty"**. In: *IEEE Software* 36.2 (2019), pp. 110–117.

[UC08] G. Upton and I. Cook. ***A Dictionary of Statistics***. Oxford University Press, 2008.

[Ull+22] I. Ullah, J. Kanwal, F. Gillani, and I. Shahzad. **"Role of Agile Methodologies for Ensuring Quality in Complex Systems: A Systematic Literature Review"**. In: *Vol 4 Issue 4* (2022).

[Urg+07] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. **"Analytic modeling of multitier Internet applications"**. In: *ACM Trans. Web* 1.1 (2007), 2–es.

[Val+10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. "Soot: A Java Bytecode Optimization Framework". In: *CASCON First Decade High Impact Papers*. 2010, pp. 214–224.

[van+09] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. "An adaptation framework enabling resource-efficient operation of software systems". In: *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010 on - WUP '09*. 2009, p. 41.

[Vie+12] M. Vieira, H. Madeira, K. Sachs, and S. Kounev. **"Resilience Benchmarking"**. In: *Resilience Assessment and Evaluation of Computing Systems*. 2012, pp. 283–301.

[Vil+09] C. Villani et al. *Optimal transport: old and new*. Vol. 338. Springer, 2009.

[Voe+12] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. "Mbeddr: An Extensible C-Based Programming Language and IDE for Embedded Systems". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. 2012, pp. 121–140.

[Vög+18] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar. **"WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction - a model-driven approach for session-based application systems"**. In: *Softw. Syst. Model.* 17.2 (2018), pp. 443–477.

[Von+20] S. Voneva, M. Mazkatli, J. Grohmann, and A. Koziolek. "Optimizing Parametric Dependencies for Incremental Performance Model Extraction". In: *Software Architecture - 14th European Conference, ECSA 2020 Tracks and Workshops- The 6th International Workshop on Quality-Aware DevOps*. 2020, pp. 228–240.

[VR14] S. Visalakshi and V. Radha. "A literature review of feature selection techniques and applications: Review of feature selection in data mining". In: *2014 IEEE international conference on computational intelligence and computing research*. IEEE. 2014, pp. 1–6.

[VSP20] L. Vogel, S. Scholz, and F. Pfaff. "Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model". In: *vogella GmbH* (2020).

[WA20] E. Whiting and S. Andrews. “Drift and Erosion in Software Architecture”. In: *Proceedings of the 2020 the 4th International Conference on Information System and Data Mining*. 2020, pp. 132–138.

[Wal+16] J. Walter, A. Hoorn, H. Koziolek, D. Okanovic, and S. Kounev. **“Asking "What"?, Automating the "How"?: The Vision of Declarative Performance Engineering”**. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (2016).

[Wal+17] J. Walter, C. Stier, H. Koziolek, and S. Kounev. “An Expandable Extraction Framework for Architectural Performance Models”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 2017, pp. 165–170.

[Wan+18] T. Wang, J. Xu, W.-b. Zhang, Z. Gu, and H. Zhong. **“Self-adaptive cloud monitoring with online anomaly detection”**. In: *Future Gener. Comput. Syst.* 80 (2018), pp. 89–101.

[WC13] W. Wang and G. Casale. “Bayesian service demand estimation using gibbs sampling”. In: *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE. 2013, pp. 567–576.

[WEH15] J. Waller, N. C. Ehmke, and W. Hasselbring. **“Including Performance Benchmarks into Continuous Integration to Enable DevOps”**. In: *SIGSOFT Softw. Eng. Notes* 40.2 (2015), pp. 1–4.

[Wei09] M. Weiss. “XML Metadata Interchange”. In: *Encyclopedia of Database Systems*. 2009, pp. 3597–3597.

[Wen24] F. Weng. “Generalisation of Consistency Rules between Architectural and Code Models”. Supervised by Martin Armbruster and Manar Mazkatli. MA thesis. Karlsruhe Institute of Technology (KIT), Department of Informatics, 2024.

[Wes+13] D. Westermann, J. Happe, P. Zdrahal, M. Moser, and R. Reussner. “Performance-Aware Design of Web Application Front-Ends”. In: *Web Engineering*. 2013, pp. 132–139.

[WFP07a] M. Woodside, G. Franks, and D. C. Petriu. “The Future of Software Performance Engineering”. In: *2007 Future of Software Engineering*. 2007, pp. 171–187.

[WFP07b] M. Woodside, G. Franks, and D. C. Petriu. “The Future of Software Performance Engineering”. In: *Proceedings of ICSE 2007, Future of SE*. 2007, pp. 171–187.

[Wil+15] F. Willnecker, M. Dlugi, A. Brunnert, S. Spinner, S. Kounev, W. Gottesheim, and H. Krcmar. "Comparing the accuracy of resource demand measurement and estimation techniques". In: *European Workshop on Performance Engineering*. Springer. 2015.

[Woh+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[Woh+19] R. Wohlrab, P. Pelliccione, E. Knauss, and M. Larsson. **"Boundary objects and their use in agile systems engineering"**. In: *Journal of Software: Evolution and Process* 31.5 (2019), e2166.

[Woh21] C. Wohlin. **"Case Study Research in Software Engineering—It is a Case, and it is a Study, but is it a Case Study?"** In: *Information and Software Technology* 133 (2021), p. 106514.

[WSH15] A. Wert, H. Schulz, and C. Heger. "AIM: Adaptable Instrumentation and Monitoring for Automated Software Performance Analysis". In: *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*. 2015, pp. 38–42.

[WSK23] J. W. Wittler, T. Sağlam, and T. Kuhn. "Evaluating model differencing for the consistency preservation of state-based views". In: *Journal of Object Technology* 22 (2023), pp. 1–14.

[WWH] S. Weber, T. Weber, and J. Henss. "Integration of Performability-Model Extraction and Performability Prediction in Continuous Integration / Continuous Delivery". to be published in Softwaretechnik-Trends.

[XG18] Y. Xu and R. Goodacre. **"On Splitting Training and Validation Set: A Comparative Study of Cross-Validation, Bootstrap and Systematic Sampling for Estimating the Generalization Performance of Supervised Learning"**. In: *Journal of analysis and testing* 2.3 (2018), pp. 249–262.

[Yin19] X. Ying. **"An Overview of Overfitting and its Solutions"**. In: *Journal of Physics: Conference Series* 1168.2 (2019), p. 022022.

[ZCS07] Q. Zhang, L. Cherkasova, and E. Smirni. "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications". In: *Fourth International Conference on Autonomic Computing (ICAC'07)*. Fourth International Conference on Autonomic Computing (ICAC'07). 2007, pp. 27–27.

[Zha+02] L. Zhang, C. Xia, M. Squillante, and W. Mills. “Workload service requirements analysis: a queueing network optimization approach”. In: *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems.* 2002, pp. 23–32.

[Zha+21] G. Zhao, S. Hassan, Y. Zou, D. Truong, and T. Corbin. **“Predicting Performance Anomalies in Software Systems at Run-time”**. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30 (2021), pp. 1–33.

[Zhe+05] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai. “Tracking time-varying parameters in software systems with extended kalman filters”. In: *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research* (2005), pp. 334–345.

Miscellaneous

The titles in this bibliography are hyperlinks pointing to the online sources.

[AG24a] S. AG. ***SICK AG***. Accessed on September 20, 2024. 2024.

[AG24b] S. AG. *SICK AppEngine*. `https://www.sick.com/de/en/sick-appspace/sick-appspace-software-tools/sick-appengine/c/g547567`. last retrieved 2024-07-31. 2024.

[AG24c] S. AG. *SICK AppSpace*. `https://www.sick.com/de/de/sick-appspace/c/g555725?q=:Def_Type:ProductFamily`. last retrieved 2024-07-31. 2024.

[AG24d] S. AG. *SICK Sensor Integration Machine*. `https://www.sick.com/de/de/integrationsprodukte/sensor-integration-machine/c/g386451`. last retrieved 2023-04-14. 2024.

[Amb22] S. Ambler. *Agile Architecture: Strategies for Scaling Agile Development*. `http://www.agilemodeling.com/essays/agileArchitecture.htm`. last retrieved 2024-06-27. 2022.

[Ana22] S. Ananieva. ***Consistent View-Based Management of Variability in Space and Time***. 2022.

[App24] C. AppDynamics. ***Observe what matters with AppDynamics Cloud***. 2024.

[Bec+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. *Manifesto for Agile Software Development*. `https://agilemanifesto.org/`. 2001.

[BMC24] BMC. ***BMC Software, Inc.*** 2024.

[CA24] CA. ***CA Technologies Infrastructure Software***. 2024.

[Cha06] M. K. Chandy. *Event-Driven Applications: Costs, Benefits and Design Approaches*. 2006.

[Com24] Compuware. *Compuware*. `https://www.bmc.com/it-solutions/brands/compuware.html?301=compuware-com`. 2024.

[Dev24] T. Developers. ***TeaStore Git***. last retrieved 31.07.2024. 31, 2024.

[Dyn24] Dynatrace. ***Dynatrace LLC.*** 2024.

[Ecl24a] Eclipse Foundation. ***Eclipse Modeling Framework Homepage***. http://www.eclipse.org/modeling/emf/. last retrieved 2024-07-31. 2024.

[Ecl24b] Eclipse Foundation. *Xtext - Language Engineering for Everyone!* https://www.eclipse.org/Xtext/index.html. last retrieved 2024-07-31. 2024.

[GmB24] E. GmBH. *ASCET Developer Tool*. https://www.etas.com/en/products/ascet-developer.php. 2024.

[Has] T. Hasan. *The Melange Language Workbench*. http://melange.inria.fr/.

[IBM24] IBM. *What is Application Performance Management (APM)?* https://www.ibm.com/de-de/topics/application-performance-management. Accessed: 2024-11-19. 2024.

[Kap+21b] A. Kaplan, M. Rapp, S. Ananieva, H. Hajiabadi, R. Kühn, M. Mazkatli, L. Schmid, and S. Singh. *Towards Managing and Organizing Research Activities*. Poster präsentiert auf 8th ACM Celebration of Women in Computing: womENcourage (2021), Online, 22.–24. September 2021. 2021.

[KBH14] S. Kounev, F. Brosig, and N. Huber. *Descartes Modeling Language (DML)*. https://se.informatik.uni-wuerzburg.de/software-engineering-group/tools/dml/descartes-modeling-language/. 2014.

[Kur24] W. Kurniawan. *Teammmates Design Architecture*. https://teammates.github.io/teammates/design.html. 2024.

[Lan19] P. Langer. ***EMF Compare***. 2019.

[LF14] Lewis and M. Fowler. *Microservices, a definition of this new architectural term*. https://martinfowler.com/articles/microservices.html#footnote-etymology. 2014.

[New24] I. NewRelic. ***New Relic: Monitor, debug, and improve your entire stack.*** 2024.

[OMG06] OMG, Object Management Group. ***UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP (realtime/05-02-06)***. 2006.

[Pin24] Pinpoint. ***Pinpoint***. 2024.

[Ral+21] P. Ralph, N. bin Ali, S. Baltes, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri, B. B. N. de França, C. A. Furia, G. Gay, N. Gold, D. Graziotin, P. He, R. Hoda, N. Juristo, B. Kitchenham, V. Lenarduzzi, J. Martínez, J. Melegati, D. Mendez, T. Menzies, J. Molleri, D. Pfahl, R. Robbes, D. Russo, N. Saarimäki, F. Sarro, D. Taibi, J. Siegmund, D. Spinellis, M. Staron, K. Stol, M.-A. Storey, D. Taibi, D. Tamburri, M. Torchiano, C. Treude, B. Turhan, X. Wang, and S. Vegas. ***Empirical Standards for Software Engineering Research***. 2021.

Appendix

A. Appendix

The appendix contains supplementary material supporting the main content of this thesis, including detailed data on the metamodels, case studies, and evaluation details, which are crucial for understanding the work and replicating the package.

The appendix contains detailed supplementary material as follows: In Section A.1, more details are provided on the metamodels of the Palladio Component Model (PCM) and related diagrams. This is followed by details on the case studies, primarily focusing on historical information regarding the Git history period considered in our evaluation. The TeaStore case study is presented in Section A.2, while the Teammates case study can be found in Section A.4. Finally, the Lua applications are described in Section A.5.

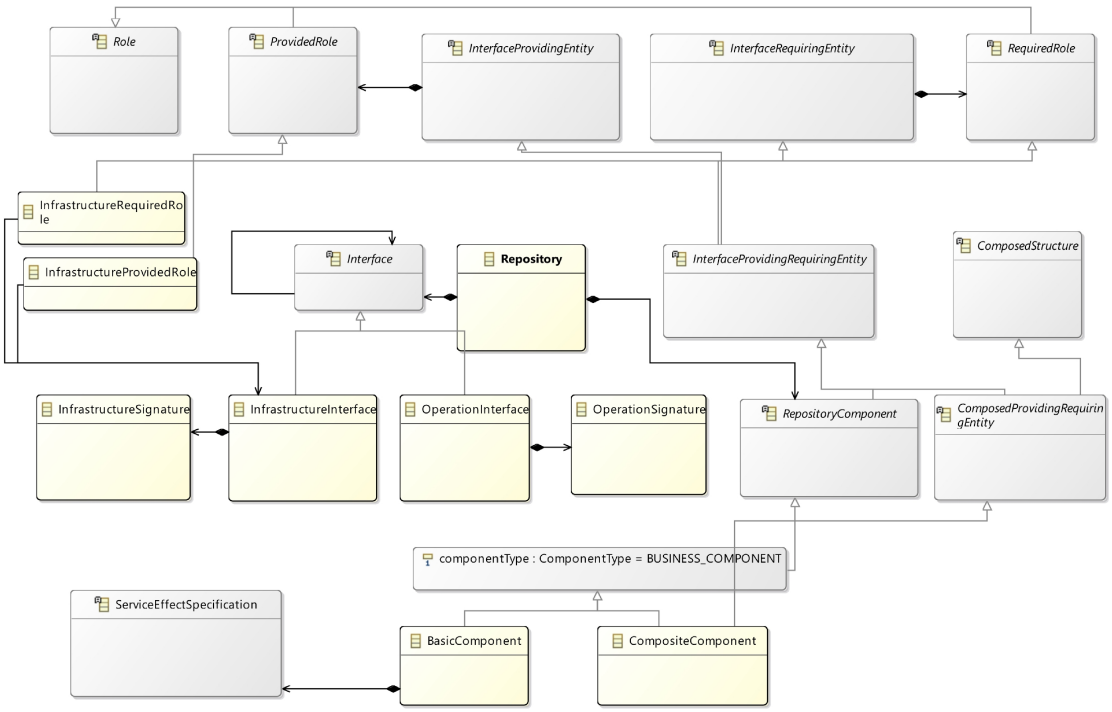


Figure A.1: Excerpt of Palladio repository metamodel.

A.1. Palladio Metamodel

In this section, we visualize some parts of `Palladio` metamodel to illustrate our approach in more detail.

A.1.1. Repository Model

The metamodel of `Repository Model` is in Figure A.2.

A.1.2. System Model

The metamodel of `System Model` is in Figure A.2.

A.1.3. Allocation Model

The metamodel of `Allocation Model` is in Figure A.3. The relationship between `Allocation Model` and `Repository Model` is clarified in Figure A.4.

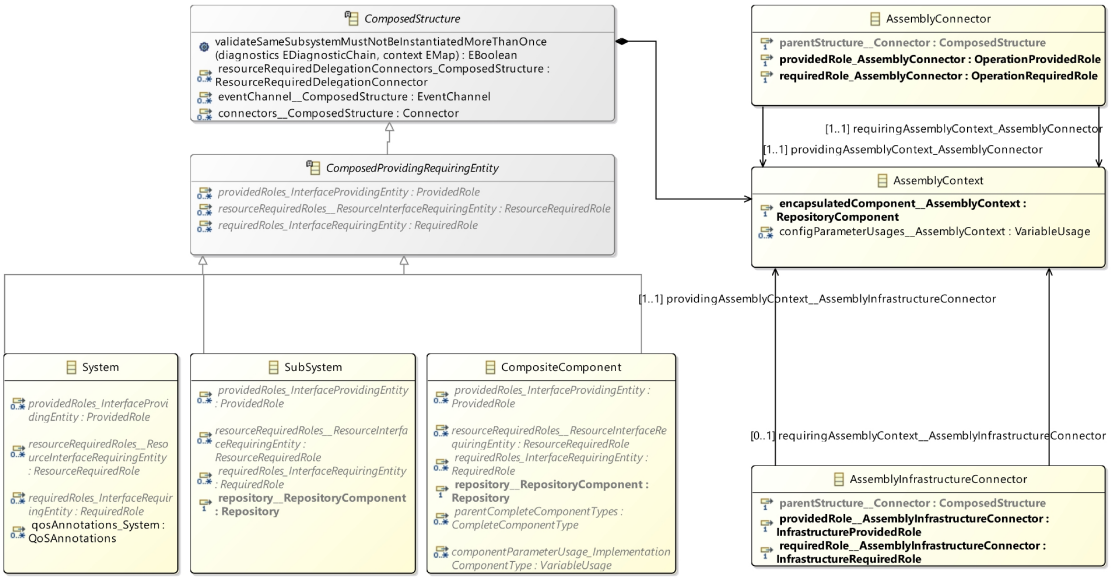


Figure A.2: Excerpt of Palladio system metamodel.

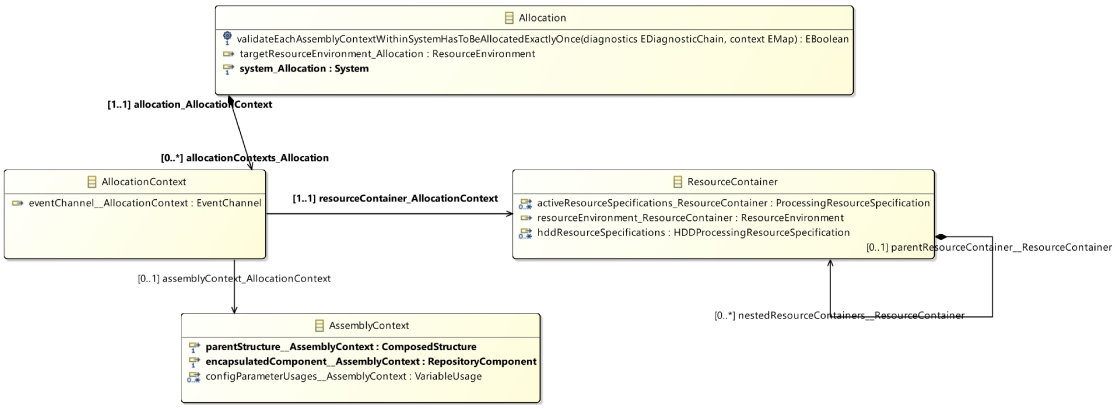


Figure A.3: Excerpt of Palladio allocation metamodel.

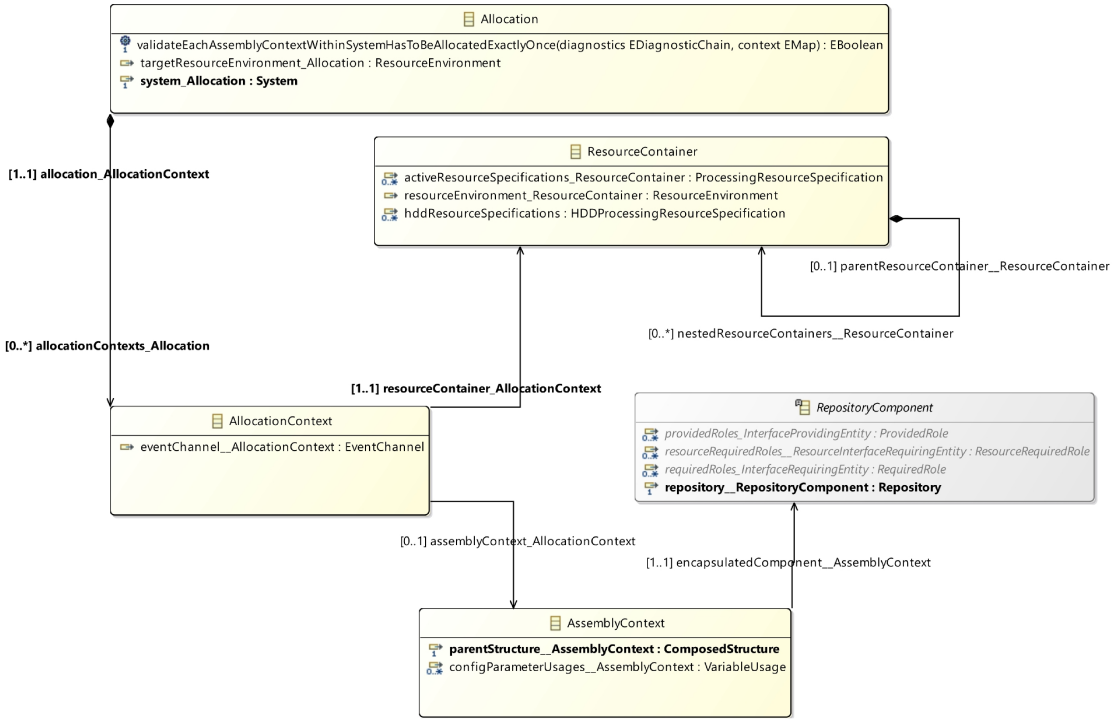


Figure A.4: Allocation class diagram with a reference to the repository model

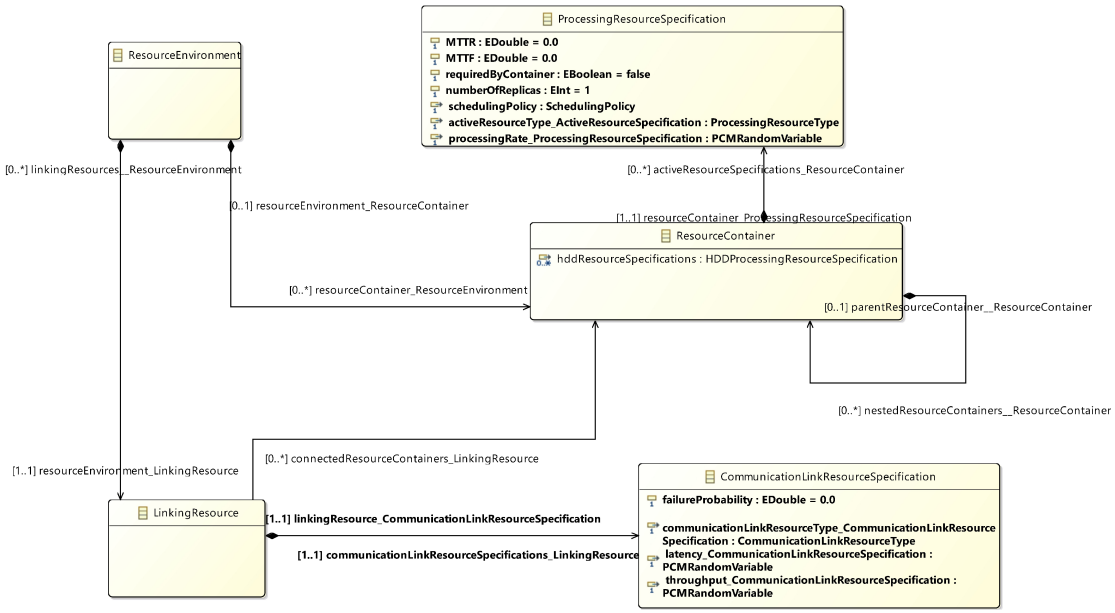


Figure A.5: Excerpt of Palladio resource environment metamodel.

A.1.4. Resource Environment Model

The metamodel of `Resource Environment Model` is in Figure A.5.

A.2. TeaStore case study

This section details the Git history used in our evaluation (see Section A.2.1) and provides an example of adaptive instrumentation in Section A.3.

A.2.1. Git History

In this subsection, we provide historical Information on the Git history period that we considered in our Evaluation.

Table A.1: The four intervals and information on their Commits. The commits are numbered continuously to ease readability. The information in this table is extracted from [Arm21].

Commit	Hash	Impact on PCM	Changed Java files	Added lines	Removed lines
Interval I					
0	77733d9c6ab6680c6cc460c631cd408a588a595c	IRE	236	26636	0
1	9c5e7aa3bfdec546740f0dc0d1a3befc36195e8e	-	0	0	0
2	585e7e9c501506580cfda0c97e5f645ac849b46e	-	0	0	0
3	38c81c974a06a2781b19395c02905ba5f604b6c0	I.A, I.B	23	1022	1022
4	115b89c518d1f281c3132ffc8d6c3e1f6293a3f5	-	1	2	1
5	362ca4028d7407156d04e8f347e90ebd1fe26004	-	0	0	0
6	439caf08f087a2093324f7f03fa8eda3eda4e25a	-	7	969	755
7	3aaa59b5b8449b214884eb24efb7e54276563c76	-	8	839	716
8	6ecfd88ecd86f4634bc95541a9fa5e5cdaf742a4	-	62	4385	4138
9	3291291cc07ca420b4c6792b1d134f08b3b0c947	-	4	20	10
10	18ef9411e4d54db780b3d47a5fee306fb4ccd8a6	I.C	3	58	53

11	bd9062b294545290c265acc81ba7e6f329a6a5b1	I.D	2	21	6
12	907cff311827f51035788544ec6bbb27abe3b9a0	-	17	496	429
13	f8752ca8b2b3c4ebc94c7e7cb4d6cfcd34d395a3	-	21	568	154
14	c834417fbd78230dc0354c22a696bbfcd3423a7b	-	10	335	26
15	492100017f535341ed047cc8e2306654ab2becc2	-	4	75	58
16	ec94531e55463c41b6a069b24205f22c42948261	-	4	58	75
17	2fe02ea07a6f702946ca66f023d35ab6fafba91a	-	4	75	58
18	baf77779976d6265745894c31a303ace226694f4	-	1	1	1
19	1cd2c8c4bac100519b5cabafc1c83a7cd7737f49	-	1	2	2
20	79e2e244ddd014fbb9df86310b7e1e97427da196	-	2	196	145
21	ec88ca71c11bcd3699ff2e965af6cea75d0ef735	-	3	432	389
22	da98b2d0056e782448f2e0022ceb0ce54228fd67	-	1	49	42
23	554ae7f76cadf85b6dd743e97d706987226fc4f0	-	1	3	0
24	447db7f7708721fcbe8b15d93a4b166d717861d6	-	0	0	0
25	ae97a5e149b7dd4bd56bbc4e0eb71ec12ac655d3	-	2	142	115
26	c7304932eef264eadbc6ef64cf7a5b46c1f8bf7a	-	0	0	0
27	9b0dbc68e0a20d4264f0d3d7b7dccc3c17d7d76c	-	1	1	1
28	16a26135a9a756ccb066e6b963808dbba1d898e9	I.E	7	174	96
29	2f97af33dea1e223e346c0e6995a71cd0cdb6d33	-	24	124	110

30	26c5a920ba08f980eb9f85b156a1accacb8a4fd8	-	1	1	1
31	22c3cc238c2ce06967dce9b83fa6e2dc7a72f3be	-	0	0	0
32	413db162b63d0d55f213439ef33553c935784726	-	0	0	0
33	039df89be3fdd63ba6557835561610120303 5fee	-	0	0	0
34	f59c83cefaaf4185b3b6cce28c3a1f8260fa8265	-	25	111	125
35	0b7e58a62f3592ba8dcb0a7b981a35cacb87a38d	-	25	125	111
36	86b977d579d0bbc46826c17cbdb8251da2bcc816	-	0	0	0
37	46c8a8155df2977caebd695a5b70ae8f7eced731	-	0	0	0
38	5da47577aee96e56ea6c3349eda1f1e63e7498d9	-	0	0	0
39	b639e0d6c6aaf23d845bbd16970fb1aeac1ee99c	-	0	0	0
40	b8779e32e91534556c23a8d1f8aecae4040fddea	-	0	0	0
41	c5d937a356f5c6a0695dd9a1956d4a4137f54ec0	-	0	0	0
42	2b96fddc1edc319bac947162e8d16e710cd2bd3e	-	0	0	0
43	af1e452c86aabd33a3e4e1a70b003931eb5850ad	-	0	0	0
44	845a610cc2dc662f4fe205e61c5e5d5cb68c29b6	-	0	0	0
45	0704e1a80882dc6f161dec85f81b656bd80f73a0	-	0	0	0
46	1b49c068aa280d74095a15dff24eb3a32bb61a26	-	0	0	0
47	0d6ba6828abf3821e1d67a45e3d12cbf33fe0c31	-	0	0	0
48	a5b3f297f6c8b936aa49025599ed61f49b8ed79c	-	0	0	0

49	504aba04279ea7f6bc4f4e62a969d8f427059a03	-	0	0	0
50	53c6efa1dca64a87e536d8c5a3dcc3c12ad933b5	-	0	0	0
Interval II					
0	53c6efa1dca64a87e536d8c5a3dcc3c12ad933b5	IRE	1088	28281	0
1	a8381b6a338e2a60db6dc5b2cd71f20444f47e0d	-	0	0	0
2	4a74699f22885d8c69d579de412d5d36c2129eca	-	0	0	0
3	3ea814eeeebc90f9408d74e8c8080b0f5ce5b538	-	0	0	0
4	4289b157b3fbf02fd4cc20d4d56c5becd844883c	-	1	5	0
5	1b63f2f9ce8a5bd9c69be3fba745281d61cd3c95	-	1	5	1
6	56a53ff7d4ab5da7ab7525bd2520234d630a94b4	-	1	2	1
7	5b3921260ed32009b3dcb5ffaf1cb95d6ef2dc03	-	1	6	0
8	787a23be2345d4b3bb18fa07bde8af84e70f10ee	-	1	1	1
9	64105610d36abe100951e6225e6788e0b89d9fa0	-	0	0	0
10	7d5b672094b48d3efe8d4c2d442a41dd0ae6b1bb	II.A	1	1	0
11	d10092e7899a2c6b614702d1b35db859f2d7195b	II.B	2	1	18
12	1514911c45a3cf28dcccd9a3db671593911badb3	-	0	0	0
13	6fd7c262a1cc9745b62a89c51d8e8ef3b30185be	II.C	3	55	1
14	62c292941ccbcc85267889e9ffc629b5f8072159	-	1	0	2
15	f695f5ae05a1f660c17680d525ec64f7579dc9f9	-	1	5	3
16	6dc58b7cd95f8606a6bac1f1ff75a0120a9a88a5	-	1	1	1
17	6462ae1fd676f6ad21c8c603b6c5a87b3363d89d	II.D	1	57	0
18	d159c35d09d69e9cf364f7ec861b6d8ce9a7fa67	II.E	2	43	14
19	0b8fca9398f69ec929e3eacb6569b4a5f8eb058d	-	0	0	0
20	f8f13f4390f80d3dc8adb0a6167938a688ddb45e	-	0	0	0
Interval III					

0	f8f13f4390f80d3dc8adb0a6167938a688ddb45e	IRE	238	28421	0
1	cf59ce3c99561fb1b83a4785305c119c720d1e73	-	0	0	0
2	624982650875d0dca231e1f8c2d76a559ffed571	-	0	0	0
3	df9eef824f74533c351036fc24b9321024d78fe0	-	2	2	2
4	d5a4464fb9eec467750b48321be89367df5263b0	-	1	2	1
5	bc322fd25c3ea2ab23c7ee45140702febb32a1f9	-	1	1	0
6	583050817139e82eb6ef99b649b2c2f09d504128	-	1	1	1
7	2e0264cedbefda1529a206e68a101764c79326ce	-	1	113	131
8	315fe19fbdd14ab96686840431035c6ba3d6d472	-	2	4	2
9	afa32739012ad86e13b4e6d76c12bc6e341d0120	-	0	0	0
10	77103bdd7d3c8c239943e24b79424315c2924f59	-	1	2	1
11	745469e55fad8a801a92b0be96dc009acbe7e3fb	-	0	0	0
Interval IV					
0	745469e55fad8a801a92b0be96dc009acbe7e3fb	IRE	238	28408	0
1	900cb91e895a801ba518939b6a472d9bfc0adf49	-	0	0	0
2	13256fb5ac64f6d5a7346817d5d2b38bb963baef	-	0	0	0
3	23b0a843735dd5d6b9174266e309624ceb861ec5	-	1	1	1
4	816b98c9c5a210d41f5cd70724fd06c0d2d48833	-	1	1	1
5	1bbab532eed113d31803150bf29ef0c8b26ea556	-	1	1	1
6	e0996dcf7fa9fe7aa1a34752ff73a2300691eabe	-	0	0	0
7	49cb2dd58e109d6402bd8f4c3fd2bc2c352cb4f1	-	0	0	0
8	cd67bbd5497222ecdde45acd8d6e3baa9feddcc6	-	0	0	0
9	bfd2b36394e9453e7020c7906de07e667776ccac	-	2	59	33
10	ad6c8557c81f0dc0601508c49ff7e0af6fe9e0fa	-	0	0	0
11	0d32426425692444ef33e26fe9a99364e88780ab	-	0	0	0
12	6843c00be99c0666195ac577177fbf956e7e332c	-	0	0	0
13	23e5f18ef1f27ed0ab36915a11d46c4945dbd03b	-	0	0	0
14	b42d44b84e49fd1039db68d391fc28d03deff2af	-	0	0	0

15	d9e1cdca5adbde0cbe28c5d15bf298fbb4815319	-	0	0	0
16	882c0481929c7f628042c9f5103165bedbe76b6f	-	0	0	0
17	7d5449f27518baaaa77f37401c82e3376efcb810	-	0	0	0
18	834d84d033a81aa6e5934a7a46f65c2a8a2f7e57	-	0	0	0
19	e1d134bd4f27754a94e5a3e96831d5bd50709a80	-	0	0	0
20	3dc265add6d18f5a2080e8b1def919d87877bfe5	-	0	0	0
21	d6f80444b18a49a75b805edd80ac8ef128d0a560	-	0	0	0
22	b0ecb45238772f06db1e11e8e7baaa72f48cdd96	-	0	0	0
23	b59562c3ea236fbd59d8361397fc7676bda256a1	-	0	0	0
24	bb6a24e0082436660055f1fd9ea11f4bafdbcf1f	-	0	0	0
25	8e182111dce10d0f84fa8b775b5a5f72c1c00b14	-	0	0	0
26	8b9c3c4b61135bbe7a2fa340913aa95472200396	-	0	0	0
27	d2b50bf17c9ea4f6d48f2511f351820c63c0053a	-	0	0	0
28	40fc2388eadd1a0194c765abc450ff10f6156dae	-	0	0	0
29	f46215c1ee9a2839e644dd69c8ca099626a2861f	-	0	0	0
30	1973d02a7a15565b368a50f865588f2e67751e75	-	0	0	0
31	00f9142a19103f063fa7da48c29d8978d5a4352d	-	0	0	0
32	3d040638f35d890808b23a846b1cc61426f96249	-	0	0	0
33	a96dd569448292fd5dd83e28eb0b303e3a0c5562	-	0	0	0

34	e9dadd5e84753e4ac832d12eaf376ef51507c903	-	0	0	0
35	2cca63aad1db3e92af334fca79d786386d5ea3d5	-	0	0	0
36	b0d62482cbd113ad1d61c374baf37031c4c5d867	-	0	0	0
37	85372b809d883c1efba4288f484e093fe49d5f61	-	0	0	0
38	815f4f64ff3d73d1211e44b6d099be4248266136	-	0	0	0
39	ffcbe96c8a2a307f96db1da4fde1d1a2ad80999c	-	0	0	0
40	0f19bc5dea0b73b623e378104abf7d73de453481	-	0	0	0
41	1da27c65d47b38d6ea3cc10bb247ff7e3c69c17f	-	0	0	0
42	d278bb55ea23f40325edfa6244a6abd6ce7a9e4e	-	0	0	0
43	014712685ae4033aa125bce7fcadc4c39ccb7c24	-	0	0	0
44	4c48631b69b9597c7cc83e16b596a4fa72bb58f2	-	0	0	0
45	57df9652d55643b0af5d4a097aabf4138b8ac1cb	-	1	1	1
46	1cbc324e1640a207032922858a9e56ea92aba779	-	0	0	0
47	bdcbb8ce1e28c8f9b708c79fe97f1d1bee4ced59	-	0	0	0
48	0bf9aa78024adc2631950ea0004b5c561d2bf5e5	-	1	1	1
49	ea62e166bc38c1ddbf76c155072aac1c5ee56c89	-	0	0	0
50	2d422b9abdd27f810eee48aa25d30b21233ef06c	-	1	1	1
51	653528e387a4fb764e48c9422ab218c65f87082f	-	6	153	193
52	0ce0cababe2590eff6d03c15f9eb3c977fdf4914	-	1	7	6

53	df597ccdd0e16f651f0113e04d014320573fee76	-	0	0	0
54	03c1f1c3a179dde5c3cc33bcf0edf60a0b49ea52	-	0	0	0
55	032167ddc872887fa1eb4c1fe2df8209a0bd7b76	-	0	0	0
56	19662c0afc9e8005917e8eba42d9b59592d0feeb	-	0	0	0
57	4cd9e06869ab671bee1d11b7abc3eebf9b582f35	-	0	0	0
58	a7727800a9879c96596e56d320420dc4f1bd8a2c	-	0	0	0
59	3e17bfbb8b39a1eba260abb9085969fb30fdb9d1	-	0	0	0
60	e0ffa6a1d477c1e1cc65f8eeb1c3bcabb6f6f5da	-	0	0	0
61	cf75936549548faa52e37c098aa16c24dde8a067	-	0	0	0
62	6d334d025b2133a92fee8d645b078f109e65f5cf	-	0	0	0
63	806e5cc946f7995256d0da40492a2050c74e5ada	-	1	1	0
64	12efccf95b372d0df94075e7797ea4b96c9b5c88	-	0	0	0
65	79330c1f9c9efc2118b018805f4c629d6096b9b9	-	0	0	0
66	9f0ec365de1cb438904530f2b90e65d9d296d6ef	-	9	227	215
67	05ade1575d5fb2276ed4e2ac0f272dbd0533b141	-	9	215	227
68	c4860ee3392cfb00e83a824b1e9c7b991e10c318	-	0	0	0
69	6d222bf82dab2e192ab7820f3d5619875b80bd31	-	0	0	0
70	8f9a8eeb2ed6f1486b5286e8dfed61ab279566ca	-	0	0	0
71	b58219ac535670f3e63700289abe8a6ce0d7dcf0	-	0	0	0

72	790452f95a9d51803afd26b36788957a0cca2c4d	-	0	0	0
73	3c674e12622a585faa39334eeca3daff98fddc80	-	0	0	0
74	637ec65a89eef33aabee8ca031d283f0e2db02b9	-	0	0	0
75	0344db1d3621df6bb5a6787973f4d9db599b480b	-	0	0	0
76	b7c4e9df95668509f7058d2fa32157229c795fc9	-	0	0	0
77	c43f0cc67b88a1f5b982c380cc6dc5b52686f471	-	0	0	0
78	87c32b70ff19df2f32f899cd659741ae60e9ac6b	-	0	0	0
79	07106148f2cedb12c01525b1586127eadefef986	-	0	0	0
80	0ffdf356e82b2e26f93a828ed4f378c505bd68f2	-	0	0	0
81	bdde10b80d195d08ed7e3617107de006210d753a	-	0	0	0
82	10f049d663e523873f27e6a794635572a975e2a5	-	0	0	0
83	a349ddeb32c58d49376cf8d165908312b5a3405b	-	0	0	0
84	e29360bd1698a6cd2f1f28a9aa6282276f582e58	-	0	0	0
85	02877a14aeaaaa9cc1365dafafeb85d8f3387e9b	-	0	0	0
86	045b1a917ac70fec559a2548e14977acc09dd76a	-	0	0	0
87	98570ca8ac398dc4d1f6e944b895bccb39726867	-	0	0	0
88	df33017194634db70c4e786b64cc1d6b386fe676	-	0	0	0
89	a543c026e9563f50ff6371912d470ec30f261ae3	-	0	0	0
90	65971f539d55fff3444a02bc31f6d3d9b4f26813	-	0	0	0

91	9a7ba550cd47ba4b7d9a652814d337f2f1f6d79b	-	0	0	0
92	6f70c3feb0640bf9d8691acdcf2dccae1710cbf0	-	0	0	0
93	febeaa537041ef7b23e479ec958586c2154e9ab4	-	0	0	0
94	d16000d2a74a299d9722c840434bedf109c04198	-	0	0	0
95	30675d11c9267eb2aee3d04e73b28a13a4eab8df	-	0	0	0
96	1a11ca0e48d59f665d30a6303ed49d14cafa13c5	-	0	0	0
97	a5f12f3bbb8a1c58c257ec0b49667cac9fca94a3	-	0	0	0
98	2c25727b02222308b21399bcd8266f732bbbe41d	-	0	0	0
99	49e2437abf782b3a618fe5ed155a2c8469557e47	-	0	0	0
100	de69e957597d20d4be17fc7db2a0aa2fb3a414f7	-	0	0	0

A.3. Example of the Adaptive Instrumentation

In this example, we focus on the adaptive instrumentation of the `clearAllCaches` service from the TeaStore case study. Initially, the method's `InternalAction` is instrumented with both enter and exit statements to monitor response times and log the unique identifier ("ID") of the internal action, as shown in Listing A.1. Typically, the unique ID of an internal action (e.g., `_YHXHhwzdEeyhr8BpjC`) is embedded in the monitoring records. This ID serves as a key reference within the monitoring system, enabling the correlation of monitoring data with the corresponding internal actions, which can then be calibrated based on the collected data.

Listing A.1: Direct instrumentation of the `InternalAction` of `clearAllCaches`.

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

To ensure successful compilation of the instrumented code, adjustments are required when the exit statement follows a return statement. As depicted in Listing A.2, the return value is assigned to a local variable (`resp1`) and returned after the exit statement.

This adjustment allows the code to compile correctly while ensuring that the monitoring data is captured throughout the method execution.

Listing A.2: Corrected instrumentation of the `InternalAction` of `clearAllCaches`.

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

A.4. Teammates case study

A.4.1. Git Histoy

We provide some historical Information on the TEAMMATES Git history period, which we considered in our Evaluation.

Table A.2: The five intervals and information on their commits. The commits are numbered continuously to ease readability.

Interval	Hash of Commits Range	Commits	Java-related Commits	Added Lines	Removed Lines	Architectural changes
I	648425746bb9434051647c8266dfab50a8f2d6a3 to 48b67bae03babf5a5e578aefce47f0285e8de8b4	17832	709	114468	0	RE
II	648425746bb9434051647c8266dfab50a8f2d6a3 to 48b67bae03babf5a5e578aefce47f0285e8de8b4	3	122	154	129	4
III	48b67bae03babf5a5e578aefce47f0285e8de8b4 to 83f518e279807dc7eb7023d008a4d1ab290fefee	2	227	3249	2978	244
IV	83f518e279807dc7eb7023d008a4d1ab290fefee to f33d0bcd5843678b832efd8ee2963e72a95ecfc9	2	65	502	340	27
V	f33d0bcd5843678b832efd8ee2963e72a95ecfc9 to ce4463a8741840fd25a41b14801eab9193c7ed18	20	147	3457	1293	176

A.5. Lua-based sensor applications

In this section, we present historical information about the Git history of Lua-based sensor applications. Section A.5.1 provides details regarding the Git history of the BarcodeReaderApp, while Section A.5.2 includes information on the ObjectClassifierApp.

A.5.1. Git History of BarcodeReaderApp

Table A.3 includes details on Commits of BarcodeReaderApp.

Commit Index	Hash	Added Lines	Removed Lines	Changed Files	Description
1	e25fb6b	575	0	4	Adding BarcodeReader, HTTPClient, and HTTPServer.
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 A.3: The commits of the BarcodeReaderApp.

A.5.2. Git History of ObjectClassifierApp

Table A.4 includes details on Commits 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

Table A.4: The selected Git commit history of ObjectClassifierApp.