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 2010
Actual submission to EC date: 10th September 2010

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

Lead contractor for this deliverable: FZI Forschungszentrum Informatik
Revision: V.1.6 (20th August 2010)
Document Status

<table>
<thead>
<tr>
<th>Deliverable Lead</th>
<th>FZI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer 1</td>
<td>Alfonso Castro (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>2010-06-22</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>2010-07-02</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>2010-07-13</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>2010-08-20</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>2010-08-30</td>
</tr>
</tbody>
</table>

Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>FZI</td>
<td>Franz Brosch</td>
</tr>
<tr>
<td>CITY</td>
<td>Davide Lorenzoli</td>
</tr>
<tr>
<td>INTEL</td>
<td>Victor Bayon, Phil Tian</td>
</tr>
<tr>
<td>SAP</td>
<td>Jens Happe, Dennis Westermann, Christof Momm</td>
</tr>
<tr>
<td>PMI</td>
<td>Sam Guinea</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.

Document History

<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</td>
</tr>
<tr>
<td>Release</td>
<td>Date</td>
<td>Author</td>
<td>Notes</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
<td>----------------------</td>
<td>----------------------------------------------------------------------</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>
</tbody>
</table>
Executive Summary

This document, SLA@SOI Deliverable D.A6a, represents the second of three annual reports documenting the progress of the topics and challenges addressed by Work Package "WP A6 - Predictable Service Engineering".

This work package aims to support the engineering process for predictable software services and 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.

There are four main contributions to SLA@SOI resulting out of the work within WP A6:

- **Software Performance and Reliability Prediction**: Provides a means for software service providers to evaluate the expected performance and reliability of their services before their run-time. May be used at (i) service offering / SLA template design stage to determine realistic quality parameters to offer, or at (ii) (automated) service / SLA negotiation, to determine realisable quality parameters to agree upon.

- **Resource Usage Prediction**: Provides a means for infrastructure service providers to predict expected infrastructure resource demands at system run-time. May be used at service provisioning stage to determine the best option for deployment of a new VM.

- **Run-time SLA Violation Prediction**: Provides a means for software service providers to anticipate possible near-future software SLA violations at system run-time. May be used at service operations / SLA runtime stage to trigger adjustment activities in order to avoid an actual SLA violation.

- **Manageability Modelling and Design**: Provides a design-time methodology and a set of tools for enhancing service components with management capabilities. In particular, service components are augmented with the capabilities needed to (i) extract the runtime data needed to monitor the quality of running services, and to (ii) adjust a service’s behaviour and/or its configuration at run time.

These contributions are included in the components provided by the SLA management framework, and by stand-alone tools that complement the capabilities of the framework. This deliverable describes each contribution, including its embedment into the SLA framework architecture, as well as its application during the stages of the SLA and service lifecycles. Compared to the M12 version of the deliverable, the following main changes and additions have been included:

- The WP A6 contributions are more explicitly set into the context of the overall SLA framework and the service lifecycle. We show how the individual goals of each contribution support the overall objectives of the project. According to the current description of work for A6 (see DoW, Amendment 2), we do not longer intend to provide an integrated
prediction service which includes all individual contributions of this work package. Instead, we appreciate the individual scope and the goals of each contribution, adding to the overall predictability of software services and components.

- For each contribution, we present the additional results that have been achieved during the second project year. Furthermore, we describe how the contribution supports the requirements of the industrial use cases, illustrating their concrete application.

Concluding, the M26 version of the deliverable highlights the challenges of predictable systems engineering, and describes how the A6 contributions address these challenges to support the predictable, transparent and automated SLA management in SLA@SOI.
# Table of Contents

1 Introduction ............................................................................................................. 11

2 Contributions Overview ......................................................................................... 13

   2.1 Key Innovations .................................................................................................. 14

       2.1.1 Software Performance and Reliability Prediction .......................................... 14

       2.1.2 Resource Usage Prediction ........................................................................... 15

       2.1.3 Run-time SLA Violation Prediction ................................................................. 16

       2.1.4 Manageability Modelling and Design ............................................................. 17

2.2 Framework Contributions ...................................................................................... 17

       2.2.1 Software Performance and Reliability Prediction .......................................... 19

       2.2.2 Resource Usage Prediction ........................................................................... 19

       2.2.3 Run-time SLA Violation Prediction ................................................................. 19

       2.2.4 Manageability Modelling and Design ............................................................. 20

2.3 Task-level Activities ............................................................................................. 20

       2.3.1 Task A6.1: Meta-models for Prediction Services ........................................ 21

       2.3.2 Task A6.2: Performance Prediction Service .................................................. 21

       2.3.3 Task A6.3: Resource Usage Prediction ......................................................... 22

       2.3.4 Task A6.4: Reliability Prediction Service ...................................................... 22

       2.3.5 Task A6.5: Run-time SLA Violation Prediction ............................................. 22

       2.3.6 Task A6.6: Manageability Modelling and Generation for (Composite) Business Services ................................................................................................. 23

       2.3.7 Task A6.7: Prediction Model Adjustment, Integration, and Validation Methodology .............................................................................................................. 23

3 Software Performance and Reliability Prediction .................................................. 24

   3.1 Introduction ........................................................................................................... 24

   3.2 Overview ............................................................................................................... 25

       3.2.1 Goals and Scope .............................................................................................. 25

       3.2.2 Preliminaries .................................................................................................. 26

       3.2.3 Roles and Responsibilities ............................................................................. 27

       3.2.4 Prediction Workflow ..................................................................................... 28

3.3 State of the Art ....................................................................................................... 29

       3.3.1 Architecture-based Performance Prediction .................................................... 29

       3.3.2 Architecture-based Reliability Prediction ....................................................... 32

3.4 QoS Meta-Model for Performance and Reliability Prediction ............................ 35

       3.4.1 Overview ......................................................................................................... 35

       3.4.2 Performance-specific Aspects ....................................................................... 36

       3.4.3 Reliability-specific Aspects .......................................................................... 37

3.5 Prediction Models Adjustment ............................................................................... 38

       3.5.1 Performance Cockpit ..................................................................................... 39

       3.5.2 Experiment Definition ................................................................................... 39

       3.5.3 Automated Measurements ............................................................................ 40

       3.5.4 Statistical Inference ....................................................................................... 41

       3.5.5 Model Integration .......................................................................................... 42

3.6 Performance Prediction Service ............................................................................ 42

       3.6.1 Overview ....................................................................................................... 42

       3.6.2 Prediction Engine ......................................................................................... 43

       3.6.3 Prediction Process .......................................................................................... 44

3.7 Cost Prediction Service ......................................................................................... 46

       3.7.1 Modeling CPU Costs with Multi-Core .......................................................... 46

       3.7.2 Modeling Power Consumption ..................................................................... 49

       3.7.3 A Cost Model for Enterprise Applications .................................................... 50

3.8 Multi-objective Optimization ............................................................................... 50

       3.8.1 A SLA-Driven Planning Framework ............................................................... 51
3.8.2 A Multi-Objective Optimizer ................................................................. 52
3.9 Use Cases for Performance Prediction .................................................... 53
  3.9.1 WP B2: Open Reference Case ........................................................... 53
  3.9.2 WP B3: ERP Hosting ....................................................................... 57
  3.9.3 WP B6: E-Government ................................................................. 63
4 Resource Usage Prediction .................................................................. 66
  4.1 Introduction ....................................................................................... 66
  4.2 Prediction User Interfaces .................................................................. 71
    4.2.1 Standalone User Interface .......................................................... 72
  4.3 Runtime Prediction Integration Scenario .......................................... 73
  4.4 Design rationale................................................................................ 76
    4.4.1 Prediction overhead ..................................................................... 76
    4.4.2 Scalable Architectures ................................................................. 77
  4.5 Architecture and Implementation ..................................................... 78
    4.5.1 ClientAgent ................................................................................ 79
    4.5.2 PredictionAgent ......................................................................... 81
    4.5.3 PredictionWorker: Prediction (Network) Provisioning ............... 84
5 Run-time SLA Violation Prediction ....................................................... 85
  5.1 Introduction ....................................................................................... 85
  5.2 State of the Art.................................................................................. 86
  5.3 The Overall Approach ...................................................................... 89
    5.3.1 QoS Specification ....................................................................... 90
    5.3.2 QoS Predictor And QoS Predictor Configuration ....................... 91
    5.3.3 Prediction Specification ............................................................... 92
    5.3.4 QoS Predictors ........................................................................ 93
    5.3.5 Experimental Results ................................................................ 94
6 Manageability Modelling and Design .................................................. 95
  6.1 Overview ......................................................................................... 95
  6.2 The Management Meta-models ........................................................... 96
    6.2.1 The Raw Data Sampling Meta-Model ......................................... 96
    6.2.2 The KPI Meta-Model ................................................................ 98
    6.2.3 The Adjustment Meta-Model ....................................................... 99
  6.3 Automatic Synthesis of Basic Instrumentation Directives ............... 100
  6.4 SLA@SOI Management Modeling Tool ........................................... 101
  6.5 Example Models and Instrumentation Configurations ..................... 102
7 Conclusions ......................................................................................... 105
8 References .......................................................................................... 106
Appendix A: Standard QoS Terms for Prediction .................................... 116
Appendix B: QoS Meta-Model ................................................................. 118
Appendix C: Service Evaluation .............................................................. 124
Appendix D: Glossary ........................................................................... 127
Appendix E: Abbreviations .................................................................... 129
Table of Figures

Figure 1: Overview of WP A6 Contributions and A-line Interrelations ............... 13
Figure 2: SLA Framework Architecture Overview ........................................ 18
Figure 3: Abstract Performance and Reliability Prediction Workflow ............... 28
Figure 4: QoS Meta-Model Structure (Overview) ...................................... 35
Figure 5: Goal-oriented Systematic Measurements .................................... 38
Figure 6: Performance Cockpit Idea ....................................................... 39
Figure 7: Experiment Definition Meta-Model ........................................... 40
Figure 8: Automated Measurements ...................................................... 41
Figure 9: Prediction Tooling (Overview) ................................................. 43
Figure 10: Evaluation of Software Services ............................................ 45
Figure 11: TPC-C Benchmark Results .................................................. 47
Figure 12: Power Function Fitting .......................................................... 48
Figure 13: Normalized Power vs. CPU Utilization .................................... 48
Figure 14: A SLA-driven planning framework using multi-objective optimization, and incorporating the performance and cost models. .................. 51
Figure 15: Open Reference Case System ............................................... 54
Figure 16: Payment Service RDSEFF ...................................................... 55
Figure 17: ORC Infrastructure Model ..................................................... 55
Figure 18: ORC Usage Model ............................................................... 56
Figure 19: Book Sale Usage Scenario .................................................... 57
Figure 20: Regular Steps Within The SD Benchmark .............................. 58
Figure 21: Components and Interfaces of the SD-Application Model ........... 59
Figure 22: ERP Use Case Component Diagram ...................................... 59
Figure 23: Transaction createCustomerOrder of SAP ERP 6.0 ................... 60
Figure 24: RD-SEFF of the Netweaver’s write Operation ......................... 61
Figure 25: RD-SEFF of MaxDB’s writeData Operation ............................. 61
Figure 26: Execution Environment ...................................................... 62
Figure 27: Deployment ........................................................................... 62
Figure 28: Usage Model of the SD Benchmark Application ....................... 63
Figure 29: Service Component Types and Interfaces ............................... 64
Figure 30: The Health Care System ...................................................... 64
Figure 31: Usage Profile of the Health Care System ............................... 65
Figure 32: Handling Booking Requests at the CallCenter ......................... 65
Figure 33: Internal Architecture of Prediction ........................................ 66
Figure 34: Basic Training/Prediction Workflow ...................................... 67
Figure 35: Dynamic Prediction With Future Error .................................. 68
Figure 36: Training and Re-training Workflow ....................................... 69
Figure 37: Prediction versus Real Instrumentation ................................... 70
Figure 38: Steady-State Metric with Spikes ............................................ 70
Figure 39: Prediction versus Real Instrumentation (Running Average) ....... 71
Figure 40: Resource Usage Prediction User Interface: Training ................. 72
Figure 41: Resource Usage Prediction User Interface: Testing .................... 72
Figure 42: Prediction Dynamic User Interface ...................................... 73
Figure 43: Prediction Usage Scenario .................................................. 74
Figure 44: SQL-like Query for Prediction Scenario ................................. 75
Figure 45: Resource Usage (cpu_user) Over Time ................................... 76
Figure 46: Prediction Data Gathering and Processing ............................. 80
Figure 47: Prediction for cpu_usage* Metric ........................................ 83
Figure 48: Prediction Instrumentation on the B4 Use Case ....................... 84
Figure 49: Prediction Provisioning ....................................................... 85
Figure 50: Prediction Framework Components ...................................... 89
Figure 51: Prediction Specification to Monitoring Specification .................. 89
1 Introduction

SLA@SOI aims at developing a comprehensive multi-layer SLA management framework for service-oriented software systems. Across all layers of an IT stack and across the various stakeholder perspectives the framework supports the provision of software systems or parts of them as services with a contractually fixed quality based on service-level agreements. Noteworthy, quality attributes span across multiple non-functional domains such as security, performance, availability, and reliability.

The A6 work package (“Predictable Systems Engineering”) specifically focuses on the challenge to predict service quality properties before they are actually observed. 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. 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, the A6 work package 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 predicted service quality.

The common work of the partners that participate in the A6 work package can be categorized into four main contributions. Three of these contributions are directly focused on predictability, whereas the fourth contribution deals with manageability design as a prerequisite and necessary context for predictable systems engineering. In detail, the WP A6 contributions are as follows:

- **Software Performance and Reliability Prediction**: Allows for software service providers to predict the expected performance and reliability of their services at system-design time. The challenge of this task is to enable quality prediction in advance before the actual assembly, deployment and operation of the service components. To realize this prediction, an architectural model of the system under study is built, which specifies the involved service components, along with their interfaces, composition, usage, and deployment onto the underlying infrastructure. The model is then evaluated to predict the system performance (response time, throughput, resource utilization) and reliability (probability of successful execution). The approach may be used at (i) the service offering / SLA template design stage to determine realistic quality parameters (i.e. parameter values) to offer, or during the (ii) (automated) service / SLA negotiation, to determine realisable quality parameters to agree upon.

- **Resource Usage Prediction**: Provides a means for infrastructure service providers to (a priori) evaluate expected infrastructure resource demands at system run-time. Prediction is based on continuously observed resource usage metrics and uses machine learning techniques to predict the (near) future trends for those metrics. A high degree of flexibility is achieved by providing a prediction framework, where multiple prediction algorithms can be plugged in and applied. The quality and appropriateness of these algorithms can be evaluated using a test module which is part of the
framework. The approach may be used at service provisioning stage to determine the best option for deployment of a new VM to a server.

- **Run-time SLA Violation Prediction**: Provides a means for software service providers to anticipate possible near-future software SLA violations at system run-time. The approach translates guarantee terms of SLA’s into rules according to a given rule expressions language, and identifies the relevant event types that have to be monitored to evaluate the rules. Based on monitoring data gathered at system run-time, a predictive engine calculates the likelihood of the violation of each rule in the near future. This knowledge may be used at service operations / SLA runtime stage to trigger adjustment activities in order to avoid an actual SLA violation.

- **Manageability Modelling and Design**: Provides a design-time methodology and a set of tools for enhancing service components with management capabilities, which are required for an SLA-driven management. In particular, service components are augmented with the capabilities needed to (i) extract the run-time data needed to monitor the quality of running services, and to (ii) adjust a service’s behaviour and/or its configuration at run time. Manageability is a cross-cutting concern, related to predictability (WP A6), as well as service management (WP A3). Manageability modelling and design in WP A6 focuses on the design-time activities and models that are a prerequisite enabling service management at run-time.

Figure 1 sets the contributions of WP A6 into the context of the overall A-line work and 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.

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 VM provisioning decisions on expected future trends of resource utilization.
The structure of this deliverable document follows the abovementioned contributions of WP A6. For each of these contributions the document provides a description of the activities, progress and achievements up to the 26th project month. The overall direction of activities in the second project year was to refine the SLA management framework which resulted from the first year, taking into account the new framework architecture, which emerged out of a substantial rework of the "ad hoc" architecture of the first year. The deliverable describes how the results of WP A6 are applied in the industrial use case demonstrators realized through B-line work packages B3 to B6.

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 an outlook towards the third project year.

## 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 chapter is organized along three questions: What are the scientific innovations of the individual WP A6 contributions (Section 2.1)? How do the contributions enrich the overall SLA
management framework (Section 2.2)? What activities have been taken and what goals have been reached in each of the tasks of the work package (Section 2.3)? Each of these questions is answered in the following, taking the four main contributions of WP A6 into account, as they have been introduced in Chapter 1: software performance and reliability prediction, resource usage prediction, runtime SLA violation prediction, and manageability modelling and design.

2.1 Key Innovations

As a large-scale integrated project, SLA@SOI aims at bringing cutting-edge research approaches together, and developing them further towards the overall vision of "a business-ready service-oriented infrastructure empowering the service economy in a flexible and dependable way", where business-readiness requires the three main characteristics of predictability & dependability, transparent SLA management, and automation (DoW, p8).

The A6 work package, at its core, provides the techniques and methodology for the development and operation of predictable services, where predictability mainly refers to the non-functional (quality) aspects of services. Such predictability can generally not be achieved by a single technique or approach alone. The evaluation of different quality attributes (e.g., performance and reliability) requires substantially different methods. For example, different approaches are needed when dealing with the end-to-end service quality, as perceived by the service customers, in contrast to the more indirect indicators that guide a service provider in making well-informed decisions and avoiding SLA violations. Also, the phase in the service lifecycle where a prediction is to be made (service design-time or service run-time) greatly influences the underlying assumptions and specific challenges of a prediction technique.

While individual problems have been tackled by existing approaches, the challenge remains to bring these approaches into a common context, to develop a common terminology, and to identify and fill the conceptual and technological gaps that exist in related work. To this end, the main innovation brought up by the A6 work package is a comprehensive set of techniques and tools that, as a whole, enables predictability of services, and supports the further goals of transparent and automated SLA management.

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

2.1.1 Software Performance and Reliability Prediction

Design-time performance and reliability prediction in SLA@SOI aims at a-priori evaluation of software service quality. It is based on the Palladio Component Model (PCM) [1], a performance modelling and prediction tool for component-based software architectures. In the course of the project, we extend and further develop the PCM approach to deal with service-based systems and architectures, we enable modelling and prediction of service reliability, and we realize a prediction service, i.e. a “ServiceEvaluation” component within the SLA management framework that can be used to evaluate service quality as part of automated negotiation processes. For a detailed description of the approach, see Chapter 3.
Compared to the state of the art in performance and reliability modelling and prediction, we see our approach as highly innovative. For a detailed state of the art overview, see Section 3.3. In summary, we see our approach as being advantageous in the following regards:

- The approach considers architectural influences on the performance and reliability of a service-based system in a comprehensive way, namely, service component implementation, service usage, quality of required services, and execution environment (physical resources). Other approaches are generally limited in their expressiveness to a subset of these factors.

- The approach supports a distributed development process as typical for service-based systems. Modelling responsibilities are split between software provider(s) and service provider(s), who can contribute their respective parts of the necessary modelling information independently from each other. This independence of multiple roles enables the systematic consideration of service quality already in early phases of the development process.

- The approach is particular strong in consideration of service usage and its influence on the expected quality. A usage model captures system workloads, call sequences and input parameter values for service calls. All parameters may be specified using arbitrary probability distributions. The usage profile is automatically propagated from the system-level throughout the service-based architecture to derive the component-level usage profiles. Thus, high prediction accuracy with regards to service usage is achieved.

Further benefits of the approach lie in the adoption of one common modelling language for performance and reliability prediction (providing a foundation for trade-off analyses), the support for complete probability distributions as a result of performance prediction (instead of only mean-value analysis), and the combined consideration of hardware- and software-induced failures for reliability prediction.

Statistical Inference of Software Performance Models

This work complements the model-based 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.

2.1.2 Resource Usage Prediction

The main innovation with regards to Runtime Prediction has been the integration and development of 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.
2.1.3 Run-time SLA Violation Prediction

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

The EVEREST+ framework is substantially different from the prediction abilities developed within the monitoring framework EVEREST as part of the SERENITY project. These differences are both theoretical and technical.

From a theoretical point of view, the prediction approach developed within SERENITY was based on Dempster-Shafer theory that is a mathematical theory of evidence [191]. It allows one to combine evidence from different sources and arrive at a degree of belief (represented by a belief function) that takes into account all the available evidence. Moreover, to predict about a property an Event Calculus (EC) [192] model of the property to predict was needed.

The approach to prediction developed within SLA@SOI project is based on statistical distribution functions inferred from historical data. Prediction algorithms use this statistical information to predict SLA violations. Statistical models are automatically inferred and updated by EVEREST+.

<table>
<thead>
<tr>
<th>SLA@SOI</th>
<th>SERENITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Based on statistical distribution functions inferred from historical data</td>
<td>- Based on Dempster-Shafer theory</td>
</tr>
<tr>
<td>- Set of prediction algorithms based on statistical models embedded in EVEREST+</td>
<td>- Prediction rules specified in event calculus and executed by EVEREST.</td>
</tr>
<tr>
<td>- User defined prediction algorithms can be plugged into EVEREST+</td>
<td>- No use of historical data</td>
</tr>
</tbody>
</table>

Table 1. Theoretical differences between SLA@SOI and SERENITY prediction

From a technical point of view, the prediction approach developed within SERENITY was tightly coupled with EVEREST monitoring engine. EVEREST collected and made it available to the prediction system. EVEREST was also in charge of reporting both SLA violations and prediction results.

The prediction approach developed within SLA@SOI, despite using EVEREST as main source of historical data, supports data coming from any other relational database, or other sources. In the latter case, a user should provide a suitable driver for data retrieving. EVEREST+ also provides a set of APIs for developing user-defined predictors (prediction algorithm implementations) and for extending the set of automatically inferred statistical models.

<table>
<thead>
<tr>
<th>SLA@SOI</th>
<th>SERENITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Supports by default data coming from relational databases; provides additional mechanisms to support</td>
<td>- Uses EVEREST as only source of data</td>
</tr>
<tr>
<td></td>
<td>- Supports event calculus prediction</td>
</tr>
</tbody>
</table>
Compared to the state of the art in run-time prediction, we see our approach as highly innovative for the following aspects:

- Our research does not only focus on the design of QoS predictors, but it provides a prediction framework able to support different and new QoS terms. We create a general and extensible framework for QoS terms prediction.

- The framework is general because it doesn’t support a limited set of QoS only. On contrary it provides a support for automatically collecting and analyzing data and reasoning about actions and their effects over time.

- The framework is extensible because the definition of data to be collected and its analysis can be specified using models (QoS specifications) and pluggable components (QoS predictors) respectively.

### 2.1.4 Manageability Modelling and Design

We provide a model-driven engineering approach that explicitly supports designing service-based systems that are manageable. This includes the specification of the sensors that will be deployed to the system to collect and process the run-time data needed for monitoring. It also includes the specification of the effectors that will be deployed to the system to provide the control capabilities needed to adjust a system dynamically.

In terms of innovation with respect to the state of the art we provide 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

The SLA management framework is the central technological outcome of SLA@SOI, delivering a comprehensive management solution for service-based systems. During the first project year, an ad-hoc version of this framework has been realized and demonstrated in the M12 project review. This ad-hoc version was mainly focused on the needs and requirements of the Open Reference Case (ORC). In the second project year, the architecture of the framework has been further enhanced in order to achieve more generality, defining a clear separation of concerns and responsibilities of individual framework components, as well as assuring reusability of those components in various use cases and contexts. Figure 2 shows an overview of the current framework architecture. For a detailed description of the architecture, see Deliverable D.1a “Framework Architecture”.

| Table 2. Technical differences between SLA@SOI and SERENITY prediction |
|-----------------------------|-----------------------------|
| data from other sources     | specification only          |
| - Is not only a prediction  | - Does not allow any user-defined |
|   framework, but also a    |   extensions                |
|   framework for developing |                           |
|   and executing prediction |                           |
|   algorithms               |                           |
Figure 2: SLA Framework Architecture Overview

Figure 2 shows how the components released by WP A6 are integrated in the overall architecture of the SLA@SOI framework. However, as the framework is executed during service offering, negotiation, and later stages, the framework components do not cover all stages of the service lifecycle, especially not the service design-time, when design-time models are created for quality prediction and manageability configuration. Thus, whereas some WP A6 contributions are directly realized as a component within the SLA management framework, other contributions result in stand-alone tools, which complement the framework functionality at service negotiation and later stages, in order to achieve overall predictability of services.

In the following, the outcomes of individual WP A6 contributions in terms of SLA management framework components or accompanying stand-alone tools are described.
2.2.1 Software Performance and Reliability Prediction

The outcome of the software performance and reliability prediction approach is twofold: On the one hand, a stand-alone tool is provided that can be used by software and service providers to create QoS model instances for their service components and service-based system architecture. On the other hand, WP A6 provides an implementation of the Service Evaluation component (see Figure 2) within the SLA framework architecture.

The Service Evaluation component provides a general means to determine service quality a-priori (i.e. before the services are actually used). This functionality 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. Within WP A6, a concrete implementation of Service Evaluation is provided for performance and reliability evaluation of software services.

The stand-alone tool provided for creation of QoS model instances is an Eclipse Modelling Framework (EMF) based tool with graphical editors supporting the different modelling tasks of software and service providers. The tool allows for conducting performance and reliability predictions, and thus can be used for determining initial service offers during SLA template design stages. 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 stages.

2.2.2 Resource Usage Prediction

Resource runtime usage prediction can be used within the framework as a standalone component that is executed as a service. Once deployed and configured, it can provide 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.

2.2.3 Run-time SLA Violation Prediction

The contributions of run-time SLA violation prediction to the SLA@SOI framework are both theoretical and technical.

Theoretical contributions:
- We have surveyed the literature searching for predictive approaches to understand which of them best suited our objectives so to analyze them to find their weakness and strength. Starting from this analysis we modelled a new approach to prediction.
- Considering the QoS terms defined within SLA@SOI we looked for a way of grouping QoS terms together by their communalities. We found that
different QoS terms belong to the same prediction strategy within our approach. This means that we can create a generic QoS predictor for set of QoS terms.

- We extended the SLA Abstract Syntax to create a prediction specification, e.g., a machine-readable document to request QoS term predictions.

Technical contributions:
- Design and implementation of EVEREST+, a general and extensible model-driven prediction framework. We implemented a prototype for proofing our approach. We also implemented QoS predictors for some of the QoS terms specified with SLA@SOI.
- 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.

2.2.4 **Manageability Modelling and Design**

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 **Task-level Activities**

This section provides an overview over the activities and reached milestones of each individual task within the A6 work package. This overview takes the consolidation and refinements of the DoW for WP A6 from Amendment 2 into account and presents the achievements of the work package according to the current DoW description and task structure.

The four main WP A6 contributions as described in Section 1 are provided by the WP A6 tasks as follows:

- Software performance and reliability prediction is a common effort of tasks A6.1, A6.2, A6.4, and A6.7
- Resource usage prediction is provided by Task A6.3
- Run-time SLA violation prediction is provided by Task A6.5
- Manageability modelling and design is provided by Task A6.6

A more detailed description of contents and achievements of each task is given in the following subsections.
2.3.1 Task A6.1: Meta-models for Prediction Services

This task lays the foundation for software performance and reliability prediction by providing a common modelling language for service-based software architectures covering relevant aspects of both quality attributes. It is thus a necessary prerequisite for the prediction services developed in tasks A6.2 (see Section 2.3.2) and A6.4 (see Section 2.3.4).

Although the goal of supporting multiple quality attributes existed from the beginning on, we focused on performance in the first project year. Upon the foundations of the Palladio Component Model (PCM) [1], we developed a QoS meta-model for the specification of service components, along with their composition, behaviour, deployment and usage. We incorporated performance-relevant annotations into the meta-model (system workload specification, physical resource demands of service execution, network latency and throughput, processing speed of physical resources) in order to enable performance predictions.

In the second project year, we concentrated on the extension of the QoS meta-model for the purpose of service reliability prediction. We adapted mainly the behavioural specifications and introduced reliability-relevant annotations (failure definitions, behavioural and network failure probabilities, physical resource availability).

The QoS meta-model is described in Section 3.4. As of M26, the main work of the task is done, and in the third project year, we'll consider refinements of the QoS meta-model, if necessary for certain use cases or for reliability prediction support.

2.3.2 Task A6.2: Performance Prediction Service

This tasks builds upon the QoS meta-model developed in Task A6.1 (see Section 2.3.1) and aims at the realisation of performance prediction for software services based on consideration of the service-based software architecture.

In the first project year, we focused on realizing performance prediction through an automated transformation of QoS meta-model instances in Queueing Networks (QNs) and simulations of these QNs. The performance metrics resulting from prediction are the service completion times, system throughput and utilization of physical resources. Furthermore, we developed a first basic version of a performance prediction service, which could be used during SLA negotiation in the ad-hoc SLA management framework.

The second project year was devoted to the development of a more generic performance prediction service, which is applicable to be used within the new SLA framework architecture, providing the IServiceEvaluation interface of the Service Evaluation component. To this end, we strengthened the relation to the WP A1 core meta-model and the WP A5 SLA meta-model by creating the necessary mappings to the WP A6 QoS meta-model.

The core results of our work are described in Sections 3.2 (foundations) and 3.6 (performance prediction service).

In the third project year, further refinements and extensions of the performance prediction service are planned, which may be specifically targeted towards special needs of individual industrial use cases.
2.3.3 Task A6.3: Resource Usage Prediction

This task builds on the work carried out during year 1 of the project. The main work has focused on extending and integrating the Y1 code base within a deployable framework. For Y2 of the project the most relevant features are:

- Self-contained instrumentation and monitoring of low level resources to extract the required basic metrics necessary to provide runtime prediction
- Pre and post processing of raw metrics using different techniques and approaches to prepare the data sets for training and prediction
- Pluggable approach to allow the deployment of different prediction implementations (algorithms) backend at deployment time such as heuristics based implementations or machine learning based
- Design and implementation of a scalable architecture that can be used for providing on-demand provisioning services for many SLA aware resources such as servers or VMs.

The results are described in Section 4.

2.3.4 Task A6.4: Reliability Prediction Service

This task was originally named “Holistic Prediction Services” and should combine the outcomes of Tasks A6.2 (see Section 2.3.2) and A6.3 (see Section 2.3.3). Thus, software-level and infrastructure-level predictions should be combined to a common prediction service. However, this turned out to be an infeasible goal. The two prediction approaches differ in their scope, their goals, and their methods. It is more natural to consider them as two measures working towards the overall predictability of service-based systems. Accordingly, the task was renamed as part of DoW Amendment 2, and now explicitly refers to the second quality attribute of design-time prediction, namely reliability. The redefinition of the task did not invalidate existing results, because work on this task began not before the second project year (as stipulated in the DoW effort distribution description).

The work on this task in the second project year is an overlapping effort to task A6.1 (see Section 2.3.1), where the QoS meta-model extensions for reliability were realized. These extensions needed to be done with the planned transformations and evaluation methods for reliability in mind. The main work of this task, however, remains for the third project year, when a transformation from the QoS-meta-model to a Markov chain model will be realized, as well as the evaluation of this Markov chain for reliability. The existing design-time prediction service for performance will be enhanced to deliver also reliability metrics, thereby enabling trade-off analyses between both quality attributes.

2.3.5 Task A6.5: Run-time SLA Violation Prediction

This task provides a run-time prediction service to anticipate potential violations of SLA guarantee terms. The prediction service is integrated into the SLA@SOI monitoring architecture and builds upon the monitoring data collected during service run-time.

To realize the prediction service, a general and extensible framework for reasoning about potential violations of QoS properties has been implemented. The framework allows for consideration of different properties through QoS predictors.
that may be plugged in as software components. SLA guarantee terms are automatically translated into monitoring rules based on Everest, the event calculus monitoring engine that is used as part of the SLA@SOI monitoring infrastructure. During service run-time, the collected monitoring data are evaluated to predict potential violations of the guarantee terms (e.g., the probability that a guarantee term – for instance “service throughput” – will be violated in the next N time units). The focus is on aggregated SLA guarantee terms rather than the prediction of occurrences of individual events. Two key features of the approach are that (i) monitoring is performed in parallel with the operation of the service-based system, without affecting its performance, and that (ii) no instrumentation of the composition process of the service-based system or individual services is required.

The results of the work of this task are described in Section 5.

2.3.6 Task A6.6: Manageability Modelling and Generation for (Composite) Business Services

This task provides a Model-driven Engineering approach to the development of manageable service compositions. In it we advocate that management is a fundamental aspect of a service-based application, and that it must be considered throughout the application’s lifecycle, starting from design time. In particular, designers need to be able to specify the non-functional extensions required to provide a service that is manageable, that is a service that is monitorable and controllable, to ensure that an SLA is met.

In the project’s second year we extended the results initially provided in year one, by introducing new meta-models and new tools. In particular, the meta-models in year one have evolved to become the basis of our raw data sampling meta-model, which is used to define what sensors need to added to a system for it to be manageable. On top of these meta-models, we have added a KPI meta-model for defining what raw data correlations and transformations need to be supported to calculate high-level QoS indicators; and an adjustment meta-model for describing the dynamic binding strategies that are supported by the system. For the raw data sampling and the KPI meta-models we also provide an Eclipse-based tool support for defining compliant models, and for automatically synthesizing the instrumentation configurations needed to deploy the sensors into the system. For the description of results, see Section 6.

In the project’s third year we will continue to refine our meta-models, and complete our tools with support for the automatic synthesis of control configurations for the effectors deployed within the system.

2.3.7 Task A6.7: Prediction Model Adjustment, Integration, and Validation Methodology

This task contributes to the software performance and reliability prediction approach, developing a methodology for adjustment of QoS meta-model instances, and thus ensuring accuracy of predictions. We developed guidelines of how to estimate performance characteristics (e.g. physical resource demands and processing speeds) in the first two project years. The resulting prediction model adjustment methodology is described in Section 3.5.
In the third project year, we will further investigate adjustment and validation techniques for reliability characteristics, considering the most relevant applications for individual industrial use cases.

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 offers) of potential customers. General and individual offers 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 of software service performance and reliability prediction supports both the creation of general SLA templates and the reaction to individual SLA offers 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 system 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 created a prototype version of the performance prediction service and demonstrated its applicability to the Open Reference Case (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 E-Government (WB B6) industrial use cases. In the third project year, we will once more generalize prediction to consider reliability as a new quality attribute beyond performance. We will provide a reliability prediction service and apply this service to industrial 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 the performance prediction service (Section 3.6). The following two Sections deal with additional investigations regarding cost prediction (Section 3.7) and multi-objective optimization (Section 3.8). Finally, Section 3.9 shows how our work is applied to B-line use cases.

3.2 Overview

This section provides a high-level overview over 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 Offers) 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 (working by M26) and reliability (will be 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 software 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.

Regarding reliability, we will extend our approach for evaluation of the **Target Service Reliability**, namely the probability that upon a service invocation request, the service completes its execution without failures (where we assume that failures may happen due to software faults in service components, as well as unavailable hardware and network resources). This property may be negotiated in an SLA through the standard QoS term `qos:reliability`. Further results of reliability prediction are planned (e.g. **Subservice Reliability**), but not strictly decided yet.

### 3.2.2 Preliminaries

The following discusses essential preconditions for 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 `qos:arrival_rate` and `qos: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](image)

**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 modeling internal parallelism inside a component. The above works were combined in the Palladio Component Model (PCM) [53], a metamodel for specifying performance-relevant information in component-based architectures. PCM is designed with the explicit capability of dividing the model artifacts 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 modeling 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 plan to mitigate 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 intend to mitigate 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 [X40] 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 probabilities [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 would be 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.

A prediction model 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:

- **Service Component Model**
  - **Performance**: Resource demands
  - **Reliability**: Software failures

- **Usage Model**
  - **Performance**: System workload
  - **Reliability**: Service invocations, Service parameters

- **Infrastructure Model**
  - **Performance**: Resource performance, Network performance
  - **Reliability**: Hardware resources, Network topology

- **Allocation Model**
  - **Performance**: Service deployment
  - **Reliability**: Hardware availability, Network failures

Figure 4: QoS Meta-Model Structure (Overview)
• 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 evaluation of software services regarding performance and reliability. 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 arbitrary big 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 behavior, 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.

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.

Figure 5: Goal-oriented Systematic Measurements

The technical core of the approach is a framework that allows for systematic performance evaluations, the Performance Cockpit. Around that technical core, there are four conceptional blocks: Experiment Definition, Automated Measurements, Statistical Inference, and Model Integration. In what follows, 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 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 artifacts. 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 requires 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 to meet 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 artifact 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 behavior, 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.

### 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 analyzing. 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 analyze 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 will automate this process as depicted in Figure 8.

Figure 8: Automated Measurements

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 inbetween 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

We target at integrating our measured functional dependencies into the QoS meta-model (Section 3.4). The meta-model is parameterizable for parameter values as well as for the deployment platform. Moreover, the transformation into queuing networks for performance prediction (Section 3.6.2) supports the use of performance completions, which allow software architects to annotate an architecture model [147]. During the transformation, low-level performance influences, e.g. of a certain middleware [148], are injected as a refinement of the original prediction model. The completions are parametric with respect to resource demands of the annotated element. For each implementation and each execution environment, the demands have to be determined explicitly. The integration of the inferred models into the QoS meta-model allows for design-time performance predictions for systems that build on a large basis of existing components. In [143], we applied our approach to build a performance completion for the message-oriented middleware ActiveMQ 5.3.

3.6 Performance Prediction Service

This section describes the current tooling that we provide for performance prediction (as of M26). Notice that in year 3, we will extend this tooling to also provide reliability prediction capabilities. 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). Last, we present the prediction process for the case that prediction is invoked as part of the SLA management framework during automated negotiation (Section 3.6.3). A specification of the corresponding Service Evaluation component, as well as the Evaluate interaction, is given in Appendix C.

3.6.1 Overview

Figure 9 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 for performance prediction (as of M26) is based on the existing PCM tooling. Here, a very short overview of the way performance prediction works is given. For more details, see [1].

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 a queueing network
model, which is represented as a Java implementation. This queueing network is then simulated, using the discrete-event Java simulation framework SSJ.

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 high-level control and data flow throughout the service components is executed as specified in the prediction model. Furthermore, message transports over network and the consumption of resources are considered by the simulation. Contention effects caused by concurrent service execution and resulting waiting times can be 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. This way, it is ensured that the simulation follows the distributions that have been specified in the prediction model.

Throughout the simulation, sensors are placed that record the simulated start and end times of each service invocation, as well as the history of resource demands and waiting times for resources. 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.

### 3.6.3 Prediction Process

Figure 10 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 10 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 `IllegalArgumentException` 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 `PredictiveSoftwareServiceEvaluator`.
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 `PredictiveSoftwareServiceEvaluator`. 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.

### 3.7 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.

#### 3.7.1 Modeling 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.

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 11. 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 11(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 11(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.

Secondly let us examine the price versus the per-core performance given the same number of cores. In Figure 11(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 12 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:

\[ C_{\text{core}} = g(T_{\text{core}}, N_{\text{core}}) = c_1 T_{\text{core}}^{c_2} + c_3 N_{\text{core}}^{c_4} + c_5, \]  

(2)

where \((c_1, ..., c_5)\) 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:

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

(3)

<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 3: 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 1. 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.

3.7.2 Modeling Power Consumption

Power consumption and associated costs become increasingly significant in modern datacenter environments [142]. In this section we analyze and model the server power consumption of business applications. We study the relationship between system power consumption (Ps, 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 Ps we use a normalized power unit Pnorm, which is defined as follows:

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

where the measured Pidle (U = 0) and Pbusy (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 13.

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 13).

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:

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

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:

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

The fitting result is shown in Figure 13 and the fitted model parameters are listed in Table 2. 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 Pbusy, the overall system power consumption $P_{sys}$ can be obtained by substituting $P_{norm}$ (6) in (4).

<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>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 4: Power Consumption Model Parameters

### 3.7.3 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:

$$Cost(T_{core}, N_{core}, U, I) =$$

$$p_0 + p_1 C_{cpu} + p_2 \int_{t \in I} P_{sys}(U(t)) \, dt, \quad (7)$$

where $t$ is the measurement time, $I$ is the measurement period ($t \in I$), $p_0$ is an adjusting constant, $p_1$, and $p_2$ 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 $P_{sys}(U)I$. The model in (7) uses an additive form to combine server hardware costs and operational costs, in which parameters $p_1$ and $p_2$ have to be set properly to reflect different cost structures.

To summarize from a mathematical modelling perspective, we can conclude that the power function ($c_1x + c_2 + c_3$) 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 $p_1$ (fixed cost) and $p_2$ (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.

### 3.8 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.

### 3.8.1 A SLA-Driven Planning Framework

Firstly we present a framework for SLA-driven planning and optimization, which is shown in Figure 14. 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.

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).

### 3.8.2 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 $y = y'$. The set of non-dominated solutions of a set $Y \subseteq \mathbb{R}^m$ is defined as: $Y_N = \{y \in Y | \nexists y' \in Y : y' \prec y\}$. Given a multi-objective optimization (minimization) problem $f_1(x) \rightarrow \min, \ldots, f_m(x) \rightarrow \min, x \in \mathbb{R}^n$, (8) the image set $Y(S)$ of this problem is defined as $\{y \in \mathbb{R}^m | \exists x \in \mathbb{R}^n : f_1(x) = y_1, \ldots, f_m(x) = y_m\}$. 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.

### 3.9 Use Cases for Performance Prediction

This section demonstrates the application of our prediction approach to the Open Reference Case (Section 3.9.1), as well as the ERP Hosting and E-Government industrial use cases (Sections 3.9.2 and 3.9.3). So far, only performance prediction is available and could be applied to use cases. In the 3rd project year, we will focus on reliability and its application to industrial use cases.

#### 3.9.1 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. This section describes the ORC scenario very 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 throughput of the involved service operations.

**Service Component Model**

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 15: Open Reference Case System](image)

Figure 15 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. Figure 7 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.
As an example for a behavioural specification, Figure 16 depicts the control flow for the `handlePayment()` operation of the composed payment service. Two calls to the card validation and payment debit services are surrounded by internal actions representing internal processing of the payment service.

**Infrastructure Model**

The ORC infrastructure model describes the resource environment to which service and legacy components are allocated. Figure 17 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 separate from the database. Accordingly, three resource containers have been specified that can be used for deployment – the `CompleteRetailSolution_VM`, `Database_VM`, and `ServicesAndLegacyComponents_VM`. Each resource container is equipped with a CPU resource. The processing rates of the CPUs are set to the default value 1.0 (which means one CPU work unit per time unit). These values may be substituted with concrete values derived from infrastructure SLA templates.
To support the distributed deployment option, a connection between the Database_VM and ServicesAndLegacyComponents_VM is needed. Therefore, a linking resource is specified to connect these containers. Latency and throughput values of the connection are specified and assumed to be fixed for the ORC scenario.

**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.

**Usage Model**

The usage model is either created by the software service provider anticipating the behaviour of potential customers, or dynamically created upon a concrete customer request. We have created an example usage model, to reflect a typical user behaviour. Figure 18 shows this model visualized by an EMF editor. System usage is restricted to the `bookSale()` and `getProductDetails()` operations of the basic inventory service, as well as the `handlePayment()` operation of the composed payment service. Both services are provided by the retail solution, as can be seen in Figure 15.

![Figure 18: ORC Usage Model](image)

All operations are invoked concurrently, each with an open workload specifying an inter-arrival time, i.e., a time interval between two consecutive invocations. Performance prediction takes all interferences between the concurrently executed service operations into account.
Each of the three specified usage scenarios is not only equipped with an open workload, but also with a usage behaviour specifying how the service operation is invoked. As an example, Figure 19 shows the usage scenario for the `bookSale()` operation of the inventory service. The corresponding behaviour consists of a single invocation of the `bookSale()` operation. The call includes a description of the list of sales items given as an input parameter to the operation. It specifies that the list has 1 item with probability 0.3, 5 items with probability 0.4, and 10 items with probability 0.3. The specification of number of items is necessary as it influences the control flow of service execution.

### 3.9.2 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. For overcoming these drawbacks, the paradigm of service-orientation has been successfully applied by SAP for building a new ERP solution that allows for simple and flexible composition and configuration of business functionality for creating customer-specific solutions in a highly efficient way.

In this section, we focus on the Sales and Distribution (SD) application which covers a sell-from-stock business process which includes the creation of a customer order with five line items and the corresponding delivery with subsequent goods movement and invoicing. It consists of the following six transactions:

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

There are fifteen dialog steps with 10-second think time in-between. The detailed benchmark steps are shown in Figure 20.
Components and Composition

In order to predict the performance of the SD application for a given workload, we defined a set of components that reflect the major components of an ERP system. Please note that this is a performance abstraction of the system. Therefore, the models presented here do not (necessarily) reflect the actual implementation.
Figure 21: Components and Interfaces of the SD-Application Model

Figure 21 illustrates the components and interfaces necessary for modelling the SD application. The component on the top left resembles the ERP system itself. In order to support the SD scenario, this component implements the ISalesAndDistribution interface that contains the most important operations of the SD application. The ERP component relies on a middleware platform which in case of SAP is the Netweaver ABAP stack. The Netweaver in turn requires a database for which we chose SAP’s MaxDB. Figure 22 shows the assembly of the components introduced above that form the SD application.

Figure 22: ERP Use Case Component Diagram

Behavioural Specification

In the following, we illustrate the behaviour of the three components by means of an example of service createCustomerOrder. Starting from this service we
furthermore describe the RD-SEFFS of the service write of the Netweaver and the service update of the MaxDB.

![Diagram of Transaction createCustomerOrder of SAP ERP 6.0]

**Figure 23: Transaction createCustomerOrder of SAP ERP 6.0**

Figure 23 represents a performance abstraction of service createCustomerOrder. The service receives a number of line items that are modelled by a collection of strings each of which represents a single item. Its variable characterisation NUMBER_OF_ELEMENTS contains the number of line items to be processed. In the first step the order is stored calling the write operation of the middleware. Next the line items are processed internally which includes for example the generation of UI presentations etc.

On the middleware layer the write action is translated into appropriate database statements as shown in Figure 24. In this case there is a probability of 30% that a new entry has to be created in the database and a probability of 70% that the customer order leads to an update.

The update and create request can be directly processed by the database. Figure 25 illustrates the behaviour of the database actions. The operations consist basically of a single internal action that requires a certain amount of processing time on the CPU and the HDD. The required processing times of course depend on the amount of data that has to be stored.
Deployment Models

The execution environment and deployment of the SD application is shown in Figure 26 and Figure 27. The execution environment consists of an application server and a database server. The deployment of the ERP system, the Netweaver ABAP stack, and the MaxDB are accordingly. The ERP system and the Netweaver platform are deployed on the application server and the MaxDB runs (as to be expected) on the database server.
Usage Model

The usage model shown in Figure 28 resembles the behaviour of the SD benchmark that we use as a reference load for our SD application. It is basically a simplification of the full flow shown in Figure 20. The user first creates a new order for a customer with five line elements. Based on the order a new delivery is created in the second step. The delivery is then changed before the user lists all orders of that customer. Finally, an invoice for the customer is created. Please note that this is a simplified representation of the behaviour.
3.9.3 WP B6: E-Government

A special application of our prediction approach could be demonstrated for the E-Government 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 model human operators that serve the booking requests of the calling customers. Our performance prediction shows how long calls take (to be negotiated as \( \text{qos:completion\_time} \) in the SLA), depending on the type of customer request (\( \text{qos:data\_volume} \)) and the call centre capabilities (determined through the prediction model). We can also deduce how many operators are actually busy at a time, and how many of them are required to serve customer requests without waiting times. The results of this application are of special interest as they point towards a new domain of application of our performance and reliability prediction approach. In the following, we give an overview over the prediction model created for WP B6.

Components and Composition

The specified service component types and interfaces are shown by Figure 29. Two types of components are specified: \text{CallCenters} and \text{ExternalCallCenters}. Both provide the functionality to do a booking (i.e., they provide the interface \text{IBook}). In addition, \text{CallCenters} can also delegate the booking functionality to another entity (i.e., \text{CallCenters} also \text{require} the interface \text{IBook}).
have a pool of human operators (the OperatorPool) to serve booking requests by customers. The capacity is originally set to 25 (i.e., there are 25 operators available in a CallCenter).

The concrete health care system is shown in Figure 30. It contains 1 CallCenter and 1 ExternalCallCenter. Booking requests from customers first arrive at the CallCenter. The CallCenter either serves the requests itself or delegates them to the ExternalCallCenter. Notice that other configurations could be possible, e.g. with multiple ExternalCallCenters to which the main CallCenter delegates calls.

**Figure 30: The Health Care System**

**Behavioural Specification**

The behavioural specification has two aspects: (i) the behaviour of customers, and (ii) the behaviour of CallCenters. The customer behaviour is specified by the usage model as shown in Figure 31. A new customer "arrives" every 7.2 seconds (qos:arrival_rate, reflects 500 calls per hour) and requests a booking. Customers require a health booking, which takes between 120 seconds and 600 seconds. Additionally, they may require a mobility booking, also taking 120 seconds to 600 seconds. In summary, a customer request takes between 120 seconds and 1,200 seconds (qos:data_volume).
As shown in Figure 30, customer requests are first handled by the CallCenter. Figure 32 shows how the CallCenter handles the requests. First of all, a decision is made if the CallCenter serves the request itself or delegates it. In the default case, all calls are directly served, and none are delegated. If the call is directly served, an operator from the pool of available operators answers the call and performs the booking (health and mobility). If no operator is available, the customer has to wait until the next operator becomes available. Any request delegated to the ExternalCallCenter would be served there. In the current version of the model, the booking would be done instantaneously by the ExternalCallCenter, without any waiting times for customers.
**Performance Prediction**

Using the described prediction model as an input for performance prediction, we get the probability distribution of the total serving time of a customer booking request \( \text{qos:completion\_time} \), which is the sum of the following 3 factors:

- **Waiting time for the next free operator:** If the request is delegated to the \text{ExternalCallCenter}, there is no waiting time. If the request is directly served by the \text{CallCenter}, the waiting time depends on the "system load".

- **Health booking time:** 120 / 160 / 600 seconds, depending on the type of request \( \text{qos:data\_volume} \)

- **Mobility booking time:** 0 / 120 / 600 seconds, depending on the type of request \( \text{qos:data\_volume} \)

The total serving time is influenced by customer inter-arrival times, request types for health and mobility booking, the number of available operators, and the percentage of calls delegated to the \text{ExternalCallCenter}. See the WP B6 deliverable for prediction results.

## 4 Resource Usage Prediction

### 4.1 Introduction

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 \text{cpu} usage (such as user, system or process), \text{load}, \text{memory} (free, cached, etc), \text{disk} and \text{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 33 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 34 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 34: Basic Training/Prediction Workflow**

**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.

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 35: Dynamic Prediction With Future Error](image)

Figure 35 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 36: Training and Re-training Workflow

Figure 36 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 37: Prediction versus Real Instrumentation

Figure 37 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 \texttt{cpu\_user} metric while the red line represents the predicted value of the metric. As observed, the \texttt{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 38 illustrates a spike scenario whereby the behaviour of the system is on average, constant and steady. On a system whose behaviour (on average) is flat, short spikes cannot be predicted easily as they could be considered random events and it will contribute little to the overall behaviour.

Figure 38: Steady-State Metric with Spikes

Figure 39 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 \texttt{cpu\_user} is 12.5 (std=9.7) and for \texttt{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%.

Figure 39: Prediction versus Real Instrumentation (Running Average)

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 39 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 40.

![Figure 40: Resource Usage Prediction User Interface: Training](image)

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.

![Figure 41: Resource Usage Predictiction User Interface: Testing](image)

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 42 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 43 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.

Figure 44 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.

```
SELECT ALL servers WHERE
MemFree > 2GB AND
CPUNum >= 2, AND
RealAverageCPUUsage1Hour <= Low AND
RealAverageCPUUsage1Day <= Medium AND
PredictionAverageCPUUsage1Hour = Low
PredictionError < 0.2
```

**Figure 44: SQL-like Query for Prediction Scenario**

**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]
- VM [1…N] [PastMetrics, PresentMetrics, FutureMetrics]
- User[id] SLA[id] VM[1..N] [PastMetrics, PresentMetrics, FutureMetrics]

Historical information could be used to characterise provisioned SLAs in terms of past provisioned resources and usage profiles that link SLAs, customers and resources. From an Enterprise IT perspective, it is important that resources are not under-utilised and to record SLAs that require high or low amount of utilization in order to maximise the return-on-investment (ROI) on computing assets and this information should be available for all the components that manage the resources during provisioning or at runtime. In turn, the historical information could also used for generating training models for prediction in terms of past provisioned SLAs and the customer’s usage and prediction profile.

Within this scenario, the main idea presented was to show, in a clear and simple way, how prediction services could be consumed by the different use cases that require infrastructural services. In Y2 of the project the work has been geared towards realising this scenario by means of architecting and developing a working prototype that is self-contained and can be run as a service and integrated within the SLA@SOI platform.
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

Running prediction on-demand is not a cost free operation as prediction is a very cpu-intensive process. Figure 45 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.

![Figure 45: Resource Usage (cpu_user) Over Time](image)

As illustrated in Figure 45, 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 43 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 message-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]: Asynchrony, 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 ClientAgent

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 46 illustrates how the raw metrics are gathered locally and pre-processed. For the current implementation, the local gathering subsystem has 3 separate functions: **Instrumentation**, **Local Historical Database (HistoricalDB)** and **Query**.

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 daemon 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).

![Diagram: Prediction Data Gathering and Processing]

**Figure 46: Prediction Data Gathering and Processing**

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 PredictionAgent

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).

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:

```
[historytotal, lasthour, lastMinute, prectionerror, 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 change slowly in general depending on 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 47 illustrates the output (xml) of the prediction service REST call.

```xml
<Service>
  <timestamp value="1274255593126">
    <metric name="cputotal" value="15345284288"/>
    <metric name="cpuhistorylast60seconds" value="3732000000"/>
    <metric name="cpuhistorylast5minutes" value="2930000000"/>
    <metric name="cpuappropriatederror" value="50"/>
    <metric name="cpuappropriatedlast30seconds" value="160"/>
    <metric name="cpuappropriatedprediction" value="257"/>
  </timestamp>
  <timestamp value="12742555620763">
    <metric name="cputotal" value="15345284288"/>
    <metric name="cpuhistorylast60seconds" value="3732000000"/>
    <metric name="cpuhistorylast5minutes" value="2930000000"/>
    <metric name="cpuappropriatederror" value="50"/>
    <metric name="cpuappropriatedlast30seconds" value="160"/>
    <metric name="cpuappropriatedprediction" value="257"/>
  </timestamp>
  <timestamp value="1274255550542">
    <metric name="cputotal" value="15345284288"/>
    <metric name="cpuhistorylast60seconds" value="3732000000"/>
    <metric name="cpuhistorylast5minutes" value="2930000000"/>
    <metric name="cpuappropriatederror" value="50"/>
    <metric name="cpuappropriatedlast30seconds" value="160"/>
    <metric name="cpuappropriatedprediction" value="257"/>
  </timestamp>
  <timestamp value="12742555508515">
    <metric name="cputotal" value="15345284288"/>
    <metric name="cpuhistorylast60seconds" value="3732000000"/>
    <metric name="cpuhistorylast5minutes" value="2930000000"/>
    <metric name="cpuappropriatederror" value="50"/>
    <metric name="cpuappropriatedlast30seconds" value="160"/>
    <metric name="cpuappropriatedprediction" value="257"/>
  </timestamp>
</Service>
```

*Figure 47: Prediction for cpu_usage* Metric*

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 48 illustrates the B4 UseCase enterprise IT displaying monitoring and prediction data for provisioned services.
4.5.3 **PredictionWorker: Prediction (Network) Provisioning**

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 49) illustrates the protocol used as a sequence diagram of message passing required to find PredictionServices (ServiceAgents or PredictionWorkers) nodes.
Figure 49: 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

Our requirements 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 requirements and view any deviations from these requirements 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 requirements 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 the monitoring framework is a monitoring policy that contains the formulas to be monitored and other monitoring parameters.

5.2 State of the Art

Monitoring the preservation of QoS properties during the operation of systems is an important verification measure for checking if the current service usage is compliant with the agreed SLAs. However, it does not always provide sufficient scope for taking control actions against violations as it only detects problems after they occur. Prediction techniques aim to detect potential violations of QoS properties before they occur for taking control actions to avoid the violations.

Intrusion Detection Systems (IDS) first attempted to solve the problem of identifying a violation of a system's rule. An IDS is software and/or hardware designed to detect unwanted attempts to access, manipulate, and/or disable of computer systems, mainly through a network, such as the Internet. These attempts may take the form of attacks, as examples, by crackers, malware and/or disgruntled employees.

Many different IDS techniques and approaches have been studied [170]. Existing approaches to intrusion detection have been distinguished into anomaly-based and misuse-based. Anomaly-based approaches [164, 165, 166] assume that attacks involve some abnormal behaviour of the system that is being monitored. Intrusions are, thus, detected as deviations from the expected normal behaviour of the system. Misuse-based approaches [167, 168, 169], on the other hand, are based on models of known attacks.

Unfortunately, these approaches are only able to detect an intrusion/violation at the moment it happens not always providing sufficient scope for taking control actions against violations. Therefore, something more it is needed, something that provides the system with enough time to react before a violation happening. Predictive techniques are thus needed.

Both academic and industry have been spending their efforts in studying the challenging field of predictive analytics. Predictive analytics is an area of statistical analysis that deals with extracting information from data and using it to predict future trends and behaviour patterns. The core of predictive analytics relies on capturing relationships between explanatory variables and the predicted variables from past occurrences, and exploiting it to predict future outcomes.

The approaches and techniques used to conduct predictive analytics can broadly be grouped into regression techniques and machine learning techniques.

Regression models are the mainstay of predictive analytics. The focus lies on establishing a mathematical equation as a model to represent the interactions
between the different variables in consideration. Depending on the situation, there is a wide variety of models that can be applied while performing predictive analytics. Some of them are: linear regression model [171], discrete choice models [172, 173], logistic regression [174], probit regression [175], time series models [176], survival analysis [177], and multivariate adaptive regression splines [178].

Machine learning was originally employed to develop techniques to enable computers to learn. Today, since it includes a number of advanced statistical methods for regression and classification, it finds application in a wide variety of fields including medical diagnostics, credit card fraud detection, face and speech recognition and analysis of the stock market. In certain applications it is sufficient to directly predict the dependent variable without focusing on the underlying relationships between variables. In other cases, the underlying relationships can be very complex and the mathematical form of the dependencies unknown. For such cases, machine learning techniques emulate human cognition and learn from training examples to predict future events. Some machine learning techniques are: neural networks [179], radial basis functions [180], support vector machines [181], naive Bayes [182], and k-nearest neighbours [183].

Even if both regression and machine learning techniques can be used for runtime prediction of future system’s behaviours, there are many factors which affect the decision of using one technique instead of another, e.g., its application fields, its preconditions, and its computational time. The latter property is very important in failure/violation prediction. Indeed, the smaller is the computational time required for making a prediction, the bigger is the scope for taking control actions against likely failure/violation.

From a temporal point of view, prediction techniques can be roughly classified in long-term and short-term prediction. Long-term prediction techniques use statistical models to foresee events that might happen after months or years from the time in which the prediction is made. These techniques are mostly used in financial and social sciences to predict economic or social trends, e.g., stock market quotations or populations evolution. On the other hand, short-term prediction techniques use statistical model to foreseen events that might happen after days, hours, minutes or even seconds. These techniques are mostly used in fields characterised by a high level of dynamicity, e.g., networking and runtime systems. Both long- and short-term prediction techniques base them predictions on past data.

Most of the approaches discussed above focus on the prediction of single properties, e.g., CPU load, throughput, and disk usage. This kind of properties can be well modelled using statistical models. Moreover, historical data about these properties are easy to collect using many available monitoring techniques (static and dynamic code instrumentation) and tool (Ganglia [184], Nagois [185], MonALISA [186]).

There are many other properties that are useful to be predicted at the service level, e.g., correctness of an interaction protocol. Unfortunately, these properties are often complex and cannot be predicted with the above techniques, if used separately.

Approaches addressing these issues can be found in the autonomic computing field [187] that, according to IBM, can be divided in the following four functional areas:

- Self-Configuration: Automatic configuration of components;
- Self-Healing: Automatic discovery, and correction of faults;
• Self-Optimization: Automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;
• Self-Protection: Proactive identification and protection from arbitrary attacks.

The two fields that are mostly related to predictive issues are self-healing and self-protecting. Moreover, that latter explicitly take into account proactive abilities to identify likely threats that might lead the system to an unstable state.

A self-protective approach for identifying likely dangerous actions is presented in [188]. The approach is named From Failure To Vaccine (FFTV). FFTV observes values at relevant program points. When the technique detects a software failure, it uses the collected information to identify the execution contexts that lead to the failure, and automatically enables mechanisms for preventing future occurrences of failures of the same type. Thus, failures do not occur again after the first detection of a failure of the same type.

FFTV uses different techniques combined together to produce its prediction. It uses interaction and data model. The former is represented using Extended Finite State Machines (EFSM), whilst the latter is represented using invariants over collected data.

A predictive approach using predictive data mining for intrusion detection (ID) and www prediction (WWW) applications is presented in [189]. Data Mining is an analytical process to analyze, explore, and summarize large amounts of data in order to uncover new patterns and/or to discover new relationships between variables. Predictive data mining is the most common type of data mining and it has the most important business applications.

ID uses Support Vector Machines (SVM) for classification. The SVM is one of the most successful classification algorithms in the data mining area, but its long training time limits its use. The authors also present a study for enhancing the training time of SVM, specifically when dealing with large data sets, using hierarchical clustering analysis. They use the Dynamically Growing Self-Organizing Tree (DGSOT) algorithm for clustering. Clustering analysis helps find the boundary points, which are the most qualified data points to train SVM, between two classes. They present a new approach of combining SVM and DGSOT, which starts with an initial training set and expands it gradually using the clustering structure produced by the DGSOT algorithm.

WWW Prediction is the problem of predicting the next page(s) a user might visit after surfing a web site. The improvement of many applications depends on surfing prediction. They propose a hybrid model that combines three classification techniques, namely, Support Vector Machines, Markov model, and Artificial Neural Networks, to resolve prediction using Dempster's Rule. Such fusion overcomes the inability in predicting the unseen data in the case of Markov model and the complexity of multi-class problem in the case of Artificial Neural Networks and Support Vector Machines, especially when dealing with large number of classes.

A predictive approach for intrusion detection is presented in [190]. The authors present a network intrusion attempts prediction model based on fuzzy neural network which is based on the observation of network packet sequences.

Also in this case the authors do not use only one technique for addressing the consider issues but use a combined approach based on classic neural network enhanced with the fuzzy logic theory.
### 5.3 The Overall Approach

Our approach for detecting potential violations of QoS properties is model-based, i.e., it uses models to establish what to predict and how to predict it. These models are used as input for the prediction framework (PF) that, together with the monitoring framework (MF), and through the use of a set of QoS predictors, detects potential violations of QoS properties. Figure 50 shows the overall architecture of our approach together with the specifications that need to be provided to them.

![Figure 50: Prediction Framework Components](image)

More specifically, our approach uses two frameworks: MF and PF. MF is used to reason over received events, e.g., events reporting about service operation calls, responses, or failures. By analysing sequences of received events, it also computes and stores QoS values according to QoS specifications, e.g., the mean time to repair for a service (MTTR), the time to fail for a service (TTF), or the service throughput. PF is used to predict QoS violations according to prediction specifications (PS) and the QoS prediction configurations that are provided as input to it. To predict violations, PF uses a set of QoS predictors. These are pluggable software components implementing specific prediction algorithms. PF also uses historical QoS data provided by MF. From these data it infers statistical models, e.g., probability distributions that are used by QoS predictors.

Given a PS, MF automatically generates a monitoring specification (MS) needed by the MF to start monitoring and computing QoS term values, as shown Figure 51.

![Figure 51: Prediction Specification to Monitoring Specification](image)
A PS is composed of a guaranteed state specifying the QoS term that should be monitored and predicted, and any additional parameters indicating additional QoS terms that must be monitored in order to be used in the generation of predictions (cf. parameters for prediction in Figure 51). For instance, to predict the MTTR QoS term’s violations a predictor may need to know TTF QoS term values as well. This additional information is specified by the prediction specification parameters.

From the “Guaranteed State for QoS Term” section of a PS, MF generates state monitoring rules by initialising state monitoring rule templates chosen with respect to the QoS term to be monitored. For instance, for the guaranteed state for QoS term MTTR<K will be instantiated the state monitoring rules for monitoring specific event sequences to compute MTTR values. Once a rule has been instantiated, MTTR values are computed and stored into the QoS database. If MF doesn’t have the mean for computing a specific QoS term, a user can send to PF, along with the PS, an event calculus rule for computing the desired QoS term. For instance, a user can create its own rule for monitoring events to compute time to fail (TTF) QoS values and send it to MF along with a PS. In this way MF is able to compute TTF values and store them into the QoS database.

Our approach provides a general and extensible framework for reasoning about potential violations of QoS properties. It is general because it doesn’t support a limited set of QoS only. On contrary, it provides a support for automatically collecting and analysing data and reasoning on actions and their effects over time. Our approach is extensible because the definition of which data to collect and how to analyse them can be specified using models (QoS specifications) and pluggable components (QoS predictors). In this way there is no need for modifying PF core components (MF and PF).

### 5.3.1 QoS Specification

QoS specifications are formal specifications that are used by MF to identify which QoS properties to monitor and store data for at run-time. QoS specifications are expressed using Event Calculus (EC), a logical language for representing and reasoning about actions and their effects over time.

An example of QoS specification is given by in formula (1). The formula checks whether the mean-time-to-repair (MTTR) for a service, i.e., the mean of the time periods between a time point when a service became unavailable and the time point following it when the service became available again, is less than a given threshold value K.

\[
\begin{align*}
\text{Happens}(e(\text{id}_1, \text{Svc}, \text{UUID}, \text{Call}(O), \text{UUID}), t_1, [t_1, t_2]) & \wedge \\
\text{Happens}(e(\text{id}_2, \text{UUID}, \text{Svc}, \text{Response}(O), \text{UUID}), t_2, [t_1, t_2 + d]) & \wedge \\
\exists \text{PN}, \text{STime}, \text{TTRs}[] : \text{HoldsAt(Unavailable(}_{\text{PN}, \text{UUID}, \text{STime}}), t_1)) & \wedge \\
\text{HoldsAt(Unavailable(Pe}r(\text{oids(}_{\text{UUID}, \text{PN}, \text{TTRs}[]}), t_2)) & \Rightarrow \\
\text{LESS\_THAN(average(}_{\text{TTRs}[]), K)}
\end{align*}
\]

(1)

The first two conditions in the rule check whether an invocation of an operation of service UUID has been served, i.e., if an event \(e(\text{id}_1, ...)\) representing the call of an operation of service UUID that occurs at time \(t_1\) is followed by an event \(e(\text{id}_2, ...\) representing the response of service UUID to the same call that occurs at a time \(t_2\) in the range \([t_1, t_1 + d]\). This event pattern assumes that an operation call is served if a response to it occurs within \(d\) time units following its occurrence. Formula (1) checks also if this sequence of call and response events occurs at a time when the relevant service has been known to be unavailable. The
unavailability of the service is checked by the condition \texttt{HoldsAt(\ldots,t1)} that checks if the fluent \texttt{Unavailable(\_PN,UUID\_STime)}, which represents that a service UUID is in an unavailability period that started at time \_STime, is unavailable at time \texttt{t1}, i.e., the occurrence of the operation call that was served. If the latter condition holds, the rule retrieves the record of the historic record of the times-to-repair of the UUID that is stored in the fluent \texttt{UnavailablePeroids(UUID\_PN\_TTRs[])} and checks if the average value of these times, i.e., \texttt{average(_TTRs[])} is less than \texttt{K}. The \texttt{LESS THAN} operator is a function that evaluates any of the usual relational operators, e.g., $\leq$, $\geq$, $=.$

In the formula (1), the variable \texttt{TTRs[]} holds the values of all the time to repair (TTR) time intervals that have been computed. Hence, \texttt{average(TTRs[])} computes the MTTR value over an actual array of TTR. Given the formula (1), MF automatically computes and stores TTR and MTTR values into a database that can be queried to retrieve this information.

### 5.3.2 QoS Predictor And QoS Predictor Configuration

A QoS predictor is a PF software component that implements a prediction algorithm. PF defines the interface every QoS predictor must implement, and provides mechanisms for loading and unloading QoS predictors. From an architectural point of view a QoS predictor can be seen as a PF plug-in. This way the PF can be easily extended by simply adding and removing its QoS predictors (plug-ins).

QoS predictors implement prediction algorithms and can access the historical data about QoS values and past events provided by MF. For instance, given a PS specific id, a QoS predictor can query MF for MTTR and TTR values by invoking the operations: \texttt{select(PSid,"qos.term.MTTR")} or \texttt{select(PSid,"qos.term.TTR").} QoS predictors can also access statistical models computed by FM by analyzing MF historical data.

Since the available QoS predictors are not known a priori, a mechanism to provide them with their configurations is also provided. QoS predictors can be configured by using specific configurations containing QoS predictor specific configuration parameters. An example of PS is given in Figure 52, where QoS predictor \texttt{predictor1} is associated with parameter \texttt{history.window} equal to 400 events.

```plaintext
predictor_configuration{
    id = predictor1
    configuration_property{
        key = history.window
        value = 400events
    }
}
```

**Figure 52: QoS Predictor Configuration Example**

Figure 53 shows the BNF representation of the QoS predictor specification.
5.3.3 Prediction Specification

A prediction specification (PS) tells PF which kind of predictions have to be performed. For expressing PSs we adopted the SLA(T) syntax defined by the EU project SLA@SOI and extended it to make it suitable for expressing prediction requirements. An example of PS is given in Figure 54.

In this example each PS specifies a service level agreement term whose id field identifies the service the prediction is about. An agreement term encloses also a guaranteed state element which specifies the QoS term that the prediction is required for (service1.MTTR in our example), the constraint that should hold for this term and whose violation will be the subject of prediction (i.e., qos.term.MTTR < 20sec in our example), and the window of the prediction (i.e., the time period in the future that the prediction should be concerned with). This window is set to 10 mins in our example, meaning that the prediction required in this instance should be whether the MTTR of service1 will be greater than or equal to 20 seconds within 10 minutes following the prediction request. Note that a prediction specification uses a QoS name that is also used in a QoS specification, and therefore, enables the QoS predictor to retrieve historical QoS data for the term in order to compute the statistical prediction model for it.

The BNF grammar of the extended SLA(T) syntax for the specification of a prediction is shown in Figure 55.
5.3.4 QoS Predictors

QoS predictors (QoSP) implement prediction algorithms. There can be many QoSPs depending on the predicting features exposed by the PF. A QoSP can be specific or generic. Specific QoSPs generate predictions about single QoS, e.g., throughput of a service, whilst generic QoSPs generate predictions about classes of QoS sharing common predicting algorithms, e.g., mean-time-to-* (MTT*) prediction algorithm that is used to predict the mean time to repair (MTTR) and the mean time to failure (MTTF) for a service (the latter is the mean period between the time point when a service becomes available after a failure to respond to an operation call and the next time that it fails again).

At a very high level, all predicting issues can be generalised in the following problem: given a request of prediction about a QoS parameter at a time point \( t_c \) and a set of historical data composed of \( N \) observations, compute the probability to satisfy the constraints of a prediction specification at a time point \( t_e \) in the future. Predictive algorithms share a framework of common definitions presented in Figure 7, where \( t_c \) is the time point a prediction is requested, \( t_e \) is the time point in the future a prediction refers to, \( p \) is the prediction window (i.e., \( t_e - t_c \)), \( N \) is the number of QoS observations between \( t_c \) and \( t_e \), \( Y \) is the number of future QoS observations between \( t_c \) and \( t_e \), \( QoSc \) is the value of the observed QoS at the time point \( t_c \), and \( QoS_y \) is the value of the predicted QoS at the time point \( t_e \).

During the monitoring phase, new TTR and MTTR values are recorded by the monitor when a service responds to a call of one of its operations (after having processed it) for the first time after a period of “failure”, i.e., the period since the last time that it failed to respond to one of its operation calls. The MF stores the collected values into a QoS database.
After the QoS update, PF performs a data analysis to determine the best-fit distribution for each stored QoS term, e.g., MTTR, TTR. The computed best-fit distribution parameters are stored into the model database to be retrieved and used during the prediction phase.

During the prediction phase, PF executes the suitable predicting algorithms to produce a prediction result for the requested QoS. QoS predictors implement those algorithms.

### 5.3.5 Experimental Results

This section presents our preliminary experimental results focusing on the detection of potential violations of the MTTR QoS property. To predict about MTTR violations we created a specification of the MTTR QoS property, that is the EC formula that tells to MF what to compute, e.g., MTTR, TTR, and TTF values. Then we implemented our MTTR QoS predictor. As data source we generated a set of events simulating service-customer interactions, i.e., service operations calls performed by a customer and corresponding service responses. To compute MTTR values we needed the notion of failure; we defined failure a situation in which the time difference between a response event and its associated request event is greater than a threshold $T : t_{\text{res}} - t_{\text{req}} > T$.

To compute QoS statistical models used by MTTR QoS predictor we analyzed historical data by using EasyFit, a software tool that infers the best-fit probability distribution to a given data set chosen among over 40 different distributions.

In these experiments we wanted to validate our approach with respect to three dimensions: precision, recall, and performance.

**Precision and Recall**

To evaluate the MTTR QoS predictor precision and recall we randomly generated 5 event sets and we considered the PS in Figure 47, where the value 20sec in the constraint qos.term.MTTR < 20sec is replaced with the variable $K$ and the value 10min in the constraint prediction.window = 10min is replaced with the variable $H$. Values for $K$ and $H$ are assigned at run-time. $K$ is determined by starting from a chosen event history size. For instance, if the event history size is 500, we compute $MTTR_c$ that is the value of MTTR at the time point the 500th is received and we select a random number $K$ such that $K \in [85\% \times MTTR_c, 115\% \times MTTR_c]$. The $H$ value is selected between 5000, 50000, and 500000 milliseconds.

For each event set we ran 9 experiments with different event history sizes and prediction windows and for all of them we computed the probability of violating the PS in Figure 5 with the actual values for $K$ and $H$ assigned at run-time. After
every execution we checked whether the prediction was wrong or correct. For instance, set the event history in 500 events and the prediction windows in 5000ms, we ran the PF against 5 different event sets randomly generated in order to obtain 5 prediction values. The results have been categorized into four groups: true positive, false negative, true negative, and false negative. Where a case is positive (negative) when the prediction value is greater than (less than) 0.5, and it is true (false) if the PS was (was not be) violated.

After having ran all the experiments, for each history size and prediction window pair we obtained a set of 5 categorized results, one for each event set. Using these results we computed precision and recall according to formulas (12) and (13).

\[
\text{precision}_{pos} = \frac{TP}{TP + FP} \quad \text{recall}_{neg} = \frac{TP}{TP + TN} \\
\text{precision}_{neg} = \frac{TN}{TN + FN} \quad \text{recall}_{neg} = \frac{TN}{TP + TN}
\]

The overall experimental results are reported in Table 1.

<table>
<thead>
<tr>
<th>Prediction window</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>100.00%</td>
<td>50.00%</td>
</tr>
<tr>
<td>50000</td>
<td>100.00%</td>
<td>40.00%</td>
</tr>
<tr>
<td>500000</td>
<td>100.00%</td>
<td>40.00%</td>
</tr>
</tbody>
</table>

Table 1. From column 3 to 5 are reported prediction and recall values of positive signals. From column 6 to 8 are reported prediction and recall values of negative signals

We observe that the precision increases as the size of the prediction periods increases. This could be because in a longer period there will be more cases of failure and therefore a more extended data set for calculating an MTTR consistent with the model fitted to historic data. Moreover, precision of negative signals higher in general because we registered \(|TN| \gg |FN|\). Perhaps this is due to the limited number of performed experiments. Future experimentations will be performed considering more and bigger event sets so to fully explore how precision and recall are affected by event history and prediction window.

# 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 run-time measurements necessary for monitoring the performance as well as control operations used for adjusting the performance at runtime; (ii) the implementation of a corresponding instrumentation of the service components in terms of sensors and effectors; (iii) the configuration of the SLA management framework (at least with SLOs, metrics as well as data computation and monitoring/control rules).

In this document, we detail the work on an engineering methodology for manageable service components that extends the initial results obtained during
the project’s first year. In particular, we present the meta-models that support the monitoring and analysis of key indicators of the managed system, detail how design-time models can be created using an Eclipse SLA@SOI modelling plugin, and present how they can be used to automatically synthesize the instrumentation configurations that will provide intelligent sensors in the execution environment.

We also give insight into our ongoing work on modelling the control capabilities of a service-based system. We present an initial meta-model that covers dynamic binding. This notion will be further investigated throughout the project’s third year, as we will extend our tools to provide the automatic synthesis of instrumentation for control.

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, SLA analysis. Indeed, the idea is to define at design-time the set of elements that will be needed to successfully manage SLAs at runtime.

In particular, the meta-model is composed of three distinct, but complementary, parts: (i) the Raw Data Sampling (RDS) meta-model that allows designers to define the data sources (elements) to be monitored at run-time, (ii) the KPI meta-model that allows designers to specify the Key Performance Indicators to be calculated on the data sources defined by the RDS model, and (iii) the Adjustment meta-model that supports dynamic binding in our service-based systems.

6.2.1 The Raw Data Sampling Meta-Model

Our approach relies on the assumption that the system to be monitored is modeled by means of SCA. It does not describe the functional properties of the system itself. Rather it represents a complementary and incremental view to the SCA-based functional design.
Figure 57: The RDS Meta-Model

The cornerstone element of the meta-model is the ManagedSystem, which identifies the system under development. It specifies the ServiceComponents that constitute the system, and that represent the main ManagedElements of interest. Each ServiceComponent exposes a set of ServiceComponentActions that implement its 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. This distinction allows designers to express the need to monitor and analyze a specific action in the context of a specific scope. Among the possible internal actions, Figure 57 shows a few key examples, namely conditional flows (ConditionalFlow), loops (Loop), and variables (Variable). However, the RDS 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 RDS meta-model distinguishes between provided (ServiceOperationCall) and required (ReferenceOperationCall) operation calls.

The ServiceComponentActions will contribute raw data to the calculation of the KPIs of interest. The data are identified by ManagedElementProperties, which can be general in nature (StartTime and EndTime) or ServiceComponentAction-specific. In particular, operation calls can identify incoming (IncomingMsg) and outgoing OutgoingMsg messages, as well as start and end times.

Samplers define how to collect the raw data and provide them for further processing. To this extent, the RDS meta-model currently provides two kinds of samplers. InterruptSamplers monitor the property of interest and provide it for
processing if and only if the its status changes. These samplers support the
notion of interval to limit their outputs –e.g., output once every x changes in the
property’s status. PollingSamplers check a property and provide it for processing
periodically. They also support a notion of interval to limit their outputs –e.g.,
output once every x minutes. Both kinds of samplers can be further 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.

6.2.2 The KPI Meta-Model

While the RDS meta-model allows designers to specify the sampling of raw data,
the KPI meta-model specifies how the raw data can be correlated and aggregated
to produce the desired performance indicators, and how these indicators can be
analyzed.

![Figure 58: The KPI Meta-Model](image)

Referring to Figure 58, all the data that are processable in the model are obtained
through a DataSource. Samplers are one example of a DataSource; DataProcessors are another. They read data from one or more sources, process
them and, if needed, provide the processed data as a new source. This
mechanism allows designers to combine different DataProcessors in a pipe-and-
filter fashion to achieve more complex data processing.

The KPI meta-model defines two different kinds of DataProcessors: KPIs and
Filters. KPIs read data from one (or more) sources, combine them by means of
well-defined rules, and provide their output as a new data source. In the model
we introduce the ResponseTime KPI, which computes the time elapsed between
two time instants (e.g., the end time of a service operation call and its
 corresponding start time), the Rate KPI which calculates the rate of arrival of a
certain event, and the Reliability KPI, which computes the number of correct
interactions with a service over the total number of interactions attempted. When
using a Rate or a Reliability KPI 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. Note that the KPI meta-model is extensible and that new KPI patterns can easily be added.

Filters read data from a single source and provide them as a new source if they satisfy a well-defined constraint. These are useful to define simple KPI analyses. For example, if we want to be notified when a response time is higher than a given threshold we can use a Highpass Filter, and if we want to be notified when a service’s reliability drops below a given threshold we can use a Lowpass Filter.

### 6.2.3 The Adjustment Meta-Model

The Adjustment Meta-Model is used to specify the Adjustment actions that can be taken on the designed system. Currently, we support dynamic binding for BPEL processes.

![Figure 59: The Adjustment Meta-Model](image)

Figure 59 shows a generalized approach to dynamic binding capable of supporting different implementations of this feature. Dynamic binding is a type of adjustment that can be applied to the binding that ties a service’s ReferenceOperationCall and a ServiceOperationCall provided by a partner service. The goal is to design the possible dynamic binding strategies that will be available in the system at runtime. This is why, in our meta-model, we allow more than one DynamicBinding to be defined per binding.

Each DynamicBinding has one SelectionCriteria. A SelectionCriteria defines the criteria that is applied to ServiceOperationCalls that are being considered as candidates for the ReferenceOperationCall. The criteria can be as simple or as complex as desired, and may predicate over Indicators. The candidate ServiceOperationCalls must make the indicators available, and for coherence we support the same indicators we are capable of producing using our KPI modelling. The SelectionCriteria produces a metric value for each candidate ServiceOperationCall, indicating its degree of alignment with respect to the criteria.
Each DynamicBinding also has an OrderingCriteria. An OrderingCriteria defines a strategy for ordering the metric values produced applying the SelectionCriteria to the candidate ServiceOperationCalls. Currently we support the Ascending and Descending criterias, although more will be added in the project’s next year. The goal of a criteria is to establish the order in which the candidate ServiceOperationCalls should be considered when dynamically changing the service’s binding.

6.3 Automatic Synthesis of Basic Instrumentation Directives

The platform-independent RDS model specifies “when” and “where” to perform the sampling, and “what” data to collect. Table 3 summarizes how these notions are mapped into instrumentation-specific data collection configurations. In particular, we illustrate how they relate to the instrumentation techniques we have provided for the Dynamic Orchestration Engine in A3.

<table>
<thead>
<tr>
<th>When to collect…</th>
<th>Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>startWindow</td>
<td>validity.from</td>
</tr>
<tr>
<td>endWindow</td>
<td>validity.to</td>
</tr>
<tr>
<td>repetitionUnit</td>
<td>validity.repUnit</td>
</tr>
<tr>
<td>repetitionValue</td>
<td>validity.repValue</td>
</tr>
<tr>
<td>NA</td>
<td>correlationKey</td>
</tr>
<tr>
<td>NA</td>
<td>correlationValue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Where to collect…</th>
<th>Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceComponent name</td>
<td>processID</td>
</tr>
<tr>
<td>ServiceComponent -&gt; ServiceComponentAction</td>
<td>operationID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What to collect…</th>
<th>Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageInstanceProperty</td>
<td>dataExtraction</td>
</tr>
</tbody>
</table>

Table 5: Mapping the RDS Model to Instrumentation Directives

The Sampler’s activation window (startWindow, endWindow, repetitionUnit, and repetitionValue) defines “when” to stop a process to perform data collection. The instrumentation approach supports a similar notion that goes under the name of validity. If a process execution occurs within a validity time-frame, data collection is performed; if not, it is ignored. If no validity is given, the default behaviour is to keep data collection on at all times.

Our instrumentation supports the definition of different data collection specifications for different process clients. This is achieved by specifying a correlationKey and a correlationValue, which are unique to a given client. The correlationKey is the name of an XML element that is present in the SOAP message that initiates the process (e.g., “clientName”), and the correlationValue is the value we expect to find therein at runtime (e.g., “name”). These data are checked every time a new process is instantiated to determine whether an appropriate sensor needs to be instantiated as well. If no correlationKey and no correlationValue are given, the default behaviour is to keep data collection on. Our models do not acknowledge this capability since its goal is to model what will be present in the system to ensure it is manageable. Once an SLA has been negotiated with a certain client, the sensors that are actually activated for that
client can be selected using this notion of correlationKey and correlationValue. However, this is a decision that is not made during the design-time modeling of the manageable system, but later on in the life-cycle.

“Where” to perform sampling depends on where the sampler is positioned in the platform-independent RDS model. A Sampler is attached to a ManagedElementProperty of a specific ServiceComponentAction, which is recursively contained within a ServiceComponent. In our instrumentation approach, a processID and an operationID specify where data collection must be performed. The former is a unique identifier of the process deployed within the Dynamic Orchestration Engine; its value corresponds to the name of the ServiceComponent in the platform-independent model (e.g., loanApproval). The latter is an XPath expression that uniquely identifies a BPEL activity within the process’ XML specification; its value is extracted by analyzing in the RDS model the path that leads from the ServiceComponent down to the ServiceComponentAction being sampled (e.g., /process/flow/receive). The kind of MessageInstanceProperty used in the RDS model determines “what” has to be collected. When the instrumentation collects data, the data are always timestamped. Therefore, when the designer uses a StartTime or an EndTime MessageInstanceProperty, no further configuration is needed. If the designer uses an IncomingMsg or an OutgoingMsg MessageInstanceProperty, this information is translated into a configuration field in the instrumentation called dataExtraction. This field is used when we need to collect data regarding a BPEL activity that communicates with an external service. It specifies the part of the message that needs to be collected, and is given using an XPath expression. When transforming the platform-independent model we always select the entire message (i.e., expression = “/”).

6.4 SLA@SOI Management Modeling Tool

We will now briefly describe the design-time tools we have developed for specifying RDS and KPI models (Adjustment models are not currently supported since they will be part of our work in the project’s third year).

The main component of the design-time tool is an Eclipse plugin for specifying RDS and KPI models, and for automatically translating their contents into configurations for our instrumentation approach. The tool was developed using EMF-based technology. Indeed, all our meta-models have been defined as Ecore models, which represent the starting point for the automatic generation of the Java code that allows us to create, edit, serialize, and deserialize RDS and KPI models, and of the Eclipse plugin itself.

The plugin supports the tree-like definition of RDS and KPI models. For example, in Figure 60 the plugin is being used, in the case of the project’s Open Reference Case’s PaymentService BPEL process, to define the need to support the calculation of the service’s response time.
Once a model is complete, the tools automatically generate an XML file containing the instrumentation configurations for all the sensors that need to be deployed together with the system. The Dynamic Orchestration Engine supports two ways of deploying new sensors to the system. Either the configuration files are dropped into the hot-deploy folder, or they are sent to the Dynamic Orchestration Engine’s SOAP Admin interface. Concrete examples of the generated configurations will be given in the following section.

### 6.5 Example Models and Instrumentation Configurations

The examples shown here relate to the project’s Open Reference Case (ORC) demonstrator, whose SCA-based design is shown in Figure 61.
Here we will show how to model the ORC application to include the capability to produce reliability and response time values at runtime. In particular, we are including in the application the capability to produce alarm events for when the Payment Service’s reliability, calculated over the last 12 hours, drops below 95%, and its response time is higher than 5 seconds.

**Figure 62: Modeling the Response Time and Reliability of the Payment Service**

Figure 62 illustrates the RDS and KPI modelling that needs to be performed to achieve our two goals. In both cases the raw data we need are the start and end times for each invocation of the ServiceOperationCall offered by the Payment Service. For both of these ManagedElementyProperties 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 both to the Reliability KPI DataProcessor and to the ResponseTime DataProcessor. The attributes in the ReliabilityKPI instruct it to calculate the reliability considering what has happened in the last 12 hours (intervalUnit = hour and intervalValue = 12), and to output a new value every 5 minutes (outputUnit = minute and outputValue = 5). The new values are passed to a LowPassFilter that will produce alarm events every time it receives a reliability whose value is less than 95%. The ResponseTimeKPI DataProcessor 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 into a HighpassFilter DataProcessor, which will produce a new value every time it receives a response time that is higher than 5 seconds (5000 milliseconds).

Our tool automatically produces the Instrumentation configurations required by the Dynamic Orcherstration Engine to capture the start and end time of each Payment Service BPEL instance. The instrumentation configuration that is created is:

```xml
<def>
  <workflow>
    <eventType value="StartTime"/>
    <processID value="PaymentService"/>
    <operation value="/process/flow/receive "/>
    <validity_from value="2010-06-14T09:00:00.000+02:00"/>
  </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 PaymentService 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 for the reasons explained in Section 6.3.

The modelling tool has also been adopted in the context of Industrial Use Case B6: E-Government. In this case we define, at design time, the sensors that need to be installed into the system to manage a virtual operator’s functional and non-functional behaviour. Figure 63 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 request arrival rate for the Health and Mobility Process, the response times of the requests made to the Health and Mobility Process, the reliability of the Internal Call Center, the number of requests to the Health Care Structure that were delayed, the average time it takes the Mobility Provider to move a patient, etc. Due to lack of space we cannot give the full details of the modelling. Instead, Figure 64 shows the RDS and KPI modelling that is performed for the first three of the above aspects.

The RDS modelling is actually quite similar to the modelling performed in the case of the ORC. Indeed we express the need to capture, at run time, the start and end of a process’ exposed service operation call. However, this time we are also interested in capturing the start and end times of the service operation call exposed by the Internal Call Center component. The KPI modelling is slightly different since we introduce the use of a Rate KPI to calculate, once every five minutes, the arrival rate of process requests witnessed over the last 12 hours. We also use a ResponseTime KPI to calculate the process’ response time. Finally we use a Reliability KPI to calculate the Internal Call Center component’s reliability over the last 12 hours.

7 Conclusions

In this document, we have presented our common efforts within WP A6 towards methodologies for predictable systems engineering, at the state of M24. We have 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. Four main contributions emerge out of the work in WP A6, which are software performance and reliability prediction, resource usage prediction, run-time SLA violation prediction, and manageability modelling and design. The clear formulation and confine of those four contributions is the result of a reshaping of the WP A6 goals and objectives after the first project year (see DoW amendment #2). Consequently, the place and responsibilities of WP A6 components in the revised year 2 SLA framework architecture are more clearly...
defined, the relation to other A-line WPs has been strengthened, and ultimately, the relevance and applicability to B-line industrial use cases could be proved.

In the third project year, we aim towards a comprehensive and fully-featured set of tools for predictable systems engineering. We will enhance the design-time prediction with consideration of service reliability, complement the manageability tools with automatic synthesis of control configurations, and extend the functionality of the run-time SLA violation prediction. Furthermore, we will collaborate intensively with industrial use cases to provide the solutions that are needed in real-world environments and development processes. We believe that achieving predictability of service quality is an important milestone towards better acceptance and success of service-based systems operated across multiple provider and technological domains. To this end, we are confident that our work in WP A6 substantially contributes to the overall SLA@SOI goal of empowering the service economy in a flexible and dependable way.

8 References


[57] Evgeny Eskenazi and Alexander Fyukov. Quantitative Prediction of Quality Attributes for


Component-based Enterprise Applications. In Proceedings of the 5th ICSE
Workshop on Component-Based Software Engineering: Benchmarks for
Predictable Assembly, 2002.

[61] Tauseef Israr, Murray Woodside, and Greg Franks. Interaction tree
algorithms to extract effective architecture and layered performance

Automated performance modeling of software generated by a design

[63] Shuping Ran. A model for web services discovery with QoS. SIGecom

QoS integration in Web services. In Proceedings of the Fourth International

[65] Andrea D’Ambrogio. A Model-driven WSDL Extension for Describing the
QoS of Web IEEE International Conference on Web Services (ICWS’06),

[66] Michael E. Maximilien and Munindar P. Singh. A Framework and Ontology
for Dynamic Web Services Selection. IEEE Internet Computing, 8(5):84–
93, September 2004.

A QoS Broker Based Architecture for Efficient Web Services Selection. In
ICWS ‘05: Proceedings of the IEEE International Conference on Web
Services, pages 113–120, Washington, DC, USA, 2005. IEEE Computer
Society.

Composition of Web Services: A Look at Selection Algorithms. In ICWS’05:
Proceedings of the IEEE International Conference on Web Services, pages

[69] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria L.
Villani. An approach for QoS-aware service composition based on genetic
algorithms. In GECCO ’05: Proceedings of the 2005 Conference on Genetic
and Evolutionary Computation, pages 1069–1075, New York, NY, USA,
2005. ACM.

[70] Rainer Berbner, Michael Spahn, Nicolas Repp, Oliver Heckmann, and Ralf
Steinmetz. Heuristics for QoS-aware Web Service Composition. In ICWS
‘06: Proceedings of the IEEE International Conference on Web Services,

Oriented Architectures. In IEEE International Conference on Web Services

[72] S. Bleul, T. Weise, and K. Geihs. Large-Scale Service Composition in
Semantic Service Discovery. In The 8th IEEE International Conference on
E-Commerce Technology and the 3rd IEEE International Conference on

Service Composition System Faster. In The 9th IEEE International
Conference on E-Commerce Technology and the 4th IEEE International
Conference on Enterprise Computing, E-Commerce, and E-Services


[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 4 and Table 5.

<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 | target | • reliability metrics |
Table 6: Software Performance and Reliability Prediction Outputs

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

Table 7: Software Performance and Reliability Prediction Inputs

While prediction results can be used for SLA negotiation for a given target service to evaluate (Table 4), prediction also needs input information as an enabling precondition (Table 5). 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.

![Diagram of Components, Interfaces, and Roles](image)

Figure 65: Components, Interfaces, and Roles

Figure 63 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.
As Figure 64 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 ). 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 66).

![Diagram of Service Effect Specifications](image)

**Figure 68: Service Effect Specifications**

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 67).

![Diagram of Resource Demanding and External Actions](image)

**Figure 69: Resource Demanding and External Actions**

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

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 \textit{VALUE}, \textit{BYTESIZE}, \textit{NUMBER_OF_ELEMENTS}, or \textit{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 \textit{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:

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

specifies that (for instance) a compression algorithm reduces the size of \textit{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 \textit{RandomVariables}.

![Figure 70: Overview of Actions](image)

Various specialisations of \textit{AbstractResourceDemandingActions} (see Figure 68) 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 \textit{InternalActions}, keeping the abstraction from the component’s implementation.

\textit{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 67) 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: System Allocation](image)

Figure 72: System Allocation

Figure 73 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.

**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).

![Figure 73: Scenario Behaviours](image)

Figure 73: Scenario Behaviours

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 71) its flow of user actions. The ScenarioBehaviour is analogous to RDSEFFs, but does not contain any resource consumptions.
Figure 74: Usage Scenario Workloads

The Workload, as shown by Figure 75, 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.

Appendix C: Service Evaluation

This section contains a specification of the Service Evaluation component, which is part of the SLA management framework. The specification has been provided as a collaborative effort of work packages A1 (architecture) and A6. Within WP A6, an implementation for the special case of Predictive Software Service Evaluation (P-SSE) has been provided, in order to realize software performance prediction during SLA negotiation.

Overview

The ServiceEvaluation component is a top-level component within the FrameworkCore composite component. It is used by SLAManager components at each level (business, software, and infrastructure) to determine a-priori evaluation of service quality parameters (see Figure 73). 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.
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. Instead, Figure 77 gives an overview over the class structure that can be taken as a basis for contributing an implementation. The next section "Datastructures and Classes" then refers to one concrete implementation that is realized within the WP A6 work package to determine software service quality through model-based predictions.

The classes depicted in Figure 74 are:

- **ServiceEvaluator**: Abstract base class from which all concrete implementations of ServiceEvaluation inherit. This class implements the IServiceEvaluation interface, offering the core evaluation functionality.
- **BusinessServiceEvaluator**: All implementations of ServiceEvaluation that deal with business services / products should inherit from this class.
- **SoftwareServiceEvaluator**: Base class for all implementations of ServiceEvaluation that focus on software services. Within the A6 work package, a concrete implementation is realized (see next section).
- **InfrastructureServiceEvaluator**: Base class for all infrastructure-related implementations of ServiceEvaluation.
Datastructures and Classes

This section describes a concrete implementation of the ServiceEvaluation component for the purpose of evaluation of software service quality (performance, reliability). The evaluation is done on the basis of a QoS model instance of the system under study. The QoS model has been created by Software / Service Providers at system design-time and stored as an artifact in the design-time repository. Figure 75 depicts the corresponding class diagram.

Figure 77: Class Structure of Predictive Software Service Evaluation

The following classes are involved:

- **SoftwareServiceEvaluator**: Base class for all implementations of ServiceEvaluation related to software services, see previous section.
- **PredictiveSoftwareServiceEvaluator**: Main class of the concrete implementation for predictive evaluation of software services. Inherits from SoftwareServiceEvaluator. Exists as a singleton instance in the SLA@SOI Framework. Contains references to the other core classes PredictionScenarioGenerator, SoftwareQualityPredictor, and DesignTimeRepository.
- **EvaluationResult**: Contains the predicted service quality for a particular ServiceRealization. See the evaluate() interaction (specified in the WP A1 deliverable) for details.
- **PredictionScenario**: Represents a QoSModelInstance configured with the parameters of a given ServiceRealization. Is taken as a basis for predictive evaluation by the SoftwareQualityPredictor.
- **PredictionScenarioGenerator**: Generates PredictionScenarios from ServiceRealizations. This includes fetching a QoSModelInstance from the DesignTimeRepository, and adapting it to the parameters of the ServiceRealization (i.e., quality descriptions and usage constraints of required software and infrastructure services). Exists as a singleton instance and is referenced by PredictiveSoftwareServiceEvaluator.
- **SoftwareQualityPredictor**: Contains the algorithms for analysis and / or simulation of QoS model instances for evaluation of expected target service quality under the assumption of a certain ServiceRealization. Exists as a singleton instance and is referenced by PredictiveSoftwareServiceEvaluator.
- **DesignTimeRepository**: Contains a set of QoSModelInstances for all target services / systems that are to be evaluated during negotiation. Exists as a singleton instance and is referenced by PredictiveSoftwareServiceEvaluator.
- **QoSModelInstance**: Contains an instance of a QoS model as described in Section 3.4, modelling the architecture of a software system including a set of hierarchically composed service components, interfaces, data types,
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.

Service Exposure: Services can be exposed either internally (within the same administrative domain) or externally.

See also internal service, external service

Service Implementation: A service implementation is a possible concrete realization of a given service type.

Service Instance: 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.

Service Interface Type: 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

Service Level Consequence: An action that takes place in the event that a service level objective is not met.
Service Level Agreement 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.

Service Level Objective 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.

Service Provider An organization supplying services to one or more internal customers or external customers.

SLA Manager A specialization of service provider: person/system that is responsible for managing SLATs and SLA relationships.

Software Designer A specialization of software provider: person that designs/develops the architecture and components of a specific SLA based application.

Software Manager 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.

Software Provider An organization producing software components which might be used by a service provider to assemble actual services.

Software Service 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.

Software Component Software components are the entities produced at design-time by a software provider.

Service Type 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.

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
FCR Finite capacity regions
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infr-SLAM</td>
<td>Infrastructure SLA Manager</td>
</tr>
<tr>
<td>Infr-SM</td>
<td>Infrastructure Service Manager</td>
</tr>
<tr>
<td>IoC</td>
<td>Inversion of Control</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
<tr>
<td>LLMS</td>
<td>Low Level Monitoring System</td>
</tr>
<tr>
<td>LQN</td>
<td>Layered Queueing Networks</td>
</tr>
<tr>
<td>MA</td>
<td>Manageability Agent</td>
</tr>
<tr>
<td>MRE</td>
<td>Monitoring Result Event</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>NFP</td>
<td>Non-functional property</td>
</tr>
<tr>
<td>ORC</td>
<td>Open Reference Case</td>
</tr>
<tr>
<td>OVF</td>
<td>Open Virtualization Format</td>
</tr>
</tbody>
</table>