Deliverable D.A6a
Predictable / Manageable Service Engineering Methodology and Prediction Services

Keywords:
QoS Meta-Model, Design-time prediction, Run-time prediction, Manageability Design

Due date of deliverable: 31st July 2011
Actual submission to EC date: 29th July 2011

Start date of project: 1st June 2008
Duration: 38 months

Lead contractor for this deliverable: FZI Forschungszentrum Informatik
Revision: 2.7 (22nd July 2011)
Document Status

<table>
<thead>
<tr>
<th>Deliverable Lead</th>
<th>FZI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer 1</td>
<td>Agustín Escamez (TID)</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Liliana Pasquale (PMI)</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Wolfgang Theilmann (SAP)</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td>2011-06-17</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>2011-07-19</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>-</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>2011-07-25</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>2011-07-27</td>
</tr>
</tbody>
</table>

Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FZI</td>
<td>Franz Brosch, Giovanni Falcone, Samuel Kounev, Benjamin Klatt</td>
</tr>
<tr>
<td>CITY</td>
<td>Davide Lorenzoli, George Spanoudakis</td>
</tr>
<tr>
<td>INTEL</td>
<td>Victor Bayon, Phil Tian</td>
</tr>
<tr>
<td>SAP</td>
<td>Jens Happe, Dennis Westermann, Christof Momm, Christian Heupel, Alexander Wert, Hui Li, Giuliano Casale, Tariq Ellahi, Daniel Scheibli</td>
</tr>
<tr>
<td>PMI</td>
<td>Sam Guinea, Luciano Baresi</td>
</tr>
</tbody>
</table>

Notices

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2009 by the SLA@SOI consortium.

* Other names and brands may be claimed as the property of others.

This work is licensed under a Creative Commons Attribution 3.0 License.
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>2009-03-31</td>
<td>Franz Brosch</td>
<td>Outline created</td>
</tr>
<tr>
<td>0.1</td>
<td>2009-04-15</td>
<td>Franz Brosch</td>
<td>Added FZI contributions (run-time prediction, manageability)</td>
</tr>
<tr>
<td>0.2</td>
<td>2009-04-17</td>
<td>Davide Lorenzoli</td>
<td>Added CITY contribution (service level prediction and CITY monitor)</td>
</tr>
<tr>
<td>0.3</td>
<td>2009-04-27</td>
<td>Phil Tian</td>
<td>Added INTEL contributions</td>
</tr>
<tr>
<td>0.4</td>
<td>2009-05-01</td>
<td>Christof Momm, Jens Happe</td>
<td>Completed FZI contributions</td>
</tr>
<tr>
<td>0.5</td>
<td>2009-05-21</td>
<td>Davide Lorenzoli</td>
<td>Addressed review comments for CITY contribution</td>
</tr>
<tr>
<td>0.6</td>
<td>2009-05-22</td>
<td>Christof Momm, Franz Brosch, Jens Happe</td>
<td>Addressed review comments for FZI contribution</td>
</tr>
<tr>
<td>0.7</td>
<td>2009-05-24</td>
<td>Phil Tian</td>
<td>Addressed review comments for INTEL contribution</td>
</tr>
<tr>
<td>0.8</td>
<td>2009-06-03</td>
<td>Jens Happe</td>
<td>Addressed review comments of PMT, finalized deliverable</td>
</tr>
<tr>
<td>1.0</td>
<td>2010-04-21</td>
<td>Franz Brosch</td>
<td>Updates for M26 version: Executive Summary, Introduction, Contributions Overview</td>
</tr>
<tr>
<td>1.1</td>
<td>2010-06-01</td>
<td>Franz Brosch</td>
<td>Added FZI contributions for M26 version</td>
</tr>
<tr>
<td>1.2</td>
<td>2010-06-12</td>
<td>Davide Lorenzoli</td>
<td>Added CITY contributions for M26 version</td>
</tr>
<tr>
<td>1.3</td>
<td>2010-06-20</td>
<td>Sam Guinea</td>
<td>Added PMI contributions for M26 version</td>
</tr>
<tr>
<td>1.4</td>
<td>2010-06-22</td>
<td>Victor Bayon</td>
<td>Added INTEL contributions for M26 version</td>
</tr>
<tr>
<td>1.5</td>
<td>2010-08-18</td>
<td>Davide Lorenzoli, Sam Guinea, Victor Bayon, Franz Brosch, Jens Happe</td>
<td>Revised contribution according to internal review comments; added SAP statistical inference contribution</td>
</tr>
<tr>
<td>1.6</td>
<td>2010-08-20</td>
<td>Franz Brosch</td>
<td>Corrected format issues and literature references</td>
</tr>
<tr>
<td>1.7</td>
<td>2010-10-12</td>
<td>Davide Lorenzoli</td>
<td>Added lost updated CITY contribution</td>
</tr>
<tr>
<td>2.0</td>
<td>2011-06-03</td>
<td>Franz Brosch</td>
<td>Updated general Chapters 1/2/7 to reflect the whole project’s lifetime.</td>
</tr>
<tr>
<td>2.1</td>
<td>2011-06-08</td>
<td>Davide Lorenzoli</td>
<td>Updated Chapter 5 to account for progress in third project period.</td>
</tr>
<tr>
<td>2.2</td>
<td>2011-06-09</td>
<td>Jens Happe Alexander Wert</td>
<td>Updated Section 3.5.5</td>
</tr>
<tr>
<td>2.3</td>
<td>2011-06-14</td>
<td>Dennis Westermann</td>
<td>Updated Section 3.8.2</td>
</tr>
<tr>
<td>2.4</td>
<td>2011-06-16</td>
<td>Franz Brosch</td>
<td>Revised Chapter 3</td>
</tr>
<tr>
<td>2.5</td>
<td>2011-07-13</td>
<td>Sam Guinea</td>
<td>Revised Chapter 6 according to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y3 progress</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>2011-07-20</td>
<td>George Spanoudakis</td>
<td>Revised Chapter 5 according to internal review comments</td>
</tr>
<tr>
<td>2.7</td>
<td>2011-07-22</td>
<td>Franz Brosch</td>
<td>Finalized deliverable for submission</td>
</tr>
</tbody>
</table>
Executive Summary

This document describes the advances regarding predictable systems engineering which have been reached in SLA@SOI as an outcome of the efforts made within the A6 work package. The document serves as a formal deliverable D.A6a, reporting the overall progress of WP A6 during the project’s lifetime, and explicitly denoting the parts that are specific to the third project year, which have not yet been covered by the two earlier versions of the report in project months 12 and 26.

The A6 work package realizes a top-level objective of SLA@SOI, to advance the engineering of predictable service-oriented systems by methodologies, modelling techniques, and prediction tools covering SOA and SOI components. Predictability is an important feature that helps service and infrastructure providers to make well-informed decisions throughout service design, offering, negotiation, provisioning and run-time. At its core, predictability is the capability to anticipate quality-related properties of services and service hierarchies, before those properties can actually be observed.

The methodologies and techniques developed in WP A6 can be grouped into four main contributions, namely (i) software performance and reliability prediction, (ii) resource usage prediction, (iii) run-time SLA violation prediction and (iv) manageability modelling and design. Each contribution covers the needs of specific roles, such as software service providers or infrastructure providers, at specific stages in the SLA and service lifecycles. Concrete tool realizations have been included in the SLA management framework, which constitutes the main technical outcome of SLA@SOI. In addition, stand-alone tools have been provided supporting further design-related activities, which are not directly in the focus of the framework.

This document describes the nature of each contribution, detailing how it supports the predictable systems engineering, how it is embedded into the SLA framework architecture, and how it is applied during the stages of the SLA and service lifecycles. It presents the involved data models and their interrelations, as well as the application of the contributions to the industrial use cases of the project. Furthermore, it describes conceptual relations to other scientific work packages, lists relevant publications over the project’s lifetime, and summarizes main achievements and lessons learned from the results of WP A6.
# Table of Contents

1 Introduction ................................................................. 11
   1.1 Context and Scope .................................................. 11
   1.2 Document Overview ............................................... 12
2 Contributions Overview ................................................ 13
   2.1 Key Innovations .................................................. 13
   2.2 Framework Contributions ....................................... 14
   2.3 A-Line Interrelations ............................................ 15
   2.4 Task-level Activities (3rd Year) ............................... 16
3 Software Performance and Reliability Prediction ................... 17
   3.1 Introduction ..................................................... 17
   3.2 Overview ........................................................ 19
      3.2.1 Goals and Scope ........................................... 19
      3.2.2 Preliminaries .............................................. 20
      3.2.3 Roles and Responsibilities ............................... 21
      3.2.4 Prediction Workflow ..................................... 21
   3.3 State of the Art .................................................. 23
      3.3.1 Architecture-based Performance Prediction ............... 23
      3.3.2 Architecture-based Reliability Prediction ............... 25
   3.4 QoS Meta-Model for Performance and Reliability Prediction .... 28
      3.4.1 Overview .................................................... 29
      3.4.2 Performance-specific Aspects ............................. 30
      3.4.3 Reliability-specific Aspects .............................. 31
   3.5 Prediction Models Adjustment ................................... 31
      3.5.1 Performance Cockpit ....................................... 32
      3.5.2 Experiment Definition .................................... 33
      3.5.3 Automated Measurements .................................. 34
      3.5.4 Statistical Inference ..................................... 35
      3.5.5 Model Integration ........................................ 35
   3.6 The Prediction Service .......................................... 38
      3.6.1 Overview .................................................... 38
      3.6.2 Prediction Engine .......................................... 39
      3.6.3 Prediction Process ........................................ 41
      3.6.4 Supported Scenarios ...................................... 42
   3.7 Complementary Prediction Services ............................... 43
      3.7.1 Cost Prediction Service ................................... 43
      3.7.2 Multi-objective Optimization ............................. 48
   3.8 Use Cases for Performance and Reliability Prediction ......... 50
      3.8.1 WP B2: Open Reference Case ............................... 50
      3.8.2 WP B3: ERP Hosting ........................................ 55
      3.8.3 WP B6: eGovernment ....................................... 58
4 Resource Usage Prediction .............................................. 62
   4.1 Introduction ........................................................ 62
   4.2 Prediction User Interfaces ....................................... 67
      4.2.1 Standalone User Interface ................................ 67
   4.3 Runtime Prediction Integration Scenario ......................... 68
   4.4 Design Rationale ................................................. 71
      4.4.1 Prediction Overhead ....................................... 71
      4.4.2 Scalable Architectures .................................... 72
   4.5 Architecture and Implementation ................................ 74
      4.5.1 Client Agent ............................................... 74
      4.5.2 Prediction Agent ........................................... 77
      4.5.3 Prediction Worker .......................................... 79
5 Run-time SLA Violation Prediction ................................................................. 81
  5.1 Introduction ......................................................................................... 81
  5.2 Evolution of the EVEREST Framework ............................................... 81
  5.3 Prediction Model .................................................................................. 83
       5.3.1 Overview .................................................................................... 83
       5.3.2 Prediction Model for Software Services ......................................... 84
  5.4 Runtime Computation Of MTTR/MTTF Models ...................................... 86
       5.4.1 Prediction Specifications ............................................................... 87
       5.4.2 Computation of MTTR/MTTF Prediction Models ......................... 90
  5.5 Experimental Results ........................................................................... 91
       5.5.1 Evaluation of Precision and Recall ............................................... 91
       5.5.2 Efficiency .................................................................................... 93
  6 Manageability Modelling and Design ....................................................... 94
       6.1 Overview ......................................................................................... 94
       6.2 The Management Meta-Models ....................................................... 94
            6.2.1 The System Meta-Model .......................................................... 95
            6.2.2 The Data Collection Meta-Model .............................................. 96
            6.2.3 The Adjustment Meta-Model ..................................................... 98
       6.3 Synthesis of Data Collection ........................................................... 101
            6.3.1 Synthesis of Sensors ................................................................. 101
            6.3.2 Synthesis of Correlation ......................................................... 102
       6.4 Synthesis of Adjustment ................................................................. 103
            6.4.1 Synthesis of Binding Rules ..................................................... 103
            6.4.2 Synthesis of Model Changes .................................................. 105
  6.5 Example Models .................................................................................. 106
  7 Conclusions ......................................................................................... 109
       7.1 Contributions and Achievements ...................................................... 110
       7.2 Lessons learned ............................................................................... 110
  7.3 Outlook ............................................................................................... 111
  8 References ........................................................................................... 112
  Appendix A: Standard QoS Terms for Prediction ........................................ 123
  Appendix B: QoS Meta-Model ................................................................. 124
  Appendix C: Service Evaluation ............................................................... 132
  Appendix D: Glossary .............................................................................. 142
  Appendix E: Abbreviations ...................................................................... 145
Table of Figures

Figure 1: WP A6 Contributions Overview .................................................. 11
Figure 2: Overview of WP A6 Contributions and A-Line Interrelations ........ 16
Figure 3: Abstract Performance and Reliability Prediction Workflow .......... 22
Figure 4: QoS Meta-Model Structure (Overview) ..................................... 29
Figure 5: Goal-oriented Systematic Measurements ................................... 32
Figure 6: Performance Cockpit Idea .......................................................... 33
Figure 7: Experiment Definition Meta-Model .......................................... 34
Figure 8: Automated Measurements .......................................................... 35
Figure 9: Integration Concept ................................................................. 36
Figure 10: Integration Meta-Model ............................................................ 37
Figure 11: Simulation Behaviour in Pseudo Code .................................... 38
Figure 12: Prediction Tooling (Overview) ................................................ 39
Figure 13: Evaluation of Software Services ............................................ 41
Figure 14: TPC-C Benchmark Results ...................................................... 44
Figure 15: Power Function Fitting ............................................................ 45
Figure 16: Normalized Power vs. CPU Utilization .................................... 45
Figure 17: SLA-driven Planning Framework Overview ............................. 49
Figure 18: ORC Components ................................................................. 51
Figure 19: Payment Service SEFF ............................................................ 52
Figure 20: ORC System Specification ....................................................... 53
Figure 21: ORC Infrastructure Model ....................................................... 54
Figure 22: ORC Allocation Model ............................................................ 54
Figure 23: ORC Usage Model ................................................................. 55
Figure 24: Regular Steps within the SD Benchmark ................................. 56
Figure 25: Experiment Selection Strategies ............................................ 57
Figure 26: Case Study Results ............................................................... 58
Figure 27: eGovernment Booking Process ............................................. 59
Figure 28: Service Component Types and Interfaces ............................... 59
Figure 29: Service Architecture for the Booking Process ......................... 60
Figure 30: Usage Profile of the Booking Service ...................................... 61
Figure 31: Behavioural Specification of Acquiring Operators .................... 62
Figure 32: Internal Architecture of Prediction ........................................ 62
Figure 33: Basic Training/Prediction Workflow ...................................... 63
Figure 34: Dynamic Prediction with Future Error .................................... 64
Figure 35: Training and Re-training Workflow ....................................... 65
Figure 36: Prediction versus Real Instrumentation ................................... 65
Figure 37: Steady-State Metric with Spikes ............................................. 66
Figure 38: Prediction versus Real Instrumentation (Running Average) ....... 66
Figure 39: Resource Usage Prediction User Interface: Training ................. 68
Figure 40: Resource Usage Prediction User Interface: Testing ................. 68
Figure 41: Prediction Dynamic User Interface ........................................ 69
Figure 42: Prediction Usage Scenario ...................................................... 70
Figure 43: SQL-like Query for Prediction Scenario ................................... 70
Figure 44: Resource Usage (cpu_user) Over Time .................................... 71
Figure 45: Prediction Data Gathering and Processing .............................. 75
Figure 46: Prediction for cpu_usage* Metric .......................................... 78
Figure 47: Prediction Instrumentation of the B4 Use Case ....................... 79
Figure 48: Prediction Provisioning ......................................................... 80
Figure 49: Prediction Framework Common Definitions ............................ 83
Figure 50: TTR and TTF Values .............................................................. 85
Figure 51: EVEREST+ Architecture ....................................................... 86
Figure 52: Prediction Specification for MTTR/MTTF ............................... 88
1 Introduction

1.1 Context and Scope

The European research project SLA@SOI aims at “a business-ready service-oriented infrastructure empowering the service economy in a flexible and dependable way”. To this end, one of the defined top-level objectives of the project is “to advance the engineering of predictable service-oriented systems by methodologies, modelling techniques, and prediction tools covering SOA and SOI components”. Within the project, the scientific work package A6, “predictable systems engineering”, is devoted to this objective. This document serves as a formal deliverable D.A6a, reporting the overall progress of WP A6 during the project’s lifetime, and explicitly denoting the parts that are specific to the third project year (see Section 1.2), which have not yet been covered by the two earlier versions of the report in project months 12 and 26.

WP A6 specifically focuses on the challenge to predict service quality properties before they can be actually observed. Predictability is an essential feature that helps service and infrastructure providers to make well-informed decisions throughout service design, offering, negotiation, provisioning and run-time. To achieve predictability, traditional component-based and service-oriented development techniques have to be enriched and complemented by systematic consideration of quality aspects. On a broader scale, WP A6 also considers the design-time activities that are necessary to enrich service components with monitoring and control capabilities. Monitoring of quality properties is a prerequisite for predictability at run-time, and control actions enable the adjustment of the running system in response to the predicted service quality.

![Figure 1: WP A6 Contributions Overview](image)

Figure 1 summarizes the common work of the partners that participate in the A6 work package into four main contributions, namely (i) software performance and reliability prediction (supported by the statistical inference of software performance curves), (ii) resource usage prediction, (iii) run-time SLA violation...
prediction, and (iv) manageability modelling and design. The figure puts these contributions into the context of the service lifecycle, shows the central questions of service and infrastructure providers that are answered by the contributions, and links the contributions to the providing tasks of the work package. Contributions (i) to (iii) are directly focused on predictability, whereas contribution (iv) deals with manageability design as a prerequisite and necessary context enabling predictable systems engineering. For each contribution, this document provides a description of the activities, progress and achievements during the project’s lifetime, covering all resulting methodologies, tools, techniques and data models. Additionally, the document describes the application of the WP A6 contributions in the industrial use cases realized through B-line work packages B3 to B6.

**Main Achievement**

The main achievement of this work package is a comprehensive set of methodologies and tools enabling the engineering of predictable systems, allowing service and infrastructure providers to make well-informed decisions throughout the stages of the SLA and service lifecycles.

**1.2 Document Overview**

This document describes the overall outcome of the A6 work package. Table 1 explicitly denotes the parts that are specific to the third reporting period, which have not been covered by earlier versions of this report. Overall, the WP A6 activities in the third project year were devoted to the completion of the scientific work, as well as the orientation towards the industrial use cases, in order to apply the scientific results. Notice that the contents of Chapter 4 (Resource Usage Prediction) have not been changed as the work on this approach was already completed at the end of the second project year.

<table>
<thead>
<tr>
<th>Section</th>
<th>Change overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapters 1/2/7</td>
<td>General revision to reflect the progress, experiences and lessons learned of the project’s whole lifetime</td>
</tr>
<tr>
<td>Section 2.3</td>
<td>The description of task-level activities has been updated to refer to the activities of the third project year.</td>
</tr>
<tr>
<td>Section 3.2</td>
<td>The list of predicted quality attributes has been completed by adding the reliability-related attributes (Section 3.2.1).</td>
</tr>
<tr>
<td>Section 3.5</td>
<td>Has been updated to reflect the new integration of model-based prediction and measurements-based performance curves (Section 3.5.5).</td>
</tr>
<tr>
<td>Section 3.6</td>
<td>The description of the prediction service has been updated to reflect the new capabilities for reliability prediction (Section 3.6.2) and to discuss how prediction supports scenarios of varying complexity (Section 3.6.4).</td>
</tr>
<tr>
<td>Section 3.8</td>
<td>Use case descriptions have been updated to include the consideration of reliability aspects and other enhancements during the third project year.</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Chapter has been rewritten with additional experimental results and a more detailed contribution about run-time prediction underpinning statistical models.</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>The descriptions of the manageability meta-models have undergone heavy changes with respect to those presented in year 2. Besides a different overall organization, we have</td>
</tr>
</tbody>
</table>
introduced new support for event correlation in the context of runtime data collection, and we have redefined completely our notion of runtime adjustments. We now support various types of dynamic binding, and service model adjustments.

Table 1: Document Changes against Previous Version

The reminder of this document is structured as follows. Chapter 2 provides an overview over key innovations of the A6 work package, the contributions of A6 to the overall SLA management framework, and the task-level activities that were taken in order to achieve the envisioned results. Chapter 3 presents achievements regarding software performance and reliability prediction, including an overview of the predictable service engineering methodology and a description of the realized prediction service. Chapter 4 describes the scope, goals and application of resource usage prediction, and its implementation as part of the infrastructure management architecture. A description of run-time SLA violation prediction and its place in the monitoring architecture is given in Chapter 5. The next Chapter 6 is devoted to the fourth contribution of WP A6, manageability modelling and design. Chapter 7 concludes the deliverable with a summary of reached results and lessons learned over the project’s lifetime.

2 Contributions Overview

This chapter summarizes the activities and outcomes of the A6 work package, and illustrates how this work package contributes to the overall goals and outcomes of the SLA@SOI research project. The discussion includes scientific innovations (Section 2.1), contributions to the SLA management framework (Section 2.2), interrelations between WP A6 and other A-line work packages (Section 2.3) and task-level activities in the third project year (Section 2.4).

2.1 Key Innovations

In the following, the key innovative aspects of each of the main WP A6 contributions are described, which distinguish the contribution from related work and enable it to be part of the WP A6 toolset for predictability.

Software Performance and Reliability Prediction

This contribution provides a means for software service providers to evaluate the expected performance and reliability of their services before their run-time. The approach may be used at the service offering / SLA template design stage to determine realistic quality parameters to offer, or at the (automated) service / SLA negotiation stage, to determine realisable quality parameters to agree upon. The most important innovations connected to the approach are a comprehensive consideration of architectural influences on service quality (including the service component implementations, the quality of required services, and the execution environment), the support of a distributed development process through the separation of modelling concerns between software and service providers, and a detailed modelling of the service usage including system workloads, call sequences and input parameter values for service calls. See Section 3.3 for further details on the state of the Art.
**Statistical Inference of Software Performance Models**

This work complements the model-based software performance and reliability prediction with a measurement-based approach that eases the integration of existing systems into the analysis. This extension is especially useful for the application in the WP B3 industrial use case, where a large stack of existing software has to be considered. Dependencies are identified (semi-) automatically by systematic measurements and observations of a real system, and corresponding performance models are derived through statistical inference. A detailed understanding and manual architectural specification of the whole software stack is not required. For a description of the approach, see Section 3.5.

**Resource Usage Prediction**

The resource usage prediction provides a means for infrastructure service providers to predict expected infrastructure resource demands at system run-time. As its main innovation, the approach realizes a scalable architecture that can be used for providing on-demand provisioning services for many SLA aware resources such as servers or VMs. The main feature of this architecture is the usage of an agent based implementation to facilitate the distribution of prediction computations over a network of agents.

**Run-time SLA Violation Prediction**

This contribution provides a means for software service providers to anticipate possible near-future software SLA violations at run-time. The approach may be used at the service operations / SLA runtime stage to trigger adjustment activities in order to avoid an actual SLA violation. The most important innovative features of the approach are the integration of monitoring and prediction capabilities into a single coherent framework, the realization and experimental evaluation of the implemented prediction models prediction model (e.g., the service mean-time-to-repair (MTTR) and mean-time-to-failure (MTTF) models), and the flexibility of the framework whose design enables extensions with new predictors aimed to cover other quality-of-service (QoS) properties.

**Manageability Modelling and Design**

This contribution provides a model-driven engineering approach that explicitly supports designing service-based systems that are manageable. This includes the specification of the sensors to be deployed to the system to collect and process the run-time data needed for monitoring, as well as the specification of the effectors that provide the control capabilities needed to adjust a system dynamically. In terms of innovation with respect to the state of the art the approach provides a coherent solution that can be used both with black-box services, and white-box services. For the former we support the definition of monitoring and control at the interface level; for the latter we support the monitoring and control at the internal code level (e.g., BPEL).

**2.2 Framework Contributions**

Within the context of the main WP A6 contributions, concrete tool realizations have been included in the SLA management framework, which constitutes the main technical outcome of SLA@SOI (see deliverable D.A1a for a detailed description of the framework architecture and D.A1b for the implementation). In addition, stand-alone tools have been provided supporting further design-related activities, which are not directly in the focus of the framework. The following
The software performance and reliability prediction provides an implementation of the Service Evaluation component within the SLA framework architecture. This component supports the SLA planning and optimization process of SLA Managers at the SLA negotiation stage, with the goal to agree upon SLAs with feasible parameters. When invoked, Service Evaluation gets information about a given configuration and envisioned usage of the service-based system as an input; it then calculates and returns the expected quality characteristics of the system.

Additionally, service providers can use a stand-alone tool environment based on the Palladio Component Model (PCM) [1], with graphical editors supporting the different modelling tasks of the providers. The tool allows for conducting performance and reliability predictions, and thus can be used for determining initial service offers during SLA template design stage. Furthermore, the created QoS model instances are an input to the design-time repository, which is used by the Service Evaluation component for performance and reliability prediction during SLA negotiation stage.

The resource usage prediction can be used within the framework as a standalone component that is executed as a service. Once deployed and configured, it provides predictive metrics over low level resources such as CPU usage or memory consumption to other services of the framework such as planning, optimization or adjustment components. The metrics are accessible via a http REST interface that allows queries in terms of resource and time, facilitating easy integration and reuse. Standalone, the different parts of resource usage prediction have been implemented as separate components and they can be reused according to the need of the use case.

The framework contributions of the run-time SLA violation prediction become manifest through the design and implementation of EVEREST+, a general and extensible model-driven prediction framework, including several QoS predictors for standard QoS terms specified in the SLA model. EVEREST+ exposes its prediction functionalities via EVEREST+ reasoning component gateway (RCG), a component implementing the SLA@SOI interface for reasoning components. EVEREST+ RCG translates monitoring system configurations, provided by Monitoring Manager, into prediction specifications. It also provides prediction results in the form of SLA@SOI events.

The manageability modelling and design task does not explicitly contribute a component to the SLA@SOI framework. It does however provide a standalone tool for modelling management issues that is compliant with SLA@SOI, and our meta-models. The tool is based on the Eclipse Modelling Framework (EMF), and consists of an Eclipse plugin. Through appropriate transformation code the tool supports the automatic synthesis of configurations for the instrumentation-based sensors that need to be deployed to the running system. This deployment is achieved using the SLA@SOI framework’s Manageability Agent.

### 2.3 A-Line Interrelations

In order to set the contributions of WP A6 into the context of the overall A-line work, Figure 2 shows the main dependencies among A6 and the other A-line work packages. WP A6 builds upon WP A1 (Architecture and Integration) and embeds itself into the overall architecture of the SLA management framework. The meta-
models developed in WP A6 use fundamental concepts provided by the WP A1 core meta-model, and add further concepts specific to the domain of quality prediction and manageability design. The implemented WP A6 prediction services go as prediction components into the SLA framework.

![Figure 2: Overview of WP A6 Contributions and A-Line Interrelations](image)

WP A6 is also related to all the other A-line work packages. The software performance and reliability prediction service is used by the SLA planning and optimization functionality realized by WP A5 (SLA Management) in order to automatically detect and negotiate feasible SLA agreement terms. The data models of the prediction service interface build upon the SLA meta-model developed in WP A5. The approach to service usage modelling taken in WP A6 is related to the business SLA definition of WP A2 (Business Management), as service usage ultimately has to be agreed upon with service customers. The WP A6 run-time SLA violation prediction is part of the overall monitoring architecture developed in WP A3 (Service Management), and takes the monitored service quality data as an input for prediction. The outcomes of prediction form a decision basis for control activities to adjust the running system, or to trigger an SLA renegotiation as considered by WP A5. The monitoring activity itself is enabled with the help of WP A6 manageability modelling and design, which equips service components with monitoring and control capabilities.

The A4 work package (Infrastructure Management) is also related to WP A6, as the resource usage prediction enriches the infrastructure management architecture with the possibility to base virtual machine (VM) provisioning decisions on expected future trends of resource utilization.

### 2.4 Task-level Activities (3rd Year)

This section provides an overview over the activities and reached milestones of each individual task within the A6 work package in the 3rd project year. The overview reflects the DoW task structure and description resulting from the latest Amendment 3.

**Task A6.1 – Meta-models for Prediction Services**: Due to its foundational character, activities related to this task already took place in the first two project
years. In the third year, a dedicated effort has been made to clarify the relation of the QoS meta-model and the SLA model (as documented in D.A1a).

**Task A6.2 - Performance Prediction Service:** Efforts in this task were devoted to the further adjustment of the performance prediction service to the SLA framework requirements and the needs of industrial use cases. The implementation of the service has been refactored to allow for a flexible configuration according to the complexity of the predicted scenario (Section 3.6).

**Task A6.3 - Resource Usage Prediction:** The results of this task have been achieved in the first two project years, and reused in the 3rd project year to support the industrial use case B4 (Enterprise IT).

**Task A6.4 - Reliability Prediction Service:** In the third project year, a reliability prediction service was realized that can be used in an integrated fashion with the performance prediction service (Section 3.6.2). Prediction now also includes reliability metrics as a result (Section 3.2.1). Reliability prediction can be performed analytically based on a Markov modelling formalism, or by simulation. The prediction service was applied to the Open Reference Case (Section 3.8.1) and the industrial use case B6 (eGovernment, Section 3.8.3).

**Task A6.5 - Run-time SLA Violation Prediction:** Efforts in the third year were devoted to a refining of the run-time prediction model and its validation against public domain data (Section 5.3). Efforts were also put in the enhancement of Everest-Plus implementation and integration with the existing Everest monitoring tool.

**Task A6.6 - Manageability Modelling and Generation for (Composite) Business Services:** The main efforts in the project’s third year were to define meta-models for supporting event-correlation during runtime data collection, and for defining various aspects of runtime adjustments the target service may offer. In particular, we now support various types of dynamic bindings, from simple to multiple binding and from concrete to abstract binding, as well as various ways to modify the service’s models.

**Task A6.7 - Prediction Model Adjustment, Integration, and Validation Methodology:** Efforts in the third year were devoted to advancing the statistical inference of software performance models and its integration with the software performance prediction as developed within TA6.2 (see Section 3.5).

## 3 Software Performance and Reliability Prediction

### 3.1 Introduction

In the vision of SLA@SOI, providers of software services can give some guarantees on specific quality of service properties for the services they offer. Such guarantees may concern, for example, the completion time and throughput of a service, or its probability of failure. However, the quality of a service depends on several roles (software providers, service providers, and service customers) involved in the service life-cycle. Therefore, strict contracts, called Service Level Agreements (SLAs), between representatives of these roles are an essential part of SLA@SOI. SLAs determine the quality properties that can be expected from
services provided by different parties. For example, an SLA captures the completion time of a service for a specific customer. SLAs are negotiated before a service is deployed (and may be renegotiated at runtime). For this purpose, service providers need to create general offers for services (SLA templates) and react on individual requests (SLA requests) of potential customers. General offers and individual requests have to be based on sound data to ensure that services will eventually show the quality that has been negotiated. Providers of software services have to know in advance which infrastructure resources and external software services are required in order to fulfil a customer’s request. This knowledge allows them to estimate costs and to acquire necessary resources. Furthermore, customers need detailed information on the quality of a service. This information helps them, for example, to identify the optimal trade-off between costs and offered quality. Customers may want to choose among multiple offers that provide the same functionality but differ with respect to quality and cost.

Our approach to software service performance and reliability prediction supports both the creation of general SLA templates and the reaction to individual SLA requests from customers. In both cases, the service provider needs to know what quality can be expected from the software service that shall be offered, given the underlying service-based architecture and the intended usage profile of the service. To this end, prediction builds upon a model of the service-based system, and evaluates this model in order to determine the expected service quality. Using a model for our approach, rather than the system itself, allows for early quality prediction, without the need for the services being already deployed and operating. Input data for the model may be collected during service design and implementation; some parameters referring to quality aspects of individual system parts may be obtained by estimation, historical data or measurements.

The Palladio Component Model (PCM) [1] serves as a foundation on which we build our solution for SLA@SOI. The PCM supports performance modelling and prediction of component-based architectures. In the context of SLA@SOI, we transfer this approach into the service-based world, and realize “prediction as a service” within the automated service negotiation workflow. In the first project year, we have created a prototype version of the performance prediction service and have demonstrated its applicability to the Open Reference Case (ORC, WP B2). In the second year, we extended and generalized prediction to be applicable to industrial use cases. More concretely, we applied performance prediction to the ERP Hosting (WP B3) and eGovernment (WB B6) industrial use cases. In the third project year, we have once more generalized prediction to consider reliability as a new quality attribute beyond performance. We have extended the prediction service to account for reliability and apply this extended service to the ORC and eGovernment use cases.

The reminder of this section is structured as follows: We give an overview over the most important concepts of our approach (Section 3.2), investigate the State of the Art (Section 3.3), introduce the QoS meta-model used to specify service-based architectures (Section 3.4), and examine possibilities and challenges of prediction models adjustment (Section 3.5). We then present our QoS prediction service (Section 3.6). The following Section 3.7 deals with additional investigations regarding cost prediction and multi-objective optimization. Finally, Section 3.8 shows how our work is applied to B-line use cases.
3.2 Overview

This section provides a high-level overview of our approach to software performance and reliability prediction and the methodology of its application. It describes the prediction goals and scope (Section 3.2.1), some preliminaries that are required for prediction to work correctly (Section 3.2.2), lists involved roles and responsibilities within the service life-cycle (Section 3.2.3) and introduces the prediction workflow (Section 3.2.4).

3.2.1 Goals and Scope

Software performance and reliability prediction allows for a-priori evaluation of the quality of software services before they are actually deployed and executed. Prediction results may be used by software service providers to (i) create general offers for services (SLA templates) at service offering time, and to (ii) react to individual requests (SLA requests) of potential customers at service negotiation time (the reaction may be to accept the offer, to reject it, or to make a counter-offer). Our prediction approach may be applied to evaluate a single software service, or a bundle of software services, if these services share common system resources. For simplicity and consistency, in the following we just refer to the target software service or the target service as the service evaluated through prediction (even though it could be a bundle of services).

Our approach evaluates software services with respect to two dimensions or quality attributes: performance (realized by M26) and reliability (realized in the third project year). Regarding performance, prediction yields the following results (for a detailed mapping of prediction results to standard QoS terms, see Appendix A):

- **Target Service Completion Time**: The time between the arrival of a service invocation request at the service provider’s system boundary and the completion of service execution. This corresponds to the standard QoS term `qos:completion_time` in the SLA model. Prediction provides a full probability distribution over the completion time, rather than only a mean value.

- **Subservice Completion Time**: If the target service is locally (i.e. within the same provider’s domain) composed of subservices, prediction also yields the completion times (i.e. time between invocation and completion) of those subservices as a probability distribution. This result is not directly negotiated in an SLA, but helps the service provider to identify critical parts of the service-based architecture (e.g. performance bottlenecks) and to improve the system configuration in order to avoid such bottlenecks.

- **Target Service Throughput**: The maximal arrival rate of service invocation requests that the system will serve without dropping requests. Corresponds to the standard QoS term `qos:throughput` in the SLA model. When working with a closed system workload specification (see Appendix B), prediction yields the throughput of the target service as a result.

- **Resource Utilization**: Prediction yields a probability distribution over the utilization of virtual or physical infrastructure resources (e.g., CPUs) on which the service-based system is executed. This result is not directly negotiated in an SLA, but gives the service provider information about the required sizing of infrastructure resources in order to achieve a given target service performance.
• **Target Service Reliability:** The probability that a service invocation completes without failure, i.e. the result of service execution is not a message, return value or signal indicating that a failure has occurred. We assume that failures may arise due to software bugs in service components, as well as failures of the execution environment of a service (e.g., application servers, operating systems, underlying hardware resources). Corresponds to the standard QoS term `qos:reliability` in the SLA model.

• **Subservice Reliability:** The probability of failure-free completion of any subservices a target service is locally composed of. Not directly negociated in an SLA, these results still indicate critical parts or components of the service-based architecture of the service provider's domain.

### 3.2.2 Preliminaries

The following discusses essential preconditions of our approach to software performance and reliability prediction, and how prediction results can be interpreted.

**Required Prediction Inputs:** The prediction approach is based on a model of the service-based system architecture, rather than the system itself. The model must conform to the QoS meta-model that we have defined for our approach (see Section 3.4) and constitute a complete instance of this meta-model. Conceptionally, this means that prediction needs a complete view onto the service-based system (or the part of the system that belongs to the software provider's domain) and its envisioned usage. If the target service requires external software services outside the provider's domain, prediction needs to know the offered quality of those services as an input. The same holds for the quality properties of infrastructure services used for software service execution (for a complete list of prediction inputs regarding external services quality, see Appendix A). Only if all required input information is given, prediction can produce a meaningful result. This pre-condition cannot be avoided because prediction results depend on all those inputs.

**Interpretation of Results:** Prediction results (i.e. performance and reliability of the target service) are valid for the system configuration and system usage that have been given as an input to prediction in terms of a QoS meta-model instance. The results cannot be generalized for any change of the system configuration or usage, like another deployment of service components to virtual machines, or the choice of another external software service with different quality properties. A linear change in one factor of the system configuration or usage can affect the resulting service performance and reliability in non-linear ways. Thus, the influence of such changes can only be determined through repeated evaluation of each configuration and/or usage that might be of interest.

**Services with Shared Resources:** If multiple target services share common physical or logical resources within the service-based architecture, they may influence each other's performance and reliability, and thus need to be considered together for the sake of prediction. Restricting the prediction to isolated evaluation of a single service might lead to over-optimistic results, as the concurrent execution of other services and resulting contention effects are neglected.
3.2.3 Roles and Responsibilities

For the creation of a complete prediction model (i.e. a QoS meta-model instance) that can be used as an input for prediction, several roles involved in the service lifecycle contribute parts of the required information. The following lists the roles and their contributions:

**Infrastructure Service Providers** offer infrastructure services, i.e. (virtual) machines, to software service providers through SLA templates. These SLA templates specify the quality (or range of quality) of the offered (virtual) machines, namely the number of processors and/or cores, the clock frequencies, as well as the size of the main memory (through the standard QoS terms `infra:CPU_Cores`, `infra:CPU_Speed`, `infra:Memory`). Combining the given infrastructure offers, a software service provider can create an *infrastructure model* as part of the prediction model, as well as the *allocation model*, which maps software service components to infrastructure (virtual) machines.

**Software Providers** implement service components and offer them to software service providers for use. To enable prediction, software providers also create a high-level specification of their components, interfaces, and behaviour. This specification might already be in terms of the QoS meta-model, or in some other form, and then mapped to the QoS meta-model by the software service provider. The behavioural specification does not reveal the details of the implementation, but describes how each service component uses hardware and software resources and calls other services. The software service provider chooses from the offered service components, and may also compose them to higher-level services. The specification of components and their composition to a service-based architecture constitutes a part of the prediction model, namely the *service component model*.

**Service Customers** request software services through SLA offers. The customer’s request is usually issued at service negotiation time in response to an already existing SLA template from the software service provider. An SLA offer specifies the software service(s) that shall be used, together with the intended workload (specified through the standard QoS terms `gos:arrival_rate` and `gos:data_volume`). The software service provider creates a *usage model* out of this information, which becomes another part of the prediction model.

**Software Service Providers** offer software services to customers, using service components from software providers, (virtual) machines from infrastructure providers, and potentially external software services from other software service providers. They collect all information from the other roles and choose between the possible system configurations as described above, create the different parts of the prediction model (namely service component model, infrastructure model, allocation model, and usage model), perform predictions, and use prediction results to determine feasible parameters for an SLA template or as a reaction to an SLA offer by a customer.

3.2.4 Prediction Workflow

Performance and reliability prediction of software services follows a general workflow, which is performed by the software service provider, and can be categorized into 3 major steps or phases as depicted in Figure 3: model creation, identification of system configurations, and the actual performance and reliability prediction. The three phases are basically sequential, but may interfere with each other; steps forth and back between the phases are possible.
**Figure 3: Abstract Performance and Reliability Prediction Workflow**

**Model Collection:** In this phase, the software service provider collects information from other roles and creates prediction model parts as described in Section 3.2.3. We reuse existing tooling from the Palladio Component Model (PCM) to provide an integrated environment for the software service provider for creation of the prediction model parts, including graphical model editors. The software service provider may map multiple choices regarding service components, infrastructure services, and external software services to multiple variants of the corresponding prediction model parts. Notice that the software service provider may also anticipate parts of the information if it is not yet available. For example, at service offering time, concrete customer requests do not yet exist. In this case, the software service provider anticipates typical service usages based on historical data of similar services, or based on experience. If prediction is performed automatically during service negotiation, it still mainly relies on the model parts manually created by the software service provider. However, service usage information is automatically incorporated into the prediction model on the basis of a concrete SLA offer by a customer.

**Identification of System Configurations:** In this phase, the software service provider determines the possible and feasible system configurations. He decides how to compose the available service components, which infrastructure and external software services to use, and how to allocate service components to infrastructure (virtual) machines. Thereby, what is a feasible solution depends on the available offers from infrastructure and external service providers, but also on other factors that the software service provider might take into account (e.g., legal restrictions or best practices for component deployment). For each system configuration, the software service provider adjusts and combines the existing prediction model parts to create a complete prediction model, including a specification of the intended system usage. This prediction model serves as an
input for the actual performance and reliability prediction. If prediction is performed automatically during service negotiation, it relies on the feasible system configurations that have been determined by the software service provider beforehand. However, the choice between those configurations is automated and performed by the Software SLA Manager component.

**Performance and Reliability Prediction:** In this phase, prediction actually takes place. Prediction results are either used to determine feasible parameters for initial SLA templates, or to determine an appropriate reaction upon concrete SLA offers by customers. In the first case, prediction is carried out manually by the software service provider using the PCM tooling environment. In the second case, prediction is invoked programmatically and carried out by the Service Evaluation component. In both cases, prediction might be carried out repeatedly in order to evaluate several system configurations and enable the selection of the best alternative through the software service provider or the Software SLA Manager component.

### 3.3 State of the Art

This section reviews the State of the Art regarding the approaches to software performance and reliability prediction. During the first year of the project, we evaluated the related work on performance prediction, while in the second year, we investigated related work on reliability prediction. An overview of the approaches that provide performance and reliability prediction is presented respectively in Sections 3.3.1 and 3.3.2.

#### 3.3.1 Architecture-based Performance Prediction

Over the last fifteen years, many approaches have been proposed for integrating performance evaluation and prediction techniques into the software engineering process. Efforts were initiated with Smith’s seminal work pioneered under the name of Software Performance Engineering (SPE) [6]. Since then, a number of meta-models for describing performance-related aspects [7] have been developed by the SPE community, the most prominent being the UML SPT profile [8] and its successor the UML MARTE profile [9], both of which are extensions of UML as the *de facto* standard modelling language for software architectures. Other proposed meta-models include SPE-MM [10], CSM [11, 12] and KLAPER [13]. The common goal of these efforts is to enable the automated transformation of design-oriented software models into analysis-oriented performance models making it possible to predict the system performance. A recent survey of model-based performance prediction techniques was published in [14]. Furthermore, other techniques exploit the combination of different performance models including standard queueing networks [15, 16, 17, 18], extended queueing networks [19, 20, 13], layered queueing networks [22, 23], stochastic Petri nets [24, 25], queueing Petri nets [26, 27], stochastic process algebras [28], Markov chains [21], statistical regression models [29, 30] and general simulation models [31]. In recent years, with the increasing success of component-based software engineering (CBSE), the performance evaluation community has focused on adapting and extending conventional SPE techniques to support component-based systems. Since component-oriented technologies are used as foundation for building modern SOA applications, their performance is essential for managing Quality-Of-Service (QoS) in SOA environments. Techniques for component-based performance prediction [32] are surveyed in detail in the next section.
Performance Prediction Techniques for Component-based Systems

A number of performance prediction methodologies and tools for component-based systems have been proposed [32]. Early attempts towards compositional performance analysis of component-based systems include the works of Sitaraman et al. [33] and Hissam et al. [34, 35]. However, the proposed component specifications and analysis methods are rather simple and do not cover all the information needed for accurate performance prediction. In [36, 37], a more sophisticated approach, called CB-SPE (Component-Based SPE), is presented as a generalization of the conventional SPE method [38, 39, 40]. Annotations based on the UML SPT profile [41] are used to augment component specifications with performance-related properties depending on platform parameters. In [42, 43], the authors propose a language called Component-Based Modeling Language (CBML) based on XML and UML2. However, no explicit context model is defined for capturing variations in input parameters, deployment and configuration.

In [44, 45], a compositional method for performance analysis of component-based systems is proposed, which, however, does not consider stochastic parameter specifications and does not provide a comprehensive component context model taking into account system configuration and deployment aspects. Further advances are the introduction of an explicit context model as part of the component specification [46] and parametric contracts modifying a component’s “provides-and-requires-interfaces” depending on its context [47, 48]. A modelling notation based on extensions to the UML SPT profile [41] is proposed in [49] allowing component developers to explicitly specify the influence of parameters on the component resource demands as well as on their usage of external services. The authors show how their approach can be integrated into the CBSE process model by Cheeseman and Daniels [50] to explicitly include early-cycle model-based performance analysis [51]. In [52], this approach is further extended by introducing constructs for modelling internal parallelism inside a component. The above works were combined in the Palladio Component Model (PCM) [53], a meta-model for specifying performance-relevant information in component-based architectures. PCM is designed with the explicit capability of dividing the model artefacts among the developer roles involved in the CBSE process.

Measurement-based Approaches to Performance Prediction

In [54, 55], it is argued that while the use of performance models in the early stages of system development could help the identification of bottlenecks in the system design, models often fail to capture important execution aspects that can only be determined at run-time. The authors propose a performance analysis method based on early cycle empirical testing. The approach, however, provides limited automation and does not consider integrating empirical measurements with performance models. In [56, 57], the authors use statistical regression techniques to model the relationship between performance-relevant parameters of software components (e.g., use of service calls, input parameters) and their resource demands. The proposed method, however, can only be applied if the adapted components are sufficiently “similar” to existing ones.

In [59, 60], a simple benchmark is used to extract a performance profile of the underlying component-based middleware (e.g., Java EE or .NET) used to build an application. A significant drawback of this approach is that application-specific behaviour is not modelled explicitly and only very rough estimates of the system behaviour can be obtained. In [61, 62], the authors describe a technique to perform automated analysis of system architectures and extraction of
performance models based on traces obtained during operation. A limitation is that components having internal parallelism (supporting forking and joining of the execution flow) are not supported. Furthermore, a number of requirements are placed on the tracing tools which make it difficult to apply the technique in large distributed systems spanning multiple administrative domains.

**Performance Prediction Techniques for Web Services and Service-oriented Architectures**

A number of approaches for introducing QoS support in Web services have been proposed, for example [63, 64, 65]. These studies, however, do not address the issue of how the service provider guarantees its QoS claims. An approach to dynamically select a service provider that best meets the consumer’s needs is presented in [66]. An agent framework coupled with a QoS ontology is used, however, the framework does not support the ability to reserve the resources required for providing a selected QoS, and therefore again no QoS guarantees are provided. In [67], a lightweight extension to WSDL (Web Service Description Language) introducing QoS characteristics was proposed. It can only be used to model services at a very high-level considering each service as a black box. In [68, 69, 70], several methods for dynamic selection of services with the goal to optimize the overall QoS of a composition are proposed.

A different approach to QoS brokering and service selection is presented in [71], where analytic queueing models are used to predict the QoS of alternative services that could be selected under varying workload conditions. In [72, 73], a service discovery system enabling service compositions from semantic descriptions stored in a knowledge base is proposed. An approach to modelling the performance of composite SOA services composed by means of BPEL (Business Process Execution Language) [74] was presented in [75]. Some further approaches based on simulation were proposed in [76, 77, 78]. These approaches, however, only consider static service compositions. Several larger efforts in the Web services arena have focused on describing, advertising and signing up to Web services at defined levels of QoS, for example, HP’s Web Services Management Framework (WSMF), IBM’s Web Service Level Agreement (WSLA) framework, the Web Services Offerings Language (WSOL) and the WS-Policy. These efforts consider QoS in its broader meaning (not limited to performance properties) and specifically target Web service management activities. Performance properties are defined at a very high level and their enforcement at the network and infrastructure layers is not dealt with.

### 3.3.2 Architecture-based Reliability Prediction

In this section, we examine relevant related work for our approach to reliability prediction for software services. We see the approach as an instance of architecture-based software reliability prediction, which is embedded into the broader scope of reliability engineering and prediction. The section is divided into two parts: First, we describe the most important milestones and developments of reliability engineering and prediction. Second, we point out shortcomings of current approaches to architecture-based reliability prediction, which we have mitigated by our work in the third project year.

**History of Reliability Engineering and Prediction**

The need to reason about the reliability (i.e. the ability of failure-free operation) of software systems is widely recognized, and possibilities for reliability modelling and analysis have been investigated for several decades. Early works were mainly
concerned with hardware-related aspects of reliability. First important contributions during the 1950s stem from Shannon [79], Hamming [80], Von Neumann [81], and Moore [82]. The first software-related reliability considerations evolved in the 1960s; examples are the works of Haugk et al. at Bell Laboratories [83], Floyd [84], Hudson [85], London [86] and Sauter [87]. Subsequently, the notion of computer system reliability was coined, acknowledging the fact that both software and hardware effects can cause system failures. Overview articles summarizing the work done until and during the 1970s have been provided by Carter et al. [88], Avizienis [89, 90], Ramamoorthy [91], Hsiao et al. [92], Short [93], Goldberg [94] and Dhillon et al. [95, 96].

A further important step in these early years was the insight that the reliability of a running system depends on the quality of its development process. The term reliability engineering [97, 98] was established to denote the systematic consideration of reliability aspects throughout the hardware and software design and development process. Within reliability engineering, one of the most important tasks is reliability prediction, i.e. the anticipation of the expected system reliability before the system is actually running and its reliability can be empirically measured. Input data for prediction, as well as prediction results, are afflicted with uncertainty. Prediction approaches deal with this uncertainty by employing statistical models and methods. Denson [99] gives a hardware-oriented overview over reliability prediction work done until the end of the 1990s.

Early approaches to reliability prediction were mainly focused on hardware, which was then the dominant reason for system failures [99]. Even if they claimed to include software into the consideration, they used the same modelling and prediction techniques as for hardware. Over the years, with the increase of software-intensive systems, new models were created to accommodate for software-specific failure causes and behaviours. The disciplines of software reliability engineering (SRE) and software reliability prediction emerged as own sub topic within reliability engineering with growing importance. Comprehensive overviews have been given by Lyu [100] and Musa [101]. The field gained maturity, and SRE was applied in major companies including IBM and AT&T [100]. However, the view onto a software system was essentially a monolithic one, without taking the inner structure and architecture of the system into account. Thus, the approaches are limited in their applicability for today’s highly distributed and modularized service-oriented architectures.

Another important area of research focuses on component based systems, and investigates how to compose the reliabilities of each component (component-level reliability) to infer the reliability of the whole system (system-level reliability). The earliest works on this field started from the 1950s [81]. Since then, the problem of finding optimal (i.e. most reliable) system architectures along given degrees of freedom (e.g. selection from a set of functionally equivalent components, or allocation of redundancy levels to components) has been extensively investigated. Overviews have been given by Kuo et al. [102, 103]. However, although very sophisticated optimization algorithms have been developed over time, the problem statements remain rather abstract. The notion of “component” is mostly a very basic one; in many cases, there is not any differentiation between software and hardware components. The employed failure models for components are mainly tailored to hardware components or electronic control units (ECUs) in embedded systems. Thus, the applicability for software-intensive systems and service-oriented architectures remains limited.

The paradigm of component-based software engineering has strongly influenced the view on software-intensive systems from the 1990s until today. The need for reliability prediction methods considering system structure and software
component failure characteristics was widely recognized and led to the discipline of architecture-based software reliability prediction. Approaches in this area explicitly consider the software components within the system, and the control and data flow among those components. Based on individual component reliabilities given as an input, the system-level reliability as perceived by the user is determined. Markov models in different variations (e.g. Discrete-time Markov chains, Continuous-time Markov chains, Markov reward models) are a popular way to represent the system architecture and possible system states, as well as probabilistic transitions between those states. Important contributions stem from Cheung [104], Goseva-Popstojanova [105, 106], Sharma [107, 108], Sato [109], Trivedi [105, 107, 108, 109], Gokhale [110] and Grassi [111].

**Limitations of Current Approaches to Architecture-based Software Reliability Prediction**

Several authors like Goseva-Popstojanova et al. [112], Gokhale [113], and Immonen et al. [114] discussed the current limitations of the approaches on architecture-based software reliability prediction. From our point of view, there are three important limitations of current approaches that we have tackled by our work:

- **Usage profile modeling**: Existing approaches do not explicitly consider the influence of the system usage profile on the system reliability.

- **Consideration of the execution environment**: Many approaches do not take into account the execution environment for reliability prediction.

- **Modeling notation**: Existing approaches use low-level modeling notations (mainly Markov models) which may be difficult to use by software engineers.

In the following, we discuss each issue in greater detail.

**Usage Profile Modelling**: The actual control and data flow happening upon a service invocation may heavily depend on the values of input parameters given to the call. Faulty code may or may not be executed depending on system usage. Thus, from the point of view of the user, the same system may be perceived as highly reliable, or as very unreliable, depending on the way they use it. Existing models do not treat the control and data flow as a variable property, but typically encode it in transition probabilities between states of a Markov model. The system usage (parameter values) is not explicitly represented in the model. Thus, for each change in system usage, the model has to be re-created (see Goseva et al. [105]), and the transition probabilities have to be established again. Different proposals for estimating the transition probabilities exist: Deploying the service components and executing the expected usage profile against them [104], using testing data or the software architect's intuition [115, 116], or just assuming fixed probabilities [119]. All of these proposals tightly bind the model to one assumed usage profile, and some of them even require executing the system. What would be needed is to model the call propagations within components and make the dependencies to input parameter values explicit. Hamlet [120] provides the first part; but the second is still open.

**Consideration of the Execution Environment**: Even if the software part of a service-based system would be totally free of faults, failures could occur due to unavailability of underlying hardware resources and communication failures across network links. These influences are typically not represented in existing approaches to software reliability prediction. Neglecting factors that could lead to
a failure generally means that prediction tends to be over-optimistic. Furthermore, when choosing between multiple system configurations for a service-based architecture, existing approaches cannot distinguish between alternative hardware reliability levels or alternative deployments of service components to hardware. Although some approaches do include properties of the execution environment into software reliability models, they do not have the goal of predicting system reliability [107], do not combine hardware reliability with the software reliability level [120, 121], are limited to network connections [122, 123], neglect software failure probability [109], or do not target component-based software architectures [124, 125, 126]. Thus, an approach is needed that combines hardware-level reliability with software failures to predict overall system reliability for component-/service-based architectures.

**Modelling Notation:** During our survey of related work, we found that Markov models in different flavours (Discrete-time, Continuous-time, Markov-Reward models etc.) are a very popular and dominant means to model a service-based architecture for the purpose of reliability prediction. Many of these approaches require direct manual creation of those Markov models, which may not be feasible for software engineers and architects. Depending on the size of the system and the envisioned accuracy of prediction, the Markov model might easily have thousands of states. Thus, a method is needed that limits the effort and the risk of producing wrong or inconsistent models. Some approaches do have recognized this and typically use a modelling notation that is different from the quality model itself, providing UML-based or UML-like modelling of the software architecture (e.g., [122], [123], [127], [128], [129]). In theory, a tool could automatically transform the user-oriented model into a Markov model, and thus substantially increase the usability of the approach. In practice, however, such an automated transformation may be difficult and has not been included into existing publications. Rather, the way how the transformation should work is only sketched [127, 128], or the transformation is just done manually [123]. Even if a transformation is described, an implementation is not offered [129]. We doubt that the automated transformation is as straightforward as claimed by some authors [128]; especially an UML-based transformation is difficult because of the size and semantic ambiguities of the UML specification. Thus, an implemented automated transformation as provided by our approach is a substantial step forward.

### 3.4 QoS Meta-Model for Performance and Reliability Prediction

In this section, we illustrate the core concepts of the QoS meta-model. This meta-model allows for capturing a service-based software architecture as a special case of a component-based architecture. It allows for specifying performance-relevant and reliability-relevant information in a way such that automated transformations can generate quality models for performance and reliability out of a QoS meta-model instance, and evaluate those quality models.

In the following, we give an overview over the structure and main parts of the meta-model (Section 3.4.1) and discuss meta-model aspects specific for performance (Section 3.4.2) and reliability (Section 3.4.3). For a more detailed description of the involved meta-model classes and constructs, see Appendix B.
3.4.1 Overview

Figure 4 provides a high-level overview over the structure of the QoS meta-model. Thereby, we refer to a prediction model as a QoS meta-model instance, describing a certain service-based architecture.

![Prediction Model (QoS Meta-Model Instance)](image)

**Figure 4: QoS Meta-Model Structure (Overview)**

A prediction models consists of four parts, describing different aspects of the service-based architecture. The contents of each part are generally relevant for performance and reliability. However, some concepts are specifically relevant for one of the two attributes. The four parts are as follows:

- **The Service Component Model** specifies the service components of the architecture, their interfaces, and their hierarchical composition. A high-level behavioural specification captures the performance- and reliability-relevant aspects of the control and data flow. This information stems from the software providers who implement service components and offer them to software service providers.

- **The Infrastructure Model** specifies the execution environment of the architecture in terms of computing nodes (containing hardware resources, e.g. CPUs) and network links. The relevant information comes from infrastructure service providers, who offer their infrastructure to software service providers.

- **The Usage Model** refers to a given service component model and specifies the usage of the system and its services. It indicates how fast users “arrive” at the system, which services they invoke, and what kind of
input data can be expected. This information is either specified directly by
the software service provider (anticipating a certain usage profile), or can
be deduced from a concrete customer request at service negotiation time.

- **The Allocation Model** provides a link between a given service component
  model and infrastructure model. It maps service components to computing
  nodes and thus determines the actual topology of the distributed system.
The allocation is generally decided and specified by the software service
provider.

In order to predict performance or reliability, a complete prediction model is
required containing all four parts as described above. However, the individual
parts may be created independently, and several variants for a part may exist
(e.g. several deployment variants may be specified through multiple allocation
models). The software service provider is free to examine different system
configurations by exchanging individual parts and perform predictions for each
configuration. This way, the provider can identify the most beneficial
configuration without actually deploying and executing all configuration variants.

### 3.4.2 Performance-specific Aspects

This section introduces and discusses some special aspects of the QoS meta-
model that are specifically relevant for and tailored to performance: resource
demands, system workload, and network performance.

**Resource Demands & Resource Performance:** Service execution times
depend on the computational demands or *resource demands* for physical
resources (e.g. CPU’s), as well as logical resources (e.g. semaphores). The time
that it takes to serve a resource demand depends on the size of the demand itself
(determined by the service implementation) and on the computational power or
*resource performance* of the system (e.g. CPU processing speed, semaphore
count). In the QoS meta-model, resource demands can be annotated to internal
actions within a behavioural specification. The annotations are provided as
*stochastic expressions* (i.e. random variables with arbitrary probability
distributions and dependencies to input parameter values). The processing speed
of physical resources can be specified as random variables as well. This kind of
specification considers involved uncertainties and can accurately reflect service
usage dependencies.

**System Workload:** The workload of a service-based system is a part of the
system usage specification. It influences system performance, because a high
workload in a system with limited resources leads to contention effects and thus
increases service completion times. In the QoS meta-model, workloads are either
*open* or *closed*. An open workload models users arriving at the system,
performing one or multiple service invocations, and then leaving the system
again. The inter-arrival time between two users is specified as a random variable
with an arbitrary probability distribution. A closed workload models a fixed
number of users invoking services in an endless loop. Between two loop
iterations, users may pause some time. This “think time” may be specified again
as a random variable with arbitrary distribution. Concluding, the QoS meta-model
provides a highly flexible way to specify different kinds of workload, and allows
for consideration of uncertainty in the workload specification.

**Network Performance:** In a distributed service-based system, network
performance may have a very significant influence on system performance, as
each single service invocation by a system user may involve many message
transports over communication links. The QoS meta-model allows for specification of two performance properties of a communication link: throughput and latency. The throughput specifies how many bits per second may be sent to or received from the link; the latency is the transfer time of a single bit over the link. Both properties may be specified as random variables with arbitrary probability distributions, in order to reflect uncertainties regarding message transfer (e.g. communication protocol delays).

### 3.4.3 Reliability-specific Aspects

This section discusses special aspects of the QoS meta-model that are relevant for reliability: software failures, hardware availability, and network failures.

**Software Failures**: One main reason for a failure during service execution is a faulty service implementation. As the exact location and activation pattern of faults in the implementation is unknown, the QoS meta-model deals with software faults in a probabilistic way. It allows for specification of a failure probability of an internal action, i.e. the probability that an internal action within the service execution flow leads to a failure while it is executed. Taking into account overall system behaviour (the possible execution paths and their probabilities of occurrence) as specified in the corresponding QoS model, the probability of failure due to software faults can be predicted.

**Hardware Availability**: Even if a service implementation would be totally free of faults, service execution might fail due to unavailable hardware resources (e.g. virtual or physical CPU's). The QoS meta-model considers this potential source of failure by annotating hardware resources with their Mean-Time-To-Failure (MTTF) and Mean-Time-To-Repair (MTTR). Both values together allow the calculation of the steady-state availability of the hardware resource as \( A = \frac{MTTF}{MTTF + MTTR} \), which can be interpreted as the probability that a hardware resource is available just at the point in time when it is required by service execution.

**Network Failures**: Beyond service implementation faults and hardware resources being unavailable, lost or corrupted messages sent over communication links may also lead to a failure during service execution. This potential source of failure is also reflected in the QoS meta-model, which allows annotating communication links with a failure probability, i.e. the probability that a message sent over the link during service execution is lost or corrupted and thus leads to a failure. Notice that communication failures may happen in spite of reliable transmission protocols, e.g. because of routing configuration errors, network overload or physical damage of communication links and equipment.

### 3.5 Prediction Models Adjustment

The QoS meta-model, as described in Section 3.4, provides a sound foundation for evaluating performance and reliability of software services. The explicit consideration of all main influencing factors (service behaviour, composition, usage, external service quality, and execution environment) allows for accurate prediction results. This accuracy, however, can only be reached if the created prediction model actually reflects the real system and its quality characteristics. Modelling errors introduced during the manual model creation process may lead to large deviations of predicted versus actual service quality, depending on the part in which the model deviates from the real system.

The uncertainties during model creation can hinder the application of performance engineering in practice. Especially in large enterprise applications, the
performance of a system is affected by a variety of parameters. Often, these parameters are distributed across various layers (infrastructure, virtualization, database, application server, etc.) involving many different technologies. Thus, evaluating such systems is a time and resource consuming process. The approach presented in this section, handles the complexity of large enterprise applications by abstracting those parts of the system that cannot be modelled (or only with high effort). The goal is to capture the dependencies between the system's usage (workload and parameters) and performance (timing behaviour, throughput, and resource utilization). In the following, we propose a combination of measurement-based and model-based performance engineering techniques to evaluate the performance of large enterprise applications.

![Goal-oriented Systematic Measurements](image)

**Figure 5: Goal-oriented Systematic Measurements**

The main idea is to abstract from system internals by applying a combination of systematic goal-oriented measurements, statistical model inference, and model integration. Figure 5 illustrates the major building blocks of the approach.

The technical core of the approach is a framework that allows for systematic performance evaluations, the *Performance Cockpit* [196, 197]. Around that technical core, there are four conceptional blocks: *Experiment Definition, Automated Measurements, Statistical Inference, and Model Integration*. In the following, we describe the building blocks of the approach in detail.

### 3.5.1 Performance Cockpit

Besides the sheer size of today’s enterprise application systems, the complexity and heterogeneity in terms of technology, distribution, and manageability complicates the application of performance evaluations. Since the performance of a system is affected by multiple factors on each layer of the system, performance analysts require detailed knowledge about the system under test. Moreover, they have to deal with a huge number of tools and techniques for benchmarking, monitoring, and data analyses. In practice, performance analysts try to handle this complexity by focusing on certain aspects, tools, or technologies within the system. However, these isolated solutions are inefficient due to the small reuse and knowledge sharing and do not provide reliable performance predictions for the overall system. The goal of the Performance Cockpit [196, 197] is to encapsulate knowledge about performance engineering, the system under test, and statistical analyses in a single application. Therefore, the framework implements best practices and guides the performance analyst in conducting systematic performance evaluations [130]. Moreover, the framework provides a flexible, plug-in based architecture that allows the separation of concerns and
supports the reuse of performance evaluation artefacts. Figure 6 illustrates the idea of the Performance Cockpit.

![Performance Cockpit Idea](image)

Each stakeholder contributes to those parts of the performance evaluation he is an expert in. The basic functionality to control the performance evaluation is provided by the framework. This plug-in based approach enables the Performance Analyst to reuse the adapters implemented by the System, Benchmark, and Tool Experts or the Analysis Experts, respectively. Moreover, the Performance Analyst can reuse adapters in multiple scenarios. Furthermore, if a component in the system under test is changed one can easily switch the plug-ins without changing the actual measurement application. The resulting benefits are (i) less effort for setting up performance tests, (ii) better knowledge transfer, (iii) flexible measurement environment, (iv) better usability, and thus making performance evaluations available to a broader target group.

3.5.2 Experiment Definition

The approach introduced in this chapter has to deal with a huge number of measurements. Moreover, the approach should be applicable for various systems. In order to keep the approach feasible, we have to abstract from the concrete system under test and automate the measurements as far as possible. The Model-Driven Architecture (MDA) [131] is a design approach that allows addressing these challenges. We implement the MDA approach by designing a platform-independent meta-model for the definition of experiments. Experiment includes the system under test, workload, monitoring, analysis, measurement procedures, evaluation goals, etc. The definition of a platform-independent meta-model allows us to provide a single point of configuration to the performance analyst. Based on the meta-model, we can automatically create configurations for different parts of the performance evaluation (e.g. via model-to-model or model-to-text transformations). Figure 7 illustrates the idea.

The generic Experiment Definition Meta-Model allows us to perform multiple evaluations in a consistent and integrated way without having effect on the flexibility of the approach. Garcia, Mora, and others successfully applied such a meta-model for software artefact and process measurements [132, 135, 137]. In our approach, we focus on the configurations necessary to perform measurements of performance metrics. This includes the following points:
1. **Performance Cockpit Configuration**: Information concerning the execution of measurements by the Performance Cockpit, e.g. number of experiment runs, stop criteria for the experiments, notification event receiver, and plug-in selection (load driver, system control, monitoring, analysis, etc.).

2. **System Under Test Configuration**: Information concerning the setup of the system under test, e.g. system parameters, system topology including addresses, and system control information.

3. **Load Driver Configuration**: Information concerning the generation of load on the system under test, e.g. the number of concurrent users, and the variation of parameters.

4. **Monitoring Configuration**: Information concerning the monitoring infrastructure and behaviour, e.g. monitored metrics, sampling intervals, and hold-back time of monitoring data.

5. **Analysis Configuration**: Information concerning the statistical analysis of the monitored data, e.g. analysis technique, assumptions about the expected functions, expected accuracy of the results and desired output format.

![Figure 7: Experiment Definition Meta-Model](image)

### 3.5.3 Automated Measurements

The experiment definition meta-model described in the previous section is an approach to automate configuration and setup of measurement environments. In this section, we focus on the automated execution of measurements. Due to the size of the considered systems and the resulting huge number of necessary measurements, the automated execution is a critical success factor. In order to automate the measurements, we have to link the different areas of performance measurement by an intelligent and efficient algorithm. If setup and configuration of the system under test and the measurement environment are completed, the following steps remain for the actual measurements: determining the actual experiment setup (i.e. how to vary the parameters in each experiment run), running the experiment and measure, and analysing. Typically, these steps are triggered manually. For example, if performance analysts want to evaluate the performance of a middleware component, they generate or adopt a certain load profile (such as provided by the SPEC benchmarks [138]) as the experiment setup and execute it, monitor the relevant metrics and parameters during execution, and finally analyse the monitored data. Often, this process is not only manually triggered but also executed only once due to the effort involved. In our approach, we automate this process as depicted in Figure 8.
The Performance Cockpit generates the experiment setup, automatically deploys the load drivers on the corresponding nodes, and starts the measurements. During the measurements, the Performance Cockpit captures information about the parameters and performance metrics of interest provided by existing monitoring infrastructures. The information is aggregated and saved in the cockpit’s measurement data repository. The Performance Cockpit uses the data to run its statistical analyses in predefined intervals. Depending on the results of the analysis the Performance Cockpit (i) reruns the load profile analyzed in that interval (e.g. because of insufficient monitoring information) or (ii) generates and executes new load profiles (e.g. in order to detect effects not covered by the actual load profile). The presented procedure allows us to implement highly dynamic and efficient algorithms. This is an essential issue towards the feasibility of our approach in large, real-world enterprise applications.

### 3.5.4 Statistical Inference

In the previous section, we described the automated measurement process used in our approach. In the analyses phase of the process, we use statistical inference [139] to capture the dependencies between the system’s usage and performance. The data collected by the monitoring is used to infer (parameters of) a prediction model. In one of our recent work [143], we derived the dependencies between the usage and the performance of a message-oriented middleware using Multivariate Adaptive Regression Splines (MARS) [145] and genetic optimization [146]. While statistical inference does not require specific knowledge on the internal structure of the system under test, it might require assumptions on the kind of functional dependency between independent and dependent variables. The main difference between the multiple inference approaches is their degree of model assumptions. For example, the nearest neighbor estimator makes no assumptions on the model underlying the observations, while a linear regression makes rather strong assumptions (linearity). Most other statistical estimators lie in between both extremes. In general, methods with stronger assumptions need less data to provide reliable estimates, while methods with fewer assumptions need more data, but are also more flexible. In our approach, the concrete technique used depends on the considered problem. For example, the identification of performance relevant parameters requires other techniques than the derivation of the actual impact of a certain parameter on a certain performance metric. Additionally, the chosen technique might differ depending on the system under test, as in some cases we might have good estimators for the underlying model while in other cases the system under test is a complete black-box.

### 3.5.5 Model Integration

In this section, we describe the integration of the statistical models (resulting from the previous step of statistical inference) into the QoS meta-model and explain the Prediction Service’s extensions to solve such hybrid models. We call these statistical models performance curves (German: “Kennlinie”) in analogy to physics where a performance curve describes the dependency between input and output parameters. In the following, we describe the integration requirements
and idea, the extensions of the QoS meta-model, and the adjustments of the simulation engine.

**Integration Requirements**

To support statistical models in the Prediction Service, we have to describe the used performance curve, its features and the usage of the performance curve in a certain context. This information is described in a so-called performance curve specification (PC-SPEC). As performance curves can comprise an arbitrary number of input parameters with different types, the integration concept must be able to provide interfaces with flexible number and types of parameters. Moreover, a mapping from model parameters to performance curve parameters is needed. Generally, performance curves can have different representations, for example as discretized point lists, formulas or a program. Thus, different representation formats have to be supported. Additionally, a PC-SPEC can be used by different services and, hence, should depend on the service’s interface only.

**Integration Idea**

Figure 9 depicts the integration idea. At the concept layer, a model of the service implementation depends on an external service. We assume that the external service is a black-box, for which a performance curve is available (that may be provided as part of its SLA). A performance curve interpreter (PC-Interpreter) concept provides an interface to be linked to the external service dependency in the QoS model. A PC-Interpreter is capable of loading any performance curve, calculate the response time and simulate a delay. The interface provided by the PC-Interpreter is generic, such that a mapping is required for simulation. At the instance layer, the simulation code, generated from the QoS prediction model has to be connected to the interface the PC-Interpreter implementation provides. For this purpose, a bridge (PC-Bridge) links the required interface of the prediction model to the provided interface of the PC-Interpreter. As the usage context of the PC-Interpreter varies from model to model, the PC-Bridge code has to be generated for each use case of a performance curve. To achieve this, information about the used performance curve and the usage context is needed. This information is provided by a PC-QoSAnnotation which contains the PC-SPEC (described in the next section) of the external service. For generating the code of the PC-Bridge, a model to text transformation is applied on the PC-SPEC model.

![Figure 9: Integration Concept](image-url)
**PC-SPEC Meta-Model**

So far, we have considered the performance curve specification (PC-SPEC) as an abstract unit, describing the features and the kind of usage of a performance curve. In this section, we substantiate the term PC-SPEC (or Performance-CurveSpecification), by introducing a meta-model for PC-SPEC.

![Figure 10: Integration Meta-Model](image)

Figure 10 illustrates the meta-model for PC-SPECs. Essentially, a performance curve specification comprises four parts: a `SignatureReference` referring to the model interface and signature, a `PerformanceCurve` representing the performance curve, a set of input parameters (`AbstractInputParameter`) representing independent parameters of the performance curve and target curve dimension for the response time. `PerformanceCurve` is an abstract type having two concrete sub-types: `DataTable` and `Formula`. A `DataTable` represents a performance curve by a discretized list of points, residing as a CSV-file. The attribute `strategy` specifies an interpolation strategy used for interpolation over the discretized points list. Independent from the concrete type, each `PerformanceCurve` comprises a set of `Dimensions`. Each `Dimension` has a name and a type, whereas the dimension type can be one of the types enumerated by `PrimitiveTypeEnum` form the QoS meta-model. Regarding the input parameters, there are three types: While `ConstantParameter` allows for configuring the PC-Interpreter, an `InputParameterDelegator` provides a simple mapping between a QoS model parameter and a curve dimension. `TrackedParameters` are used when aggregation of the value of a QoS model parameter over the whole queue of the PC-Interpreter is needed. `TrackedParameters` have to be computed by the PC-Interpreter, for example the PC-Interpreter has to track the `QueueLength` as it is a parameter whose value is not known outside the PC-Interpreter. Finally, a `PerformanceCurveSpecification` has to specify a `QueueingStrategy` which determines whether the requests to the PC-Interpreter are processed sequentially (`INITIAL`) or concurrently (`EFFECTIVE`).

**Simulation Using PC-Interpreter**

The simulation logic is another essential part of the integration concept. After the PC-Bridge is generated by interpretation of the PC-SPEC model, all artifacts needed for simulation of the overall scenario are available: the QoS prediction model describing the service and its behaviour, the PC-Interpreter implementation used for describing the external system through a performance curve and the PC-Bridge configuring the PC-Interpreter for the specific context.
The interesting aspect of the simulation execution for the performance curve integration is, when the simulation reaches external service calls. At this point, the PC-Bridge calls the PC-Interpreter, which simulates the behaviour of the external service using the specified performance curve.

![Pseudo Code](image)

**Figure 11: Simulation Behaviour in Pseudo Code**

Figure 11 depicts the pseudo code of the simulation. For each request, the PC-Interpreter creates a new event and saves the input parameters and the process of the new event. As a new request influences the response time of all other requests, the PC-Interpreter updates the input parameters for all active events and calculates a new response time for each active event. Then the PC-Interpreter schedules the chronologically next event and passivates the process of the new event. As soon as an event is executed the corresponding process is activated and the thread leaves the simulate method. The response time of an external call is computed by measuring the simulated time of the call, when the corresponding simulated process is passivated.

In the following section, we describe the structure of the prediction service in more detail.

### 3.6 The Prediction Service

This section describes the current tooling that we provide for performance and reliability prediction. In the following, we first present a high-level overview over the available tooling (Section 3.6.1). Then, we describe how the actual prediction engine works (Section 3.6.2). Section 3.6.3 presents the prediction process for the case that prediction is invoked as part of the SLA management framework during automated negotiation, and Section 3.6.4 discusses how prediction supports scenarios with varying complexity. A specification of the corresponding Service Evaluation component, as well as the Evaluate interaction, is given in Appendix C.

#### 3.6.1 Overview

Figure 12 shows the architecture of the available prediction tools and their integration into the SLA management framework. We have chosen this architecture in a way such that (i) the existing tooling of the Palladio Component Model (PCM) could be reused, and (ii) both envisioned scenarios of applying prediction (manually during service offering / automated during service negotiation) are supported.
In the first scenario, a software service provider uses prediction to determine feasible quality parameters for the software services to be offered. Prediction results are used for the creation of the corresponding software SLA templates. The software service provider uses an integrated environment for the graphical creation of prediction models, the actual prediction, and the graphical visualisation of prediction results (Figure 5, right-hand side). This prediction environment is realized in terms of Eclipse plug-ins running on an OSGi platform, based upon the existing PCM tooling. It is a self-contained tool offering everything the software service provider needs for performing the prediction.

In the second scenario, prediction is performed automatically as part of the SLA negotiation workflow conducted by the SLA management framework. To this end, we have extended the prediction engine of the integrated environment with a web service interface. Thus, the environment becomes a prediction server application, and prediction can be triggered programmatically. Within the SLA management framework, prediction is offered as an implementation of the Service Evaluation component for the special case of predictive software service evaluation (P-SSE). It is invoked through the Software Planning and Optimization sub component (S-POC) of the Software SLA Manager component, in order to determine a proper reaction to a concrete SLA offer coming from a potential customer. P-SSE invokes the predictive engine with a prediction model as an input and retrieves prediction results back as an output.

Both scenarios use the same prediction engine; consistent results independent from the phase of application (service offering or service negotiation) are thus ensured.

### 3.6.2 Prediction Engine

The prediction engine as shown in Figure 12 provides the core functionality to evaluate prediction models regarding their performance and reliability. While the performance evaluation is based on the existing PCM tooling, the capabilities for reliability evaluation have been added through the work in SLA@SOI. Here, a very short overview of the way the prediction engine works is given. For more details see [1, 199].
The prediction engine takes a full QoS meta-model instance as an input, including a service component model, an infrastructure model, an allocation model, and a usage model. Using the openArchitectureWare (OAW) framework, an automated transformation is applied on the prediction model, resulting in an extended queueing network model, which is represented as a Java implementation. This queueing network is then simulated, using the discrete-event Java simulation framework SSJ. The simulation is used to evaluate both the performance and the reliability of the services under study. From the system workload specification, the transformation generates a workload driver, which upon execution spawns threads to simulate arriving users that invoke system services. The control and data flow throughout the service component architecture is executed as specified in the prediction model.

The simulation considers performance-relevant aspects of the execution, such as the consumption of resources and the latency of message transports over network links. Contention effects caused by concurrent service execution and resulting waiting times are observed. Whenever a probabilistic decision has to be made (e.g. to determine the arrival time of the next user, the size of a resource demand, or which branch in the control flow to take), a sample is drawn from the specified probability distribution, and the decision is based on the sample. Hence, it is ensured that the simulation follows the distributions that have been specified in the prediction model.

To simulate a software failure, an exception may be raised during the execution of a control flow action. A random number is generated according to the given failure probability, which decides about success or failure of the action. Communication link failures are handled in the same way. Furthermore, the simulation includes the notion of hardware resources and their failure behaviour. It uses the given MTTF/MTTR values as mean values of an exponential distribution and draws samples from the distribution to determine actual resource failure and repair times. Whenever an internal action requires a currently unavailable hardware resource, it fails with an exception. Taking all possible sources of failure into account, the simulation determines system reliability as the ratio of successful service executions to the overall execution count.

Throughout the simulated Java code, sensors are placed that record the simulated start and end times of each service invocation, the history of resource demands and waiting times for resources, and the execution results (success or failure) of service execution. After the simulation, these data are available for visualization (e.g. time series diagrams or histograms) or further aggregation (e.g. determining the 90% percentile of service completion time). The integrated prediction environment provides capabilities for such visualization of results, allowing the software service provider to derive the relevant information about feasible quality parameters, or to make a sophisticated choice between multiple system configurations.

As an additional evaluation method in cases where only service reliability is of interest, the prediction engine provides an analysis method that does not use simulation but mathematical calculations. The method provides an alternative transformation from the prediction model to a discrete-time Markov chain (DTMC). The Markov chain represents all possible execution paths through the service-based architecture, together with their occurrence probabilities. An initial state marks the beginning of the execution; a set of result states marks either a successful service completion or the occurrence of a failure-on-demand. Applying results from Markov theory, the analysis determines the success and failure probabilities of the service execution and thus the service reliability. Compared to the simulation, the analysis is restricted to reliability evaluation only, but it is
significantly faster and does not include any numerical errors due to the sampling of probability distributions.

### 3.6.3 Prediction Process

![Diagram](image)

**Figure 13: Evaluation of Software Services**

Figure 13 illustrates the prediction process that is executed when the Predicted Software Service Evaluation (P-SSE) component is invoked via the Evaluate interaction during service negotiation. The invocation comes from the Software Planning and Optimization (S-POC) component, which in turn has been triggered upon a customer SLA offer with the goal to establish an SLA accordingly, to reject the offer, or to create a counter-offer. P-SSE helps S-POC in this decision-making by evaluating the performance and reliability for individual system configurations and usage profiles. S-POC can compare the predicted quality of a given system configuration with the customer request and decide if the configuration satisfies all given requirements, and if the SLA can thus be established based on this configuration.

The actors shown in Figure 13 are a running instance of the S-POC, as well as several entities belonging to a P-SSE instance. The sequence diagram shows an invocation of P-SSE by S-POC, with the following steps:

1. S-POC issues an `evaluate()` request to P-SSE, containing information about the system configuration(s) to evaluate, as well as service usage and external services quality parameters. The request is received by the PredictiveSoftwareServiceEvaluator.
2. The PredictiveSoftwareServiceEvaluator checks the input for being valid, i.e. being consistent and complete. If the input turns out to be invalid, an `IllegalArgument`Exception is thrown.
3. A given ServiceRealization (i.e. system configuration) is forwarded to the PredictionScenarioGenerator, which retrieves the corresponding QoSModelInstance (the prediction model) from a DesignTimeRepository, creates a PredictionScenario (i.e. a complete and adjusted prediction model, considering also service usage and external services quality parameters), and returns this scenario to the PredictiveSoftware-ServiceEvaluator.

4. The generated PredictionScenario is given as an input via a web service interface to the SoftwareQualityPredictor, which is part of the prediction server (also see Figure 5). The SoftwareQualityPredictor performs a simulation in order to evaluate the expected performance of the target service. The evaluated quality parameters are returned to the Predictive-SoftwareServiceEvaluator. If the PredictionScenario cannot successfully be evaluated because of an unexpected error, an EvaluationException is generated and returned to the S-POC as the caller of the evaluate() operation.

5. The PredictiveSoftwareServiceEvaluator checks the evaluation result for being valid. This includes adherence to usage bounds of required software services, as well as indication that the system was not overloaded by the envisioned target service usage.

6. If the result is valid, the PredictiveSoftwareServiceEvaluator creates an EvaluationResult instance and stores the results for the current PredictionScenario there.

7. Steps 3 to 6 are repeated until all given ServiceRealizations have been evaluated.

8. The list of evaluated results is returned to S-POC as the caller of the evaluate() operation.

### 3.6.4 Supported Scenarios

The approach to software performance and reliability prediction presented in this chapter puts no strict limit on the complexity of the supported application scenarios; however, its practical applicability depends on their complexity. Thereby, the term “complexity” refers to the structure of a local service hierarchy (e.g. number of involved service components), but even more to the number of external software and infrastructure service dependencies, as well as the number of different available offerers for each dependency. These aspects influence the effort for the creation of prediction models, as well as the time that it takes to evaluate all possible service realizations.

As a lesson learned during the evolution of the approach, flexible solutions have to be offered to account for the varying complexity of the targeted application scenarios. While in the eGovernment use case, a prediction model of manageable size could be created in a straightforward manner to reflect a health care and mobility booking process (see Section 3.8.3), modelling the whole SAP-based ERP Hosting use case including all legacy components would have required unfeasibly high effort. Three main measures taken to improve the applicability of the prediction approach are (i) the complementation of model-based prediction with a measurements-based prediction approach, (ii) the configurability of the simulation depth during the prediction, and (iii) the decoupling of the prediction process from the concrete point in time of the SLA negotiation.

The measurements-based prediction approach (as introduced in Section 3.5) is applicable in situations where the service architecture is very complex and includes potentially big legacy parts. In such cases, in may be infeasible to create
a prediction model of the whole architecture because of the high efforts associated with the determination of all relevant input parameters. Instead, systematic measurements of individual system parts or the system as a whole are conducted, to observe service quality for certain service configurations and usage profiles. The resulting data allow for estimating the quality of further configurations and profiles through the use of statistical inference techniques. This method has successfully been applied to the ERP Hosting use case (see Section 3.8.2).

To reduce the duration of the model-based prediction when invoked during the SLA negotiation, we have added further configuration parameters to the «evaluate» interaction. These parameters specify stop criteria for the simulation, and they refer either to the number of produced simulation measurements or to the simulated execution time. The simulation ends as soon as a stop criterion is reached and returns the results that have been collected up to this point. If the simulation stops early, the number of measurements is lower, and there is less confidence in the results. On the other hand, if the results very clearly suggest that a certain service configuration is not satisfactory, it can be rejected even if the confidence is low. The user of the «evaluate» interaction can determine independently for each invocation how intensive the simulation shall be.

Further flexibility is provided by the fact that there is no strict coupling between the points in time when a prediction is performed and when an SLA is negotiated. SLA managers can use idle times during the run-time of the SLA management framework to predict the quality of certain scenarios (i.e. requested services with certain configurations and a certain usage profile). The selection of scenarios to predict can be based on experience about which scenarios are generally most probable or most likely to occur in the near future. The SLA managers can store the prediction results and use them for future SLA requests where applicable. A corresponding strategy has been implemented by the Software Planning and Optimization (S-POC) sub component of the software SLA manager.

Together, these measures provide the required flexibility to cope with the complexity-related challenges of different application scenarios. As illustrated in Section 3.8, service quality prediction has successfully been applied to the relevant use cases, namely the Open Reference Case (ORC), the ERP Hosting use case and the eGovernment use case.

3.7 Complementary Prediction Services

This section describes further results that have been achieved as part of the design-time prediction efforts of WP A6. Although these results are not directly included in the SLA management framework, they complement the existing methodology of service performance and reliability prediction. More concretely, Section 3.7.1 describes an approach to cost prediction of services, and Section 3.7.2 tackles the challenge of optimizing a service configuration with respect to multiple conflicting objectives, such as reliability and performance.

3.7.1 Cost Prediction Service

In this section, we propose an analytical cost model that jointly accounts for fixed hardware costs and dynamic operational costs related to power consumption. This model is cross cutting with respect to the software and infrastructure layer. However, it proved to be useful in the assessment of enterprise applications such as the Sales and Distribution application described in deliverable D.B3.a.
For the service providers to specify SLAs and optimize their service/infrastructure landscapes, it is important to analyze, understand, and model the cost components within the so-called "Total Cost of Ownership" (TCO). TCO is intrinsically complex and involves a great number of tangible/ intangible factors. As is pointed out in [133], the TCO of a large-scale hosting center can be broken down into four main components: hardware, power (recurring and initial datacenter investment), recurring datacenter operations costs, and cost of the software. Normally the operations costs (incl. human capital/consulting) and software constitute a large percentage of TCO for commercial deployment, however, it is very difficult to develop a generic quantitative cost model for these components. In this section, we focus on more tangible cost factors such as server hardware, and we incorporate power consumption into the cost model as a server’s energy footprint becomes an increasingly important cost factor in large-scale hosting environments.

Not aiming at a comprehensive TCO model, we focus on the quantitative aspects and develop an analytic cost model that consists of two tangible cost components: server hardware and power consumption. Firstly, a pricing model for CPU is proposed as a function of per-core performance and the number of cores. The per-core performance is based on the published results of industry-standard OLTP (online transaction processing) benchmark TPC-C [149] on Intel DP/MP platforms. The fitted CPU pricing model also manifests the current multi-/many-core trend. Secondly, server power consumption is modelled as a function of CPU utilization using a customized power function. By combining the fitted models for both server costs and power consumption, we develop a simplified analytic model that can be used in the studies of optimizing the enterprise system landscape with multiple objectives.

**Modelling CPU Costs with Multi-Core**

Among the many components of server hardware, namely CPU, memory, storage, and network, we focus on the CPU costs in this paper and make simplified assumptions that costs of other components remain constants or scale with the CPU costs. We are particularly interested in the price performance relationship on multi-/many-core platforms, as the general trend in processor development has been from single-, multi-, to many cores. Our goal is to investigate and model the relationship between the objective, namely the price per-CPU (Ccpu) or price per-core (Ccore), and the two related parameters: number of cores (Ncore) and benchmark results per-core (Tcore). Tcore also corresponds to the processing speed of the core, and thus the resource demands of the measured OLTP applications.

![Figure 14: TPC-C Benchmark Results](image)

We examine the certified TPC-C [149] benchmark results on Intel DP/MP platforms and associate them with CPU price information [144], which are shown in Figure 14. As there are two independent parameters (Ncore and Tcore) we study one of them by fixing the value of the other, and vice versa.
Price, Performance, and Number of Cores

Firstly let us look at the price versus the number of cores given a similar per-core performance. In Figure 14(a), we can see that the per-core price decreases as the number of cores per CPU increases on the Intel Xeon DP platform. As the per-core performance of TPC-C remains the same, the price/performance ratio improves by adding more cores. Generally this trend is also observed for TPC-C on Intel MP, as is shown in Figure 14(b). We notice that the per-core tpmC decreases slightly as the number of cores increases. This is because that the core frequency scales down as the number of cores scales up. Nevertheless, as the chip design becomes more efficient, the per-core performance/frequency ratio improves with the evolution of CPU generations.

**Figure 15: Power Function Fitting**

**Figure 16: Normalized Power vs. CPU Utilization.**

Secondly let us examine the price versus the per-core performance given the same number of cores. In Figure 14(c), as predicted, we can see that the price increases as the CPU frequency and throughput numbers increase. Some abnormal behaviour happens between 2.33 GHz and 2.83 GHz. This may be explained partially by the noise in the data as there is only one available measurement each for CPU frequency at 2.33 GHz and 2.83 GHz. Nevertheless, the general trend of price increasing with speed (core frequency) still holds.

Figure 15 gives a better view on the pattern of how price changes with the per-core performance for TPC-C. On both DP and MP platforms with different cores, the per-core price scales with the per-core throughput like a power function. We studied different functions for curve fitting, including polynomial, exponential, power, and other custom functions. It is found that the power function, shown in (1), gives the overall best fit for different data sets.

\[ f(x) = c_1 x^{c_2} + c_3 \]  

(1)

It is also shown that the price per-core decreases like a power function while increasing the number of cores per-CPU. This indicates that the power function in (1) can be used to model the relationships between price per-core and Tcore or Ncore individually.

**A CPU Price Model**

The next step is to study per-core performance (Tcore) and number of cores (Ncore) jointly and model their relationship with price. Since the power function is the best fitted model for Tcore and Ncore individually, we can extend this model to a multi-variable case. A power function with two variables can be formulated as follows:
where \((c_1, \ldots, c_5)\) are the parameters to be fitted. The price per-CPU \(C_{\text{cpu}}\) is readily obtained by multiplying price per core with the number of cores:

\[
C_{\text{cpu}} = N_{\text{core}}C_{\text{core}} = N_{\text{core}}g(T_{\text{core}}, N_{\text{core}}).
\]

<table>
<thead>
<tr>
<th>model param.</th>
<th>(c_1)</th>
<th>(c_2)</th>
<th>(c_3)</th>
<th>(c_4)</th>
<th>(c_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCC/DP</td>
<td>36</td>
<td>2.0</td>
<td>261</td>
<td>-0.9</td>
<td>-105</td>
</tr>
</tbody>
</table>

**Table 2: CPU Cost Model Parameters**

A non-linear least-squares method in the Matlab Optimization toolbox (lsqcurvefit) is used for curve fitting, and the fitted parameters are shown in Table 2. The fitted model gives an overall good interpolation of real benchmark results. Although different benchmarks on different platforms may yield different parameters, the model shown in (3) is general and flexible enough for estimating a wide range of CPU cost information.

It should be noted that the power-function based model for CPU costs developed in this section depends on the Intel pricing schemes for its multi-/many-core platforms. Our contribution is to fit such price information with mathematical models, in relationship to real OLTP benchmark results. This gives the planners/architects at the provider side a convenient tool for estimating hardware costs given the desired performance level of their applications.

**Modelling Power Consumption**

Power consumption and associated costs become increasingly significant in modern datacentre environments [142]. In this section we analyse and model the server power consumption of business applications. We study the relationship between system power consumption (\(P_{\text{sys}}\), measured in Watts) and CPU utilization (\(U\)), which is used as the main metric for system-level activity. We run a customized application similar to sales and distribution business processes on a 64-bit Linux server with 1 Intel dual-core CPU and 4 GB main memory. The system power is measured using a power meter connected between the server power plug and the wall socket. The CPU utilization data is collected using Linux utilities such as sar and iostat. Monitoring scripts in SAP performance tools are also used for correlating power and CPU utilization data. Before data fitting and modelling, we first perform a data pre-processing step called normalization. Instead of directly modelling \(P_{\text{sys}}\) we use a normalized power unit \(P_{\text{norm}}\), which is defined as follows:

\[
P_{\text{norm}} = \frac{P_{\text{sys}} - P_{\text{idle}}}{P_{\text{busy}} - P_{\text{idle}}},
\]

where the measured \(P_{\text{idle}}\) (\(U = 0\)) and \(P_{\text{busy}}\) (\(U = 1\)) for our test system are 42W and 84W, respectively. Different systems may have different idle and peak power consumptions. The normalized measurement results are shown in Figure 16.

Generally speaking the server power consumption increases as the CPU utilization grows. One important finding from the measurement data is the so-called power capping behaviour [142], which means there are only a few times that the highest power consumption is reached by the server. Additionally we find that such highest power points are drawn mostly when the CPU utilization is higher than 80% and they have very similar peak values. Most of the functions, such as
quadratic polynomial, power, exponential, and Gaussian, cannot fit such flat curve of power values in the high-utilization interval (see the quadratic fitting in Figure 16).

We developed a model that can fit such power-capping behaviour well. The model is inspired by the frequency response curve of a linear filter called Butterworth filter [150]. It has such desired "flat" behaviour in the passband of the frequency. We replace the polynomial part of the transfer function with the following customized power function with two U components:

\[ h(U) = c_1 U^{c_2} + c_3 U^{c_4} + c_5, \]  

(5)

where (c1, ..., c5) are the parameters to be fitted. The model that relates normalized power (Pnorm) and CPU utilization U can be formulated as follows:

\[ P_{\text{norm}}(U) = 1 - h(U)^{-1}. \]  

(6)

The fitting result is shown in Figure 16 and the fitted model parameters are listed in Table 3. We can see that the proposed power model fits the measurement data well, especially during the high utilization period. Given the measurements for Pidle and Pbussy, the overall system power consumption Psys can be obtained by substituting Pnorm (6) in (4).

<table>
<thead>
<tr>
<th>model param.</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
</tr>
</thead>
<tbody>
<tr>
<td>business app.</td>
<td>276.7</td>
<td>15</td>
<td>7</td>
<td>2.1</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 3: Power Consumption Model Parameters

**A Cost Model for Enterprise Applications**

By combining the cost models for CPU and power consumption in previous section (equations (3), (4), and (6)), we developed a cost model for business applications:

\[ \text{Cost}(T_{\text{core}}, N_{\text{core}}, U, I) = p_0 + p_1 C_{\text{cpu}} + p_2 \int_{t \in I} P_{\text{sys}}(U(t)) dt, \]  

(7)

where t is the measurement time, I is the measurement period (t \in I), p0 is an adjusting constant, p1, and p2 are the weighting parameters that scale the individual model outputs. If during the measurement period only average utilization is available, the output can be written as Psys(U)|I. The model in (7) uses an additive form to combine server hardware costs and operational costs, in which parameters p1 and p2 have to be set properly to reflect different cost structures.

To summarize from a mathematical modelling perspective, we can conclude that the power function (c1 x c2 + c3) and its variants have attractive properties for fitting a wide range of curves, including both single- and multi-variable case. Thus, the power function family represents a general and flexible modelling library from which different cost models can be fitted and derived.

In practice when using the cost model for the optimization of enterprise systems, we need to determine the weighting parameters p1 (fixed cost) and p2 (operational cost). These parameters are chosen in a way to reflect the real numbers obtained in case studies in [133]. There are two situations under study
in this section. On one hand, for a typical “classical” data centre the ratio of fixed cost versus operational cost (r) is set to 7:3, which indicates that the high server capital costs dominate overall TCO by 70%. For a modern commodity-based data centre, on the other hand, the ratio r is set to 3:7. This means operational costs including power consumption and cooling become the dominating factor.

### 3.7.2 Multi-objective Optimization

As a further extension of our prediction service, we also propose to use multi-objective optimization to find the Pareto-optimal solutions that describe the best trade-off solutions between conflicting performance and cost-saving goals. Experimental validation demonstrates the accuracy of the proposed models and shows that the attained Pareto-optimal solutions can be efficiently used by service providers for SLA-driven planning decisions, thus making a strong case in favour of the applicability of our methodology for deployment decisions subject to different SLA requirements.

We adopt a multi-objective approach towards SLA-driven planning of enterprise applications. A framework is introduced for formulating the problem with multiple objectives and describing the design paradigm. What lies in the core of the framework is a multi-objective optimizer, and we apply a state-of-the-art evolutionary multi-objective optimization (MOO) algorithm. We show how the performance and cost models can be used in an optimization process of the planning phase.

**An SLA-driven Planning Framework**

Firstly we present a framework for SLA-driven planning and optimization, which is shown in Figure 17. The system planner interacts with the planning tool via a dashboard-based User Interface (UI). The planner starts with defining the objectives, namely, system end-to-end response time and infrastructure cost. In this case the problem is formulated as a minimization problem: minimizing both response time and cost. The planner then follows several main steps in the planning phase:

1. Define default constraints or extract them from the customer SLAs. Such constraints are considered as fixed constants in the optimization process, and they are mostly related to the user workloads. For a closed queueing network model used in this paper, the constraints of interest are number of users and think time.
2. Define parameters to be optimized. In the context of this paper most of the parameters are configuration parameters in the enterprise system landscape. These include hardware resource specifications, namely, Resource Demand (D) and number of cores K. It also includes application server configurations such as W, number of WPs (dialog work processes).
3. Formulate the problem for optimization. The performance and cost models developed in previous sections can take configuration parameters as inputs and generate/predict performance and cost outputs. The utility functions scale the model outputs as utilities for a unified representation of objective values. The decoder, on the contrary, maps the encoded parameters into model-specific formats.
4. Run the optimization and interpret the results. With the set of “optimal” trade-off solutions obtained via optimization, the planner can make educated decisions for planning the system landscape according to different levels of SLAs.
Figure 17: SLA-driven Planning Framework Overview

The central component of the framework is an evolutionary MOO algorithm called SMS-EMOA, which will be elaborated in the next section. Here we give more explanations on utility and encoding/decoding functions. Firstly, for scaling the diverse objective values into unified utilities (e.g. $[0, 1]$), we adopt Derringer's individual desirability function [141]. In case of a minimization problem, to which our problem belongs, the desirability value is increasing along with the value of the objective, bounded by a maximum value. For the sake of simplicity linear scaling is used in practice. Secondly, like other evolutionary algorithms the configuration parameters is encoded in the individuals as continuous double values. The number of WPs is discretized by rounding up to the closest small integer. The number of cores is encoded as a double variable $x \in (0, 3)$, and is decoded by $2 \text{floor}(x)$ (1, 2, or 4 cores).

A Multi-Objective Optimizer

In the previous section, the MOO algorithm is treated as a black-box: iteratively evaluate the objective values, generate new parameters, and hopefully after some generations (sub)optimal solutions could be found. In this section, we explain the rationale behind a true multi-objective optimization and describe how a state-of-the-art evolutionary MOO algorithm works.

Multi-objective optimization (MOO) is the process of simultaneously optimizing two or more objectives. Most problems in nature have several, possibly conflicting, objectives. In the context of this section, for instance, we are aiming at maximizing the system performance at the same time minimizing the infrastructure cost. On one hand, common ways of dealing with MOO problems include treating them as single-objective by turning all but one objective into constraints, or combining multiple objectives into one. A MOO algorithm, on the
other hand, tries to find good compromises (or trade-offs) rather than a single global optimum. Therefore, the notion of “optimum” in multi-objective optimization changes accordingly and the most commonly accepted term is called Pareto optimum [140].

The concept of Pareto optimum and Pareto front are explained as follows. Given a parameter vector \( X \in \mathbb{R}^n \), an evaluation function \( f : X \rightarrow Y \) evaluates the quality of the solution by mapping the parameter vector to an objective vector \( Y \in \mathbb{R}^m \). The comparison of two parameter vectors \( x \) and \( x' \) follows the well-known concept of Pareto dominance. We say that an objective vector \( y \) dominates \( y' \) (in symbols \( y \prec y' \)), if and only if \( \forall i \in \{1, \ldots, m\} : y_i \leq y'_i \) and \( y \neq y' \). The set of non-dominated solutions of a set \( Y \subseteq \mathbb{R}^m \) is defined as: \( YN = \{ y \in Y | \not\exists y' \in Y : y' \prec y \} \). Given a multi-objective optimization (minimization) problem \( f_1(x) \rightarrow \min, \ldots, f_m(x) \rightarrow \min, x \in X \subseteq \mathbb{R}^n \), (8) the image set \( Y(S) \) of this problem is defined as \( \{ y \in \mathbb{R}^m | \exists x \in X : f_1(x) = y_1, \ldots, f_m(x) = y_m \} \). The non-dominated set of \( Y(X) \) is called Pareto front. In other words, the Pareto front consists of a set of optimal solutions representing different trade-offs among the objectives. The knowledge of Pareto front helps the decision maker in selecting the best compromise solutions.

In order to approximate a continuous Pareto front that typically consists of infinitely many points, we can compute an approximation set that covers the Pareto front. In general, an approximation set is defined as a set of mutually non-dominated solutions in \( Y(X) \). A common indicator for the quality of an approximation set, measuring how well it serves as a well-distributed and close approximation of the Pareto front, is the hypervolume indicator (or: S-Metric) [136]. The problem of finding a well distributed approximation of the Pareto front can be recasted as the problem of finding an approximation set that maximizes the S-Metric. Evolutionary algorithms possess several characteristics that are naturally desirable as the search strategies for multi-objective optimization [140]. Among other indicator-based MOO algorithms, the S-Metric Selection Evolutionary Multi-objective Optimization Algorithm (SMS-EMOA) approximates such S-Metric maximal approximation sets. The SMS-EMOA algorithm implements a steady-state (\( \mu + 1 \)) evolutionary strategy: keep a population of \( \mu \) individuals, remove one “bad” individual and add a new one in each generation. SMS-EMOA can also be parallelized by distributing function evaluations to different processors. We follow the algorithmic details for the hypervolume computation and variation operators as described in [136], and integrated both sequential and parallel SMS-EMOA implementation in SLA-driven planning.

### 3.8 Use Cases for Performance and Reliability Prediction

This section demonstrates the application of our prediction approach to the Open Reference Case (Section 3.8.1), as well as the ERP Hosting and eGovernment industrial use cases (Sections 3.8.2 and 3.8.3). In addition to the performance prediction which was available and could be applied to use cases in the first two project years, the updated section also includes reliability prediction as realized in the third project year.

#### 3.8.1 WP B2: Open Reference Case

In the first project year, we have demonstrated the capabilities of software performance prediction within the context of SLA@SOI through the Open Reference Case (ORC), which is developed in WP B2, as a QoS meta-model
instance, and have predicted its performance in order to derive a-priori knowledge of the completion times and throughputs of its services. In the third year, we have revised and enhanced the prediction model to also take into account reliability aspects, and performed predictions of both quality attributes. This section describes the ORC scenario briefly (for details, refer to the WP B2 deliverable of M12), and presents the created prediction model along its individual parts – service components, infrastructure, allocation, and system usage.

The Sales Process

The ORC scenario focuses on the sales process at individual cash desks of a supermarket. Several services are invoked during each sales process. For each item captured by the scanner, product information has to be retrieved from the inventory database operated by the service provider. If the customer at the cash desk decides to pay with credit card, the card has to be validated and debited according to the summarized sales value. This requires an additional bank service provider, which in turn is used by the ORC service provider. Finally, the sales process is accomplished by booking the sale with the inventory database, transferring information about all sold items (ID’s and quantities).

The sales process as described above is supported by the inventory service and the payment service, where the latter is a composition of two basic services: card validation service and payment debit service. The service components for these services are included in the prediction model to support QoS prediction for the sales process. Thereby, prediction focuses on the completion times and reliability of the involved service operations.

Service Component Model

![ORC Components Diagram](image)

Figure 18: ORC Components
The ORC service component model contains a specification of the service components that are involved in the sales process, as well as the software components of the underlying legacy application (the trading system). The ORC comprises 15 components – 8 service components and 7 legacy components. The service components include basic services (such as the inventory service), and composed services (such as the payment service). Access to the inventory database is modelled through a database component.

Figure 18 shows the specification of the ORC service and legacy components. As the QoS meta-model is defined using the Eclipse Modelling Framework (EMF), prediction models can be developed and visualized through EMF editors, as shown in the figure. All ORC components are contained in the ServicesAndLegacy-ComponentsRepository. The figure shows the contents of the TradingSystem-Webservices.InventoryService component, which provides the interface InventoryServiceIf and requires the interfaces StoreIf and CashDeskConnectorIf. The interfaces themselves are defined in the lower part of the figure. For example, the InventoryServiceIf contains six service operations (changePrice, getAllProducts, etc.). As the inventory service component provides this interface, it must contain ServiceEffectSpecifications (SEFFs) for each of its operations. The SEFFs define the behaviour of the component when invoked with a certain operation.

![Payment Service SEFF](image)

**Figure 19: Payment Service SEFF**

To give an example of a SEFF specification, Figure 19 shows the handlePayment operation of the composed payment service. The specification includes a sequence of actions, surrounded by a StartAction and a StopAction. Two InternalActions represent internal processing steps of the composed service. The specification does not reveal the details of the processing, but denotes the associated resource demands and failure probabilities. The resource demands are specified through probability distributions. For example, the handlePayment-Postprocessing requires between 0.0 and 0.55\(^1\) work units of the underlying CPU, as well as between 0.0 and 0.7 hard disk drive (HDD) work units, and it exhibits an InternalProcessingFailure in approximately 7 of 1,000 cases. These annotations may have been derived from software tests or historical data of the

---

\(^1\) More concretely, the resource demand is uniformly distributed within [0.0,0.3] with 20% probability, and within [0.3,0.55] with 80% probability.
service execution. Besides the *InternalActions*, the SEFF contains two *ExternalCallActions* representing invocations of the external card validation and payment debit services. Beyond linear action sequences, SEFF specifications can also include loops, branches and other control flow constructs (not shown in the figure). Together, all SEFF specifications in the prediction model completely describe the behaviour of the service-based system under study (i.e. the ORC installation).

![Diagram of ORC System Specification](image)

**Figure 20: ORC System Specification**

Figure 20 illustrates the system view on the ORC service component model, showing individual service and legacy components, as well as their composition. The system as a whole provides a set of interfaces (e.g., the payment interface), and also requires a service interface (the bank interface), which indicates the need of a software service provided by an external software service provider. The figure shows that the payment service is composed of the card validation and payment debit services, which in turn make use of the bank service (offered by an external provider). The inventory service is based on the application (store) legacy component.

**Infrastructure Model**

The ORC infrastructure model describes the resource environment to which service and legacy components are allocated. Figure 21 shows the model contents visualized by an EMF editor. The model is prepared to support two deployment options: the complete retail solution running on one virtual machine, or service and legacy components separated from the database. Accordingly, three resource containers have been specified that can be used for deployment – the *AllInOne-Infrastructure*, *Database_VM* and *ServicesAndLegacyComponents_VM*. Each resource container is equipped with a CPU and a hard disk drive (HDD) resource. In the context of SLA@SOI, resource containers represent infrastructure services that may include either physical or virtualized resources. The model only captures the performance and availability of the resources, through processing rates and Mean Times To Failure (MTTF) / Mean Times To Repair (MTTR). In the example, the CPU is set to 2.0 GHz. This value can be adjusted in a concrete negotiation scenario to correspond to a given SLA template specification.
**Allocation Model**

The allocation of service and legacy components to infrastructure resources allows for the calculation of concrete time demands from abstract resource demands specified in component behavioural specifications. Two allocation models have been specified and represent two alternative deployment options. One allocation puts all service and legacy components to a single resource container; the other allocation separates the database from the other components. As an illustration, Figure 22 shows the All-in-one allocation variant, where all components are deployed on AllInOne_Infrastructure, while no components are deployed on the Database_VM and ServicesAndLegacy-Components_VM.

**Usage Model**

![Figure 22: ORC Allocation Model](image)
The usage model represents a system usage profile, which is initially created with default parameters, and adapted during the SLA negotiation to reflect given SLA template specifications. Figure 23 shows the initial version visualized by an EMF editor. For each of the required service operations of the sales process, a UsageScenario represents the repeated invocation of the operation with a certain frequency and certain input parameter values. To this end, the figure shows the ORC_CustomerConstraintInventoryBookSale scenario containing the Inventory-Service.bookSale invocation, as well as two other scenarios for InventoryService-getProductDetails and PaymentService.handlePayment. The frequency of bookSale is set to a default value of an exponential distribution with parameter λ=1/1000 (i.e. one invocation in 1,000 time units on average), and the number of sold items is assumed to follow a probability distribution evaluating to 1 item (30%), 5 items (40%) or 10 items (30%). Notice that the number of items may impact service performance and reliability; for example, a processing step may have to be repeated for each item, leading to an individual resource demand and failure potential per item.

![Usage Model Diagram](image)

**Figure 23: ORC Usage Model**

### 3.8.2 WP B3: ERP Hosting

In this section, we describe the Enterprise Resource Planning (ERP) use case presented by SAP. Traditional ERP systems tend to be very large, rather monolithic and hard to set up and manage. The goal of this case study is to demonstrate that the Performance Cockpit approach described in Section 3.5 is applicable on real data measured on a large enterprise application. We address the problem of customizing an SAP ERP application to an expected customer workload. The workload of an enterprise application can be coarsely divided into batch workload (background jobs like monthly business reports) and dialog workload (user interactions like displaying customer orders). This workload is dispatched by the application server to separate operating system processes, called work processes, which serve the requests [195]. At deployment time of an SAP system the IT administrator has to allocate the available number of work processes (depending on the size of the machine) to batch and dialog jobs, respectively. With the performance curve derived in this case study, we enable IT administrators to find the optimal amount of work processes required to handle the dialog workload with the constraint that the average response time of dialog steps should be less than one second.
The system under test consists of the enterprise resource planning application SAP ERP2005 SR1, an SAP Netweaver application server and a MaxDB database (version 7.6.04-07). The underlying operating system is Linux 2.6.24-27-xen. The system is deployed on a single-core virtual machine (2.6 GHz, 1024KB cache). To generate load on the system we used the SAP Sales and Distribution (SD) Benchmark. This standard benchmark covers a sell-from-stock scenario, which includes the creation of a customer order with five line items and the corresponding delivery with subsequent goods movement and invoicing. Each benchmark user has his or her own master data, such as material, vendor, or customer master data to avoid data-locking situations. It consists of the following six transactions:

1. Create an order with five line items. (VA01)
2. Create a delivery for this order. (VL01N)
3. Display the customer order. (VA03)
4. Change the delivery (VL02N) and post goods issue.
5. List 40 orders for one sold-to party. (VA05)
6. Create an invoice. (VF01)

There are fifteen dialog steps with 10-second think time in-between. The detailed benchmark steps are shown in Figure 24.

In the context of the Performance Cockpit, an experiment (or configuration point) is defined as one configuration of all parameters (i.e., it corresponds to one point in the configuration space). The configuration space is spanned by the configuration parameters and their corresponding domains. In our experiments, the dependent variable is the average response time of dialog steps \(\text{AvgResponseTime} \). The independent variables in this setup are (i) the number of active users \(\text{NumUser} \) where the domain ranges from 60 to 150 and (ii) the number of work processes for dialog workload \(\text{NumWP} \) varied from 3 to 6. Thus, we are looking for the function \( f(\text{NumUser}, \text{NumWP}) = \text{AvgResponseTime} \). The full configuration space consists of 360 measurement points. In order to get statistically stable results we repeated each measurement multiple times. All in
all, the determination of a single measurement point takes approximately one hour which means that in the worst case the IT administrator has to measure 15 days in order to determine the optimal configuration.

We measured the system using different experiment selection strategies provided by the Performance Cockpit. The strategies aim at the automated derivation of a software performance curve with the least possible number of measurements. The iterative algorithms

1. Select new experiments for each iteration,
2. Infer a statistical model based on the available data after each iteration, and
3. Are aware of the quality of the inferred model.

![Figure 25: Experiment Selection Strategies](image)

The three experiment selection methodologies (see Figure 25) that we applied in this case study implement the following strategies:

- The random experiment selection strategy randomly selects a fixed number of new experiments.
- The equidistant experiment selection strategy splits the parameter space in equidistant areas.
- The adaptive experiment selection strategy selects new experiments in those areas of the parameter space that show the worst predictions.

Each of the three methodologies can be combined with various model inference techniques. In this case study, we combined them with Kriging and MARS.

The results of the case study are summarized in Figure 26. The table contrasts the different combinations of experiment selection method and analysis method. To determine the quality of the derived prediction model we compared the prediction for each measurement point in the configuration space with its actual value and calculated the mean relative error (MRE). The results show that our adaptive experiment selection methodologies can provide very accurate prediction results.

<table>
<thead>
<tr>
<th>Random</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Kriging</td>
<td></td>
<td>Mars</td>
</tr>
<tr>
<td>#M/Full</td>
<td>MRE</td>
<td>#M/Full</td>
</tr>
<tr>
<td>104/360</td>
<td>24,50%</td>
<td>90/360</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equidistant</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Kriging</td>
<td></td>
<td>Mars</td>
</tr>
<tr>
<td>#M/Full</td>
<td>MRE</td>
<td>#M/Full</td>
</tr>
<tr>
<td>123/360</td>
<td>7,60%</td>
<td>122/360</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adaptive</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Kriging</td>
<td></td>
<td>Mars</td>
</tr>
<tr>
<td>#M/Full</td>
<td>MRE</td>
<td>#M/Full</td>
</tr>
<tr>
<td>64/360</td>
<td>1,60%</td>
<td>67/360</td>
</tr>
</tbody>
</table>
The combination AdaptiveSelection/Kriging required only 64 measurement points (~18% of the full configuration space) to derive a prediction model with a mean relative error of 1.6%. This reduces the time necessary to derive an optimal configuration from 15 to ~2.5 days.

### 3.8.3 WP B6: eGovernment

A special application of our prediction approach could be demonstrated for the eGovernment industrial use case (WP B6), where we extended the scope of the approach beyond pure software services towards human services and resources. Instead of a software architecture that arises as a composition of software service components, we built a prediction model to reflect a health care system, where “components” are call centres that provide “services” in terms of booking capabilities. Instead of computing resources that are demanded during software service execution, we modelled human operators that serve the booking requests of the calling customers. The concept of “service failure” was mapped to unsuccessful booking attempts of customers due to a lack of available free operators and phone lines. In the following, the booking process, the created prediction model and the predicted result metrics are described. For further description of the eGovernment use case, see Deliverable D.B6a.

#### The Booking Process

Figure 27 gives an overview of the booking process as considered in the eGovernment use case. The process involves customers calling a Citizen Service Centre (CSC) wishing to book health-care and / or mobility services. The outcome of the process may either be a “SUCCESS” (the booking is performed as requested) or a failure of type “BUSY LINE” (the customer found a “busy” signal because no phone line was available) or “LOST CALL” (a phone line was available, but it took too long for an operator to become available). In case of a success, the duration of a call is determined by the waiting time for an operator (if any), as well as the time needed for performing the bookings. To gain additional flexibility, the CSC can delegate customers to an external call centre to avoid overload situations. The external call centre can perform bookings just like the CSC.

As with software services, the booking service of the CSC is subject to SLA negotiation, where the service usage refers to the arrival rate of customers at the CSC (negotiated through the standard term qos:arrival rate), and service quality refers to the percentages of lost calls and busy lines (qos:reliability), as well as the booking time in the success case (qos:completion time). A similar negotiation takes place between the CSC as service customer and the external call centre as a further provider. After creating a prediction model that represents the process as outlined above, the CSC can use our prediction approach to determine the relevant quality parameters of the booking service, depending on the service usage and the quality delivered by the external call centre. In addition, the prediction provides further interesting information for the CSC, such as the average number of busy operators at a time, or the average waiting time for the next free operator. The CSC can use such information for capacity planning in terms of the number of employed operators.
**Figure 27: eGovernment Booking Process**

**Components and Composition**

The specified service component types and interfaces are shown by Figure 28. The booking process is modelled through 3 service component types: CallCenterControl, BookingBackend and OperatorBackend. The CallCenterControl models the overall booking process and the pool of available phone lines (the capacity of the pool is initially set to 100, but can be adapted to reflect different installation sizes). The BookingBackend models the core booking activities and the required booking time depending on the type of request. The OperatorBackend models the pool of available operators (initially set to 25), as well as the acquiring and release of operators for individual booking calls. The
figure also depicts the service interfaces provided and required by the components and their contained service operations.

The concrete health care service architecture is shown in Figure 29. The system provides the booking functionality to its users through an instantiation of the CallCenterControl component. This component uses an OperatorBackend instance, as well as 2 BookingBackend instances (named “HealthBookingBackend” and “MobilityBookingBackend”) to represent all aspects of the booking process.

![Figure 29: Service Architecture for the Booking Process](image)

**Behavioural Specification**

The behavioural specification has two aspects: (i) the behaviour of costumers, and (ii) the system behaviour (i.e., the behaviour of the CSC and external call centre). The customer behaviour is specified by the usage model as shown in Figure 30. A new customer "arrives" every 7.2 seconds (qos:arrival_rate, reflects 500 calls per hour) and requests a booking. Further properties of the scenario are encoded into the input parameters of the booking call. This includes a required health care booking time between 120 seconds and 600 seconds, a mobility booking time between 0 seconds (if no mobility booking is required) and 600 seconds, the probability of a booking request being delegated to the external call centre (initially set to 0.2), the quality characteristics delivered by the external call centre, and the maximal waiting time that a customer tolerates waiting for a free operator. The CSC can vary these parameters and conduct predictions to evaluate different scenarios.

The system behaviour consists of the behavioural specifications of the service component instances as shown in Figure 29. As an example for a behavioural specification, Figure 31 illustrates the “acquireOperator” operation of the OperatorBackend component. As the figure shows, the acquiring procedure depends on the question whether the call is accepted by the CSC or delegated to
the external call centre. These two options are modelled through a BranchAction “acquiringDispatcher” with two BranchTransitions “internalHandling” and “delegatedHandling”. The value of the input variable “delegated” decides which transition to take (this value ultimately stems from the parameter “delegate” of the usage profile as shown in Figure 30). In case of internal handling, an operator is acquired from the modelled CSC operator pool through the “acquireOperator” action. During the simulation, a waiting time will be introduced if no operator is available when performing the action. For delegated handling, no operator pool is modelled (because from the point of view of the CSC, the externally provided booking service is a black box). Instead, a certain waiting time is directly included, according to the negotiated properties of the external booking service. Other parts of the booking process are modelled in a similar way, to reflect all aspects and constraints included in the process.

![Figure 30: Usage Profile of the Booking Service](image)

**Other Model Parts**

The service usage and behaviour specification as explained above is complemented by an infrastructure and allocation model, in order to create a complete QoS model instance that can be used for prediction. In this case, the infrastructure does not represent actual physical resources such as computing nodes with CPUs and hard disks. Rather, it represents a logical execution environment for the health care and mobility booking process as outlined above. In summary, the prediction model accurately reflects the process, which shows that the prediction approach described in this chapter can successfully be applied beyond the pure software domain.
4 Resource Usage Prediction

4.1 Introduction

This section describes an innovation realized within the first two project years, namely the resource usage prediction. During the first year of the project, the Runtime Resource Prediction task focused on the research, development and improvement of several pluggable algorithmic approaches to resource consumption prediction within the context of SLA aware infrastructures.

Within the SLA@SOI context, infrastructure runtime prediction refers to providing a framework for predicting the behaviour of basic infrastructural metrics of resource consumption such as cpu usage (such as user, system or process), load, memory (free, cached, etc), disk and network input/output, etc, based on historical data as the main data source and machine learning techniques that must process the data to generate predictive models.

Figure 32 illustrates the internal architecture of the Infrastructure Runtime Prediction (IRP).

The techniques developed during Y1 are:
- **MPI** (Multiple Predictor Integration): MPI integrates multiple types of predictors, such as Auto Regression, ARM, and Multi-Resource Model, and chooses the best predictor regarding the current inputs. It suits for short-term prediction and can be used for prediction of the metrics with high variance (i.e., the metric changes its value up and down very sharply and frequently).

- **FLC** (Fuzzy Logic and Clustering): FLC combines fuzzy logic and clustering techniques, which can model multiple relevant inputs and capture dependency between inputs and output. The FLC models can be used for both short- (e.g., 5 minutes CPU utilization) and long-term prediction.

- **PPP** (Periodic Pattern Prediction): PPP algorithm and models can discover periodic behaviours existing in historical tracking of the metrics. It captures the trend and the periodicity within the data to make long time range prediction (e.g., I/O prediction, such as 1 hour Byte_in and Bytes_out).

Figure 33 represents the basic components and workflow of Infrastructure Runtime Prediction According to the implementation carried out. Two stages are required to perform dynamic a prediction over a dataset: Training and Prediction. These stages are described below.

![Figure 33: Basic Training/Prediction Workflow](image)

**Training Stage**

In the training stage, a training dataset (**training.csv**) with representative data from the systems under observation is required. The data format of the training set has to be structured as the time series of a metric (or a set of metrics) with a specific sample rate (for example, one measurement every 30 seconds).

Although it is possible to choose a random training data set, the assumption of prediction is that the choice of data characterises the behaviour of that particular metric (or metrics) during a particular period of time. For example, for a metric such as **cpu_user**, it is necessary to assume that if today is a Monday, last Monday’s data is an approximation of how this metric will behave today.

For the training and the prediction stages, the main underlying hypothesis is that the past behaviour will predict the future behaviour. It is important to note also that the training should be contextualised. A particular metric (or metrics) under training should refer to a specific system under a particular workload and the same holds true for the input data. For example, in an office environment, the system that hosts an email server will probably see a spike in usage first thing on a Monday morning as people log in to check their emails. These patterns of usage will be reflected in the metrics and they could be used as a typical representation for the training stage. Although it is not a requirement for runtime prediction to know about software services, it is necessary that the behaviour of software
services is reflected on the data. Each training set represents a usage profile of the system behaviour.

**Prediction Stage**

After the training stage is performed, a temporary model is produced and stored locally. This model is subsequently used during the prediction stage. To this aim, a new live dataset *(input.csv)* that matches the format, length and sample rate of the data set adopted during the training stage is given in input to the system. After this stage the system generates a prediction of the future demand of a particular metric *(output.csv)*. The prediction result is then only considered valid for a very specific period of time (e.g., for 5 minutes), depending on the configuration, and also the current variance of the live data over the previous training set.

![Prediction Dynamic](image)

**Figure 34: Dynamic Prediction with Future Error**

During the training stage, a prediction error is also calculated. This error represents the average percentage of how far off the prediction is. For example, if the training data has 100 data points, it is possible to select 20% (by configuration) of the data points and this data set of 20 points is fed into the prediction algorithm, comparing the output of prediction with the real data used for training. The error is calculated as the average of the deviation of the prediction result from the input data. The error calculation is only valid if the current live data follows the same pattern of behaviour that is present on the training set.

Figure 34 shows the prediction error displayed as an upper bound and (green) lower bound (yellow) for the bytes_in metric (network I/O), using the Prediction UI.

**Training and Prediction Workflow**

As mentioned earlier, it is required to train prediction with a specific data set and to use live input data to perform the prediction. For a given prediction output, the prediction output is only valid for a specific short-term window of time. For this reason it is necessary to re-run prediction with a new live data set as new data is available, or as often as required.
Figure 35: Training and Re-training Workflow

Figure 36: Predicton versus Real Instrumentation

Figure 35 describes the basic training and prediction process, running prediction with input data as often as required and re-training the models. As new data arrives and the conditions of the system changes, it is necessary to retrain the models often enough so that the training model reflects the system’s current state. Potential reasons for re-training the prediction models are the following:

- Periodical: as a “rule of thumb” and with assumption that the system state changes over time continuously, re-train model as often as required such as every day, every hour or 10 minutes (for example) with latest live data. This is the current default configuration of prediction training.

- The state of the system changes due to the deployment of new applications or workloads on the systems, making the current prediction model inaccurate.

- Prediction does not reflect the reality of the system, when the prediction output and the live system data does not match satisfactorily (according to a configurable percentage).
• Runtime reconfiguration (e.g. addition of new metrics, flushing caches and prediction models, etc.).

Figure 36 illustrates a typical configuration and execution of prediction. The data is obtained from a running resource executing a repetitive (over periods of hours) workload. The X-axis represents the time, measured in incremental time steps, where each step corresponds to 1 or more (configurable) seconds. The Y-axis represents the value of the metric at that point in time.

The blue line represents the actual value of the cpu_user metric while the red line represents the predicted value of the metric. As observed, the cpu_user_prediction is flat for a particular period of time (roughly 8-9 steps) as the computed predicted value is an average of the time series of the predicted data set rather than instant values. The predicted value changes every time there is a new prediction computation.

The aim of this configuration of prediction is to focus on what is, on average, the predicted behaviour or trend rather than the short “spikes” of the system (such as bursts of 100% utilization of a resource), as those ones are more difficult to predict.

![Figure 37: Steady-State Metric with Spikes](image1)

![Figure 38: Prediction versus Real Instrumentation (Running Average)](image2)
short spikes cannot be predicted easily as they could be considered random events and it will contribute little to the overall behaviour.

Figure 38 shows the same data in a different way. The data has been aggregated in a running average and shows how prediction roughly follows up the basic trend (upwards or downwards) of the system (especially in the first 2/3 of the graph). For this particular data set, which has high variance, the average value of cpu_user is 12.5 (std=9.7) and for cpu_user_prediction the average value is 16.5 (std=11.5), indicating that prediction could approximate the average value of a data set with a high variance. The error estimated for this particular dataset was approximately ±5%.

At some point in time, the output of the prediction shows the average trend and this information could be potentially used for planning, optimisation or provisioning among many other possible tasks.

For example, the results of the prediction shown in Figure 38 show that at time step (X axis) $t=1$ the average value for the next steps will be a trend downwards for the following instants of $t=t+1$, $t+2$, $t+3$, etc up to $t=25$, where the graph for the real instrumentation and prediction show a trend upwards. At this point it time (t), a provisioning scheduler could potentially take a decision about whether to provision a new VM (that has a “high” cpu_usage) on a server that, in the short term, prediction has estimated a downward trend for the cpu_user parameter. It is also important to note that prediction can only provide basic trend averages with an error that is computed during training and that the workload of the system under prediction requires stable workloads.

## 4.2 Prediction User Interfaces

In order to be able to test and visualise the results of prediction two different approaches have been developed. The first approach consists of a basic standalone desktop application (UI) where users can explore the functionality of prediction with offline data. Although strictly not an user interface, the second approach is a data provider that can query live resources that have prediction metrics from any application that has http data import, computation and charting facilities, such as Excel or OpenOffice, statistical packages such as R and also SLA@SOI components. This service will be described with more detail in the subsequent architecture and implementation section.

### 4.2.1 Standalone User Interface

The user interface implements functionality that allows users to use prediction interactively via a desktop application. For example, the MPI algorithm could be selected to train a model together with training data and the metric(s). After training, the prediction model is generated the error is displayed as shown in Figure 39.

The testing implementation and user interface is designed to allow users to load arbitrary datasets and to test them against existing training data sets as shown below.

The prediction implementation of the user interface is used to perform prediction on a specific metric using the models built beforehand and to visualise the results in a graph. The following figure shows the dynamic prediction simulation with PP models including selected data, specified parameters, simulation chart and up to date simulation error. Figure 41 shows again the full Prediction Dynamic.
4.3 Runtime Prediction Integration Scenario

For driving the use case requirements gathering and consolidation, a basic hypothetical integrated scenario was presented to the relevant parties illustrating a deployment scenario of prediction and its potential benefits. The goal of this presentation is to collect feedback and suggestions of how runtime prediction might be used by the different use cases.

Figure 42 illustrates the main scenario:

- From a given infrastructural SLA, a ProvisionRequest is processed by the DeploymentPlanner (one of the components of the A4 architecture for Y1).
- The infrastructure terms are then extracted. Examples of these terms are: cpuProfile (Fast, Slow, etc related to an SLA level such as Gold, Silver, etc), mem (memory amount in Gigabytes), cpuNum (cores), image types, number of VMs, etc.
- From this request the DP could then contact the InfrastructureLandscape and issue a query to search and allocate PhysicalServers (depicted as blue
square boxes) that could satisfy the provision request requirements. At this point the InfrastructureLandscape is a cache containing the latest known state of the PhysicalServers.

- Each PhysicalServer could be running an arbitrary number of virtual machines (VM - depicted as orange squares in Figure 7). Each server and each VM is then instrumented to gather infrastructure metrics and these are published to the PredictionService; which, after computing the training and prediction, inserts the prediction results periodically in the infrastructure landscape. The state of the PhysicalServer is also saved in the InfrastructureLandscape.

- As the InfrastructureLandscape is a DBMS (SQL), standard queries could be performed over the past and present status of the landscape to identify, for example, which servers are free/busy at a specific time instant.

- Once the query results are returned to the DP, the DP takes the decision of where to allocate (on which server(s)) and instantiate the new VMs according to allocation policies, such as the Enterprise Capabilities Framework (ECF) policies in the specific scenarios of B4 rules & policies or, for example, an energy aware policy that maximises the usage of the PhysicalServers while minimising power usage by means of VM consolidation and powering off servers.

![Prediction Dynamic User Interface](image)

**Figure 41: Prediction Dynamic User Interface**

Figure 43 illustrates a hypothetical search query in pseudo-SQL to search for servers that have been under-utilised over the last hour and that, within the next hour, will have a load level low (with the error below a certain threshold) and that the pattern of usage for the last day has been medium.
**Past/Historical Data**

From the use case perspective (and especially from B4 Enterprise IT), it is important to include historical data as well as prediction data together as a requirement to analyse the past usage of the infrastructure and provide data that could be used as inputs for the planning, optimization and adjustment approaches that are being developed in some of the other A-line work packages. Data regarding the past behaviour are also important to potentially create usage profiles based on the SLAs and the users of the system that can help to further optimise and forecast the behaviour of the services deployed.

From a technical perspective, the result of the queries could be seen as vectors that include the following values for each *PhysicalServer* and *VirtualMachine*, SLAs and Users within the infrastructure landscape:

- PhysicalServer [1...N] [PastMetrics, PresentMetrics, FutureMetrics]
**4.4 Design Rationale**

The work developed during the second project year has been focused on providing an architecture and implementation of which prediction algorithms developed during Y1 could be run in a scalable, reliable and deterministic way and can be offered as a service to the different SLA@SOI components.

**4.4.1 Prediction Overhead**

![Figure 44: Resource Usage (cpu_user) Over Time](image)

Running prediction on-demand is not a cost free operation as prediction is a very cpu-intensive process. Figure 44 illustrates the cost of running runtime prediction (training and prediction) for 1 metric and 1 resource during 12000 time steps.
The X-axis represents the time while the Y-axis represents the resource utilization (cpu_user). The resource usage of prediction shows a pattern of very short bursts (100%) of utilization with idle gaps in between. On average and for a particular configuration (2007 class Xeon processor, single threaded application), to compute prediction vectors 5+2 times per hour required around 5% of resources.

As illustrated in Figure 44, running prediction is not a free process and requires a careful analysis of the trade-offs of running it, such as prioritising prediction computation (with the current implementation of prediction) over giving those resources as examples to the consumers of the infrastructural services.

Figure 42 illustrates the scenario with 12 resources (3 servers and 9 VMs) in total are represented. With 12 resources in total, it would take approximately 60% of 1 single core resource (ignoring potential queuing effects and uniform distribution) to provide basic prediction capabilities. Extrapolating this result, it would take 1 full (4 core) server to provide prediction capabilities for 50 resources (Physical or Virtual, as the cost is the same assuming that the configuration is uniform.

The SLA@SOI framework should be designed to cope with the management of large amounts of provisioned SLAs and must be able to potentially run thousands of SLA aware services within one single administrative domain. In order to provide runtime prediction services for the SLA@SOI components that can be accessed on demand we have started to architect prediction services with scalable computing principles [151] in mind; principles such as elasticity whereby we can “provision” prediction computation for large amounts of SLA aware services.

4.4.2 Scalable Architectures

Typically, a Service Oriented Architecture (SOA) follows a semi-centralised (or decentralised [152]) managed approach deployed within very specific boundaries (i.e. single enterprise) with well-defined contracts (service’s interface). Also, among the components, there are means of search, discover, bind and execute atomic and composite services. However, SOAs, as a high level concept, does not often address the operational parts of an architecture that can contribute to scalability and these aspects must be considered within the architecture, such as how it can be “elastic” to accommodate internet-scale workloads. For example some of the concerns that scalable architectures require are [153]:

- Network design (how the services are wired together)
- Configuration of the different synchronous and asynchronous parts of the system.
- Redundancy mechanisms.
- Design for failure.
- Caching policies and approaches (memory, disk, distributed, etc).
- Load Balancing.
- Shared State/Stateless services.
- Database replication (Master/Slave, Master/Master, Replication Lag, etc).
- Database partitioning (Clustering, Federation, Shards, etc).
- Monitoring, scalable monitoring.
- etc.

One key concept within the A4 architecture from Y1 and SLA@SOI is the concept of infrastructural Agents. The goal of the Agent architecture was to facilitate the creation and management of distributed message-driven entities across the network that can be addressed to perform computations. During runtime, the Agents form an "Agent Landscape" and this landscape can be dynamically re-
configured in order to change its behaviour. Agents could execute a different number of tasks, such as provisioning, or, in this case, they could execute a prediction task (training and prediction) that is network addressable via messaging.

Within A4 Y1 architecture, the provisioning requests and the act of provisioning was managed by agents, whereby agents were running on physical servers that provided VM provisioning or acted as managers. Similar concepts could be applied to the provisioning of prediction computation. This concept of the agent architecture will be also reused in WP A3, task 3.5 where the architectural concepts of Agent and tasks have been reused.

One critical aspect of the Agent architecture is that we must convert provisioning requests (such as VMs or prediction computation) from synchronous to asynchronous as the Agent architecture is messaging based. When a synchronous request from the client side arrives, the requests are processed, de-coupled and put into a managed messaging queue. The queue is then processed for each request and a series of messages are sent to specific agents within the system to reserve and provision the resources specified. In Y1, the reservation/provision messaging workflow was managed by a transaction manager to guarantee successful provisioning or a managed failure. Since the Agents implemented has a notion of self-awareness (such as an agent does not respond to a provisioning request if its resources are exhausted), the implementation of the agent architecture follows some of the practical principles of autonomic computing [154].

This conversion of time domains (from synchronous to asynchronous then back to synchronous) is a critical aspect to be able to facilitate "elastic" SOAs whereby the services required to satisfy a request that can grow or shrink to accommodate current and future demand. Some research projects have studied, in the past, how the autonomies agent approach can be used to provide an elastic SOA backend infrastructure that can adjust to different levels of demand in an ad-hoc manner [155][156].

There are many systems in the literature that implement the paradigms of scalable de-coupled and messaged based. One example of such systems is Amazon’s Dynamo [154] whereby to complete a request (to generate the content of a web page, a synchronous operation) more than 150 asynchronous services including services with dependencies on other services need to be executed. The properties that scalable services should have by design are [159]: Asynchronous, controlled concurrency, controlled parallelism, decentralised, Decomposition (simple reusable/composable services), failure tolerant, local responsibility, built-in recovery mechanism and symmetry [158]. For the architecture and implementation of the prediction service, we have aimed to follow such principles.

Dynamo has been implemented following the “Eventually Consistency” philosophy [157] to overcome the “CAP” theorem limitation [160]. As the systems become asynchronous and distributed, the state of a request can only converge and be consistent at some point in time. The CAP theorem also highlights an important limitation of the traditional SOA model: how an SOA can be flexible enough to be able to accommodate large amounts of users. Many other systems are emerging, including Google’s approach of consider computation not only as a distributed system, but as an entity that encompasses a whole data centre with all its subsystems working as a unified platform for providing scalable computing infrastructure [161].
Another area of work that is very relevant to runtime prediction providing distributed scalable computation is a system such as Hadoop and its sibling projects [162] and reference implementations of standard machine learning algorithms that can run on such infrastructure [163], as these systems and algorithms could be integrated and deployed within the SLA@SOI framework to offer computation services for prediction and many other tasks.

4.5 Architecture and Implementation

This section will briefly illustrate the architecture and implementation of prediction and its subcomponents. This architecture is strictly related to the work of WP A4 and use case B4.

The architecture of prediction is heavily based on the Agent architecture developed during Y1 in A4 and it has been extended with custom profiles, tasks and actions to support the required prediction functionality.

Prediction is mainly structured around three Agents: PredictionAgent, PredictionWorker and ClientAgent. PredictionAgent is responsible for providing an interface to the other parts of the SLA@SOI framework. ClientAgent consumes prediction data locally. Multiple PredictionWorker(s) are responsible to provide distributed prediction computations and speed up the prediction process when multiple prediction requests arrive, by having a cluster of services (servers) listening for prediction provisioning requests via the messaging bus that can.

In terms of implementation, the three agents are very similar, changing only the type of profiles that they types and the tasks that they implement. As Tasks are independent entities.

4.5.1 Client Agent

The next section will discuss briefly the implementation of some of the tasks that ClientAgents integrate. Each ClientAgent can integrate and arbitrary number of tasks, depending on the specific requirements of the use case. The following generic tasks have been developed for the specific context of the B4 use case:

- **ClientTask**: Responsible for the management of the ClientAgent, start-up and runtime configuration, local and remote prediction, landscape registration, messaging handling and protocol implementation.

- **InstrumentationTask**: Capture of raw metrics, data pre-processing and preparation for prediction.

- **PredictionTask**: calculates and/or delegates prediction from the pre-processed metrics (delegation via messaging).

- **AnalysisTask**: Takes instrumentation and/or prediction data and analyses them based on local rules, if local analysis is required.

- **LandscapeTask**: publishes the historical and predicted (averages for the current implementation) metrics to the infrastructure landscape.

- **FindTask**: Finds free PredictionWorkers (see below) to offload prediction computations to worker nodes.
As part of implementing some of the self-management capabilities, one very important task that the ClientAgent executes is the gathering and processing of raw data metrics. Figure 45 illustrates how the raw metrics are gathered locally and pre-processed. For the current implementation, the local gathering sub-system has 3 separate functions: Instrumentation, Local Historical Database (HistoricalDB) and Query.

**Figure 45: Prediction Data Gathering and Processing**

The *InstrumentationTask* part has been integrated by simply deploying Collectl [163] on each of the resources that require prediction. Collectl is a very powerful, low overhead and configurable instrumentation component for server/workstation monitoring. Once the Collectl deamon is configured and running, it saves raw metrics to a local log file that can be then processed by the prediction tasks.

For the first integrated prototype of prediction, we have chosen Collectl over Ganglia (as used in A4 Infrastructure Management) as Ganglia is oriented towards network gathering of raw metrics in a client/server type of architecture, whereas Collectl provides simpler mechanism to manage the logs locally. However the same subsystem could be used for processing any other monitoring system if the data is supplied as comma separated values (CSV).

A typical default configuration of the Collectl daemon will save the raw metrics every 30 seconds. The *HistoricalDB* is a simple collection of files, one file per day, of all the raw metrics captured by Collect. For the current use case implementation, each log file is archived after two weeks. The local query functionality is where the raw metrics are pre-processed and prepared to compute prediction. The pre-processing can be configured to process the raw
data based on different time windows such as weekly, daily, group of days, and
depends on the specific needs of the use cases.

In order to speed up the prediction process (the computation time for training
and prediction increases linearly with the amount of data available), raw data is
pre-processed by using a smoothing function that reduces the number of data
points.

The smoothing function has also the desired side effect of removing some of the
high variability (such as spikes) of the data. The current implementation for the
smoothing function is a moving average such as the ones considered and with the
current configuration reduces the number of data points by 66% (configurable).

After the data is pre-processed, locally on the ClientAgent, 5 directories are
created: Total, Prediction, Training, LastHour and LastMinute. Each directory is
respectively associated with:

- **Total**: contains metrics for the total period of time under consideration
  (currently two weeks).
- **Prediction**: the predicted values that are computed locally or remotely
  are placed here.
- **Training**: training data sets for the window of time considered values the
  average values of the metrics for the period of time considered for
  training.
- **LastHour**: the average values if the metrics for the last hour of
  observations.
- **LastMinute**: the average metrics for the last minute observations.

This implementation not only allows for computing the average value of the
metrics but also allows for performing more sophisticated statistical analysis. The
table (below) illustrates the contents of the directories and properties files of
some of these directories (left hand side of table and samples of the contents of
the files .properties (right hand side of the table) for each metric under
observation for the directories training (Today+Last week=Training) and
LastHour.

As discussed earlier, the metrics for past, present and future take the same
format, helping the integration and processing by other components. Once the
prediction values are computed, a message is sent via the LandscapeTask to
PredictionAgents containing the status of the resource, including the past, present
and future (prediction) metrics.
4.5.2 Prediction Agent

The tasks executed by the PredictionAgent are:

- **PredictionServiceTask**: General management, start, stop, configuration of the agent.

- **PredictionServiceLandscape**: Process the messages arriving to the landscape messaging channel. Records and keeps track of the ClientAgents in a local database. Saves prediction data to a local historical database.

- **PredictionEndpointTask**: Provides query services for the consumption of prediction services by SLA@SOI components.

- **PredictionTask**: This is the actual task that is used to compute models for training and prediction. It can be disabled or enabled, and is the only non-management task enabled in the PredictionWorker agent.

The PredictionEndpointTask presents several query interfaces which allows other components to obtain prediction information for resources. Currently the main bulk of the implementation and customisation of this task has been developed for the specific needs of use case B4.

The figure below illustrates the output web service based interfaces developed and customised for the B4 UseCase. For a Service (a virtual machine for example
in B4 terminology), a series of time-stamped metrics can be obtained by querying for the virtual machine ID that was obtained the Infrastructure Service Manager (ISM).

![XML code for prediction metrics]

**Figure 46: Prediction for cpu_usage** Metric

A query against the PredictionAgent, can take the form of the following http REST web service call:

```
http://predictionAgent:port/instrumentation?id=545200457a4c&metric Name=cpu&minutes=4
```

A query like this with the current implementation will retrieve all the cpu related metrics with the following format:

```plaintext
[historytotal, lasthour, lastMinute, predictionerror, last, prediction]
```

The meaning of the individual parts is as follows:

- **historytotal**: The average value measured for the metric (cpu) since monitoring of this service began. This metric is useful to determine over time what is the service utilization. For example an adjustment algorithm could look at this metric and optimize its usage by means of making sure this value is maximised. This metric changes slowly over time.

- **lasthour**: The average value of this metric for the last minute. For example a planning component can evaluate this metric for a service and provision only on services (physical servers) that have a medium utilization.)
- **lastMinute**: The average value of this metric for the last minute. For example planning and optimization components could look at this metric and determine the services with lowest utilization and perform a provisioning that requires a high instant utilization (based on knowing the usage profile of the service that needs to be provisioned). This metric can change very rapidly

- **predictionError**: The calculated error for prediction, based on the training stage. This value is only estimation.

- **last**: The average value of this metric for the training period considered. This metric can changed slowly in general depending the period of time considered for training, such as few hours, or a day.

- **prediction**: The predicted average value for the metric

With these vectors, it is possible to design and implement different types of heuristics that can act upon the past, the present, or the predicted future of a metric, or combinations of these metrics depending on the requirements. Figure 46 illustrates the output (xml) of the prediction service REST call.

At the very basic level, the instrumentation part of prediction provides a useful framework on which components could be built that take advantage of the information provided by prediction. The REST interface also makes prediction easy to integrate with other components. Figure 47 illustrates the B4 UseCase enterprise IT displaying monitoring and prediction data for provisioned services.

![Prediction Instrumentation of the B4 Use Case](Image)

**Figure 47: Prediction Instrumentation of the B4 Use Case**

### 4.5.3 Prediction Worker

By configuration on the ClientAgent, prediction can be processed locally or remotely. When it is configured, ClientAgents will try to offload the computation(s) of prediction by means of finding a ServiceAgent or PredictionWorker on the network that is capable of performing the prediction computations.
It is important to know that although there could be an arbitrary number of ServicesAgents on the network, only one can be configured to act as the unique central point of access to the prediction subsystem. It is possible to have an arbitrary number of PredictionWorkers with the only requirement that they all shared the same messaging channel.

The sequence diagram below (Figure 48) illustrates the protocol used as a sequence diagram of message passing required to find PredictionServices (ServiceAgents or PredictionWorkers) nodes.

![Sequence Diagram](image)

**Figure 48: Prediction Provisioning**

One aspect not shown in this diagram is how the PredictionServices choose to respond (or not) to the different ClientAgents requests searching for free or available resources that can compute prediction. As the ServiceAgents are also instrumented and have access to their own metrics, they can determine how many active prediction requests have, and, and how many spare resources (cpu mainly) they have free/available.

On the ClientAgent side, for each Find “Prediction Service Agents” request issued to the messaging bus, an arbitrary number of ACKs from ServiceAgents could potentially arrive. It is the responsibility of the Client Agent to choose only one (either randomly or by policy) from the list of available Service Agents that responded to the Find message. However the Find operation could be executed every time a new training/predict cycle is started. In the current implementation, the Find operation is executed as Task.
5 Run-time SLA Violation Prediction

5.1 Introduction

The EVEREST SLA monitoring framework has been designed with the objective to support two different monitoring scenarios for service-based systems (SBS) using a non-intrusive approach. The two key features of this approach are that monitoring is performed in parallel with the operation of an SBS without affecting its performance and does not require the instrumentation of the composition process of an SBS system or the individual services deployed by it.

In the first of the assumed monitoring scenarios (Scenario 1), a human user (typically the provider of an SBS) can request the framework to monitor whether the runtime operation of the system satisfies certain SLA guaranteed terms and view any deviations from these terms as soon as they are detected.

In the second scenario (Scenario 2), the monitoring can be requested by the environment that executes the process of an SBS. In this scenario any deviations of the SLA guaranteed terms which are being monitored are reported back to the system which requested the execution of the monitoring activity.

In both these scenarios, the input to EVEREST is a monitoring specification that contains the formulas to be monitored and other monitoring parameters. These formulas are generated automatically from SLA guaranteed terms by the reasoning component gateway (RCG) through which EVEREST is connected to the SLA@SOI framework. Depending on the form of the SLA guaranteed term that is to be monitored, the operational monitoring specification of EVEREST can include Event Calculus formulas for maintaining the values of monitoring parameters and Event Calculus formulas for checking conditions about these parameters and terms.

5.2 Evolution of the EVEREST Framework

The objective we aimed to achieve was to perform run-time prediction of SLA violations in an automatic and efficient manner, without the need of supervision. To accomplish our objective, we designed and developed a framework, called EVEREST+, which supports the prediction of potential violations of QoS properties in SLAs. EVEREST+ has been developed as part of a generic monitoring framework, called EVEREST. EVEREST+ provides an integrated architecture for SLA monitoring and prediction through the deployment of a built-in set of internal model-based predictors.

The prediction capabilities of the EVEREST+ framework are different from the relevant capabilities of the initial EVEREST that had been developed as part of the SERENITY project. These differences are both theoretical and technical.

From a theoretical point of view, the prediction approach of the original EVEREST framework that had developed within SERENITY was based on Dempster-Shafer theory of evidence [191] and allowed the combination of evidence from different sources in order to compute a degree of belief (represented by a belief function) in the occurrence or not of atomic runtime events that could violate a given monitoring rule. The basis of the prediction approach of EVEREST+ was
maintenance of historical data enabling the computation of conditional beliefs in the occurrence of runtime events (i.e., beliefs that an event E1 would occur within some period in the future, given that another event E2 or set of events has occurred). This approach could support the prediction of non-aggregate properties of services (e.g., integrity, liveness) as discussed [193] but could not support predictions of aggregate quality-of-service properties as those that are normally found in SLAs (e.g., average service performance, various properties related to service availability and maintainability (e.g., mean-time-to-failure and mean-time-to-repair for a service).

The new prediction framework of EVEREST+ [201, 203, 204] allows the development and automatic maintenance of runtime prediction models for estimating the probability of potential violation/satisfaction of constraints over aggregate QoS properties. The development of these prediction models is based on prediction specifications determining: (a) key variables underpinning the quality properties that need to be predicted, (b) the patterns of events that need to be monitored in order to collect runtime data about these variables, and (c) the computations that need to be performed to generate predictions for the properties. Based on these specifications, EVEREST+ can infer automatically the probability distribution functions of the indicated variables at runtime using monitoring data, and uses these functions to calculate the probability of future violations/satisfactions of the relevant property, as determined by prediction specifications. EVEREST+ can also dynamically update the probability distribution functions of the variables underpinning a model following the accumulation of new monitoring data.

Furthermore, from a technical point of view, the prediction approach developed within the EVEREST was tightly coupled to data generated and maintained by the monitoring engine of the framework. The prediction approach developed in EVEREST+, can use not only historical QoS monitoring data collected and generated by EVEREST but also QoS data coming from any other relational database or other sources, as long as they conform to a given format determined by a prediction specification. In the latter case, a user should provide a suitable driver for data retrieval. EVEREST+ also provides a set of APIs for developing user-defined predictors (i.e., prediction algorithm implementations) and for extending the set of automatically inferred probability distribution functions.

Table 4 summarises the differences between the prediction approach realised in EVEREST+ and the earlier prediction approach that had been developed as part of EVEREST.

<table>
<thead>
<tr>
<th>EVEREST+ (SLA@SOI)</th>
<th>EVEREST (Serenity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Prediction can be used for aggregate QoS properties, (i.e., properties whose value is computed as a function of a series of – as opposed to atomic – events) such as service MTTR, MTTB.</td>
<td>- Prediction can only be used for properties expressed as patterns of atomic events.</td>
</tr>
<tr>
<td>- Prediction is based on the automatic identification and update of probability distribution functions for variables that affect the property of concern from historical data and the use of the probabilities returned by</td>
<td>- Prediction is based on computing beliefs in the conditional occurrence or not of atomic events based on Dempster-Shafer theory</td>
</tr>
<tr>
<td>these functions</td>
<td>- Non extensible prediction capabilities</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>- Incorporates a built-in set of prediction models for QoS properties.</td>
<td></td>
</tr>
<tr>
<td>- The set of built-in prediction models is extensible (new prediction models can</td>
<td></td>
</tr>
<tr>
<td>be developed as EVEREST+ plug-ins and supported by adequate prediction</td>
<td></td>
</tr>
<tr>
<td>specifications).</td>
<td></td>
</tr>
<tr>
<td>- Prediction can be based on monitoring data imported from external monitors</td>
<td>- Prediction can only be based on monitoring data maintained by the monitoring engine of EVEREST.</td>
</tr>
<tr>
<td>and not only EVEREST, provided that the latter comply to a specific format and</td>
<td></td>
</tr>
<tr>
<td>the form as well as the use of the data are described in a prediction</td>
<td></td>
</tr>
<tr>
<td>specification.</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Differences between EVEREST and EVEREST+

5.3 Prediction Model

5.3.1 Overview

A QoS property prediction in our approach is the computation of the probability that the QoS property will violate a constraint some future time point \( t_e \), given a request for it received at a time point \( t_r \). Figure 49 illustrates this general formulation of the prediction problem. In particular, \( t_e \) in the figure is the time point at which a prediction for a property QoS is requested; \( t_r \) is the time point in the future that the prediction is required for; \( p \) is the prediction window (i.e., \( p = t_e - t_r \)); \( N \) is the number of QoS values observed between \( t_r \) and \( t_e \); \( Y \) is the number of future QoS property values that are expected between \( t_r \) and \( t_e \); \( QoS_e \) is the value of the observed QoS property at the time point \( t_e \); and \( QoS_p \) is the value of the predicted QoS property at the time point \( t_e \).

![Figure 49: Prediction Framework Common Definitions](image)

Given this generic formulation, the computation of the probability of violating a QoS property constraint is based on estimating the probability of occurrence of different values of variables that affect the QoS property in the prediction window, and can make it violate the constraint. The probabilities of the values of the underpinning variables are determined by finding the probability distribution functions (PDFs) that have the best fit to sets of historic values of these variables.
5.3.2 Prediction Model for Software Services

MTTR

The overall approach outlined above has been used to develop the prediction model for the mean-time-to-repair (MTTR) and mean-time-to-failure (MTTF) of a service. More specifically, the MTTR of a software service is defined as the average time from a failure of a service to respond to an operation call until it restarts responding to operation calls normally. MTTR needs to be bounded to ensure the timely reactivation of a service after periods of unavailability. In an SLA, this would be typically specified as a boundary constraint of the form: MTTR ≤ K where K is a constant time measure.

The estimation of the probability of violating the boundary constraint MTTR ≤ K at a future time point \( t_e \) is based on identifying the probability distribution functions of two variables: (1) the MTTR of the service of concern, and (2) the time between non-served calls of service operations (i.e., service failures) that occur in a period during which a service has been available (referred to as "time-to-failure" or "TTF" henceforth). MTTR and TTF values correspond to the periods in the operational life of a service shown in Figure 50. More specifically, MTTR is computed as the average of TTR values, i.e., the time difference between the timestamp of the first served call of a service following a period of unavailability and the timestamp of the initial non served call (NS Call) of the service that initiated this period. TTF is the difference between the timestamps of two NS calls of the service that initiate two distinct and successive periods of unavailability.

Assuming that \( N \) is the number of MTTR values recorded until the time \( t_e \) at which the prediction is requested, \( t_c \) is the future time point which the prediction is requested for, and \( y \) is the – yet unknown – number of TTF values that will be recorded during the prediction horizon \( p \) (or, equivalently, the number of cases where the service fails again following a period over which it has been available), to violate the MTTR constraint at \( t_e \) the following condition must be false:

\[
MTTR_e = (N \times MTTR_c + y \times MTTR_y)/(N + y) \leq K \quad (1)
\]

According to (1), the MTTR value at \( t_e \) (i.e., \( MTTR_e \)) is computed by replacing each of the \( N \) TTR-values that have been recorded until \( t_c \) by the average value \( MTTR_c \) that has been recorded until \( t_c \), and each of the \( y \) TTR-values from \( t_c \) to \( t_e \) by their average value \( MTTR_y \), since:

\[
\sum_{i=1}^{N} TTR_i = N \times MTTR_c \quad \sum_{j=1}^{N+y} TTR_j = y \times MTTR_y
\]

From formula (1), however, it can be deduced that for the MTTR constraint to be violated it must be that:

\[
MTTR_y \geq (K \times (N + y) - N \times MTTR_c)/y \quad (2)
\]

Given (2), however, there are two factors to take into account to predict MTTR:

(a) The probability to observe \( y \) reactivations of the service in the time period \( p \) from \( t_c \) to \( t_e \), denoted as \( Pr(y) \) in the following, and

(b) The probability of having \( MTTR_y > MTTR_{crit} \) (i.e., \( Pr(MTTR_y > MTTR_{crit}) \)) where \( MTTR_{crit} = (K \times (N + y) - N \times MTTR_c)/y \).
Based on the probabilities (a) and (b), the probability to violate the constraint MTTR ≤ K at the end of the time period p can be estimated approximately by the following formula:

$$\Pr \left( \bigcap_{y=1}^{M} E_y \right) = \begin{cases} 1 - \sum_{y=1}^{M} \Pr(y) \times \Pr(MTTR_y \leq MTTR_{crit}), & MTTR_c > K \\ \sum_{y=1}^{M} \Pr(y) \times \Pr(MTTR_y > MTTR_{crit}), & MTTR_c \leq K \end{cases}$$  \hspace{1cm} (3)

Formula (3) distinguishes two cases: (a) the case where the last recorded MTTR value at the time when the prediction is requested violates the constraint (i.e., the case where MTTR_c > K), and (b) the case where the last recorded MTTR value at the time when the prediction is requested does not violate the constraint (i.e., the case when MTTR_c ≤ K). In the former case, the probability of violation is computed as the probability of not seeing a value of MTTR in period p (i.e., MTTR_y) that is sufficiently small to restore the current violation. In the second case, the probability of the violation is computed as the probability of seeing a large enough MTTR_y value (i.e., a value greater than MTTR_{crit}) that would violate the constraint.

Since the value of y is not known at t, when the prediction is requested, formula (3) considers values of y up to an upper limit M. The value of M is determined by a condition over its probability. More specifically, M is set to the largest value that makes Pr(y=M) arbitrarily small (i.e., Pr(y=M) < e for an arbitrary small e^2).

**Figure 50: TTR and TTF Values**

To compute the probabilities Pr(y), Pr(MTTR_{crit} ≤ MTTR_y), and Pr(MTTR_{crit} > MTTR_y) we need to know the cumulative distribution functions (CDF) for the variables y and MTTR, referred to as CDF_y and CDF_{MTTR_y} respectively. Given CDF_{MTTR}, Pr(MTTR_{crit} > MTTR_y) and Pr(MTTR_{crit} ≤ MTTR_y) can be computed by formulas (4) and (5) below:

$$\Pr(MTTR_{crit} > MTTR_y) = CDF_{MTTR}(MTTR_{crit})$$  \hspace{1cm} (4)

$$\Pr(MTTR_{crit} ≤ MTTR_y) = 1 - CDF_{MTTR}(MTTR_{crit})$$  \hspace{1cm} (5)

To compute Pr(y) we do not use CDF_y but the probability distribution function of TTF variable (i.e., the difference between the timestamps of two NS calls of the service that initiate two distinct and successive periods of unavailability, as defined earlier). This is because over a time period of p time units TTF is related to y as shown by the formula below:

$$TTF = p/y$$  \hspace{1cm} (6)

---

2 The value of e is set by the requester of the prediction.
As (6) indicates, TTF is an invertible monotonic function $g$ of $y$ (i.e., $TTF = g(y) = p/y$ where $p$ is constant). Thus, the cumulative probability distribution of $y$ $CDF_y = g(CDF_{TTF})$ can be computed by the formula:

$$CDF_y(y) = 1 - CDF_{TTF}(g^{-1}(y)) = 1 - CDF_{TTF}(p/y) \quad \text{(7)}$$

where $g^{-1}(y) = p/y$.

Furthermore, to compute $Pr(y)$, instead of considering a single value $v$ of $y$, we consider the range $y(v) = (v - 0.5, v + 0.5]$. Thus, assuming that $x_1 = y - 0.5$ and $x_2 = y + 0.5$, $Pr(y)$ is computed by the following formula:

$$Pr(y) = CDF_y(x_2) - CDF_y(x_1) = CDF_{TTF}(p/x_1) - CDF_{TTF}(p/x_2) \quad \text{(8)}$$

In the case of MTTF, the typical SLA constraint that should be monitored and forecasted is $MTTF \geq K$ (the largest the MTTF the less frequent the failures of the given services) and the probability of violating this constraint can be estimated approximately by the formula:

$$Pr \left( \bigwedge_{y=1}^{y=M} y \right) = \begin{cases} 1 - \sum_{y=1}^{M} Pr(y) \times Pr(MTTF_y \geq MTTF_{crit}) , & MTTF_c < K \\ \sum_{y=1}^{M} Pr(y) \times Pr(MTTF_y < MTTF_{crit}) , & MTTF_c \geq K \end{cases} \quad \text{(9)}$$

The above formula is derived similarly to the case of MTTR but due to space restrictions the details of its derivation are omitted.

### 5.4 Runtime Computation Of MTTR/MTTF Models

The prediction models of MTTR and MTTF have been implemented using EVEREST+.

---

**Figure 51: EVEREST+ Architecture**
EVEREST+ includes two subsystems: (1) a core monitoring subsystem, called EVEREST (EVEnt REaSoning Toolkit), and (2) a prediction subsystem. The monitoring subsystem of EVEREST+ checks functional and QoS properties based on events intercepted from services using internal or external event captors. Whilst monitoring QoS properties, EVEREST stores QoS related information, including computed QoS property values, instances of violations and satisfaction of guaranteed constraints of QoS properties that have been set in SLAs, and the values of any other state variables that might have been taken into account in checking particular QoS properties. These types of information are made available through an API. The prediction subsystem of EVEREST+ (see prediction framework in Figure 51) supports the generation of predictions for potential violations of guaranteed constraints for QoS properties upon request. This support is available through generic functionalities including: the automatic fitting of different built-in probability distribution functions (PDFs) to different types of historical QoS data generated by EVEREST; the selection of the PDFs that have the best fit with the data, the update of PDFs following the accumulation of further QoS property monitoring data; and the generation of predictions for QoS property violations based on built-in and/or user defined functions making use of the probabilities returned by the fitted PDFs.

EVEREST+ automates the prediction generation process based on prediction specifications expressed in an extension of the SLA specification language SLA* that we have developed for this purpose.

5.4.1 Prediction Specifications

A prediction specification (PS) includes: (a) a set of generic parameters for the forecast, (b) the agreement term to be forecasted, (c) a predictor configuration, and (d) a QoS specification. In the following, we examine these parts of prediction specifications more closely and illustrate them through the example PS shown in Figure 52, which is used to derive the prediction model for MTTR (the grey text within parentheses in the figure shows the differences of the PS for MTTF prediction model).

(i) Generic parameters: The generic parameters in a PS determine: the identifier of the service and the operation that the QoS property to be forecasted relates to, the prediction window of forecasts (i.e., the time period in the future that the prediction is required for), and the history size of forecasts (i.e., the size of the historic event set that will be analysed to derive the prediction model). In the example of Figure 52, the PS refers to the operation Ping of service Srv and sets the prediction window to 10 minutes, and the history size to 500 events.

(ii) Agreement term: The agreement term in a PS specifies the constraint that should be satisfied for the QoS property of interest by the target service and operation of the PS (aka guaranteed state). In the example in Figure 52, the guarantee state refers to the QoS property MTTR (MTTF) and the specified constraint regarding is that MTTR (MTTF) must be less than (greater than) 10 seconds.

(iii) Predictor configuration: The predictor configuration in a PS indicates which prediction model to use for computing the probability of the guarantee state of the PS, and the variables whose probability distribution functions will need to be determined from historical monitoring data as they will be needed by the predictor. The former is specified as the value of the attribute predictor.id of the predictor configuration in a PS and the latter are specified by the element
prediction variables that includes a list of variables, each of which is specified by a name/value pair.

The predictor configurator in Figure 52 identifies the predictor as “MT_SV_PRED” (i.e., a parametric predictor for formulas (3) and (9)). It also identifies the two variables used as parameters for the two models (i.e., "MTTR" and "TTF" for the MTTR model and "MTTF" and "TTF" for the MTTF model). The names used for the identification of the two prediction parameters of the MTTR model correspond to names of fluent-variables used in the operational monitoring specifications of the relevant QoS properties. Based on this convention, EVEREST+ can identify the monitoring data that need to be considered in determining the probability distributions functions of the relevant variables at runtime, as we explain below.

```
1  prediction specification {
2    generic parameters {
3       service.id = Srv
4       operation.id = Ping
5       prediction.window.value = 10
6       prediction.window.unit = minute
7       history.window.size = 500
8       history.window.unit = event
9    }
10   agreement term {
11      guaranteed state {
12         expression.qos = MTTR (MTTF)
13         expression.operator = less than (greater than)
14         expression.value = 10
15         expression.unit = second
16      }
17   }
18   predictor configuration {
19      predictor.id = MT_SV_PRED
20      prediction variables {
21         variable {
22            name = EVEREST+.model.distribution
23            value = MTTR (MTTF)
24         }
25         variable {
26            name = EVEREST+.model.distribution
27            value = TTF
28         }
29      }
30   }
31   qos specification {
32      specification.name = MTTR (MTTF)
33      specification.value = MTTR-Formulas (MTTF-Formulas)
34   }
35 }
```

Figure 52: Prediction Specification for MTTR/MTTF

(iv) QoS Specification: the QoS specification within a PS provides the operational monitoring specification of the guaranteed state of the QoS property that the prediction is required for. This monitoring specification is expressed in the Event Calculus based monitoring language of EVEREST+, called EC-Assertion. In the following, however, we provide an overview of the language to enable the reader
understand the monitoring specification of MTTR listed below. In EC-Assertion, a
guaranteed state over a QoS property is expressed by a monitoring rule and a set
of zero or more assumptions. Both monitoring rules and assumptions in have the
general form: body⇒head. The semantics of a monitoring rule of this form is that
when the body of the rule evaluates to True, its head must also evaluate to True.
The semantics of an assumption is that when the body of the assumption
evaluates to True, its head can be deduced as a consequence. The body and
head of EC-Assertion rules and assumptions are defined in terms of the following
Event Calculus predicates:

(a) The predicate Happens(e,t,R(lb,ub)) which denotes that an instantaneous
event e occurs at some time t with in the time range R(lb,ub);

(b) The predicate HoldsAt(f,t) which denotes a state (a.k.a. fluent) f
holds at time t;

(c) The predicates Initiates(e,f,t) and Terminates(e,f,t) which denote the
initiation and termination of a fluent f by an event e at time t respectively; and

(d) The predicate Initially(f) which denotes that a fluent holds at the start of
the operation of a system.

The QoS specification of the MTTR of a service _Srv in EC-Assertion is shown in
Table 5. The formulas in the table check whether the MTTR is always below a
given threshold K. More specifically, the rule R1 checks for the MTTR condition
violations when a call of operation _O in service _Srv is served after a period of
service unavailability. The first two conditions in the rule (see Happens
predicates) check whether the operation call has been served, i.e., whether the
service has produced a response to the call within \(d\) time units. The third
condition of R1 (cf. predicate HoldsAt(Unavailable(_PeriodNumber,
_Srv,_STime),t)) checks whether the served operation call happened at a time
when the service has been unavailable, and the fourth condition establishes the
MTTR value at the time of the call.

The assumption R1.A1 in Table 5 initiates the fluent Unavailable(_PeriodNumber+
1,_Srv,t) to represent a period of service unavailability. This fluent is initiated
when a service call occurs (i.e., the call represented by the event _id1) without a
response to it within \(d\) time units, and at the time of the occurrence of the call the
service is not already unavailable (i.e., no fluent of the form Unavailable(_Period-
Number, _Srv, _STime) already holds). The assumption R1.A2 terminates the
fluent that represents a currently active period of service unavailability (i.e., the
fluent Unavailable(_PeriodNumber, _Srv, _STime)), when a served service call
occurs whilst the service is unavailable. The assumption R1.A3 updates the fluent
MTTR(_Srv, _PeriodNumber, _MTTR) that represents the mean length of
consecutive periods of service unavailability (i.e., the value of the variable _MTTR
of the fluent).

A QoS specification must also include a specification of the monitoring pattern
that is used to record the values of the prediction variables, which are used in the
prediction model of the relevant PS but are not required for the pure monitoring
of the property. An example of such a variable is TTF since it is required for
predicting the probability of future MTTR values but it is not necessary for
monitoring MTTR itself. Hence, the QoS specification of MTTR includes a
specification enabling the monitoring of TTF values. Table 6 shows the
assumption used to initiate fluents for keeping TTF values (R1.A4).
Table 5: QoS Specifications of MTTR

Table 6: QoS Specification of MTTR Formulas for TTF

5.4.2 Computation of MTTR/MTTF Prediction Models

The MTTR/MTTF prediction models are computed by EVEREST+ dynamically at runtime based on the predictor identified in their PSs. This predictor is identified as MT_SV in Figure 52. MT_SV is a parametric predictor realising the following formula:

\[
MTTR_{\text{new}} = \frac{\text{MTTR} \times \text{PeriodNumber} - 1 + (t_1 - STime)}{\text{PeriodNumber}}
\]
\[
Pr\left( \bigwedge_{y=1}^{M} E'_{y} \right) = \begin{cases} 
1 - \sum_{y=1}^{M} Pr(y) \times Pr(MT \ast y \bigcirc MT \ast_{crit}) & , \quad MT \ast_{c} \bigcirc K \\
\sum_{y=1}^{M} Pr(y) \times Pr(MT \ast y \bigcirc MT \ast_{crit}) & , \quad MT \ast_{c} \bigcirc K 
\end{cases}
\] (10)

(10) is a parametric form of formulas (3) and (9) for estimating the probability of a QoS constraint of the form \( MT \ast \bigcirc K \) where \( MT \ast \) is the mean time variable of interest, \( y \) is a parameter affecting it, \( \bigcirc \) is the relational operation used in the constraint (e.g., \(<, >, \leq, \geq \)), \( \bigcirc \) is the negation of this operation, \( MT \ast_{c} \) is the value of the mean time variable at the time of the prediction request, and \( MT \ast_{crit} \) is the critical boundary value that is determined by the different values of \( y \) \( (MT\ast_{crit} = (K \times (N + y) - N \times MT\ast_{c})/y \) as discussed earlier).

When a prediction request is received, EVEREST+ determines the PDF of each of the prediction variables in the PS specifications of MTTR and MTTF (i.e., MTTR, TTF for MTTR and MTTF, TTF for MTTF). This is based on computing the parameters of the alternative PDFs in its built-in PDF function set, and then selecting the one that has the best fit with the last N recorded values of each of these prediction variables (N is the value of the history.window.size variable in the prediction specification). The fit of each of the built-in PDFs is measured by the non-parametric Kolmogorov-Smirnov (K-S) goodness-of-fit test, and the probability distribution that has the smallest goodness-of-fit (GoF) value according to the test is selected for each PS variable.

EVEREST+ has built-in implementations of 43 continuous PDFs. The set of these functions is extensible provided that new implementations adhere to a fixed interface required by EVEREST+.

5.5 Experimental Results

5.5.1 Evaluation of Precision and Recall

To evaluate the precision and recall of the MTTR and MTTF predictor models, we used monitoring data generated from the invocation of the Yahoo WebSearchService. Through a Java client that we developed to invoke this service, we collected a total of 5500 invocation and 5500 response events. The service response time in these invocations varied between 800 and 5251 milliseconds (ms) with an average of 1146 ms. In the experiments, we considered all invocations with a response time of more than 1000 ms as “non-served” service calls (“failures”). Based on this filtering, we obtained 1075 “non-served” service operation calls and used them to compute MTTR, MTTF, TTR and TTF values.

The total time range of the 5500 invocations was divided in 9 sub-ranges of equal distance and for each of them we computed the MTTR\(_c\) and MTTF\(_c\) values for the end of the sub-range. We also used five different QoS constraints for MTTR and MTTF, based on different K values. The K values were determined by the MTTR and MTTF values at the end time point t\(_c\) of each of the nine sub-ranges as: 0.75 x MTTR\(_c\), MTTR\(_c\)+1, MTTR\(_c\), MTTR\(_c\)+1, 1.25 x MTTR\(_c\). For each K, we generated predictions using combinations of different prediction window sizes (i.e., 1, 10, 60 and 600 seconds) and different history sizes (i.e., 100, 300 and 500 data points). Hence, we carried out 540 predictions for each of MTTR and MTTF. The precision and recall of these predictions were measured by the following formulas:
Precision = (TP + FN)/(TP + FP + TN + FN) \quad (10)
Recall = TP/(TP + TN) \quad (11)

In these formulas, TP is the number of true (i.e., correct) positive predictions of QoS constraint violations; FP is the number of false (i.e., incorrect) predictions of QoS constraint violations; TN is the number of true predictions of QoS constraint satisfaction, and FN is the number of false predictions of QoS constraint satisfaction. The criteria for classifying a prediction as a TP, FP, TN or FN are summarized in Table 7.

<table>
<thead>
<tr>
<th>True:</th>
<th>Prob of QoS constraint violation ( \geq 0.5 )</th>
<th>Negative:</th>
<th>Prob of QoS constraint violation &lt; 0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>QoS constraint violated at: ( t_c + p )</td>
<td>TP</td>
<td>TN</td>
<td></td>
</tr>
<tr>
<td>QoS constraint satisfied at: ( t_c + p )</td>
<td>FP</td>
<td>FN</td>
<td></td>
</tr>
</tbody>
</table>

**Table 7: Criteria for TP, FP, TN and FN Predictions**

We also investigated the effect on precision and recall of: (a) the size of the historic event set (HS) that was used to generate the QoS prediction model, (b) the size of the prediction window (PW), and (c) the K-S goodness of fit measure (GoF) of the probability distribution functions that underpin the prediction model.

<table>
<thead>
<tr>
<th>MTTR</th>
<th>MTTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>Recall</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prediction window (secs)</th>
<th>1</th>
<th>0.96</th>
<th>0.94</th>
<th>0.90</th>
<th>0.93</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>0.81</td>
<td>0.71</td>
<td>0.79</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>0.77</td>
<td>0.61</td>
<td>0.60</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>0.47</td>
<td>0.39</td>
<td>0.56</td>
<td>0.67</td>
</tr>
<tr>
<td>History size (events)</td>
<td>100</td>
<td>0.74</td>
<td>0.67</td>
<td>0.65</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>0.75</td>
<td>0.60</td>
<td>0.72</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.76</td>
<td>0.58</td>
<td>0.77</td>
<td>0.83</td>
</tr>
<tr>
<td>Goodness of fit</td>
<td>0.09</td>
<td>0.70</td>
<td>0.69</td>
<td>0.75</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>0.18</td>
<td>0.77</td>
<td>0.59</td>
<td>0.69</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>0.27</td>
<td>0.76</td>
<td>0.65</td>
<td>0.75</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>0.36</td>
<td>0.73</td>
<td>0.52</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>Overall</td>
<td>74.73</td>
<td>63.18</td>
<td>65.27</td>
<td>69.820</td>
<td></td>
</tr>
</tbody>
</table>

**Table 8: MTTR and MTTF Precision and Recall**

The recorded recall and precision measures for MTTR and MTTF are shown in Table 8, grouped by the history size, the prediction window and the GoF measure for the best underpinning PDF of MTTR and MTTF. As shown in the table, the overall precision and recall of predictions for all different combinations of HS and PW were 0.75, 0.63 for MTTR and 0.65, 0.7 for MTTF, respectively.

Precision and recall improved significantly for both models when considering short prediction periods, raising to 0.96 and 0.94 for MTTF and 0.9, 0.94 for MTTR when the prediction window was set to 1 sec. This was expected since as the
prediction window gets longer, historic data become less indicative of what might happen at the window end.

Also, as shown in Table 8, the precision and recall of the MTTR and MTTF prediction models were not influenced by the history size and the GoF measure. For the history size, this was expected as HS was sufficiently large (1%, 3% and 5% of the total event log). For GoF, the expectation was that a lower GoF would result in higher recall and precision (due to a better fit of the selected PDF to the data) but the experiments did not indicate any effect.

The effect of the prediction window and the history size on the recall and precision of the two models was also tested using a 2-way analysis of variance (ANOVA) for these two factors. ANOVA confirmed that the only statistically significant effect was that of the prediction window on the precision of the MTTR and MTTF predictions. The observed differences in precision for different prediction windows were significant at $\alpha=0.01$ (the $p$-values for MTTR and MTTF precision where 0 and 0.0019, respectively). ANOVA also confirmed that the history size and the interaction between history size and prediction window had no significant effect in the recall and precision of either model.

An evaluation of the runtime prediction of aggregate QoS properties was also carried with respect to the B6 SLA. This evaluation focused on the mean operator response time (i.e., the G3 SLA guarantee term in the reference B6 SLA). The prediction was based on the model specified by formulas (1)-(8) where MTTR was replaced by the mean operator response time (MORT) and TTF was replaced by the time-to-operator-call (TTOC). Consequently, the probability distribution functions deployed by the model were the ones identified experimentally by EVEREST+ for the replacement variables (i.e., MORT and TTOC). The evaluation took into account monitoring events spanning over a period of about 3 hours and was based on predicting violations of the above terms in horizons of 15, 30, 45 and 60 minutes. In the set of monitoring events provided by B6, the precision of the predictions produced by EVEREST+ was 1 (i.e., 100%) across and all prediction horizons.

### 5.5.2 Efficiency

![Figure 53: PDF Inference Time vs. History Event Size](image)

To evaluate the efficiency of the EVEREST+ implementation of the MTTR/MTTF prediction model, we used extended sets of historic events (varying from 50 to 20000 events) and measured the total time that it took to infer the MTTR/MTTF prediction model, i.e., to fit different PDFs to the historic data. Figure 53 shows the time taken by the PDF inference process (i.e., the estimation of the
parameters of each PDF from the historic data set) against different sizes of history event sets.

As expected, the inference process took longer as the history size became larger. However, increments were linear. Also, since the size of the history event set does not affect the precision and recall of predictions, the MTTR/MTTF models can be efficiently inferred and updated from relatively small historic event sets for which the experiments demonstrated an efficient inference process (e.g., the PDF inference tool 217ms for 500 events).

6 Manageability Modelling and Design

6.1 Overview

Manageability Design targets to support additional development steps required for implementing monitoring and control capabilities that are necessary for an SLA-driven management. These additional development steps include (i) the identification of the main system components that need to be managed, (ii) the identification of the basic and correlated run-time measurements necessary to monitor the components’ functional and non-functional behaviours, (iii) the identification of the control operations needed to adjust the components’ behaviours at runtime; (iv) the implementation of a corresponding instrumentation of the service components in terms of sensors and effectors, (v) the configuration of the SLA management framework (at least with SLOs, metrics as well as data computation and monitoring/control rules).

In this chapter, we detail the work on an engineering methodology for manageable service components that extends the initial results obtained during the project’s first two years. In particular, we present the meta-models that support the monitoring and adjustment of key indicators of the managed system, and we present how they can be used to automatically synthesize the instrumentation configurations that will provide intelligent sensors and adjustment capabilities in the system.

6.2 The Management Meta-Models

This section presents the meta-models used to define the elements within a service-composition relevant with respect to the Management perspective – e.g., service monitoring and adjustment. Indeed, the idea is to define at design-time the set of elements that will be needed to successfully manage SLAs at runtime.

The management meta-model is composed of three distinct, but complementary, parts: (i) the System meta-model that allows designers to define the main service components that will need to be managed within the system, (ii) the Data Sampling meta-model that allows designers to define the raw and correlated data that need to be collected to analyze a system’s behavior, and (iii) the Adjustment meta-model that supports the run-time modification of the service-based system. This distinction of three core meta-models substitutes the meta-model organization made within the project’s second year, effectively bring greater clarity to the models.
6.2.1 The System Meta-Model

As defined in year 2 the overall approach relies on the assumption that the system to be managed is modelled by means of SCA. The cornerstone of the management model is therefore the system meta-model. Its goal is to provide a SCA-based overview of the components of the system that need to be managed. It provides a deep-dive into those components whose functionality and non-functional qualities we need to monitor, and to control to satisfy a negotiated SLA.

The system model does not describe the functional properties of the system itself. Rather it represents a complementary and incremental view to the SCA-based functional design.

![Diagram](Image)

**Figure 54: The System Meta-Model**

The system meta-model is not fundamentally different from the meta-model presented at the end of year 2. There have only been slight modifications that were introduced to better support the novelties presented in the Adjustment meta-model. Here we will briefly introduce the System meta-model, focusing on the key abstractions that the reader must understand to correctly interpret the system control meta-models that will be illustrated later in this deliverable.

The main element in the meta-model continues to be the ManagedSystem, which identifies the system under development. It specifies the ServiceComponents that constitute the system, and that represent the main ManagedElements of interest. However, the key abstraction for data collection and adjustment is the notion of ServiceComponentAction. The idea is that the raw data that will be needed at runtime for monitoring are data that are related to component actions. Only by correlating, aggregating, and in general manipulating these raw data is it possible to understand the ServiceComponent’s behaviour. The ServiceComponentAction is also the smallest concept to which we can apply adjustment to shape a ServiceComponent’s behaviour.
ServiceComponentActions implement a ServiceComponent’s UnitOfWorks – i.e., its functional elements. A functional element can be classified either as an InternalAction or an OperationCall. An internal action identifies the elements that internally implement the component’s behavior. In our models we distinguish between internal actions that define a scope (WithScope), and those that do not. An internal action defines a scope if its semantics conceptually introduce a “code” block in which other actions can be placed – e.g., the Loop action can contain other internal actions or operation calls. Other examples are the Sequence, which indicates that internal ServiceComponentActions are executed in a sequence, and the Parallel, which means that the internal ServiceComponentActions are executed in parallel. We also support conditional flows (ConditionalFlow) and loops (Loop). This distinction allows designers to express the need to monitor and analyze a specific action in the context of a specific scope. However, the System meta-model is extensible and further actions can easily be added. OperationCalls are used to specify the elements that concern the component’s interactions with external partner components. Due to the bilateral nature of these interactions, the meta-model distinguishes between provided (ServiceOperationCall) and required (ReferenceOperationCall) operation calls.

Using ServiceComponents and ServiceComponentActions it is possible to define the key elements in the system that need to be managed. In order to bridge these elements with the data that need to be collected and correlated for monitoring, and with the actual adjustment capabilities that can be used, we introduce the notion of ManagedElementProperty. A ManagedElementProperty represents a raw value or event that can contribute to monitoring. In particular we distinguish between execution points, messages, and state values. Execution points are either the StartTime or the EndTime of a specific ServiceComponentAction. They are needed to understand “when” a certain action occurs. Messages can either be IncomingMsg or OutgoingMsg, and they are needed to keep track of “when” a component communicates with another component, an “what” that communication looks like. Values refer to the values that ServiceComponent internal state variables may contain. How these ManagedElementProperties are used in practice will be the focus of the following two sections that concentrate on data collection and adjustment.

### 6.2.2 The Data Collection Meta-Model

The data collection meta-model defines the main abstractions needed to describe the data that needs to be collected at runtime for monitoring purposes. The model is a complete re-envisioning of the RDS (Raw Data Sampling) meta-model presented at the end of the project’s second year. Instead of concentrating simply on raw data and events, we have introduced mechanisms for correlation, effectively merging last year’s RDS meta-model with last year’s KPI meta-model. The idea is to gather under a single meta-model all the elements required to provide monitoring with runtime data that are truly useful in evaluating a system’s behavior. Besides a joint focus on raw and correlated data, we have also introduced new mechanisms for communicating these data. Indeed, we now support both push and pull sampling mechanisms.

The Sampler is the main element in the meta-model. It represents one of the DataSources present within the ManagedSystem. (Notice that the ManagedSystem element has a light grey background to indicate that it is shared with the System meta-model.) A Sampler is configured to be active only within certain time windows. A time window is defined by a start date (startWindow) and an end date (endWindow), and can be repeated using the repetitionUnit and
repetitionValue attributes. For example, a time window can start on Monday at 8AM and end on Monday at 10AM, and be repeated every week (repetitionUnit = week, repetitionValue = 1, startWindow = Monday 08.00AM, endWindow = Monday 10.00AM). If no activation window is specified, the default is to consider the sampler as always active.

Figure 55: The Data Collection Meta-Model

A Sampler can be either a PullSampler or a PushSampler. This distinction classifies samplers by how they communicate their data to a monitoring system. The monitoring system can either “pull” the data from the sampler, or the data can be “pushed” automatically from the sampler to the monitoring system. A PullSampler operates in two separate moments. The first moment coincides with a specific ExecutionPoint in the System. When the system’s execution reaches this point the Sampler collects the datum it is responsible for. The second moment is when the sampler is activated by the monitoring system, which requests the last datum that the Sampler collected. PullSamplers can be further classified as SystemPullSamplers or ContextPullSamplers. The former collect a datum from the system’s internal state (i.e., a Value); while the latter collect a ContextProperty, that is a datum that is external to the system’s definition itself. This is useful when the system’s behaviour is intrinsically related to some context information.

PushSamplers are divided into three main categories: InterruptSamplers, PollingSamplers, and Aggregates. InterruptSamplers “activate at” a given ExecutionPoint, meaning that when the system reaches that specific point in its execution it is momentarily stopped so that a specific datum can be sampled. Once the datum has been collected the system is free to resume its execution. The datum can pertain to the System being executed, and in this case we speak of SystemInterruptSamplers, or to the context of execution, and in this case we speak of ContextInterruptSamplers. As soon as the datum is sampled it is packaged into an event and pushed towards the monitoring system.
In the case of SystemInterruptSamplers we collect a value from the system’s internal state, an IncomingMsg, or an OutgoingMsg. Notice that the activation point must be defined in such a way as to ensure that the desired datum has been correctly initialised at the time of collection. To simplify this we require that, in the case of incoming and outgoing messages, the specified activation point refer to the message exchange’s corresponding ReferenceOperationCall or ServiceOperationCall. Finally we can also collect an ExecutionPoint itself. In this case the datum we push to the monitoring system consists of a timestamp and the information needed to uniquely identify that execution point.

In the case of ContextInterruptSamplers we collect a ContextProperty’s value. Keep in mind that the moment in which the data collection continues to be determined by the system’s execution.

PollingSamplers check a property and provide it for processing periodically. In the case of SystemPollingSamplers we only support the polling of values of the system’s internal state. In the case of ContextPollingSamplers we support the collection of ContextProperties. PollingSamplers support a notion of interval to limit their outputs – e.g., output once every x minutes.

Finally Aggregates were introduced to support the correlation, aggregation, and in general manipulation of data with the goal of providing more interesting information to the monitoring system. An Aggregate is a data manipulator that samples information streams being pushed out of two or more existing PushSamplers. It receives data from these streams, processes them, and then pushes the results out, effectively behaving like a PushSampler itself. This mechanism allows designers to combine different PushSamplers in a pipe-and-filter fashion to achieve more and more complex data correlation and manipulation. In our meta-model we currently support three kinds of pre-defined aggregates: Avg Response Time, Reliability, and Rate. The AvgResponseTime Aggregate computes the time elapsed between two time instants (e.g., the end time of a service operation call and its corresponding start time), the Reliability Aggregate computes the number of correct interactions with a service over the total number of interactions attempted, and the Rate Aggregate calculates the rate of arrival of a certain event. When using an Aggregate the designer must also state (i) over how much time it has to be calculated, and (ii) how often its output value needs to be made available. For example, the designer might want to calculate a service’s reliability considering the last 12 hours, and output a new value every 5 minutes. This is achieved through its periodUnit and periodValue, and outputUnit and outputValue attributes respectively.

In the meta-model we also support the specification of new kinds of Aggregate Push Samplers through the abstract notion of DomainSpecificAggregates. A DomainSpecificAggregate defines how the data it reads are correlated and manipulated through a statement property expressed using the Esper Processing Language (EPL). EPL is a well-known language for defining Complex Event Processing. In the EPL statement we refer to the incoming event streams using the name property of the corresponding PushSampler in the model.

6.2.3 The Adjustment Meta-Model

The Adjustment Meta-Model is used to specify the Adjustment capabilities that can be taken on the designed system. This meta-model completely redefines the corresponding meta-model presented at the end of the project’s second year in which we only concentrated on providing dynamic binding. We now support a
much wider variety of adjustment actions, and we have also redefined how dynamic binding is modelled.

The Adjustment element is the main abstraction in the meta-model. An Adjustment is any kind of action that can be taken on the System to impact its behaviour. The meta-model distinguishes between three different kinds of adjustments: SimpleAdjustments, InstanceAdjustments, and ClassAdjustments.

SimpleAdjustments consist of actions that do not attempt to impact the System being run and its behaviour. We support three different kinds of actions. The first is to simply Halt the System; the second is to Ignore that there has been any sort of problem; and the third is to notify a System administrator with an appropriate email. Instance Adjustments are all those actions that attempt to modify the behaviour of a single instance of the running System. We support three main kinds of actions: Dynamic Binding, Service Calls, and Instance Model Adjustment.

![Figure 56: The Adjustment Meta-Model](image)

Dynamic Binding extends the work presented at the end of the project’s second year with support for dynamic binding to multiple endpoints. There are therefore three different kinds of dynamic bindings that can be defined in the model. In general all three Dynamic bindings are applied to the binding that ties a service’s ReferenceOperationCall and a ServiceOperationCall provided by a partner service. (Notice that the ReferenceOperationCall and ServiceOperationCall elements and their Binding have a light grey background to indicate that they are shared with the other models.)

Simple Binding provides a way to statically define an alternative for a binding. This is useful when we have a preferred service we want to use in the System but we already have knowledge of acceptable alternatives we can switch to should there be problems. In this case it is sufficient to know the URI of the new service.

Multi Binding provides a way to statically define a set of bindings the system can switch over to. In this case the System will bind to and call all the services in the set. The results of these invocations are then joined and passed back to the System. In this case it is sufficient to know the URIs of the new services.
Finally we have Search Binding. Search Binding allows us to switch over to a service, or a set of services, we do not already know about at design time. In this case we want to model the strategy that will be used to find which services to use starting from a set of candidate services that may be extracted at runtime from a repository. The Search Binding will effectively use one or more alternative services depending on the its simple and amount attributes. If the simple attribute’s boolean value is set to true only one resulting alternative service will be considered. If is set to false the new binding will consider as many alternative services as defined in the amount attribute.

The first possibility is to use a WeightedSelectionStrategy, which in turn can be of one of two StrategyTypes: it can either try to minimize a cost value or it can try to maximize a profit value. In both cases the value to minimize or maximize is determined as the sum of a set of Weighted Properties. Each Weighted Property consists of a name and a namespace, of the weight it carries in the particular Selection Strategy, and of an actual Property definition. The Property is defined by specifying the name and namespace of a specific quality value, a FilterType (equals, lessThan, or greaterThan) and a threshold value. For example, the property definition may state that the service’s average response time (name = AvgResponseTime, namespace = http://sla_at_soi.org) should be less than 500 ms. If a candidate service satisfies this property we assign it a positive weight value corresponding to the weight attribute defined in the WeightedProperty element.

A second possibility is to use a SLAT-based Strategy. In this case the new service is chosen based on a specified SLAT. The SLAT is used as a query on services. Once again this can be used to search for a single service or for a set of services, depending on the values of the simple and amount attributes.

Another Instance Adjustment consists in a Call to a ServiceOperationCall that is offered by the service we want to adjust. This is useful when the service provider also provides management or configuration methods for the service being offered. Finally, the last Instance Adjustment we support is the Instance Model Adjustment. This kind of adjustment has a much more profound impact on the service instance we are trying to modify. Through model adjustment it is possible to alter the Data Collection (DC) model associated with the System, the Adjustment Model associated with the System, and the System model itself. In the case of a change to the data collection model we need to provide the new set of DataSource to be used. In the case of a change to the adjustment model we need to provide the new set of Adjustments. In the case of a change to the System model itself we can add or remove a ServiceComponentAction, or substitute a ServiceComponentAction with a new one. Model Adjustments are also tied to ExecutionPoints in two ways. First we need to define an activation point, i.e., the point in the system’s execution in which we activate the adjustment. Second we need to define an actual enactment point, i.e., a successive point in the system’s execution in which the new model is actually put into service.

As a third macro-area of Adjustment we support Class Adjustments. A Class Adjustment consists of a change in the data collection model, in the adjustment model, or in the System model itself. These adjustments are also subject to the definition of activation and enactment points. The only difference is that the change applies to all the running instances of the System, and to all the future instances as well.

Before we conclude we must also state that the Adjustment Meta-model allows for the definition of Adjustment Strategies. An Adjustment Strategy consists of a set of Adjustment Actions that are to be taken together. Incompatible actions,
such as a System Model Change that removes a certain ReferenceOperationCall and SimpleBinding that refers to that same ReferenceOperationCall, should not be placed together. We leave this up to the designer. At runtime the System can take the decision to enact any of the defined strategies, confident that the runtime support will be in place.

### 6.3 Synthesis of Data Collection

In the second year’s deliverable we discussed how to achieve the automatic synthesis of sensor instrumentations in BPEL processes. We will briefly go over how data collection configurations are synthesized, since we have introduced some novelties in this year’s models.

#### 6.3.1 Synthesis of Sensors

Figure 57 illustrates the Sensor Instrumentation Model for BPEL processes. The main element is the SensorConfiguration. It associates a ContextSensor or a SystemSensor to a specific BPEL process identified by a unique id. All sensors collect their data when the system’s execution reaches a certain ActivationLocation, identified by an operationID that uniquely refers to a BPEL activity within the process, and a precede boolean value which tells the system if the data has to be collected before or after that BPEL activity is executed.

![Figure 57: Sensor Instrumentation Model](image)

A ContextSensor is a sensor that will collect information from the context given a URI of the external data source, and a namespace and name that identify the desired context property. A SystemSensor is a sensor that will collect a variable from the BPEL process’ internal state, which can also mean an incoming or outgoing message. If no variable name is given, the sensor will only produce a timestamp and an indication of which BPEL activity has or was about to be executed. Once collection is complete our Sensors both push the data to an event bus, and keep it to provide pull support. Finally, our sensors also provide User Correlation so that user-specific sensors are activated depending on who is executing the BPEL process, and a Validity for indicating when and how often the sensor is to be considered active and not dormant.

The transformation needed to produce a valid Sensor Configuration is quite straightforward and is exemplified using QVT for an InterruptSampler in Figure 58. The InterruptSensorConfiguration takes its processID from the name of the ServiceComponent on the left hand side of the transformation, which comes from the System model. Since we only need the call’s StartTime we put the InterruptSensorConfiguration’s varName to null. The Sensor’s ActivationLocation is defined by an operationID that is constructed at the bottom of the QVT transformation using the name of the ServiceOperationCall of which we want to collect the start time, and by the precede value being put to false. This means we
want to intercept the process immediately after it has sent the outgoing message.

The Sensor’s Validity values are taken straightforward from the InterruptSampler’s startWindow, endWindow, repetitionUnit, and repetitionValue. It is not possible to automatically extract User Correlation values for the sensor. The designer must provide these manually.

![Figure 58: QVT Transformation of an Interrupt Sampler](image)

### 6.3.2 Synthesis of Correlation

Aggregation and correlation are supported in year three through an extension to the SLA@SOI framework’s notion of Manageability Agent, which can now subscribe to data flowing through the event bus, process them, and then publish the results back to the event bus. The plug-in consists of an Esper event processor, capable of interpreting EPL statements. We support Reliability, Avg Response Time and Rate correlations out of the box, each with its own EPL statement. Due to lack of space we shall only detail how the calculation of the Reliability of a given ServiceOperationCall.

All we need to collect are the StartTime and the EndTime of calls made to the operation. Each StartTime and EndTime contains the following information: an originID that refers to the DataSource producing the information, an instanceID that uniquely identifies the call made to the service, and a timestamp. From the model we will also use the periodUnit and periodValue to determine how much time we will consider in the calculation of the Reliability (e.g., 12 hours), and the outputUnit and outputValue to determine how often a new Reliability value will be calculated (e.g., every 5 minutes).

Reliability is generically defined as $R = 1 - \frac{Total_f}{Total_r}$, where Total$_f$ is the amount of failures associated with the operation in a given amount of time, and Total$_r$ is the total number of requests made to the operation in that same time frame. The Reliability is therefore implemented using two EPL statements, one for Total$_f$, and one for Total$_r$. Both use a 15 seconds time-window, i.e., a sliding window that extends 15 seconds into the past to limit the events they need to keep track of. The 15 seconds represent the amount of time we are willing to wait for a response from the service before we decide that it failed. The first EPL statement is used to calculate the total number of requests:
(1) SELECT * FROM StartTime.win:time(15sec)

An Esper listener receives new data as soon as the engine processes events for such statement. Its reaction is to accumulate the number of events that occurred within the window, and update the Total_r value calculated over the last 12 hours.

The second EPL statement is used to calculate the total number of failures that occur:

(1) SELECT rstream * FROM StartTime.win:time(15 sec) ST
(2) FULL OUTER JOIN EndTime.win:time(15 sec) ET
(3) ON ST.instanceID = ET.instanceID
(4) WHERE ET.callID is null

Specifically, we define a failure as the incapability to match two corresponding StartTime and EndTime events within a 15 second time window. The EPL statement uses the "rstream" keyword to select events from the remove stream. This means it would tend to notify its listener every time an event exits the time window. However, it should only notify its listener if a StartTime is exiting and no corresponding EndTime can be found. The fact that an EndTime is missing is discovered thanks to the outer join (line 2) performed on the instanceID attribute of both events. In Esper, if an outer join cannot make a match (line 3), a result is presented nevertheless, and the default value null is given in place of the missing attributes (line 4). The listener, once again, simply counts the number of notifications it receives, and updates the Total_f value calculated over the last 12 hours.

To conclude, the two listeners collaborate once every 5 minutes to calculate the new reliability, and update their Total_r and Total_f values to ensure the next reliability value will not consider requests or failures that have occurred more than 12 hours ago. As soon as a new reliability value is available it is disseminated to the event bus.

6.4 Synthesis of Adjustment

Regarding the automatic synthesis of adjustment configurations we currently support the automatic synthesis of static Simple Binding and Multiple Binding in BPEL processes, as well as the automatic synthesis of changes to the System and Data Collection models in BPEL processes, both in the case of instance model modification and in the case of class model modification.

6.4.1 Synthesis of Binding Rules

Figure 59 illustrates the model behind the binding rules supported by the Dynamic Orchestration Engine described in SLA@SOI’s A3 workpackage. The model supports two kinds of Binding Rules: Concrete Binding Rules and Abstract Binding Rules.

A Concrete Binding Rule states the process to which it refers (processID) and a User Correlation to define when the rule has to be taken into consideration. Each concrete binding rule then defines its type, which can be simple or multiple. If we are dealing with a simple binding the rule contains a single endpoint reference that points to the new and alternative service. If we are dealing with a multiple binding the rule contains a set of endpoint references and a Variable that indicates where the endpoints are saved to within the process’ internal state. An
Abstract Binding Rule is similar, except that it never contains references to concrete endpoints. Instead it refers to a SLAT that can be used to search for a list of services that can be used.

The way we support the automatic synthesis of Simple Binding and Multi Bindings from an Adjustment model is to support the automatic creation of Binding Rules. The rules can then be loaded into the binding rules database of the DOE at runtime to enable them.

Figure 59: Dynamic Binding in the Dynamic Orchestration Engine

Figure 60: QVT Translation of a Simple Binding

Figure 60 illustrates how we adopt QVT to automatically translate a Simple Binding specification in the Adjustment model to a coherent Binding Rule that can be added to the DOE. Other translations are achieved similarly.

The Binding Rule’s processID is taken from the ServiceComponent specified in the System model, and its type is automatically configured to Simple. The designated endpoint is taken from the Simple Binding in the Adjustment part of the model and is placed as the requested URI in the Binding Rule. Starting from our models it is impossible to fill out the Correlation set needed buy the Binding Rule. The Binding Rule has to be completed by hand.
6.4.2 Synthesis of Model Changes

The automatic synthesis of DataCollection Model Changes consists simply in the automatic synthesis of new Sensor configurations. Since this has been previously discussed we will not treat it here. Instead we will focus on discussing the automatic synthesis of System model changes.

![Diagram](image)

**Figure 61: Model of the BPEL Modification Rule**

Figure 61 illustrates the model behind the definition of BPEL Modification rules. A modification rule identifies the process it needs to be associated with thanks to the processID attribute. The modification can be of two types: either a Class modification, and in this case the processID is sufficient, or an Instance modification. In this case to identify the instance the rule makes use of a User Correlation definition, which is checked against the running instances.

Each rule has an ActivationPoint and an EnactmentPoint. The ActivationPoint identifies starting from when the modification should be considered. It contains an XPATH expression that identifies a specific point in the process. The EnactmentPoint identifies where exactly the new modification needs to take place.

There are different possible behaviours depending on what kind of ModificationAction is being requested. If we are requesting the addition of new behaviour, the new behaviour, expressed in the ActivityCode in terms of a BPEL snippet, is added immediately after the EnactmentPoint’s execution. If we are requesting the removal of a behaviour, we identify the activity to be removed through the EnactmentPoint. Moreover, in this case the rule does not contain any new ActivityCode. Finally, if we are requesting a substitution the EnactmentPoint identifies the BPEL activity to be replaced, while the ActivityCode identifies the new behaviour to be executed in its place.

The actual synthesis of a BPEL ModificationRule is not as simple as the previous syntheses we have shown. Although the synthesis of the ActivationPoint; the EnactmentPoint; and the processID, type, and action attributes are straightforward, the synthesis of the ActivityCode is not. In practice we currently require that a human expert look at the new ServiceComponentAction that needs to be added and implement a coherent snippet of BPEL code, one that will not conflict with the already existing process structure. Obviously this problem does not occur when we are only performing a removal, since in that case no ActivityCode is needed.
6.5 Example Models

Figure 62: The SCA Model of the eGovernment Use Case

Figure 63: System and Data Collection Models of the eGovernment Use Case
The examples shown here relate to the project’s eGovernment Use-Case. Figure 62 illustrates the SCA model of the use case, which is implemented as a process composition that orchestrates four different parties.

In this use case the “Provincia Autonoma di Trento” (PAT) wants to be able to keep track of a number of indicators: the response time for the Citizen Service Center Process, the response time for the MobilityProvider when the process attempts to book mobility, and the response time for the Health Care Structure when the process attempts to book a health treatment. The required System and Data Collection models are shown in Figure 63.

In all three cases the raw data we need are the start and end times for each invocation of the corresponding ServiceOperationCall. For both of these ManagedElementProperties we design an InterruptSampler (STSampler and ETSampler respectively) with an interval value of 1. This means that every time there is a new execution, the corresponding start or end time will be captured and provided for further processing.

The outputs of the STSampler and the ETSampler are passed to the ResponseTime Aggregator. The ResponseTime Aggregator does not require any additional attribute configurations. Its default semantics is to produce all the response times it is capable of calculating by correlating the input data it receives. The response times are then passed to the Event Bus so that they can be handed over to the Monitoring System.

Our tool automatically produces the Instrumentation configurations required by the Dynamic Orcherstration Engine to capture the start and end time of each of these ServiceOperationCalls. The instrumentation configuration that is created in the case of the overall Citizen Service Center Process is:

```xml
<def>
  <workflow>
    <eventType value="StartTime"/>
    <processID value="CitizenServiceCenter"/>
    <operation value="/process/flow/receive "/>
    <validity_from value="2011-06-14T09:00:00.000+02:00"/>
    <validity_to value="2011-06-14T10:00:00.000+02:00"/>
    <validity_repUnit value="Week"/>
    <validity_repValue value="1"/>
    <correlationKey value=" null "/>
    <correlationValue value=" null "/>
  </workflow>
  <workflow>
    <eventType value="EndTime"/>
    <processID value=" CitizenServiceCenter "/>
    <operation value="/process/flow/reply="/>
    <validity_from value="2011-06-14T09:00:00.000+02:00"/>
    <validity_to value="2011-06-14T10:00:00.000+02:00"/>
    <validity_repUnit value="Week"/>
    <validity_repValue value="1"/>
    <correlationKey value="null"/>
    <correlationValue value=" null "/>
  </workflow>
</def>
```
The specification configures two sensors: one for the start time and one for the end time. Both of them are attached to the CitizenServiceCenter BPEL process. The start time sensor is attached to the BPEL receive activity that initiates a new process instance, while the end time sensor is attached to the BPEL reply activity that concludes the instance. Both are defined with an activation window that starts Monday morning at 9 o’clock and concludes at 10 o’clock, and repeats once a week. Finally, the correlationKey and correlationValue attributes are not used meaning that the sensors are active on all process instances.

In order to be able to cope with low system performances, the designers have defined a second System model they can switch over to. The second system model costs more so it was not chosen as the main design. This is shown in Figure 64.

**Figure 64: The Adjustment Model for the eGovernment Use Case**

In the original System model the Citizen Service Center consists of a ServiceComponent containing four main steps represented by Sequences. They are the BookTreatment, the AppointmentSearch, the MobilitySearch, and the PhoneCallback. In the Adjustment model we define an adjustment strategy called Parallelize, made up of three Instance Model Adjustments. They all activate when the process reaches the EndTime execution point of the BookTreatment Sequence. The first RemoveInstanceServiceComponentAction is enated at the StartTime of Sequence AppointmentSearch, and removes the AppointmentSearch Sequence itself. The second RemoveInstanceServiceComponentAction is enacted at the same execution point and removes the MobilitySearch sequence. Finally, we have an AddInstanceServiceComponentAction called AddParallel, enacted once again in the same execution point, that adds a new ServiceComponentAction called AppointmentSearchParallel. The ServiceComponentAction contains the same two component actions that were removed, yet now they are configured to execute in parallel.

The synthesis of this model produces three modification rules that need to be executed. The rules are similar since they all have the same activation and enactment points, which are identified using XPATH expressions that uniquely identify the corresponding BPEL scopes within the process. Recall that the activity code corresponding to the new parallelized version of the process must be defined manually by a domain-expert. We do not currently support the automatic synthesis of this code.
7 Conclusions

This document serves as deliverable D.A6a and has reported on the progress of the A6 work package over the lifetime of the European research project SLA@SOI. WP A6 is devoted to the top-level SLA@SOI objective to advance the engineering of predictable service-oriented systems by methodologies, modelling techniques, and prediction tools covering SOA and SOI components. To this end, A6 has provided four main contributions, namely (i) software performance and reliability prediction, (ii) resource usage prediction, (iii) run-time SLA violation prediction and (iv) manageability modelling and design. This chapter provides some concluding thoughts about the WP A6 achievements (Section 7.1), lessons learned (Section 7.2) and directions for future work (Section 7.3).
7.1 Contributions and Achievements

The main achievement of this work package, as already stated in Section 1.1, is a comprehensive set of methodologies and tools enabling the engineering of predictable systems, allowing service and infrastructure providers to make well-informed decisions throughout the stages of the SLA and service lifecycles. WP A6 has investigated different dimensions, scopes, and goals of predictability, such as design-time prediction versus run-time prediction, infrastructure-level prediction versus service-level prediction, and prediction of quality metrics versus prediction of rule violations. It has turned out that each of these areas has its own specific requirements and challenges, and a corresponding reshaping of the original work package goals was necessary after the first project year (see DoW Amendment 2). As a result, a clear formulation of the four main WP A6 contributions as stated above was found, so that together, they constitute a comprehensive solution.

Through the work done in WP A6, software service providers can create feasible service offers in terms of SLA templates at the service offering stage, using the software performance and reliability prediction. During service negotiation, the prediction is used once more to support the service providers in finding adequate responses to concrete SLA requests. At the service provisioning stage, the resource usage prediction supports infrastructure providers in using their resources efficiently. During the operation of services, the run-time SLA violation prediction is continuously performed based on monitoring data, to help service providers avoid SLA violations. The required manageability features for the services under study are defined at the service design stage through manageability modelling and design. All these contributions include the conceptual methodology, as well as concrete tool support as part of the SLA management framework and further stand-alone tooling. They have been successfully applied to the industrial use cases of the project, and scientifically proven through a series of peer-reviewed publications [197-208].

7.2 Lessons learned

Beyond the targeted methodologies and tools, the results of WP A6 include valuable experiences which can be seen as lessons learned in the field of predictable systems engineering.

Lesson 1: Predictability is an essential ingredient to successful SLA management.

At the planning stage of the project, predictable systems engineering was set as a top-level goal assuming that it is essential for a successful SLA management. This assumption has turned out to be fully true – the predictive evaluation of services is an integral part of the SLA negotiation process, and a corresponding top-level component “Service Evaluation” takes care of such evaluation in the SLA management framework. Further predictive capabilities are part of other framework components, such as the SLA violation prediction that is integrated into run-time monitoring. Overall, SLAs can only be reliably negotiated, enforced and promoted by applying tools and methodologies related to predictability.

Lesson 2: The wide variety of goals and scopes related to predictability is covered best by a flexible set of tools.

During the 3 years of the project’s lifetime, the A6 work package went through a major reshaping and further minor adjustments, which became manifest in the DoW Amendments. While originally, the focus was more on a uniform view on the predictable systems engineering, which should be supported through holistic
prediction services, it turned out that several different perspectives on the problem have to be considered (as stated above), and that it is preferable to provide a set of related, but independently usable methodologies and tools. A major effort in the first project year went into the process of harmonizing the perspectives of all involved partners, as well as formulating a set of four main contributions that complement each other to form a comprehensive overall solution.

Lesson 3: The provided solutions have to be differentiated according to the specific properties of individual application scenarios.

A further challenge became evident regarding the provision of very concrete solutions and tool support for application scenarios as generic as possible. It turned out that a single solution is generally limited in its capacity to support all kinds of service-based systems and related SLA scenarios. For example, the high computational overhead of the software performance prediction as foreseen and developed in the first two project years may limit its use if a service-based system features a complex architecture and a high number of possible service configurations. Thus, alternative strategies had to be evaluated allowing for a flexible response to the varying architectural complexity. Through this kind of flexibility, the range of applicability of the WP A6 solutions could be significantly extended, which was demonstrated in the context of the project’s industrial use cases.

Lesson 4: Possibilities to transfer solutions to new scenarios may exist and should be investigated.

A particular interesting insight was gained through the integration of the different views of the involved partners: Some approaches showed potential for application scenarios beyond their originally intended scope. For example, it turned out that the software and performance prediction, which was intended for software services only, could be expanded to the field of human services and resources, as demonstrated in the eGovernment use case. In this respect, truly new perspectives on known problem fields and new potential for future work could be unlocked through the work done in SLA@SOI. Possibilities of such transferred application scenarios should be explicitly considered in future projects.

7.3 Outlook

Although WP A6 has substantially contributed to the field of predictable systems engineering, potential for further extensions and enhancements remains. In some areas, the methodologies developed by WP A6 are more general than the actually implemented solutions. For example, while software performance and reliability prediction provides a means to evaluate software services, similar solutions are desirable for the business layer and the specific business level services. Although the concept of the Service Evaluation component in the architecture of the SLA management framework is generic with respect to the service layer, a concrete solution was only realized for the software layer. Other possible extensions lie in the predictive evaluation of further quality attributes that were not in the central focus of WP A6, such as service security or interoperability.

Regarding the existing solutions, WP A6 partners will keep being involved in their further evolution, in particular in the context of the SLA management framework, which has been released as an open source project, providing a basis for collaborative enhancement by all interested parties.
8 References


[157] Werner Vogels, Eventually Consistent, Queue, v.6 n.6, October 2008


**Appendix A: Standard QoS Terms for Prediction**

The software performance and reliability prediction provided by WP A6 relates to the standard QoS terms of the SLA model, which are negotiated in software and infrastructure SLAs. The mapping of prediction inputs and outputs to standard QoS terms is given by Table 9 and Table 10.

<table>
<thead>
<tr>
<th>Prediction Output</th>
<th>Specified for:</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| qos:completion_time (MESSAGE_TYPE_SERVICE) | target software service | • performance metrics  
• obtained through simulation  
• corresponds to “response time” in prediction tool  
• available in year 2 |
| qos:throughput (MESSAGE_TYPE_SERVICE) | target software service | • performance metrics  
• obtained through simulation  
• can be indirectly deduced from prediction results when a single usage scenario and a closed workload was used in the prediction model (divide total simulated time through total number of response time measurements for the usage scenario)  
• available in year 2 |
| qos:reliability (MESSAGE_TYPE_SERVICE) | target software service | • reliability metrics  
• obtained through analytical calculation  
• is the direct result of reliability prediction  
• available in year 3 |

**Table 9: Software Performance and Reliability Prediction Outputs**

<table>
<thead>
<tr>
<th>Prediction Input</th>
<th>Specified for:</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| qos:arrival_rate (MESSAGE_TYPE_SERVICE) | target software service | • needed as an input for performance prediction  
• used for specification of inter-arrival time in an (open workload) usage scenario |
| qos:data_volume (MESSAGE_TYPE_SERVICE) | target software service | • may be needed as an input for performance and reliability prediction  
• used for specification of input parameter values in usage scenarios, if those values impact service performance and / or reliability |
| qos:completion_time (MESSAGE_TYPE_SERVICE) | external software | • may be needed as an input for performance prediction |
| qos:throughput  | external software services | • may be needed as an input for performance prediction  
• used for validity check of prediction scenarios, if system-external service calls exist in the model (a check is needed if the actual arrival rate of an external service is within the allowed boundary specified by the throughput term of that service) |
|----------|----------------|--------------------------------------------------|
| qos:reliability | external software services | • may be needed as an input for reliability prediction  
• used to specify the failure rates of system-external service calls, if such calls exist in the model |
| qos:mttf    | external infrastructure services | • needed as an input for reliability prediction  
• used to specify the MTTF of physical resources (CPUs) in the model |
| qos:mttr    | external infrastructure services | • needed as an input for reliability prediction  
• used to specify the MTTR of physical resources (CPUs) in the model |
| infra:CPU_Cores  | external infrastructure services | • needed as an input for performance prediction  
• used to specify the processing speed of physical resources (CPUs) in the model (together with CPU_Speed and Memory) |
| infra:CPU_Speed | external infrastructure services | • needed as an input for performance prediction  
• used to specify the processing speed of physical resources (CPUs) in the model (together with CPU_Cores and Memory) |
| infra:Memory | external infrastructure services | • needed as an input for performance prediction  
• used to specify the processing speed of physical resources (CPUs) in the model (together with CPU_Cores and CPU_Speed) |

**Table 10: Software Performance and Reliability Prediction Inputs**

While prediction results can be used for SLA negotiation for a given target service to evaluate (Table 9), prediction also needs input information as an enabling precondition (Table 10). Without this information, prediction results cannot be accurate, or prediction might not work at all. The needed information differs depending on the concrete scenario. Some information relates to customer obligations regarding the target service (arrival rate, data volume). Other information relates to quality characteristics of external software and/or infrastructure services. Both kinds of information are given as input parameters to prediction - in terms of SLA templates for the target (software) service, as well as external (software / infrastructure) service dependencies.

**Appendix B: QoS Meta-Model**

This Section describes the most important meta classes of the QoS meta-model and their semantics.

**Components and Interfaces**

For the static architecture of the QoS meta-model, we follow the concepts of the Service Component Architecture (SCA) (The OASIS Open CSA, 2008), even if our terminology deviates from the terms used in the SCA specification. Our notation adopts the terminology of the current QoS meta-model elements and gives the terms used in SCA between brackets.
In the QoS meta-model, service components can either be atomic or composed from other service components. In this subsection, we focus on atomic service components and defer composition to the next subsection.

Figure 65 shows the meta-model excerpt for components, interfaces, and roles. Service components and interfaces are first-class entities, since they can exist independently from other entities. **BasicComponents (SCA: Service Components)** model atomic components. **Roles** assign **Interfaces** to service components. Thereby, **ProvidedRoles (SCA: Services)** specify the interfaces that are offered by a component, while **RequiredRoles (SCA: References)** specify those interfaces that allow the component to work properly.

![Figure 65: Components, Interfaces, and Roles](image-url)
**Figure 66: Interface Signatures**

As Figure 66 shows, an Interface contains a list of method Signatures. Each Signature has a list of input Parameters and a return value. Each Parameter, as well as the return value, belongs to a certain DataType, which is either a PrimitiveDataType, a CompositeDataType, or a CollectionDataType. Components and interfaces are stored in a design-time Repository.

**Composite Components and Systems**

CompositeComponents (SCA: Lower-level Composites) assemble service components (basic and composite) into higher-level structures. A CompositeComponent does not contain any own business logic or implementation code itself; rather, it is a structuring element to represent different levels of abstraction. It is important to note that composite components can only be deployed as a whole. Thus, their subcomponents cannot be allocated to different nodes. The concealment of internal structures of composite components currently contradicts the concepts of SCA claiming that each subcomponent can be deployed separately.

At the highest level, Systems (SCA: Higher-level Composite) combine components that represent the modelled software system as a whole. Similar to components, Systems provide Interfaces to their users (either human or other systems) and require other software services specified in terms of required Interfaces (see Figure 67). A System presents a self-contained part of the overall QoS meta-model.
Component Behaviour

For QoS analysis, behavioural models are essential. They describe how services use available hard- and software resources and thus form the basis for performance and reliability analysis. In the context of the proposed QoS meta-model, Resource Demanding Service Effect Specifications (RDSEFFs) model the control and data flow of service components. RDSEFFs specify the behaviour of each method Signature provided by a BasicComponent (see Figure 68).

Being a ResourceDemandingBehaviour, the RDSEFF basically consists of a chain of actions which are either calls to other components (ExternalCallActions) or abstract internal computations (AbstractResourceDemandingActions). In order to assign an ExternalCallAction to a specific method, it references a Signature of a required Interface or the BasicComponent (see Figure 69).

Figure 67: Composite Components and Systems

Figure 68: Service Effect Specifications
An AbstractResourceDemandingAction may specify a resource type and the amount of resource units required. It therefore contains a ParametricResourceDemand that stochastically describes the required amount of resource units. For this purpose a RandomVariable stochastically specifies possible dependencies to input parameters. In the following, we give a brief introduction to such parametric dependencies. Please refer to [1] or [2] for further details.

Parameter dependencies allow for representing methods with input and output parameters, whose values can be characterized with focus on performance-relevant aspects. Possible characterizations include the VALUE, BYTESIZE, NUMBER_OF_ELEMENTS, or TYPE of a parameter. The characterisations can be stochastic, e.g., the byte size of a data container can be specified by a probability mass function. For example:

\[
data\.BYTESIZE = \text{IntPMF}(1000;0.8) \ (2000;0.2)\]

where \text{IntPMF} is a probability mass function over the domain of integers. Here \text{data} has a size of 1000 bytes with probability 0.8 and a size of 2000 with probability 0.2. Stochastic expressions model data flow based on parameter characterisations. For example, the stochastic expression:

\[
\text{result\.BYTESIZE} = \text{data\.BYTESIZE} \ast 0.6
\]

specifies that (for instance) a compression algorithm reduces the size of data to 60%. The expression thus yields:

\[
\text{IntPMF}(600;0.8) \ (1200;0.2).
\]

Stochastic expressions support arithmetic operations (*,+,−,/,... ) as well as logical operations for Boolean type expressions (==,>,<,AND,OR,...) on RandomVariables.

Various specialisations of AbstractResourceDemandingActions (see Figure 70) reflect control flow statements and allow software providers to specify branches, loops, forks etc. That way, performance-relevant and reliability-relevant dependencies between the control flow, external calls, and parameters, can explicitly be expressed in the model. On the other hand, independent component-internal processing can be aggregated into InternalActions, keeping the abstraction from the component’s implementation.
BranchActions represent “exclusive or” splits of the control flow, where only one of the alternatives can be taken. The choice can either be probabilistic or determined by a guard. In the first case, each alternative has an associated probability giving the likelihood of its execution. In the latter case, Boolean expressions on the service’s input parameters guard each alternative. With a stochastic specification of the input parameters provided by the caller, the guards are evaluated to probabilities.

LoopActions model the repetitive execution of a part of the control flow. A probability mass function specifies the number of loop iterations. For example, a loop might execute 5 times with a probability of 0.7 and 10 times with a probability of 0.3. The number of loop iterations can depend on the service’s input parameters. Furthermore, iterations over a collection are also modelled explicitly (CollectionIteratorAction) where the number of repetitions depends on the size of a collection.

ForkActions split the control flow into multiple concurrently executing threads. The control flow of each thread is modelled by a ForkedBehaviour. The main control flow only waits for forked behaviours that are marked as synchronised. Its execution continues as soon as all synchronised ForkedBehaviours finished their execution.

AcquireActions and ReleaseActions model the acquisition and release of limited passive resources, e.g., semaphores or connection pools. Passive resources may have a significant influence on the execution time of a service due to waiting times.

Resource Allocation

A necessary part of the QoS meta-model is the (virtual) resource environment that shall host a service. In order to separate the specification of service components (which require resources) and the infrastructure (which offers resources) both refer to abstract resource types from a global resource repository. The QoS meta-model coarsely distinguishes between processing resource types (e.g., CPU, HD, etc.) and passive resource types (e.g., semaphores etc.).
ResourceEnvironments contain a number of ResourceContainers (called nodes in UML) connected by LinkingResources. ResourceContainers bundle a set of resources. They include ProcessingResourceSpecifications (e.g., a CPU with a processing rate of 1000 work units per second) and PassiveResourceSpecifications (e.g., a data base connection pool with a capacity of 10). A component that is embedded in a specific software architecture (its so-called AssemblyContext) can be allocated to specific resources. The abstract resources referenced by the RDSEFFs included in the AssemblyContext's components can then be substituted by the concrete resources from the ResourceEnvironment to compute actual resource demands.

Within an RDSEFF, ResourceDemandingActions request ProcessingResources (see Figure 69) that always have an associated processing rate. The demand divided by the processing rate yields the processing time of the demand not considering any contention effects.

Figure 72 shows the allocation of a System to a ResourceEnvironment. Each component instance included in the System through an AssemblyContext is mapped to a ResourceContainer by an AllocationContext, representing a deployment instance of the component.

---

**Figure 71: Resource Environments**

**Figure 72: System Allocation**
Usage Profiles

To estimate the expected QoS a priori, a system’s usage is specified in terms of workload (i.e., the number of concurrent users), user behaviour (i.e., the control flow of user system calls), and parameters (i.e., abstract characterisations of the parameter instances users utilise).

Usage models contain multiple UsageScenarios, each of which models a single use case of the system. For each UsageScenario, a Workload describes its usage intensity and a behavioural model (ScenarioBehaviour, see Figure 73) its flow of user actions. The ScenarioBehaviour is analogous to RDSEFFs, but does not contain any resource consumptions.

The Workload, as shown by Figure 74, is either an OpenWorkload or a ClosedWorkload (similar to queueing networks). An OpenWorkload specifies an InterArrivalTime, that is, a time interval between the arrivals of two users. Every user who arrives at the system executes the associated ScenarioBehaviour once and exits. By contrast, a ClosedWorkload specifies a fixed number of users who are always present. Every user executes the associated ScenarioBehaviour infinitely, taking a certain ThinkTime between two executions.

Figure 73: Scenario Behaviours
**Appendix C: Service Evaluation**

**Architecture and Overview**

The ServiceEvaluation component is a top-level component within the SLA@SOI management framework. It is used by SLAManager components at each level (business, software, and infrastructure) to determine a-priori evaluation of service quality parameters (see Figure 75). The results of evaluation are used during negotiation to agree upon feasible terms and conditions regarding service quality. The interaction of ServiceEvaluation with SLAMangers is described as "<<evaluate>>" interaction.

![Figure 75: Interaction of Service Evaluation and SLA Managers](image)

Concrete implementations of the ServiceEvaluation component may completely vary in scope and solution strategy. Regarding scope, each type of service (business, software, and infrastructure) comes with its own specific characteristics and evaluation goals. Regarding solution strategy, very different approaches may be possible, such as (i) interpretation and aggregation of historical data of service quality, (ii) application of rules and constraints to calculate expected quality parameters, and (iii) analysis and / or simulation of
architectural models created during system design. Thus, there is no generic description of implementation of the ServiceEvaluation.

**Software Service Evaluation**

As described above, the general concept of service evaluation targets software, business and infrastructure services and can be extended with specific evaluations for individual scenarios. The sections below are about an implementation realized within the WP A6 work package to determine software service quality through model-based predictions. As an introduction Figure 76 gives an overview of the general interface structure and the default implementation available. As shown, the general service evaluation interface provides the two main evaluation methods

- `evaluate(Set<ServiceBuilder>, SLATemplate)`
- `evaluate(Set<ServiceBuilder>, SLATemplate, EvaluationSettings)`

When the first method is used, a default EvaluationSettings object is used. Details about those settings are provided in Section 'Evaluation Settings'.

![Figure 76: Interface Structure of the Software Service Evaluation](image)

**Software Service Evaluator**

Figure 77 shows the main class to create an instance of to access the service evaluation component. Additional to the methods of the general evaluation interface, that takes SLA templates and service builders as an input, this implementation provides evaluation interfaces with plain java objects. More details about those methods can be found below.

![Figure 77: Main Class to Access the Software Service Evaluation](image)

Internally, the implementation consists of the major sub components Extractor, Repository, ModelAdjustment and Prediction as shown in Figure 78.
Figure 78: Major Components of the Software Service Evaluation

Extractor

The Extractor is responsible to extract prediction parameters from SLA templates and service builders. With a target SLA and a service builder, or a set of service builders, he creates a PredictionModelModification object as described below in the subsection 'Plain Java Interface'. For further details about which SLA and builder format is required, please refer to the section 'Evaluation Request Parameter'.

Repository

When the prediction parameters are in place, the Repository component is triggered to load the prediction model for the requested service implementation. The service builder specifies which prediction model to load (see section 'Service Builder Specification' for details). In any case, a model is identified by a model id. The default implementation of the model repository loads the prediction models from a directory in the file system. Each model is placed in a subdirectory named by the model id. The default path of the repository directory is <ORC_HOME> / common / osgi-config / software-servicemanager / quality_model_repository /.

ModelAdjustment

As soon as the model has successfully been loaded from the repository, the ModelAdjustment is invoked with the PredictionModelModification provided by the Extractor and the model loaded by the repository. The model adjustment now adjusts the prediction model according to the parameters extracted from the provided SLA templates and service builders. Which properties can be adjusted in detail is described within the Evaluation Request Parameter section.

Prediction

When the final model to be predicted is prepared, the prediction is triggered with this model and the EvaluationSettings object. It is responsible to prepare the remote invocation of the prediction server, to manage the communication, and to
process the results returned back. The communication with the prediction server is done based on an http soap based interface.

**Evaluation Request Parameters**

The request of a software service evaluation contains the evaluation parameters within a target SLA template, one or more ServiceBuilders and an EvaluationSettings object. This section describes their details as well as the alternative plain java object based interface.

SLA templates, whether they are about target SLAs, infrastructure SLAs or external service SLAs, can contain agreement terms with variable declarations. Those declarations, in turn, can contain a fixed value as well as guarantee terms with conditions specifying the possible value ranges. However, while an evaluation always is about a specific value, only the fixed values are considered and not the possible ranges.

**Target SLA Template Specification**

The target SLA is considered to extract information related to the usage scenario, also known as usage model in the PCM. We extract the expected workload and the services to be called. We identify the services to be invoked and the according workload based on variable declarations in the agreement terms that have the data type tx_per_sec. When the SLA template is processed, we scan all agreement terms and the variable declarations in those, to pick up those declarations with the tx_per_sec datatype. If we have found one, we use the id of the AgreementTerm to identify the service to be called and use the value of the variable to specify the estimated workload for this service call. For the example snippet blow, taken from http://sourceforge.net/apps/trac/sla-at-soi/changeset/1, we would identify the service ORC_CustomeConstraint-InventoryGetDetails called with a workload of 500 transactions per second.

```xml
<slasoi:AgreementTerm>
  <slasoi:Text/>
  <slasoi:Properties/>
  <slasoi:ID>ORC_CustomeConstraintInventoryGetDetails</slasoi:ID>
  <slasoi:VariableDeclr>
    <slasoi:Text/>
    <slasoi:Properties/>
    <slasoi:Customisable>
      <slasoi:Var>Var_CustomeConstraintInventoryGetDetails</slasoi:Var>
      <slasoi:Value>
        <slasoi:Value>500</slasoi:Value>
      </slasoi:Datatype>http://www.slaatsoi.org/coremodel/units#tx_per_s
    </slasoi:Datatype>
  </slasoi:VariableDeclr>
</slasoi:AgreementTerm>
```

**Figure 79: XML Snippet with Service Request and Workload Specification**

**Note:** It is necessary to specify at least one service call in the target SLA according to the above format. Otherwise, there would be no service call to predict which would not make any sense at all.

The original quality prediction model, more precise, its usage model contains all service calls that are available in general. With the information extracted from the target template, we remove all of them that are not part of the actual request.
**Service Builder Specification**

A prediction request can contain a set of service builders. Each service builder represents a different service setup for the requested service identified by the target SLA template.

Each service builder specifies the external service (infrastructure definitions) as well as the service implementation to be used. For the latter, a quality prediction has to be provided to the model repository in advance.

**Service Implementation**

A service builder references an implementation with a service implementation name.

![Diagram of Service Builder and Service Implementation](image)

*Figure 80: Service Builder and Service Implementation*

The name of the service implementation is used as ID of the model to be loaded. The default implementation of the model repository provided by the service evaluation component is a file based model repository. It loads all prediction models from subdirectories with names according to the model IDs. For example, the prediction model for the service implementation with the name ORC_AllInOne is stored in the directory `<ORC_HOME> / common / osgi-config / software-servicemanager / quality_model_repository / ORC_AllInOne`.

**SLA Templates of Service Bindings**

Each Service Builder references a set of bindings for dependencies of the service implementations implementation artefacts. Each of those bindings references an SLA template that specifies the properties of the bound service.

![Diagram of Service Implementation Dependencies and Bindings](image)

*Figure 81: Service Implementation Dependencies and Bindings*

These SLA templates are analysed to extract the service configurations out of them.
The prediction model previously identified by the name of the service implementation contains a resource environment with one or more resource containers that can be adjusted with the information specified in the service template referenced by a service binding.

The resource container to adjust is identified by the name of the dependency. For example, to adjust the resource container Database_VM in Figure 82, the dependency should be named Database_VM as well.

**Figure 82: Resource Container in a Resource Environment Model**

Within the resource container, the parameters CPU Speed, CPU Cores, Memory, and Availability can be adjusted. To do this, values of variable declarations within agreement terms of the SLA templates are used. The agreement terms as well as the variable declared within them are identified by their name. The following table specifies the terms and variable declarations that are processed if they are present in an SLA template:

<table>
<thead>
<tr>
<th>Agreement Term Name</th>
<th>Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-in-one</td>
<td>CPU_SPEED_VALUE</td>
</tr>
<tr>
<td></td>
<td>CPU_CORES_VALUE</td>
</tr>
<tr>
<td></td>
<td>MEMORY_VALUE</td>
</tr>
<tr>
<td>OverAllAvailability</td>
<td>Overall_Availability_VAR</td>
</tr>
</tbody>
</table>

As an example, the following xml representation of an SLA template shows agreement terms and variable declarations for all the available options:

```xml
<slasoi:AgreementTerm>
  <slasoi:Text/>
  <slasoi:Properties/>
  <slasoi:ID>all-in-one</slasoi:ID>

  <slasoi:VariableDeclr>
    <slasoi:Text></slasoi:Text>
    <slasoi:Properties></slasoi:Properties>
    <slasoi:Customisable>
      <slasoi:Var>CPU_CORES_VALUE</slasoi:Var>
      <slasoi:Value>
        <slasoi:Value>1</slasoi:Value>
        <slasoi:Datatype>http://www.w3.org/2001/XMLSchema#integer</slasoi:Datatype>
      </slasoi:Value>
    </slasoi:Customisable>
  </slasoi:VariableDeclr>
</slasoi:AgreementTerm>
```
Evaluation Settings

The EvaluationSettings class specifies a data type that encapsulates all available parameters to control an evaluation run.

- **analyzeResponsibility**: specifies if the reliability of a service should be evaluated or just its performance.
• **maxMeasurementsCount**: can be used to control the number of measurements during the simulation. Fewer measurements can result in a shortened prediction time but with less accuracy of the prediction.

• **maxSimTime**: can be used to control the maximum simulation time. This can be used to achieve a shortened prediction time but with less accuracy of the prediction.

• **verboseLogging**: This parameter controls the logging in the prediction server.

• **sensorNames**: This optional parameter can be set to a list of sensor names to be returned. If it is not set, the prediction simply returns the response time for the major service calls without any internal or infrastructure sensors.

**Note**: maxMeasurementsCount and maxSimTime are both maximum definitions. So the first maximum that is reached will lead to stop the simulation.

![Diagram](image)

**Figure 84: Evaluation Settings Class**

**Plain Java Interface**

As an alternative to the evaluation methods accepting SLA templates and Service Builders, we provide a Java interface accepting plain java objects that specify the required model adjustments. This is the same data model as created by the extractor from the SLA templates and service builders. This model is located within the package

• org.slasoi.seval.prediction.configuration

and the root class of such an adjustment specification is

• PredictionModelModification

Beside the parameters that are extracted from SLA templates as described above, this interface provides an additional option to adjust service call parameters in the prediction model.
System Call Variables in the Usage Model

The usage model of the Palladio Component Model allows specify variables passed as parameters to the service calls (named EntryLevelSystemCall in the PCM). These variables can be adjusted in the original prediction model. To specify such a parameter, it has to be specified as shown in Figure 85.

This can be used to specify a service call identified by the name used in the usage model and to define variables also by their name and the value to be set in the model.

Note: The variables and system calls specified in this adjustment configuration have to be present in the original prediction model. They are not created from scratch if they are not already defined in the model.

Figure 86: Component Parameters
**Component Parameters in the Repository Model**

The repository model of the prediction model describes the components realizing a service. As part of the component specifications, component parameters can be used to specify specific properties, such as capacities. To specify such parameter adjustments, the java model provides specific classes as shown on the Figure below.

In the same way, the service call parameters are specified, the PredictionModelModification references a list of ComponentSpecifications that identify the components to adjust by their names. Those component specifications reference a list of ComponentParameters which identify the parameters of a component by their name and specifies the value to set for this parameter.

**Result Format**

Calling the evaluation service with an evaluation requests returns an EvaluationResults with a class structure presented in Figure 87.

The root object is a class implementing the IEvaluationResult interface. The default implementation provided with the SLA@SOI framework is the EvaluationResultImpl also included in the class diagram. The EvaluationResult references a service builder the result belongs to. If the evaluation service was requested to evaluate a service for different builders, this association provides the information which result belongs to which service builder in the original request.

![Evaluation Result Class Structure](image)

**Figure 87: Evaluation Result Class Structure**

In addition to this, the EvaluationResult contains different result lists. getResponseTimeResults() provides access to the response time related results. For the downward compatibility to older versions of the SLA@SOI framework, we still provide the getResults() that also returns the response time related results. Besides these results, the getReliabilityResults() method returns those evaluation results that are specific to the reliability simulation. Even if this simulation was not activated in the EvaluationSettings, this method will return an empty list.

The SingleResults in the lists contain the necessary information to describe the evaluation result as well as the point of the system they are related to. The AggregationType, that is an enumeration of the ISingleResult interface, describes the type of the result.
• Responsibility AggregationTypes
  o SUCCESS
  o FAILURE
• ResponseTime AggregationTypes
  o MAX
  o MEAN
  o MEDIAN
  o MIN
  o PERCENTILE_25
  o PERCENTILE_50
  o PERCENTILE_75
  o PERCENTILE_90
  o PERCENTILE_95
  o PERCENTILE_99
  o STDDEV

The serviceId of a single result describes the sensor it belongs to. If a list of sensor names was provided with the EvaluationSettings object, the serviceIds will be the same as those names. If none were requested, the response time results returned include the names of the provided services. The term included with the single result contains the SLA@SOI GSLAM specific quality terms, which might be http://www.slaatsoi.org/commonTerms#availability or http://www.slaatsoi.org/commonTerms#completion_time.

**Appendix D: Glossary**

The following list shows the most important entries of the SLA@SOI glossary. Note that terms that are specific for the current document and not part of the overall project wide glossary are marked with an asterix *.

**Agreement Initiator** An agreement initiator is a party to a service level agreement. The initiator creates and manages an agreement on the availability of a service on behalf of either the service customer or service provider, depending on the domain-specific signalling requirements.

**Agreement Offer** An offer is the description of the agreement relationship that is sent from agreement initiator to agreement responder during agreement creation, indicating the relationship which the initiator would like to form.

**Agreement Responder** The agreement responder is a party to a service level agreement. The responder implements and exposes an agreement on behalf of either the service provider or service customer, depending on the domain-specific signalling requirements.

**Agreement Template** An agreement template is an XML document used by the agreement responder to advertise the types of offers it is willing to accept.

**Agreement Term** Agreement terms define the content of a service level agreement.

**Business Service** A business service is exposed/invoked via at least some non IT elements.
Business Manager A specialization of service provider: person that defines the SLATs of products and joins available services in a product.

External Service External services are exposed across the boundaries of an organization, i.e. across at least two administrative domains.

Framework Administrator A specialization of service provider: person that configures/adapts the SLA@SOI framework for a specific application.

Guarantee Term Guarantee terms define the assurance on service quality associated with the service described by the service definition terms. They refer to the service description that is the subject of the agreement and define service level objectives, qualifying conditions and business value expressing the importance of the service level objectives.

Hybrid Service A hybrid service is a set or bundle of other services where all these services are exposed to the customer but have different service interface types (e.g. an IT service and a business service).

Infrastructure Manager A specialization of infrastructure provider: person/system that is interested to measure and control infrastructure properties.

Infrastructure Provider A specific kind of service provider that focuses on the provisioning of infrastructure services.

Infrastructure Service An infrastructure service is a specific IT service which exposes resource/hardware-centric capabilities.

Internal Service Internal services are exposed within the boundaries of an organization, i.e. within one administrative domain.

IT Service An IT service is exposed/invoked by means of information technology. Specific classes of IT services may be software services, infrastructure services or media services.

Offered Service An abstract service (more precisely: service type) which is offered by a specific Service Provider to its Service Customers.

Operation Level Agreements: A specification of the conditions under which an internal service or a component is to be used by its “customer”.

Service A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks. See also service interface type, service concreteness, service exposure

Service Concreteness The stage a service reaches over time from a fully abstract type to actually instantiated. See also service type, offered service, service implementation, service instance

Service Consumer Person(s) who actually consume/use the provided services. Typically they belong to the service customer.

Service Customer Someone (person or group) who orders/buys services and defines and agrees the service level targets.

Service Description Term Service Description Terms describe the functionality that will be delivered under the service level agreement. The agreement description may include also other non-functional items referring to the service description terms.
<table>
<thead>
<tr>
<th>Service Exposure</th>
<th>Services can be exposed either internally (within the same administrative domain) or externally. See also internal service, external service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Implementation</td>
<td>A service implementation is a possible concrete realization of a given service type.</td>
</tr>
<tr>
<td>Service Instance</td>
<td>A concrete realization of an offered service which is ready for consumption by service users. It relies on the instantiations of all the resources required for a given service implementation.</td>
</tr>
<tr>
<td>Service Interface Type</td>
<td>Describes the nature of an actually exposed service, i.e. about the nature of his invocation interface. See also business service, IT service, hybrid service</td>
</tr>
<tr>
<td>Service Level Consequence</td>
<td>An action that takes place in the event that a service level objective is not met.</td>
</tr>
<tr>
<td>Service Level Agreement</td>
<td>An agreement defines a dynamically-established and dynamically managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement. The management of this delivery is achieved by agreeing on the respective roles, rights and obligations of the parties. The agreement may specify not only functional properties for identification or creation of the service, but also non-functional properties of the service such as performance or availability. Entities can dynamically establish and manage agreements via Web service interfaces.</td>
</tr>
<tr>
<td>Service Level Objective</td>
<td>Service Level Objective represents the quality of service aspect of the agreement. Syntactically, it is an assertion over the agreement terms of the agreement as well as such qualities as date and time.</td>
</tr>
<tr>
<td>Service Provider</td>
<td>An organization supplying services to one or more internal customers or external customers.</td>
</tr>
<tr>
<td>SLA Manager</td>
<td>A specialization of service provider: person/system that is responsible for managing SLATs and SLA relationships.</td>
</tr>
<tr>
<td>Software Designer</td>
<td>A specialization of software provider: person that designs/develops the architecture and components of a specific SLA based application.</td>
</tr>
<tr>
<td>Software Manager</td>
<td>A specialization of service provider: person that defines software-based services, takes care of their management and supports the SLA manager in creating appropriate SLA templates.</td>
</tr>
<tr>
<td>Software Provider</td>
<td>An organization producing software components which might be used by a service provider to assemble actual services.</td>
</tr>
<tr>
<td>Software Service</td>
<td>A software service is a specific IT service which is exposed/invoked by means of software entities such as Web services, user interfaces, or software-based business processes.</td>
</tr>
<tr>
<td>Software Component</td>
<td>Software components are the entities produced at design-time by a software provider.</td>
</tr>
<tr>
<td>Service Type</td>
<td>A service type (or abstract service) specifies the external interface of a service possibly including non-functional aspects. It does not specify any means (components, resources) which are needed for the actual provisioning of that service.</td>
</tr>
</tbody>
</table>
Appendix E: Abbreviations

AOP  Aspect Oriented Programming
BM   Business Manager
B-SLAM Business SLA Manager
EMF  Eclipse Modelling Framework
ERP  Enterprise Resource Planning
IE   Interaction Event
Infr-SLAM Infrastructure SLA Manager
Infr-SM Infrastructure Service Manager
IoC  Inversion of Control
KPI  Key Performance Indicator
LLMS Low Level Monitoring System
MA   Manageability Agent
MRE  Monitoring Result Event
MVC  Model View Controller
NFP  Non-functional property
ORC  Open Reference Case
OVF  Open Virtualization Format
QoS  Quality of Service
PAC  Provisioning and Adjustment Component
POC  Planning and Optimization Component
POJO Plain Old Java Objects
SaaS Software as a Service
SE   Service Evaluation
SLA  Service Level Agreement
SLAM SLA Manager
SLAT Service Level Agreement Template
SM   Service Manager
SME  Small and Medium-sized Enterprise
SOA  Service Oriented Architecture
SW-SLAM Software SLA Manager
SW-SM Software Service Manager
TCO  Total Cost of Ownership