Abstract:

The goal of CloudScale is to aid service providers in analysing, predicting and resolving scalability issues, i.e., support scalable service engineering. The project extends existing and develops new solutions that support the handling of scalability problems of software-based services. This deliverable presents:

- concepts and terminology on scalability-relevant issues,
- methods for evaluating (a) scalability and (b) elasticity of services deployed on cloud resources are described in two chapters,
- the CloudScale Method, a collection of concrete process steps to engineer scalable cloud systems at design time and as the system evolves,
- ScaleDL representing a family of four sub languages including ScaleDL Usage Evolution and ScaleDL Architectural Template,
- ScaleDL Usage Evolution, a sub language of ScaleDL, for service providers to specify scalability properties of their offered services modelled by the usage evolution,
- HowTos, a list of design recommendations and patterns we suggest for scalability, elasticity, and efficiency,
- ScaleDL Architectural Template, a sub language of ScaleDL, manifesting HowTos such that they can be used for engineering ScaleDL models, and
- the Analyzer using ScaleDL Usage Evolution specifications as an input to predict the scalability of cloud computing applications at design time.

We discuss each of these nine results in separate chapters.

---

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement no 317704.
CloudScale consortium

CloudScale (Contract No. FP7-317704) is a Collaboration Project (STREP) within the 7th Framework Programme, Call 8, Priority 1.2 (Cloud Computing). The consortium members are: ¹

SINTEF ICT
(SINTEF, Norway)
NO-7465 Trondheim
Norway
www.sintef.com

Project manager: Richard T. Sanders
Richard.Sanders@sintef.no
+47 930 58 954

Technical manager: Gunnar Brataas
Gunnar.Brataas@sintef.no
+47 472 36 938

SAP Research, CEC
Karlsruhe
(SAP, Germany)
69190 Walldorf,
Germany
www.sap.com/research

Contact: Roozbeh Farahbod
Roozbeh.Farahbod@sap.com

Ericsson Nikola Tesla
(ENT, Croatia)
Krapinska 42
10000 Zagreb, Croatia
www.ericsson.com/hr

Contact: Darko Huljenic
Darko.Huljenic@ericsson.com

XLAB, Slovenia
Pot za Brdom 100
1000 Ljubljana, Slovenia
www.xlab.si

Contact: Jure Polutnik
Jure.Polutnik@xlab.si

Universität Paderborn
(UPB, Germany)
Zukunftsmile 1
33102 Paderborn,
Germany
www.uni-paderborn.de

Contact: Steffen Becker
Steffen.Becker@uni-paderborn.de

Technische Universität Chemnitz
(TUC, Germany)
Straße der Nationen 62
09107 Chemnitz
Germany
www.tu-chemnitz.de

Contact: Steffen Becker
Steffen.Becker@informatik.tu-chemnitz.de

¹UPB withdrew from the Consortium in October 2014, as the research team there relocated to TUC. TUC was not a partner before October 2014.
## Contents

1 Introduction ................................................. 1  
  1.1 CloudScale motivation and Background .................. 1  
  1.2 Relation to D1.1 ........................................... 2  
  1.3 Relationships with Other Deliverables .................. 3  
  1.4 Challenges .............................................. 3  
  1.5 Contributors ............................................ 4  
  1.6 Change Log .............................................. 4  

2 Concepts .................................................. 5  
  2.1 Foundation ............................................... 5  
    2.1.1 Scalability Concept Map .............................. 5  
    2.1.2 Workload = Work × Load ............................. 5  
    2.1.3 (Performance) Quality ................................ 5  
    2.1.4 Variability .......................................... 7  
    2.1.5 Multi Tenancy ........................................ 8  
    2.1.6 Price Model Classification .......................... 9  
  2.2 Deployment Characterisation .............................. 10  
    2.2.1 CloudStore Deployment Example ...................... 10  
    2.2.2 IaaS Taxonomy ....................................... 11  
    2.2.3 Resource Space ...................................... 11  
    2.2.4 Software Resources .................................. 12  
  2.3 CloudScale Metrics ...................................... 13  
    2.3.1 Capacity ............................................ 14  
    2.3.2 Scalability with respect to Cost (SC) .............. 15  
    2.3.3 Scalability Range (SR) .............................. 17  
    2.3.4 Number of SLO Violations (NSLOV) .................. 17  
    2.3.5 Marginal Cost (MC) .................................. 17  
    2.3.6 Resource Provisioning Efficiency (RPE) ............ 18  
    2.3.7 Mean Time To Quality Repair (MTTQR) .............. 18  
    2.3.8 Scalability Speed (SS) ............................... 18  
  2.4 Related Work ........................................... 18  
    2.4.1 Scalability Concepts ................................ 18  
    2.4.2 Scalability Metrics .................................. 19  
    2.4.3 Elasticity Metrics ................................... 20  
    2.4.4 Efficiency Metrics .................................. 20  
    2.4.5 Combined Metrics ................................... 21  
  2.5 Future Work on Concepts and Metrics ................... 21  

3 Scalability Evaluation Method .......................... 23  
  3.1 Introduction to Evaluation Method ....................... 23  
  3.2 Pilot Evaluation ....................................... 24  
    3.2.1 Load Generation ..................................... 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3.1</td>
<td>Multi-tenant Applications</td>
<td>55</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Scalability for Individual Tenants</td>
<td>55</td>
</tr>
<tr>
<td>8.4</td>
<td>Future Work</td>
<td>56</td>
</tr>
<tr>
<td>9</td>
<td><strong>SCALEDL ARCHITECTURAL TEMPLATE</strong></td>
<td>59</td>
</tr>
<tr>
<td>9.1</td>
<td>Example Scenario for Architectural Templates</td>
<td>59</td>
</tr>
<tr>
<td>9.2</td>
<td>Architectural Template Processes</td>
<td>61</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Applying a SPOSAD Architectural Template</td>
<td>61</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Creating the Architectural Template for SPOSAD</td>
<td>61</td>
</tr>
<tr>
<td>9.3</td>
<td>Abstract Syntax: AT Metamodel</td>
<td>63</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Type Package</td>
<td>63</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Instance Package</td>
<td>64</td>
</tr>
<tr>
<td>9.4</td>
<td>Concrete Syntax: AT Graphics</td>
<td>66</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Type Graphics</td>
<td>66</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Instance Graphics</td>
<td>67</td>
</tr>
<tr>
<td>9.5</td>
<td>Architectural Templates Repository</td>
<td>69</td>
</tr>
<tr>
<td>9.6</td>
<td>Naming of Architectural Templates</td>
<td>69</td>
</tr>
<tr>
<td>9.7</td>
<td>Future Work on the Architectural Template Language</td>
<td>70</td>
</tr>
<tr>
<td>10</td>
<td><strong>ANALYSER</strong></td>
<td>71</td>
</tr>
<tr>
<td>10.1</td>
<td>Relations to Other Chapters</td>
<td>71</td>
</tr>
<tr>
<td>10.2</td>
<td>Palladio</td>
<td>72</td>
</tr>
<tr>
<td>10.3</td>
<td>SimuLizar</td>
<td>72</td>
</tr>
<tr>
<td>10.4</td>
<td>Technical Realization</td>
<td>72</td>
</tr>
<tr>
<td>10.5</td>
<td>Future Work on the Analyser</td>
<td>73</td>
</tr>
<tr>
<td>11</td>
<td><strong>FUTURE WORK</strong></td>
<td>75</td>
</tr>
<tr>
<td>A</td>
<td><strong>ANALYSER TEST PLAN</strong></td>
<td>77</td>
</tr>
<tr>
<td>A.1</td>
<td>Prerequisites</td>
<td>77</td>
</tr>
<tr>
<td>A.1.1</td>
<td>State of the Analyser</td>
<td>77</td>
</tr>
<tr>
<td>A.1.2</td>
<td>Three Important Parts of a Test Case</td>
<td>77</td>
</tr>
<tr>
<td>A.1.3</td>
<td>Equivalence Classes</td>
<td>78</td>
</tr>
<tr>
<td>A.1.4</td>
<td>Boundary Cases</td>
<td>79</td>
</tr>
<tr>
<td>A.1.5</td>
<td>Work and Load and Quality Thresholds</td>
<td>79</td>
</tr>
<tr>
<td>A.2</td>
<td>Simple Tests</td>
<td>79</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Simple Test of Scalability Range</td>
<td>79</td>
</tr>
<tr>
<td>A.2.2</td>
<td>Simple Test with No Load</td>
<td>80</td>
</tr>
<tr>
<td>A.3</td>
<td>Usage Evolution Tests</td>
<td>81</td>
</tr>
<tr>
<td>A.3.1</td>
<td>Scalability Range (SR)</td>
<td>81</td>
</tr>
<tr>
<td>A.3.2</td>
<td>Scalability Capacity (SC)</td>
<td>82</td>
</tr>
<tr>
<td>A.3.3</td>
<td>Resource Provisioning Efficiency (RPE), Internal Comparison</td>
<td>83</td>
</tr>
<tr>
<td>A.3.4</td>
<td>Resource Provisioning Efficiency (RPE), Comparing with Capacity Metric</td>
<td>83</td>
</tr>
<tr>
<td>A.3.5</td>
<td>Number of SLO Violations (NSLOV), Internal Comparison</td>
<td>84</td>
</tr>
<tr>
<td>A.3.6</td>
<td>Mean Time to Quality Repair (MTTQR)</td>
<td>84</td>
</tr>
<tr>
<td>A.3.7</td>
<td>Marginal Cost (MC), Internal Comparison</td>
<td>85</td>
</tr>
<tr>
<td>A.3.8</td>
<td>Marginal Cost (MC), Comparing with Capacity Metric</td>
<td>85</td>
</tr>
<tr>
<td>A.3.9</td>
<td>Scalability Speed (SS)</td>
<td>85</td>
</tr>
<tr>
<td>B</td>
<td><strong>GLOSSARY</strong></td>
<td>87</td>
</tr>
</tbody>
</table>
List of Figures

1.1 The results of CloudScale. .................................................. 1
2.1 Concept map for scalability. ............................................. 6
2.2 CloudStore deployed on Amazon Web Services (AWS) ............ 10
2.3 AWS resource space for CloudStore. ................................. 12
2.4 AWS resource space with instances for CloudStore. .......... 12
2.5 YaaS provides services to XaaS and requires services from ZaaS 13
2.6 Relation between load and quality. .................................... 15
2.7 Relation between relative cost and capacity. ....................... 15
4.1 Basic elasticity test case. ................................................. 31
5.1 CloudScale Method Overview. ........................................... 34
5.2 Mapping of software engineer roles and CloudScale Method. 37
6.1 Overview of ScaleDL sub-languages and their relationships. 43
7.1 Draft meta model for ScaleDL Usage Evolution. .................. 50
9.1 Model of the designed book shop scenario according to a simplified SPOSAD variant 60
9.2 Process steps of AT application for software architects. .......... 61
9.3 Process steps of AT creation for AT engineers. .................... 62
9.4 AT Language Metamodel: Type Package ............................. 64
9.5 AT Language Metamodel: Instance Package ......................... 65
9.6 CloudScale's Architectural Templates Repository ................. 69
10.1 The Analyser Process ..................................................... 71
A.1 Figure used to compare test outputs. ............................... 78
List of Tables

2.1 An example of a quality specification for three operations A, B and C. . . . . . . . . 7
4.1 Parameters when testing elasticity. . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
8.1 Overview of Identified HowTos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
9.1 Description Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
9.2 Repository Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
9.3 Role Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
9.4 Component and Connector Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
9.5 Parameter Instantiation Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
A.1 Equivalence classes for the Analyser. . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
Executive Summary

This second WP1 deliverable summarises CloudScale’s results for the design support of scalable, efficiency and elastic cloud computing services. CloudScale’s design support results include:

**Concepts and terminology** on scalability-relevant issues: we describe our scalability, efficiency and elasticity metrics and their relation to quality.

**Scalability Evaluation Method** describing how to measure the scalability of a service deployed on cloud resources.

**Elasticity Evaluation Method** describing how to measure the elasticity of a service deployed on cloud resources.

The **CloudScale Method**, a collection of concrete process steps to engineer scalable cloud systems at design time and as the system evolves. This method has six main process steps where especially the system construction and analysis step are decomposed further.

**SCALEDL** represents a family of four sub languages including **SCALEDL Usage Evolution** and **SCALEDL Architectural Template**, which is described in this deliverable. In addition, **SCALEDL Overview** described in D3.1, is also part of **SCALEDL**. Finally, the forth **SCALEDL** sub language is **Extended PCM** building on the Palladio Component Model (PCM) and SimuLizar.

**SCALEDL Usage Evolution**, a sub language of **SCALEDL**, for service providers to specify scalability properties of their offered services modelled by the usage evolution. In this section it is also described how scalability can be connected to cost.

**HowTos** a list of design recommendations and patterns we suggest for scalability, elasticity, and efficiency.

**SCALEDL Architectural Templates** a sub language of **SCALEDL**, manifesting HowTos such that they can be used for engineering **SCALEDL** models.

**The Analyser** using **SCALEDL Usage Evolution** specifications as an input to analyze the scalability, elasticity, and efficiency of cloud computing applications at design time. **Analyser** also uses the PCM for representing the components within a service, how these components relate with each other, and how these components invoke lower level services.

We discuss each of these results in a separate chapter. In addition, this deliverable also contains a test plan for the **Analyser** as well as a glossary defining key CloudScale terms.
1 Introduction

This deliverable describes the conceptual foundation for CloudScale and is therefore a starting point for readers who want to understand the CloudScale project. Moreover, this deliverable also gives detailed knowledge of the artefacts scalability and elasticity evaluation method, SCALEDL USAGE EVOLUTION, CloudSCALE Method, HowTos, SCALEDL ARCHITECTURAL TEMPLATES language and Analyser. This deliverable also contains a test plan for the Analyser. A glossary specifying key CloudScale terms is a live document [Clo14]. A snapshot of the glossary is also available in Appendix B. This deliverable is typeset with LATEX, which eases the conversion to and from papers where LATEX is the preferred typesetting tool.

1.1 CloudScale motivation and Background

Cloud providers theoretically offer their customers unlimited resources for their applications on an on-demand basis. However, scalability is not only determined by the available resources, but also by how the control and data flow of the application or service is designed and implemented. Implementations that do not consider their effects can either lead to low performance (under-provisioning, resulting in high response times or low throughput) or high costs (over-provisioning, caused by low utilisation of resources).

CloudScale provides an engineering approach for building scalable cloud applications and services. Our objectives are to:

1. Make cloud systems scalable by design so that they can exploit the elasticity of the cloud, as well as maintaining and also improving scalability during system evolution. At the same time, a minimum amount of computational resources shall be used.

2. Enable analysis of scalability of basic and composed services in the cloud.
3. Ensure industrial relevance and uptake of the CloudScale results so that scalability becomes less of a problem for cloud systems.

CloudScale enables the modelling of design alternatives and the analysis of their effect on scalability and cost. Best practices for scalability further guide the design process.

The engineering approach for scalable applications and services will enable small and medium enterprises as well as large players to fully benefit from the cloud paradigm by building scalable and cost-efficient applications and services based on state-of-the-art cloud technology. Furthermore, the engineering approach reduces risks as well as costs for companies newly entering the cloud market.

### 1.2 Relation to D1.1

Compared to the first deliverable from WP1, D1.1, this second WP1 deliverable provides the following main changes and extensions:

**Concepts** This chapter is rewritten:

- **Foundation** Our basic concepts in Section 2.1 have been rewritten: the quality concept in Section 2.1.3 is sharpened and we have added sections on multi-tenancy in Section 2.1.5 and cost in Section 2.1.6.

- **Deployment characterisation** This section is new. New deployment concepts are described in Section 2.2.

- **CloudScale metrics** Our coverage of metrics is considerably extended in Section 2.3. While D1.1 contained sketches of both the capacity metric and the scalability with respect to cost metrics, these metrics are now more mature and described in Section 2.3.1 and Section 2.3.2, respectively. We have also included six more metrics for elasticity and efficiency: scalability range in Section 2.3.3, number of SLO violations in Section 2.3.4, marginal cost in Section 2.3.5, resource provisioning efficiency in Section 2.3.6, mean time to quality repair in Section 2.3.7, and scalability speed in Section 2.3.8.

- **CloudScale Method** Our method has been polished. It introduces the main concepts that is used for defining the appropriate granularity level for each method step. The new version also describes how to present efforts for different method steps.

- **Methods for evaluating scalability and elasticity** have been added since metrics in these areas also need a precise evaluation method.

- **ScaleDL** The short introduction chapter for ScaleDL has been extended to now also describe Descartes Load Intensity Model (DLIM), and a description and figure has been added explaining how the different sub-languages of ScaleDL relate to each other.

- **ScaleDL Usage Evolution** Our approach has been revised to use DLIM for modelling single parameters of load and work while ScaleDL Usage Evolution now focus on how these DLIM models are mapped to elements of the Palladio model to describe their evolution. The meta model of ScaleDL Usage Evolution has been updated to reflect these changes, and the description now reflects the initial working implementation. The discussions of deployment characterisation and of scalability with respect to cost have been moved to the concepts chapter. We now describe the relation to the Cloud Modelling Framework in the FP7 projects MODAClouds' and PaasSage’s, as well as to MARTE and SLA@SOI.
HowTos In this version of the deliverable, a chapter describing CloudScales efforts with respect to HowTos is added. The primary focus is on describing the rationale behind our selection of patterns we suggest as HowTos. The chapter also links to the CloudScale wiki where the detailed descriptions of our HowTos can be found. These HowTos where also the basis for realizing concrete Architectural Templates.

SCALEDL ARCHITECTURAL TEMPLATES are now integrated into the ANALYSER and several architectural templates where implemented; concepts compared to D1.1 did not change.

ANALYSER now supports architectural templates as well as novel metrics for scalability, elasticity, and efficiency. We provide a new report on the technical realization of our extensions. The ANALYSER overview from D1.1 did not change.

1.3 Relationships with Other Deliverables

This document relates to the following deliverables and white papers:

- D2.2 – Evolution support, second version: Presenting CloudScale’s support for evolution using the EXTRACTOR and the SPOTTER, which is covered by the CloudScale Method presented in this deliverable.

- D3.2 – Second version of Integrated Tools: Presenting the architecture of the tools, which is also used in the CloudScale Method. This deliverable also presents the SCALEDL OVERVIEW which focuses on deployment.

- D4.2 – Requirements and validation, second version: Which gives the requirements for SCALEDL USAGE EVOLUTION, SCALEDL ARCHITECTURAL TEMPLATES and also for the ANALYSER. CloudScale Method usage effort estimations baseline is defined based on the initial validation results from this deliverable.

- D5.2 – Second version of Showcase: Presents CloudStore, which is also used as a running example in this deliverable.

- Quality Analysis Lab (white paper)\(^1\): Describes the technical realization of the ANALYSER and serves to disseminate CloudScale results to the Palladio community.

1.4 Challenges

Below is a list of challenges related to this deliverable:

Define metrics To propose and prioritise metrics for implementation in the Analyser was demanding as this required a good overview of potential ANALYSER users as well as of their required output. This work lead to a clarification of the roles described in both D1.1 and D4.1. We now have eight candidate metrics of which three are currently implemented in the Analyser.

Scalability and elasticity evaluation method Making a scalability and evaluation method requires a good overview of all factors contributing to the quality of these evaluations. Our current

\(^1\)The Quality Analysis Lab can be downloaded from the Palladio SVN (User: anonymous, Password: anonymous; https://svnserver.informatik.kit.edu/i43/svn/code/QualityAnalysisLab/Documentation/trunk/ org.palladiosimulator.qual.docs/QualityAnalysisLab.pdf).
evaluation method is our best attempt of identifying the relevant factors and giving advice on how to reduce the manual work in scalability and elasticity evaluation.

**Large implementation effort** Because of the large amount of work to refactor and also to implement the metrics in the Analyser, we were delayed with writing this deliverable, which means that we also had little time to modify the method this year.

We have several successful results in D1.2, for example related to metrics. The scalability range metric is new compared to what we anticipated when we wrote the Description of Work (DoW). We have proposed several elasticity metrics this year and have also implemented the Number of SLO Violations elasticity metric.

Concerning unsuccessful results we had anticipated more progress on scalability with respect to services. This has proven to be hard and is also less required in the cloud where cost is such a prominent measure of service consumption. As a result, we are delayed with our composition concept. Finally, the connectivity concept which we describe in the DoW seemed less important in the cloud environment and was not prioritised.

### 1.5 Contributors

The following partners and persons have contributed to this deliverable:

- SINTEF ICT: Gunnar Brataas and Erlend Stav
- SAP: Rouven Krebs
- Ericsson Nikola Tesla: Darko Huljenić and Goran Kopčak
- Universität Paderborn: Sebastian Lehrig

### 1.6 Change Log

No change log entries.
2 Concepts

This chapter presents our concepts for scalability, elasticity and efficiency. This understanding is our starting point for defining the design support for scalability in the following chapters, including the ScaleDL Usage Evolution, Analyser, and the CloudScale Method. The chapter starts with a foundation where basic concepts like workload, quality, multi-tenancy and price are described. We then outline some basic concepts relating to deployment, before we describe the CloudScale metrics for scalability, elasticity and efficiency. Related work is also covered, before we outline future work.

2.1 Foundation

This section introduces basic concepts which is a foundation for our work in D1.2. However, we start in Section 2.1.1 with an overview of the concepts which we develop in D1.2.

2.1.1 Scalability Concept Map

The concept map in Figure 2.1 gives an overview of the main concepts presented in this chapter along with the main relations between them. The description of each concept is introduced throughout the chapter. The central concept of scalability is highlighted and shown at the top of the figure. Performance efficiency is described in this figure, as it is a central concept in this domain defined in ISO/IEC 25010:2011, see Section 2.1.3. We have not yet included elasticity and variability in this concept map.

2.1.2 Workload = Work × Load

We separate workload into work and load.

**Work** is the characterisation of the data to be processed, stored, or communicated by a service. It is about what is done in each invocation of the service and is determined by the type and nature of the services. Service parameters are typically important.

**Load** characterises how often a service is used. We distinguish between open and closed systems:

- **Open system** Load is measured in terms of throughput; the number of finished requests during a given time interval.
- **Closed system** Load is characterised in terms of number of simultaneous users in the system. The number of users is constant.

**Workload** is the combined characterisation of work and load.

2.1.3 (Performance) Quality

A good source for classification of quality is the ISO/IEC 25010:2011 standard for System and software quality models [ISO11] (and its predecessor ISO/IEC 9126-1:2001 [ISO01]). The quality model classifies quality into a set of quality characteristics and sub-characteristics of these. In the
In the context of scalability, **performance efficiency** is the most relevant of the 8 main product qualities defined. Performance efficiency is defined as "performance relative to the amount of resources used under stated conditions" [ISO11], with time behaviour, resource utilisation and capacity as sub-characteristics. Resources in this definition can include "... other software products, the software and hardware configuration of the system, and materials (e.g. print paper, storage media)".

All software and hardware systems have a certain (performance) quality that varies when we change the work and load. Typically, (performance) quality degrades when we increase either work, load, or both.

Two concepts are central when defining quality in relation to software systems:

**Quality metric** defines a standard of measurement for a quality characteristic. A quality metric is an essential part of a Service Level Objective (SLO). External metrics for the quality characteristics of [ISO01] are defined in [ISO03]. Examples of metrics for the time behaviour sub-characteristic of (performance) efficiency include response time (measured in terms of time) and throughput (measured in terms of count/time). A more complex metric used in telecom is Mean Opinion Score (MOS). The MOS is a number between one (bad) and five (best), representing quality of experience. Quality metrics may also reflect system properties like utilisation of some hardware resources.

**Quality threshold** is the border between acceptable and non acceptable (performance) quality, when applying a quality metric. With **optimal provisioning** we are satisfying the quality thresh-
Table 2.1: An example of a quality specification for three operations A, B and C.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Metric specification</th>
<th>Threshold (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Average response time</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>Average response time</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>Maximum response time</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>Average response time</td>
<td>10</td>
</tr>
</tbody>
</table>

old with an optimal cost. With under-provisioning, the (performance) quality is not acceptable, and we get an SLO violation. With over-provisioning, (performance) quality is better than the threshold, but the cost is also higher compared to optimal provisioning. As an example where the quality metric is response time, the threshold may be 1 second, meaning that users are satisfied with less than 1 second response time. More than one second will be an SLO violation. On the other hand, less than 1 second might mean that we have over-provisioning at a higher cost.

Part of the quality metric specification must also be an agreement on how this quality shall actually be measured. In the case of response time, it must be clearly defined if the actual response time threshold apply to all service invocations, to a given per cent of them, or to the average.

A service will typically have more one operation. In principle, the quality metric may vary between the operations, but it will often be the same. We may also have more than one metric for each operation. The actual limit threshold for each operation and for each metric will often vary between the different operations, e.g. some operations may have a limit of 1 second, but for other operations the limit may be 10 seconds. With several operations, we must also have a way of describing the distribution between these operations. This is described in more detail in Chapter 7.

In summary, the quality thresholds specification may look like Table 2.1. We can here see how these three operations all have the average response time metric, but with different thresholds. In addition to the average response time metric, operation B also has an absolute response time metric.

2.1.4 Variability

Variability deals with the architectural variant to choose. Variability is one way of supporting adaptation. If we want to address this, we need to define what we mean by architectural variability. One source of definitions for this is the results from the MUSIC project [MUS10], that used variation points in architectures to do self-adaptation at run-time. In MUSIC, the following concepts were used to describe the variability of a self-adaptive system:

- a set of alternative realisations for a component type / interface
- a set of different compositions, where each composition has a set of required and optional component roles (defined by type) and connectors between these.
- a set of possible deployments to nodes for components
- parameter values for components (e.g., resource allocation)
- architectural constrains that can limit the set of allowed variations, by, e.g., specifying a set of component realisations for different roles that match with each other or are mutually exclusive.
In **CloudScale** we base our modelling and analysis of self-adaptive elastic services on SimuLizar [BLB12, BBM13] together with inspiration from the MODAClouds [MOD13] EU project and concepts from MUSIC [MUS10]. We therefore integrated SimuLizar into the **ANALYSER** as described in Chap. 10.

### 2.1.5 Multi Tenancy

There are several ways to decrease the per customer costs when a SaaS application is provided:

- **At minimum a single code base** should be shared between different customers/tenants. If one single code base is used the application has to be widely configurable to be adapted for customer specific needs. Sharing the code base yields reuse and is omnipresent. Nevertheless, a single code base is not sufficient enough to reduce the operational costs. Developing widely configurable software instead of customer specific branches is a question related to product line engineering and not specific for MTAs.

- **Sharing a data center** is the lowest level of resource sharing one could imagine. Reusing the facilities environment like air conditioner or network infrastructure is the simplest way of decreasing costs. Application Service Providers already have adopted this concept for years. However, sharing the data center only has a very limited cost saving potential, e.g. workload fluctuations of different customers can't be considered for resource optimization.

- **Virtualization** provides an easy way for sharing a single server. Running a separate instance of the application within one VM for each customer is a first step towards efficient operation and probably today’s most widely adopted sharing approach. In opposite to a shared data center, virtualization allows leveraging workload fluctuations by overcommitting the servers, while allowing a good isolation. However, the overhead of this solution per customer is still quite high. Virtualization is a well-established field of research with challenges and goals on a hardware related level and should not be referred to multi-tenancy which is a concept on the applications level.

- **Another approach** shares the middleware. In this scenario the middleware becomes shared by several application instances. This means, that the implementation of the application itself has low or no overhead regarding multi-tenancy. The disadvantages are the challenges regarding isolation of tenants and the overhead due to separated application instances. Hosting different applications within one middleware is not different than hosting the same application multiple times, which is a topic that is already discussed extensively in the literature.

- **A Multi-tenant Application (MTA)** shares one application instance among different customers to reduce overhead the most. Handling different tenants within one application instance requires several modifications as every tenant needs its own view. MTAs started to become of interest as SaaS technologies arose. Therefore, we propose Multi-tenancy in the manifestation of a shared application as the most relevant pattern to provide an cost efficient way for hosting SaaS applications. Consequently, it is the best scaling solution and thus in focus of our investigations.

The contribution of CloudScale in the context of Multi-tenancy concerning work package 1 can be separated into two parts. One is related to existing patterns to build multi-tenant applications. The second conducts the question on how to provide scalability on a tenants basis. More details concerning the relevant patterns and contribution are discussed in Section 9.
2.1.6 Price Model Classification

Each service delivery model (SaaS, IaaS, PaaS) and service provider offers a wide range of business solutions and price models for service customers. For this section, our goal was to identify common features of these business solutions and price models. On the basis of these common features, we grouped them into several price model groups:

- **On-Demand price model [Wea09]:** Service customers are charged according to the actual cloud service (e.g., CPU, data storage, data transfer) usage time and quantity. This price model concept is also known as pay-per-use or pay-as-you-go price model. It is one of the most commonly used price models in cloud environments and it illustrates many features of the NIST cloud computing paradigm [MG11].

- **Package price model [MWT12]:** Service providers often provide complete service packages with predefined cloud service parameters (e.g., monitoring intervals, number of virtual CPUs, etc.) at a fixed price. Service customers can easily switch or upgrade to the more or less powerful cloud package offerings based on their needs.

- **Customer-based price model [MWT12]:** This price model charges by the number of customers that consume services (e.g., subscription-based services, licensing offers). It is mostly used in the PaaS and SaaS delivery models.

- **Auction price model [WLL12]:** In this model, service customers can compete to gain cloud services through an auction system. This type of price model provides a dynamic price depending on the demand.

- **Free-of-Charge price model [DMT12]:** This model allows services that are free of charge. There are different business models based on this model. Open source communities (e.g., Scalr) can often provide completely free-of-charge services. Commercial cloud providers can provide a try-before-buy model to their customers. There can also be a so-called freemium model (a combination of a free and premium usage) in which the service is free for basic customers (being the majority) and it is only charged for a small percentage of customers who need advanced options and service configurations.

Orthogonally to this classification, price changes during specific periods of time can be considered [KSA12]:

- **Static pricing:** All service costs are fixed during the whole usage period. From the price optimisation point of view, optimisation is done only once and there are no powerful utilisation mechanisms in this pricing model for service providers and service customers.

- **Dynamic pricing:** The service price depends on service availability and the time frame in which the service is used.

SLO violations, such as service unavailability (hard downtime) and performance degradation in the cloud computing environment, can have an additional impact on the total service usage cost for each cloud price model. SLO violation cost impacts are specified using the service credit concept [Bas12]. Service credit is the amount credited to the cloud service customer (instantly or through future service payments) if the SLO is violated. Most service providers specify service credit as a percentage value in the interval of 0-100% but there are also cases that cloud providers offer even higher service credits [oD]. For example, when the CloudStore SaaS is unreachable for 1 hour, it could receive 30 minutes of credit at the end of the billing period if the service credit is 50%.
2.2 Deployment Characterisation

The deployment profile characterises how an application uses lower-layer services. This will build on ScaleDL Overview which is described in D3.2, but some key concepts will also be mentioned here. An SaaS application could for example use PaaS services such as databases, message queues, and monitors, but it may also use IaaS [Mea13]. The deployment characterisation is specified by the architect of the SaaS application.

In this section we will introduce some important concepts which is required before defining scalability. First a CloudStore deployment example based on Amazon Web Services is described, before a simple IaaS taxonomy is explained. We then introduce the concept of resource space and describes software resources.

2.2.1 CloudStore Deployment Example

In Figure 2.2 we see how the CloudStore application server may be deployed on Amazon EC2. This example is relevant because it tells us that one SaaS service will be deployed on different IaaS as well as on PaaS services. It is also interesting to see that some of the deployed services can be replicated, while others cannot.

EC2 can be auto scaling so that it automatically can select how many instances to use, of the instance type and size specified. The Amazon S3 image server is a PaaS, meaning that it is auto scaling to be able to fulfil the demand, and then with a resulting cost. The Amazon RDS database server can be configured to have one master and several replicas, of the instance type and size specified. Note that the RDS masters and replicas cannot be replicated.

Figure 2.2 describes composition of underlying services, because CloudStore use several of them. If we want to compose across even more services, we could, for example, introduce a Payment PaaS as one more underlying service.
2.2.2 IaaS Taxonomy

On the market, there are many different infrastructure services to select from. In this section we will make a simple taxonomy of IaaS services with Amazon Web Services (AWS) and more specifically Amazon Elastic Computing Cloud (Amazon EC2). We have the implicit assumption that the IaaS competitors are in principal similar to what AWS and in particular EC2 offers, for example the AWS RDS database instances follow the same pattern. You rent instances and you pay for instances regardless of the utilisation of the instance. Either you use it with 0.1 per cent utilisation or with 99 per cent utilisation, the price is the same.

IaaS can be scaled across three dimensions:

**Instance type** optimised for a given work profile. In AWS EC2, there are five types of instances: general purpose, compute optimised, GPU, memory optimised and storage optimised. There may also be several generations of each instance type, which may have slightly different characteristics.

**Instance size** Each instance type will have different sizes and in AWS EC2 they are termed small, medium, large, large, 2xlarge, 4xlarge, 8xlarge. Typically, these instance types are replicated internally, e.g. instead of 1 CPU you get 2, 4, 8 etc. However, not all aspects of the instance are scaled in this way, like communication capabilities.

**Number of instances** We can use several instances and must therefore be able to exploit this replication in your service, for example to handle more load.

With more load, there are basically two scaling options: a larger instance size (vertical scaling) or more instances (horizontal scaling). In some cases replication (horizontal scaling) is not an option (like in Amazon RDS), and it is here an advantage to select a sufficiently large instance size (vertical scaling).

2.2.3 Resource Space

The resource space describes the resources we will explore during our scalability analysis to handle our usage evolution. The resource space is basically a variability model [BCK13] for the underlying IaaS and PaaS services. We then have to specify which instances we want to use for each EC2 and RDS.

To make the presentation simpler, we will use Amazon Web Services as an example. In CloudStore we can vary the instance types for the EC2 and for the RDS in Figure 1. Using auto scaling, EC2 can then use more of the selected instance type, whereas no auto scaling exists for the RDS.

Figure 2.3 illustrates one possible resource space for CloudStore. The x-axis shows the selected instance types for EC2 and the y-axis the selected instance types for RDS. We also see from this figure that we combine m3.medium with db.m3.medium, whereas the combination m3.medium with db.m3.large is not investigated as part of this test plan. With more dimensions than three or even four or more, the visual presentation of the resource space will be demanding.

The resource space determines the configurations we have to investigate. In our case we see that we have to investigate three different configurations, namely:

- m3.medium used for EC2 and db.m3.medium used for RDS
- m3.large used for EC2 and db.m3.large used for RDS
• m3.xlarge used for EC2 and db.m3.xlarge used for RDS

However, for EC2 we can in addition to instance type also play with the number of instances. With auto provisioning this will be handled automatically. However, even if auto provisioning is used, we have to keep an eye also on the number of instances. Number of EC2 instances will then introduce a third dimension that is illustrated in Figure 2.4. In this figure, the instances are only relevant for the x-axis, the EC2 instances, but not for the y-axis, the RDS instances.

The architect must also determine important characteristics of the service deployment. Examples are the number and types of indices in a DBMS.

2.2.4 Software Resources

Hardware resources or active resources are CPUs, disk and networks. These resources are typically provided by IaaS like Amazon EC2, but they are of course implicitly also part of PaaS and SaaS, like S3 and RDS.

Software resources or passive resources are buffers, locks or semaphores. CloudStore has the following five software resources:

• Connection pool in the database (for example MySQL)
• Worker pool in the web server (for example Tomcat)
• Three semaphores in the Database Access:
  – Synchronise cart
  – Synchronise customer
  – Synchronise order

These three semaphores exist to ensure ACID properties. Using database replicas, only parts of the database may be locked at the same time.
While hardware resources can be viewed as being outside of the CloudStore service, software resources describes the integration between the CloudStore service and its resources.

It is essential to tune the software resources. For example, the better the match between the number of database connections and both the CloudStore application and the available resources, the higher capacity for a given resource configuration and the better the quality of the measurement results.

2.3 CloudScale Metrics

As a basis for our metrics we will introduce a simple example in Figure 2.5. This figure shows how a YaaS layer provides services to an XaaS layer and requires services from a ZaaS layer. The letters X, Y and Z can be any meaningful cloud computing acronym, typically referring to SaaS, PaaS, and IaaS (from top to bottom).

![Diagram](chart.png)

Figure 2.5: YaaS provides services to XaaS and requires services from ZaaS.

We basically have three groups of metrics:

**Scalability** Relating to Figure 2.5 CloudScale currently has the following definition of scalability:

*For an Y as-a-Service, scalability is the ability of the layer to sustain changing workloads while fulfilling its SLO, potentially by consuming a higher/lower quantity of ZaaS services.*

By saying potentially, we mean that there are also other possibilities. Currently, we are aware of two other possibilities, and we discuss them for increasing workload:

**Adaptation** By adaptation in the YaaS layer that reduces the workload on ZaaS. We may, for example, change the version of one component to a component that uses less ZaaS services. To make this into a realistic scenario, there must be some drawback with this new component, otherwise it would always be used; e.g., it may be more costly or have poorer availability, security etc.

**Over-provisioning** YaaS has some degree of over-provisioning, either because the YaaS services it is using has room for more workload, or because it for some reason uses more than the required quantity of services from ZaaS. In both cases, it can, therefore, handle increasing workload by reducing this over-provisioning.

As an effect of both these two other possibilities, the SLO with its quality threshold, will be less sensitive to changes in workload.
The scalability definition above covers both workload growth and reduction in workload. Workload growth is most common. Scalability is however also relevant when the workload is reduced, and where it is important to reduce the consumption of lower level services. Since provisioning of ZaaS services has a cost, increasing over-provisioning with decreasing workload is not optimal. In Section 2.3.2 we describe this relation, between provisioning of ZaaS services and cost, in more detail.

For scalability, the YaaS is allowed to find its steady state capacity. As a result the length of the measurement interval and transients are not relevant. Scalability is not the same as capacity, but capacity is an essential building block for understanding scalability, as seen by comparing Figure 2.6 with Figure 2.7. The basis for scalability is a set of configurations, as described by the resource space in Section 2.2.3.

**Elasticity**  We use the following definition of elasticity, based on [HKR13]:

> For an as-a-Service layer, elasticity is the degree to which the layer is able to adapt to workload changes by (de)provisioning services of its underlying layers in an autonomic manner such that at each point in time the utilised services fulfil the SLOs of the layer as closely as possible.

While scalability is mostly a feature of a service itself, elasticity is mostly a feature of the underlying services. Scalability describes a relation between service workload and the amount of underlaying services consumed. Elasticity describes the ability for the underlying services to offer a higher or lower amount of itself. Even a services which scales well will have troubles if the underlying services are not elastic. On the other hand, a service with poor scalability will face problems, at least with cost, and possibly also with fulfilling its SLOs, no matter how elastic the underlying services are.

In contrast to scalability, that focuses on steady state behaviour, elasticity focuses on how transients are handled. If you compare scalability to speed, then elasticity is acceleration, the ability to change speed.

**Efficiency** expressing the amount of resources for processing a given amount of work [HKR13].

In addition, we have the capacity metric which is a step towards our scalability with respect to cost metric. The capacity metric is a performance metric and is not a scalability metric.

### 2.3.1 Capacity

By this metric we find the capacity for a given configuration. Figure 2.6 illustrates how the (performance) quality (for example response times) gradually degrades (on the y-axis) when we increase the load (on the x-axis). At some point, we reach the quality threshold (on the y-axis) and the corresponding point on the x-axis becomes the capacity of this system.

For the capacity metric an exact specification of quality thresholds is important, see Section 2.1.3. This metric finds the load which just barely satisfies the SLO. This load will then be the capacity.

It is an implicit assumption in Figure 2.6 that work is constant. We will later see in Chapter 7 that in addition to the load, we can also scale several work parameters as well as the quality thresholds. We may then make similar figures where we only vary one of the work or quality parameters, and then keep the remaining parameters fixed. We may of course also make multi-dimensional graphs, but they may be hard to visualise.

The capacity metric assumes a fixed configuration with a fixed amount of resources; i.e., no auto provisioning. For each resource configuration in the resource space described in Section 2.2.3, we
will get many curves like Figure 2.6. We can combine these different resource configurations with the scalability-with-respect-to-cost metric as described in Section 2.3.2.

### 2.3.2 Scalability with respect to Cost (SC)

This metric combines several configurations with their capacities into a graph as shown in Figure 2.7, where the x-axis is normalised cost, representing quantity of underlying ZaaS services, and where the y-axis represents their relative cost. We will now look into this in more detail.

#### Cost of Configurations

The resource space as described in Section 2.2.3 defines several resource configurations and in this metric we measure the capacity of each of them. This capacity will represent the y-axis value in Figure 2.7. To compute the corresponding x-axis value, we must know the costs of each of these configurations. The cost can be computed in two ways:

**Price lists** We will simply look in the price list for our cloud service provider. For each instance type and instance size, we will multiply the number instances with the price of this instance. We will do this for all instances used. Note that this will be easy for IaaS which typically are priced per instance, but harder for PaaS and SaaS which typically have more complex price structures.
Measurements. We will simply measure the accurate cost for the use of all underlying services, including PaaS and SaaS, as well as IaaS. We need to measure for so long time that we get meaningful results. Cost may for example only be computed once every hour.

To get a value which is less dependent on actual costs, we define one configuration as a baseline configuration and divide the costs of the other configurations with the cost of this configuration. In this way the x-axis becomes relative and not absolute cost, and the x-axis will be dimensionless. This makes the x-axis value more robust to cost changes, as absolute costs of configuration are less stable than relative costs of configurations. However, this only works if the cost of all configurations are computed at approximately the same time.

One drawback of using relative cost as a measure, is that it is flexible and may change regardless of actual changes in either the workload or the deployment. However, this drawback will be smaller when we use relative and not absolute costs, simply because relative costs are less likely to change.

Scalability with Respect to Cost Graph

The larger the resource space, the more observations we get and the more fine-grained the graph will be. Similarly, the more underlying services we include in the resource space, the larger the resource variability will be and the more configurations need to be evaluated to find the optimal capacity. When we include more underlying services in the resource space, we are composing more underlying services.

In the scalability with respect to cost diagram, it makes no sense to present more expensive configurations which have poorer capacity compared to more inexpensive configurations. It only makes sense to present cost optimal configurations that may cost more than other configurations, but then also provides a capacity increase compared to these other configurations.

If we will only vary load in the usage evolution, then this may look like Figure 2.7. When we also consider different work parameters as well as quality thresholds, it will be more complex. One solution would be to look at one parameter at a time, for example one work parameter and keeping the load as well as the other work and quality threshold parameters constant.

Scalability with Respect to Services?

This scalability metric normalises the different configurations based on cost. We could also think of a scalability metric with respect to services, which specifies the relation between usage evolution for a service and the required quantity of lower-layer service. With only one deployment option (for example the m3.medium Amazon EC2 instance), the specification will be a mapping from all the variables in the usage evolution down to the number of services instances required to fulfil the required quality thresholds. With many different deployment options, such a characterisation may be complex both to evaluate and to represent. One way forward may be to use the concept of system size described in Section 2.4.1, so that we only get one dimension.

Performance Analysis Versus Scalability Analysis

Compared to performance analysis, scalability analysis has a more explicit focus on change in resources. Figure 2.6 will typically represent the performance of a layer, while Figure 2.7 represents scalability of the same layer. As also illustrated in these two figures, during performance analysis, the quality threshold is often a result of the analysis, while the result of a scalability analysis is
capacities for the resources of interest. Of course, what we here term scalability analysis may also have been performed under the umbrella of capacity planning earlier. But we will still argue that the explicit focus on change in resources is something new.

### 2.3.3 Scalability Range (SR)

Given an infinite amount of resources, our novel scalability range [BLB15] metric captures maximum capacity. If there are several types of resources, for example an application server as well as a database server, then all these resources should be set to infinity. This means that there will be no queueing for computing resources and they can in the queuing theory sense be modelled as delay centres instead of queueing centres.

The scalability with respect to cost metric in Section 2.3.2 focuses on actual configurations, and can be applied both to real systems as well as to models representing real systems. In contrast, the scalability range metric is only applicable to models. Maximum capacity is of course an idealisation, which is hard in practice. Some resources are naturally constrained, like the network resources, and also an infinite amount of processors are not easy to offer.

It is not meaningful to define an absolute lower limit in the range, other than 0, because this has to do with handling small workloads cost efficiently, which is not covered by this metric (but by the scalability with respect to cost metric).

We have now described setting amounts of resources to infinity. However, there is also the possibility of scaling by setting resource demands close to or even equal to zero. This will not be possible to test with a real system, but it may be interesting for finding the absolute maximum capacity.

### 2.3.4 Number of SLO Violations (NSLOV)

The number of SLO violations metric [BLB15] counts SLO violations during a given time interval. Conceptually this is simple, but it of course requires a precise SLO specification. With the metric response times, we must clearly specify thresholds for each operation and also clearly state if we are interested in absolute or average response time SLO violations.

With several operations, we simply add the number of SLO violations for all of them. Given an SLO violation, the extent of the violation does not count by this metric. If the response time threshold is 1.0 seconds then 1.01 seconds is a violation, just as 1 minute will be. It may be interesting to also include the amount of SLO violation, inspired by [HKR13].

### 2.3.5 Marginal Cost (MC)

The marginal costs metric [BLB15] captures how much it costs to serve a small workload increment. It is therefore essential to have a precise understanding of costs as well as of workload increments.

Concerning costs we must understand which costs are actually counted. If we only count IaaS costs, a given configuration in the resource space will be used in a given capacity interval. In this capacity interval, marginal costs will be zero. However, if we want more capacity, another configuration is required and we will experience a jump in marginal costs. When PaaS and SaaS costs are counted, even a small increment in workload will likely have a non-zero cost. However, since there are jumps in IaaS costs, we will also have jumps in combined IaaS, PaaS and SaaS costs.

When it comes to workload increments, this can be increases in both work and load. For load, we
must separate between an open and a closed system. For an open system the arrival rate may increase by a certain amount, and for a closed system the number of users in the system may increase by a given amount. Also the work may increase, e.g. there may be an increase in the number of books.

2.3.6 Resource Provisioning Efficiency (RPE)

The resource provisioning efficiency metric [BLB15] captures the mismatch to optimal provisioning. This mismatch is potentially unlimited. The resource space described in Section 2.2.3 defines the optimal configuration for each combination of load, work and quality thresholds. If we use another configuration, we will get a mismatch. This mismatch is then quantified as part of the RPE metric.

2.3.7 Mean Time To Quality Repair (MTTQR)

This metric the mean time to repair SLO violations [BLB15]. This means that each time we get an SLO violation, we have to measure the time until we now longer have a violation.

Sometimes we will have ripple effects where a large SLO violation (both in terms of amount of violation and in terms of length or number of SLO violations) gives satisfactory service for some time and then cause further periods of SLO violations.

2.3.8 Scalability Speed (SS)

The scalability speed metric [BLB15] is a scalability metric which additionally considers the rate at which a system can scale. That is, the metric defines that a system is able to achieve its SLOs at each time when the workload changes at a maximal changing rate. The rate is defined by a maximum workload and an increase rate. For example, a scalable system can scale up to a request rate of 112 \( \text{req/\text{min}} \) with a linear increase rate of 1 additional requests per month.

2.4 Related Work

This section describes related work for scalability concepts in general, and related works on metrics for the three areas scalability, elasticity and efficiency as well as combined metrics.

2.4.1 Scalability Concepts

Scalability has traditionally focused on hardware resources, not on cloud services. The amount of resources may be termed system size [BHFL04], having the following three dimensions:

- **Processing speed** addresses computation power for all types of resources, e.g., million instructions per second for a CPU, average disk access time or network bandwidth
- **Storage amount** describe the amount of non-persistent (RAM) and persistent (flash and disk) memory, including also the amount of caching at several levels.
- **Connectivity** addresses the number of intra-system connections, e.g., the number of connections between each web server and each application server, or the number of physical connections in a router.
System size is hierarchic, and we can typically define the system level, the subsystem level (nodes like web server, app server and database server), and the device level (CPUs, disks and networks). On all of these levels, size can be increased by replication. Replication at the subsystem level (adding more servers for each server role like the app server) is typically called scaling out or horizontal scaling, while scaling up or vertical scaling refers to replication at the device level (adding more CPUs, memory, disks, etc.) [BCK13].

A case study on how to measure the scalability of a Lync application is described in [RBMM13]. This article describes a method for how to measure the capacity of different amounts of hardware resources and also reflects on the amount of human time required to follow this method.

In the design of parallel architectures, the scalability concepts of scale-up and speed-up are well established. In scale-up, both the system size and the problem size increase by the same amount. In speed-up only the size of the system increases while the problem size is kept constant. While linear speed-up is too optimistic in practice, because of serial code (Amdahl's law [Amd67]), linear scale-up is possible, but not easy (Gustafson's law [Gus88]).

Gunther has defined what he terms the "universal scalability law" which describes how the relative capacity $C$ vary with the number of users $N$ [Gun07, pp. 240 – 241] (the subscript "sw" means software, a similar law applies to hardware):

$$C_{sw}(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

In this formula, $\alpha$ is the contention parameter and $\beta$ is the coherence-delay parameter. The contention parameter $\alpha$ represents the degree of serialisation of shared writeable data. The coherence-delay parameter $\beta$ represents the penalty for keeping shared writeable data consistent. When $\beta = 0$ this formula reduces to Amdahl's law.

We base our scalability concept on usage evolution introduced in Section 7. For related work on scalability with respect to cost, which is one metric for scalability, see Section 2.4.2.

### 2.4.2 Scalability Metrics

We will primarily consider scalability related to cost. In this section, we describe the relation between our work and other similar work on scalability, which at the same time also considers cost.

Jogalekar and Woodside [JW00] give a scalability definition where they combine throughput, average value of each response, and cost into the same metric. In contrast to this work, we describe how work, specified using operation parameters, load, and quality limits can scale independent of each other.

Bondi [Bon00] divides scalability into structural scalability ("ability to expand in a chosen dimension without major modification") and load scalability ("ability of a system to perform gracefully as the offered traffic increases"). In terms of this classification, we focus on load scalability in our work. However, in contrast to Bondi, we also provide concrete metrics for load scalability and apply these to the cloud computing context.

Our view on scalability is more closely related to Duboc et al. [DLR13] who describe scalability assumptions conceptually as characteristics in the application domain. These characteristics, e.g., workload, are expected to vary over time. We make our scalability assumptions explicit by describing them as operations, operation parameters, and load. Furthermore, we describe how operations with operation parameters characterise work. Duboc et al. also describe scalability goals much in the
same way as we do. In contrast to Duboc, we also closely tie scalability to service deployment and show how scalability can be expressed both in terms of service quantities and in terms of service costs. This specialisation is required when describing the scalability of cloud applications.

Brebner and Liu [BL10] compute the actual cost and measure the real performance of a typical e-Business service using two types of workloads that all have spikes during a typical day. They consider the cost for three different scenarios: in-house using a bare machine, in-house using virtualisation, and exploiting services of three different cloud computing providers (Amazon EC2, Google AppEngine, and Microsoft Azure). They also consider typical response times also including networking for these different scenarios. However, they do not provide a scalability definition, thus, making it hard to reuse their ideas in a goal-oriented manner, i.e., towards a general, quantifiable scalability concepts.

Kossmann, Kraksa, and Loesing [KKL10] compare the scalability of a TPC-W application using three different cloud services: Amazon AWS, Google AppEngine, and Microsoft Azure. They also consider costs. In contrast to our work, they neither describe scalability using more than load nor define scalability. Furthermore, they lack a focus on SaaS applications and concentrate on different service providers. We are especially interested in the scalability of SaaS applications.

### 2.4.3 Elasticity Metrics

In this section, we present related work that targets metrics for elasticity in the context of cloud computing and SLO specification. Herbst et al. [HKR13] provide a set of elasticity metrics based on speed and precision (w.r.t. avoiding under- and overprovisioning) of scale-in and -out. Because their goal is a benchmarking methodology for elasticity, they can assume full knowledge about the resource usage of the benchmarked application. However, in our case, we assume that this knowledge is unavailable because details on resources are implementation decisions. In contrast, we consider requirements specified between cloud consumer and provider. This lack of knowledge necessarily leads to different metrics as by Herbst et al., e.g., considering SLO violations instead of resource usage transparent to consumers. Folkerts et al. [FAS+12] and Islam et al. [ILFL12] both provide elasticity metrics that meet this requirement regarding knowledge. However, they lack the distinction between elasticity and efficiency because they both use cost metrics for elasticity, thus, eliminating the possibility to investigate both properties in separation.

### 2.4.4 Efficiency Metrics

In this section, we present related work that also targets metrics for efficiency in the context of cloud computing and SLO specification.

Roloff et al. [RDCN12] define basic efficiency metrics for high performance computing in cloud computing environments. They define cost efficiency as the product of costs per hour and average performance. In contrast to our work, they neglect the context, e.g., actual workload, and only take the average performance. Berl et al. [BGDG+10] address energy efficiency in all technical components of cloud computing, e.g., servers, networks as well as network protocols. We do not address energy efficiency directly but only resource provisioning efficiency. Investigating whether efficient provisioning of resources positively correlates with energy efficiency is left as future work.

CloudScale
2.4.5 Combined Metrics

The metrics mentioned above mostly focus on single quality properties. In contrast, the Cloud Services Measurement Initiative Consortium (CSMIC) provides a standard measurement framework, called the Service Measurement Index (SMI), that covers all quality properties considered important for cloud computing [SP12]. CSMIC particularly provides metrics for these quality properties, intended to be used by cloud consumers and cloud providers. Their framework allows for a structured classification of quality properties. However, they do not discuss the derivation of suggested metrics for these properties, leaving open whether and how their metrics can be used for quantification.

2.5 Future Work on Concepts and Metrics

This deliverable presents metrics for scalability, elasticity and efficiency. In the third and final project year we will look into the following issues:

**Better integration with SCALEDL OVERVIEW** The deployment concepts especially for resource space described in Section 2.2 should be better integrated with SCALEDL OVERVIEW.

**Refined definition of scalability, elasticity and efficiency** As a result of our metrics and experiences using them, we may sharpen our definitions of scalability, elasticity and efficiency.

**Refined metrics** Based on our case studies and also based on the CloudStore Showcase we will get feedback on our metrics. Based on this feedback we get a deeper understanding of the metrics, which may lead to refined metrics.

**More metrics** In this second year we have focused on the capacity metric, the scalability with respect to cost metric, the scalability range metric and the number of SLO violations metric. The remaining four metrics will be detailed in the final year. In addition, we will also consider new metrics if needed to fulfill our case study requirements.

**Composition** We will describe how we can specify the scalability of a service so that this specification can be used when composing the scalability of a service which also includes other services.
3 Scalability Evaluation Method

This chapter describes the first version of a method for evaluating the scalability with respect to cost metric for of a service. We use CloudStore as a running example, but this method can also be applied to other services. In D5.2 we show how this method is applied to measuring the scalability of the CloudStore application deployed on Amazon Web Services.

3.1 Introduction to Evaluation Method

In this introduction we will describe the objectives of this evaluation method as well as the limitations, before we sketch the general evaluation approach.

The overall objectives for this scalability evaluation method are to:

- Investigate the scalability of CloudStore deployed on actual cloud infrastructure. It is also relevant to compare the scalability of different cloud deployments, e.g. Amazon Web Services versus SAP's PaaS and a private cloud.
- Enable comparing actual measurements with the model results, relevant for both WP1 and WP5. This is because this evaluation method can be used for models just as for real systems.
- Produce measurement results that can be used for validation in our scientific articles. The quality of these results shall be accepted by leading software journals like IEEE Transactions on Software Engineering or leading conferences like International Conference on Performance Engineering (ICPE).

We also have the following more detailed objectives:

- The process leading to our evaluation results should repeatable by others. We must therefore document all configuration parameters and other information required to reproduce the measurements.
- We assume sound engineering judgement when performing the evaluation, so that for example the load generation itself do not produce errors in the measurements because of bottlenecks.
- This evaluation method shall save both human effort as well as computer execution time.

To get good results with as little (human and computerised) effort as possible, this evaluation method is divided into pilot evaluation and exact evaluation. The pilot evaluation focuses on resolving several trade offs to avoid redoing measurements.

This evaluation method has the following limitations:

- This scalability evaluation method is tailored to CloudStore deployed on Amazon Web Services (AWS), but other applications and deployment can of course also be used. AWS has the following limitations:
  - AWS/EC2 require approximately 3 minutes to initialise new instances.
The billing period is 1 hour, which means we cannot distinguish cost on a smaller time scale.

- In this first version of the scalability evaluation method we will not evaluate different values for the work parameter or for the quality threshold parameter, but will only vary load. However, it should also be applied to work. For CloudStore work will typically mean number of books or possibly number of customers.

- At this stage the pilot evaluation method is a bit complex. It will be clearer once we get more experience.

- Using this evaluation method we can evaluation the capacity for a given configuration and we can also evaluate the scalability versus cost for a resource space. This will give some indications for maximum capacity, but we cannot find the scalability range when testing actual hardware.

- The influence of the network between our load generation and AWS will not be considered as part of this evaluation method.

The overall method for evaluation scalability is as follows:

- For each of the configurations specified by the resource space described in Section 2.2.3, we find the capacity using binary search. This is specified in Section 3.3, which also describes how we find the bottleneck leading to this capacity as well as the cost of this configuration.

- After we have measured each configuration, we make a scaling graph, which summarises the scalability of the service for all the different configurations in the resource space. This graph then shows how capacity varies as a function of configuration cost.

- Since we are in the cloud environment, we use cost expressed as $ per hour when making this graph. These costs are then normalised so that we set the lowest cost to 1, which removes the reference to absolute cost and makes the results more robust to price fluctuations.

Key concepts for this evaluation method are described in Chapter 2 and in Chapter 6 as well as in Chapter 7.

### 3.2 Pilot Evaluation

The overriding concern of the pilot evaluation is to get a feeling for the trade-off between the amount of manual work versus the quality of the measurement results. The closer to ideal measurement quality, the better, but at some point the cost will of course be too high and we have to do something more pragmatic. The effort involved in performing the pilot evaluation can be reduced by doing fewer measurements and by running each measurement for a shorter time. The output of the pilot evaluation is a set of decisions concerning these trade-offs. It is important that an outsider is involved when discussing these decisions, so that we can save time when continuing with the evaluation method and avoid redoing measurements because of change in major trade-offs.

When comparing the same CloudStore application deployed on different public or private cloud services, as well as when comparing a CloudStore model run in the Analyser with a real CloudStore application deployed on cloud services, it is essential to test the system under similar circumstances or using the same trade-offs, for example with or without auto scaling or with or without cache.
3.2.1 Load Generation

This evaluation method assumes the use of a load generator which in our case is JMeter [jme14]. The load generator must be running on a client that has enough memory, CPU power and network bandwidth, as well as low network latency, so that it does not introduce errors.

For testing one configuration with one load, we need time to evaluate a given load as specified by JMeter parameters. There are the values we have used so far:

**Start Thread Count:** similar to the load we are using during tests.

**Initial Delay:** 30 seconds has been used.

**Startup Time:** 300 seconds (5 minutes) has been used.

**Hold load for** 1800 seconds (30 minutes) should give good and accurate results when we are close to the capacity. Shorter hold load time will be use during pilots testing. If cost shall be measured, we need longer hold times.

**Shutdown time** 0 seconds which means we release cloud resources as fast as possible.

One goal of the pilot evaluation is to check that the load generation itself is not a bottleneck.

3.2.2 Quality Thresholds

The quality thresholds will be used to find the capacity for all configurations. It is therefore important to agree on the quality thresholds before starting the evaluation and likewise not to change them during the evaluation.

Ideally, quality thresholds are set as part of the negotiation between the producer and the consumer of a service. If no real consumer is available, you have to make some effort into establishing reasonable quality thresholds. On the one hand, quality thresholds should give an acceptable quality to the consumers of a service, but on the other hand, it shall not be so strict that it is impossible to fulfil.

We can also evaluate different quality thresholds, corresponding to scaling the quality threshold, but this will of course increase the effort involved in scalability evaluation, and is not part of this version of the evaluation method.

3.2.3 Work

Since we will not vary work in this version of the evaluation method, we have to select meaningful values for work parameters like number of books and number of customers. These values will be constant for all measurements.

3.2.4 Cache

Caching is a great technique for improving the performance of applications. However, the more cache, the harder it is to get repeatable results, because some results benefit more from cache than others. At least it is important to be able to tell when the cache is primed with previous similar results and when it is not. Proper restarting of experiments is part of the answer.
3.2.5 Auto provisioning

When evaluating scalability it is easier to get accurate results with auto provisioning turned off. The quality criteria that are used to determine auto provisioning, need not be in sync with the quality criteria that are specified as part of the usage evolution. For example, if a new instance is added when utilisation is above 70% and if instances are removed when utilisation is below 40%, this may not exactly correspond to a quality threshold of 3 seconds. As auto-scaling rules are designed to give good quality with fluctuating load, it will probably lead to over provisioning when measuring scalability, which depends on steady state.

3.2.6 Configuration Parameters

The CloudStore has several configuration parameters. More effort will give better values for these configuration parameters and as a result also more accurate measurement results. Examples of configuration parameters are:

- Number of software resources like database connections, as well as worker pool size. For example, the better the match between the number of database connections and both the CloudStore application and the available resources, the higher capacity for a given resource configuration and the better the quality of the measurement results.

- Used more widely, we may also include the selection of database type, since more ideal database type will affect software resources like semaphores, because a more sophisticated database will use less restrictive semaphores. For example, a relational database will use more semaphores than a non-relational database, and will as a consequence have more locking and therefore a lower performance and therefore also lower capacity.

- Number of database replicas. More replicas may give faster reading but slower writing and will therefore also be a tuning parameter.

Configuration parameters may have different values for different configurations, which will make this very demanding. Out assumption is that is is not true.

3.2.7 Resource Space

The concept of resource space is described in Section 2.2.3, and must be clarified before measurement can start. Basically, you must know the resource types which should be investigated during measurements. A larger the resource space, means more observations and a more robust scaling curve, but it also means a larger effort.

When combined by modelling a smart approach is to use the model to investigate a large resource space, so that the much more demanding measurements only need to explore a narrow resource space. If we have no model, then this smart approach of course does not work, and we are left with using measurements to explore the whole resource space of interest.

3.2.8 Estimate Time to Measure

Before finishing the pilot phase it is wise to give an estimate on the amount of human and computer time required to execute a scalability evaluation. Based on these estimates it is possible to optimise especially the use of scarce human resources, but also of computer resources.
The first step is to identify the required steps, like to populate instances with data, to set up the front end, to evaluate the capacity of one configuration and to find the bottleneck. For each of these steps, both time to do the human and the computerized part should be recorded. This time may depend on some parameters, which must then also be described.

### 3.3 Analysing One Configuration

We here consider one configuration. First we determine the capacity of this configuration. Afterwards we analyse the bottleneck in this configuration. Finally, we compute the cost of this configuration. All this is then input to the scalability-with-respect-to cost graph which is described in Section 3.4.

#### 3.3.1 Finding Capacity using Binary Search

In the pilot evaluation, we have established an upper bound on capacity. We will now find the exact capacity using binary search. If we know that the capacity is lower than $x$, we test with load $\frac{x}{2}$, where we have two outcomes:

- **SLO is obeyed:** we know capacity is within $\left[\frac{x}{2}, x\right)$. Afterwards, we test with load $\frac{3x}{4}$.
- **SLO is violated:** we know capacity is within $[0, \frac{x}{2})$. Afterwards, we test with load $\frac{x}{4}$.

This is continued until we find the capacity with the required amount of precision.

When we have found the capacity, we measure one more time, but now with a longer hold time, so that we are also able to record costs and at the same time also get more accurate results with a better confidence interval. If we measure with auto provisioning we must be sure to measure for so long that number of instances stabilise.

Apart from average response times and error rate, we also need the Box-and-Whisker plot, which presents and therefore also requires, the following information about distribution of response times: minimum, 25 percentile, 50 percentile, 75 percentile and maximum. In practice, this means that for each measurement using JMeter, we have to take care of the output generated by JMeter and that we must ensure that we here have the following parameters for each request:

- **Time stamp** for the start of the thread
- **Response time** (termed “real elapsed” in JMeter)
- **True/false** depending whether the request was successful or not

When we have these values for each request, we are afterwards able to generate all the information we need, including a Box-and-Whisker plot.

#### 3.3.2 Determining Precision of Capacity Measure

When finding the capacity for a given configuration, we basically vary the load (or work, but not in this version of the evaluation method), and stop when we have found a good match with the quality thresholds. The closer to the quality threshold we wish to come, the more experiments we must do.
The trade-off between number of measurements and length of measurements versus quality of measurements, involves several issues which are described in separate below.

**Number of Experiments**

If we want to find the exact capacity it may take up to $\lceil \log_2(x) \rceil$ experiments to find the capacity limit, in the interval $[0, x)$, so with $x = 800$, we may have to use 10 experiments. If we relax requirements and want to find the capacity with a precision of $y$, we need fewer experiments. This can be expressed by:

$$\lceil \log_2\left(\frac{x}{y}\right) \rceil$$

Therefore, if we want to find capacity with a limit of $y = 100$ and start with $x = 800$, we need 3 experiments, which is considerably less than the 10 experiments required to find the capacity with a precision of 1.

**Run Time for Each Experiment**

Generally, the longer we run each experiment, the more accurate the result is, but then we also have to wait longer to get the result. We need a trade-off between run length and accuracy. In this evaluation method we use a short run length during pilot evaluation and a long run length for the accurate measurements. As a starting point, we will use 5 minutes hold time during pilot evaluation and 30 minutes for the accurate measurements.

The run length will be checked against confidence intervals. Lilja [Lil00, p. 53] describes how many experiments which is required for a certain confidence interval. This equation requires the standard deviation, therefore we need to run some measurements before it is possible to find out how many measurements we need.

**Remove Transient Phase**

To get accurate results it is beneficial to remove the transient phase. Jain [Jai91, pp. 423 – 428] describes several approaches to remove transients so that we only focus on steady-state, where a common assumption for the last four is that the variance during steady-state is lower than in the transient phase:

- **Long Runs** Measure for a long time to dilute initial transients. This is simple, but costly. Hard to know how long time which is required.

- **Proper Initialisation** Start close to the steady state. A good techniques, but we need to consider more closely how it can be applied to CloudStore.

- **Truncation** Ignore initial observations until they no longer constitute max and mean.

- **Initial Data Deletion** Assuming several complete runs and then look at the change of the mean after removing initial observations until the change in the overall mean do not change.

- **Moving Average of Independent Replications** Similar to the previous technique except for using moving averages instead of the overall mean.
**Batch Means**  Running a long simulation and then dividing the measurements into several batches. Observe the variance of the batch mean and identify the transient as when the variance starts decreasing.

We will consider batch means and proper initialisation. If we use auto scaling, then run time must also be long enough for auto scaling to take effect.

To determine rough capacity range for all configurations is part of the pilot evaluation. We can then narrow the scope for the more demanding exact measurements. Because of transients, results with little run time will generally be optimistic. This means that quality thresholds may be broken even if a short measurement run does not tell so. One the other hand, an SLO violation will most likely be worse with a longer run time. Therefore upper bounds on capacity should be fairly robust, but they may bee too high.

### 3.3.3 Analysis

It is important to get an understanding of what is the cause of the limit in capacity. Key questions are:

- What is the bottleneck resource and is it hardware or software resources?
- Have we selected good values for the configuration parameters, like size of connection pooling?
- Is the cause of the capacity limitation the measurement set up including the machine holding the JMeter load generator more than the actual deployment for CloudStore?

Ideally, it makes sense to have the most expensive hardware resource as the bottleneck. Sometimes it may be impossible to avoid software resources because this requires a major rewrite of the application, and this rewrite will implicitly be more costly than any hardware resource.

### 3.3.4 Compute Cost

We focus on the cost for the IaaS resources. For a configuration with used for EC2 with one instance of m3.medium (assuming no auto scaling) and with one RDS master and three RDS replica instances of db.m3.medium we estimate the cost as follows, based on the prices found at Amazon:

- EC2 with m3.medium: 0.077 $/hour
- RDS with four db.m3.medium: 4 * 0.095 $/hour
- Total cost: 0.362 $/hour

A better approach is to measure the cost in AWS, but this requires running for such a long time that costs gets accurate enough. This enables to also take PaaS costs like the S3 costs into account, as well as networking costs etc.
3.4 Presenting Scalability with Respect to Cost

We want to present all configurations together with their respective capacities, as described in Figure 2.7. The y-axis in this graph is the capacity of a given configuration, while the x-axis is the normalised costs, which is the cost related to the cost of one of the configurations. This makes the cost estimate more robust to cost changes, as absolute costs are less stable than relative costs.

If in addition to testing with different instance sizes we also vary number of instances; we would have got more configurations and therefore more points on this curve. The more configurations we test, the more points we get in this graph. Note that it makes no sense to show configurations that in addition to a higher cost also have poorer capacity compare to other configurations. It only makes sense to present cost optimal configurations that may cost more than other configurations, but then also provides a capacity increase compared to cheaper configurations.

3.5 State of the Art in Scalability Evaluation Methods

The evaluation method described borrows several elements from the method by Rygg, Brataas, Millstein, and Molle [RBMM13], but is adapted to the cloud and also to a cost-based presentation of the different configurations in the resource space. Also the introduction of the pilot phase is new. As a result cloud evaluation method are structured differently.

A paper by Weber, Herbst, Groenda, and Kounav [WHGK14] focus on elasticity, but also describes how scalability evaluation must be done in advance, based on what they term benchmark calibration in their Section 4.2. This is also in line with our view.

3.6 Future Work on Scalability Evaluation

This method will be refined in the next year when we validate it also in the industrial case studies. It will also be better integrated with the CloudScale Method. We will extend it to work and quality threshold evaluation.
4 Elasticity Evaluation Method

This chapter describes the first version of a method for evaluating the number of SLO violations metric for a service. This is the elasticity metric which is currently implemented by the Analyser. We use CloudStore as a running example, but this method can also be applied to other services. In D5.2 we show how this method is applied to measuring the number of SLO violations metric for the CloudStore application deployed on Amazon Web Services.

This evaluation method builds on the scalability evaluation method in Chapter 3, and we will focus on discrepancies between these two evaluation methods in this chapter.

As before, it is essential to have a very clear definition of the SLOs, especially as the focus on elasticity testing currently is on counting the number of SLO violations. We will extend to other elasticity metrics, once they are implemented.

Apart from describing the exact version of CloudStore with all configuration variables, it is essential to also describe in detail the specific cloud resources which is used during this test.

4.1 Test Workload

Load is varied as described in Figure 4.1. Load is constant at level $A$ until time $c$. Then load is linearly increased to reach load $B$ at time $d$ and this level is kept until the test is finished at time $e$. Even though Figure 4.1 shows a load increase with $A < B$, we may of course also have a load decrease, where $A > B$. The parameters in Figure 4.1 are summarised in Table 4.1. When testing elasticity we must document these values.

For an open system, load parameters are in terms of arrival rate, whereas for a closed system, load parameters is for number of consumers in the system. We have the following special cases:

- **Step function** where $c = d$
- **Zero initial load** where $A = 0
Table 4.1: Parameters when testing elasticity.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial load</td>
<td>$A$</td>
</tr>
<tr>
<td>Final load</td>
<td>$B$</td>
</tr>
<tr>
<td>Time when load increase starts</td>
<td>$c$</td>
</tr>
<tr>
<td>Time when load increase ends</td>
<td>$d$</td>
</tr>
<tr>
<td>Duration</td>
<td>$e$</td>
</tr>
</tbody>
</table>

**Zero final load** where $B = 0$

When combining several instances of load increase or decrease, we may build more complex patterns. We may for example first increase load and then afterwards decrease the load.

As before, we hold work constant in this first version of the elasticity evaluation method.

When testing elasticity we will experiment with several values of $A$, $B$, $c$, $d$, and $e$:

- The duration of the test $e$ as well as the time for the increase $c$ will be constant. As possible values, $e$ will be set to 10 minutes and $c$ will be set to 2 minutes.
- We will experiment with both increase ($A < B$) and decrease ($A > B$).
- We will test with as close to instantaneous increase as possible ($c \approx d$), but also with slower increase and decrease rates.

### 4.2 Counting SLO Violations

As a result of this test, we simply have to count the SLO violations for the duration of the test.

### 4.3 Future Work on Elasticity Evaluation

This method is will be refined in the next year when we validate it also in the industrial case studies. It will also be better integrated with the CloudScale Method. We will also include more elasticity metrics and will accordingly make elasticity evaluation methods for them.
5 CloudScale Method

This section describes the CloudScale Method. The main changes from the previous version include polished method steps and introduction of the plan for presenting method usage effort estimation based on the validation results. The method describes how to model self-adaptive elastic services. Moreover, it also gives advice on how to develop scalability models, which can be used for "what-if" analysis. The overall CloudScale Method is a collection of concrete process steps required to engineer scalable cloud systems. CloudScale Method describes how to model scalability both during design and during evolution of cloud systems. It also defines the input of a scalability model and what output can be produced based on this input. Finally, in later version of this deliverable, the method will also describe the effort involved in each process step and gives advice on model granularity.

5.1 Method Overview

The key to the development of scalable cloud applications is an appropriate engineering method for the scalability. Initially, a service provider has an idea for an application which he wants to execute in the cloud because of the cost-efficient management and the virtually unlimited amount of hardware resources. The service provider wants to ensure that the application scales cost-effectively, i.e., that it always tries to cope with its workload with the minimum amount of cloud resources (measured in terms of costs paid for the resources).

To enable sustainable engineering of the method for scalable cloud applications it is important to support the complete application life-cycle. The proposed CloudScale Method builds on an overall system life-cycle process from initial requirements collection towards operation and monitoring process.

5.2 CloudScale Method Scenarios and Basic Elements

CloudScale provides an engineering method for building (evolving) and adapting scalable cloud applications and services. We focus on two core scenarios:

Development: Enabling software engineers to develop scalable applications and services for a cloud computing platform. We extend existing capacity planning tools for scalability analysis, and introduce ScaleDL Usage Evolution to describe, compare, and compose the scalability of services. We also provide a set of best practices and design-patterns for building scalable software systems.

Evolution: Enabling software engineers to evolve an existing software system or service into a solution that scales in cloud computing environments. We introduce a novel approach for scalability evaluation of existing systems by systematic experimentation using existing load-testing tools or making experiments on the models. This approach allows for scalability anti-pattern detection in an existing application and for extracting scalability models.

Furthermore, the CloudScale Method will enable a combination of both scenarios. The scalability evaluation of existing systems will yield scalability models that allow for scalability redesign and the evaluation of "What-if" scenarios that can be combined with measured behaviour from deployed
system. The integrated view the CloudScale Method provides on scalability allows software engineers to address scalability in all lifecycle phases of their application with minimal effort. The overall CloudScale Method overview is presented on Figure 5.1.

![CloudScale Method Overview Diagram](image)

**Figure 5.1: CloudScale Method Overview.**

On the bottom of the figure is a legend explaining the notation. It is important to form clear process steps which cover essential service life-cycle steps like Requirements, Design, Realisation, Operation and Monitoring. It is also important to show the data and control flow between the process steps. The most important service life-cycle steps are additionally elaborated with supported tools and intermediate documents. To enable experimentation and iterative construction and analysis cycles, a couple of decision points for control and data flow were introduced (e.g., a decision step where we determine if scalability requirements are satisfied after system modelling and analysis). Additionally, the method enables solution check after each service life-cycle steps and then returning to the analysis step to optimise the constructed service.

The basic processes in CloudScale Method are standard development processes in software development. We make some changes in the naming of processes to have more focus on the specific need of the CloudScale Method. The first defined process is Requirements identification, because the CloudScale Method will specifically deal with scalability issues in the system development or adaptation, and will be integrated with some other general accepted engineering method for requirements engineering. However, it may also be executed independently. This process is defined because we must always annotate and define scalability requirements for the analysed system. During the Requirements identification process, the main focus is on describing the evolution of load and work. The quality metric describes what is acceptable system quality to the users, e.g., a particular response time. For example, we may expect used service response time in less than one second. This process results in a requirement specification document that is visible as ScaleDL Usage Evolution Specification. This output document of the Requirements Identification process is used like...
main input in the System construction and Analysis process.

Based on this requirement specification, the process System Construction and Analysis starts where we will use a model for specifying the system. We can do this in two ways:

1. By Reverse Engineering using an existing code base for creating an initial or adapting an existing system model. This process is driven by our tool Extractor.

2. By (Re-)Designing a system on the model level using our Analyser tool. For specifying this model, we will use the ScaleDL Usage Evolution and Palladio Component Model (PCM), see Section 10.

We guide our design decisions along the requirements specification and support it with known patterns for good architectures regarding scalability in cloud environments by using ScaleDL Architectural Templates. The output document from Extractor or newly designed system that will support (Re-)Design of system will be unified System Model that will be representation of ScaleDL Instance as basic input for Analyser and Static Spotter tools. The next step involves the analysis of the modelled system to check whether it meets the identified requirements. This process is driven by our Analyser and Static Spotter tools as described in Section 5.4.3 and is repeated with different system alternatives until the requirements are met. The main reason why we combine these two tools is because they logically do similar tasks. And development of these two tools is still in early stage and when clear tool inputs and outputs will be defined, we will update the CloudScale Method definition.

If first model does not satisfy requirements we shall apply Spotter to make an analysis where we can compare the Anti-Patterns & Solution based knowledge/parameters and Analyser driven output parameters. Results from the Spotter is compared with Available Solutions and if we are satisfied with the proposed Solution we will apply an Anti-Pattern solution to create a new ScaleDL Instance of the System model and go to the Analyser tool with new model. In the situation when the solution is not available, we shall return to our construction and analysis process for improving the existing system model. This construction and analysis loop stops when system architect is satisfied with the system behaviour — when the requirements are met. The system construction and analysis process finally ends by reaching satisfactory results about service scalability behaviour.

We may also find that our scalability requirements are infeasible and, therefore, we have to modify them by reducing the complexity (and consequently work) of the services offered during the high load.

The two main outputs from the system construction and analysis process are the Realisation Directive and the Deployment Directive. These two directives form the basis to continue with next steps in software applications life-cycle called Realisation and Deployment.

The Realisation Directive can be used to automatic code generation in the Realisation process, either based on a developed system model or for semi-automatic code generation. The Deployment Directive contains essential requirements/parameters for services deployment, which is required to satisfy required system behaviour. In some situations, output from the System Construction and Analysis process can be only a deployment directive, e.g., in cases when we use only existing and already developed service components, and when we define only parameter reconfigurations for each component, which determine the service deployment process.

Based on a Realisation Directive we start to implement our system. The output of system realisation is a Realised System prepared for deployment.

When the Realisation process step is finished, we move to Deployment. In the Deployment process step, the application is deployed according to the Deployment Directive, and put in operation in a cloud computing environment. For operation it is important to specify well-defined resource require-
ments that enable cloud computing provider to efficiently provide cloud resources and to fulfil load, work and quality requirements of the deployed application reflecting the application evolution.

*Monitoring* is another process step that is active during the system operation and enables control of system behaviour. Collecting measurements for scalability parameters also belongs in this step. Based on the operational parameters and system quality metrics, monitoring control can require some changing system requirements and triggering the need to rerun our process cycle (adaptation loop).

Based on systematic experiments, enabled with the *SpottingByMeasuring* process, software engineers will receive the information needed to assess costs, identify scalability problems, and address such problems accordingly. This will address both deployment evolution as well as architectural evolution. In the area of identification of scalability anti-patterns, *CloudScale* will identify and formalise scalability anti-patterns for cloud applications and provide a tool-supported method to detect these patterns on existing code-bases.

### 5.3 CloudScale Method Roles and Stakeholders

In the previous overall *CloudScale Method* description we have mainly used the role of software engineer, but it is important to emphasise that in the proposed method we envision four basic roles for the software engineer:

**Product manager** Person responsible for discussing with the customer of our service and identify initial system requirements and define development goals especially from the business perspective. The product manager is always active during the decisions regarding the requirements fulfilment and business potential of the solution, i.e., in the *Requirements identification* and *System Construction and Analysis* process steps.

**System architect** Person responsible for *Requirements Identification* and the main driver of *System Construction and Analysis*. This role cooperates with Product manager and Service developer and its main responsibilities are architecture definition and to propose the main service components.

**Service developer** Person responsible for service realisation (both development and test), and for preparing the system deployment process. This role cooperates with System architect for checking realised services and with System engineer in preparing system deployment.

**System engineer** Person responsible for service deployment and monitoring of the system in operation. Based on monitoring results, the system engineer tends to optimise system operation parameters or to run the *System Construction and Analysis* process step if it is not possible to fix system by fine tuning. System engineer cooperates with all other roles during the system life-cycle.

Basic mapping of roles and *CloudScale Method* is presented on Figure 5.2.

Looking from the perspective of *CloudScale Method* usage, we can define four basic system stakeholders:

**Service consumer** wants to have supplied services for own purpose according to business needs and according an agreed SLA.

**Service provider** responsible to fulfil SLA and other requirements towards service user (according to cloud services it can be IaaS, PaaS or SaaS provider), preparing service requirements.
5.4 Overview of the System Construction and Analysis process step

This section gives the detailed overview of steps that are needed, and tools that will support process of construction and analysis during the system initial design or evolution. The essential part of the description is to define (1) what are the inputs, (2) what are the expected outputs, (3) which the tools support of transformation from input to the output and (4) who is responsible to that. The examples are taken from analysing the CloudStore application and from transferring it to the cloud environment.

5.4.1 Extractor

Process type: tool-driven
Roles: System Architect
Input: Existing System, ScaleDL instance
Output: **Extractor Output**

**Description:** Extractor extracts software architecture from source codes and visualizes with PCM model. During extraction Extractor finds CloudScale Architectural Templates in software architecture automatically. Extractor extends Tool Archimetrix, enhances its functionalities and provides support for cloud computing environment.

**Example:** The architect who wants to model CloudStore uses Extractor to generate software architecture model automatically. He simply runs Extractor using the CloudStore code base.

**Status:** Extractor is under development. It is based on Archimetrix. It enhances Archimetrix functionalities, e.g. it could generate PCM model, and extends its support for cloud computing environment.

**References:** [vDL13]

5.4.2 (Re-)Design System

**Process type:** manual

**Roles:** System Architect

**Input:** Requirements Specification, **SCALEDL Architectural Template**, **SCALEDL instance**

**Output:** **SCALEDL instance**

**Description:** The goal of the (Re-)Design System process is to design a system model describing the system to be realized on the model level. For specifying this model, we use the **SCALEDL Usage Evolution** meta model as described in Section 7.

We guide the design process along **SCALEDL Architectural Templates** promising to fulfill scalability requirements specified by the requirements specification. The design process requires to manually fill chosen **SCALEDL Architectural Templates** and, therefore, leaves some design decisions to the system architect. However, subsequent processes (e.g., the **ANALYSER process**; cf. Section 5.4.3) provide automated support for (1) checking whether the system fulfills specified requirements and (2) identifying causes for requirement validations (if any). Hereby, we assure that requirements are met even though we explicitly allow making design decisions when filling **SCALEDL Architectural Templates**.

Depending on the input to the (Re-)Design System process, we allow both, designing a system from scratch or redesigning an existing **SCALEDL Usage Evolution** instance (the **SCALEDL Usage Evolution** instance is an optional input). When redesigning an existing **SCALEDL Usage Evolution** instance, the architect starts with the existing instance and alters it, e.g., according new requirements, to evolve the system. In this evolution case, the architect does not need to re-model system parts that are not affected by the evolution.

**Example:** An architect is ordered to design an elastic CloudStore from scratch. The "scalability requirement" is specified in the requirements specification: the system should be capable of responding in less than 3 seconds for user arrival rates between 1 and 60 users per hour. As a starting point, he decides to develop the system for AWS Elastic Beanstalk (being an elastic PaaS environment). To do so, the architect creates a new **SCALEDL Usage Evolution** instance and instantiates the **SCALEDL Architectural Template** specific to AWS Elastic Beanstalk. Therefore, he has only to complement the template with the missing web, application, and data logic. For instance, the architect provides components for (1) creating a UI for best selling books, (2) the application logic that fills the UI, and (3) a database query used by the application logic.
to receive a random set of five best selling books. On the model level, he only models the scalability-relevant behavior of these components. For instance, he annotates the application logic to be stateless. The architect is not required to cope with other issues as, e.g., handling the database connection or setting up load balancers. Scalability-relevant information about these issues are included in the ScaleDL Architectural Template and allow, e.g., the Analyser to predict whether the elasticity requirements are actually met.

**Status:** The "(Re-)Design System" is described on a high level of abstraction. More details are planned for M24, i.e., once the complete CloudScale Method has been applied on first examples.

**References:** The initial idea of the (Re-)Design System process are sketched in [BSL+13].

### 5.4.3 Analyser and Static Spotter

**Process type:** tool-driven

**Roles:** System Architect

**Input:** Requirements Specification, ScaleDL Usage Evolution instance

**Output:** Analyser and Static Spotter Output

**Description:** Analyser and Static Spotter process analyses a modelled system (ScaleDL Usage Evolution instance) to check whether it meets the scalability requirements of the requirement specification. The Analyser is a tool that can automatically execute these checks and Static Spotter can detect scalability issues on model level.

To do so, outputs of this process are scalability predictions for the modelled system. Based on this first output, Analyser can further output whether a requirement is met or not, respectively. In case it is not met, additional analyses can automatically be executed by Static Spotter to detect the causes for that (e.g., Static Spotter identifies a guilty component).

**Example:** The architect who modelled CloudStore via a ScaleDL Usage Evolution instance executes the Analyser on this instance. The elasticity requirement specified in the requirement specification is checked (the system should be capable of responding in less than 3 seconds for user arrival rates between 1 and 60 users per hour). The Analyser then predicts whether the requirement is met or not.

**Status:** Analyser is currently in a conceptual phase. It will be based on Palladio and Simulizar. The first prototype allowing to analyse a simple example will be available at M12. Static Spotter is also currently in a conceptual phase.

**References:**

- The initial ideas of the (Re-)Design System process are sketched in [BSL+13].
- Sec. 5.4.3 provides details about the Analyser’s output.
- Chap. 10 describes the Analyser in detail.

### 5.4.4 Dynamic Spotter

**Process type:** tool-driven

**Roles:** System Architect

**Input:** System Model or Analyser Output, ScaleDL Usage Evolution instance
Output: **Dynamic Spotter Output**

Description: **Dynamic Spotter** is a tool which finds components which are responsible for hindering performance by inspecting software architecture and codes. If these responsible components fit certain anti-patterns, **Dynamic Spotter** also suggests solutions automatically. **Dynamic Spotter** uses System Model or **Analysyer** Output as an input with proper system operational parameters and works with running system.

Example: After **Analysyer** and **Static Spotter** find that System Model does not fulfil Requirements, the architect runs **Dynamic Spotter** to locate “bad” components. **Dynamic Spotter** inspects System Model and codes. When **Dynamic Spotter** finds anti-patterns in the existing system, it also suggests solutions to alter the flawed software architecture or codes. The architect chooses one of the suggested solutions and **Dynamic Spotter** applies the change automatically.

Status: **Dynamic Spotter** is currently in a conceptual phase.

References: [WHH13]

### 5.5 Effort for each Step

Initial effort estimation and results will be described using the following concepts:

**Activity definition:**

1. Requirement identification phase effort estimation
2. System construction and Analysis phase effort estimation
   - (a) Extractor usage effort estimation
   - (b) Analysyer usage effort estimation
   - (c) Static Spotter usage effort estimation
3. Realisation phase effort estimation
4. Dynamic analysis phase effort estimation
   - (a) Dynamic Spotter usage effort estimation
5. Deployment phase effort estimation
6. Operation phase effort estimation
7. Monitoring phase effort estimation

**Potential duration (person-hours):**

1. Best case scenario effort estimation
2. Likely case scenario effort estimation
3. Worst case scenario effort estimation
4. CloudScale partner scenario effort estimation
   - (a) ENT scenario effort estimation
   - (b) SAP scenario effort estimation
   - (c) Showcase scenario effort estimation
5.6 Granularity

One of the main tasks for the CloudScale Method is to provide advice on model granularity that is used during the system design and analysis. During the system lifecycle that is described by method steps, different types of system models are used and for each of these types we can define different levels of granularity in order to achieve more accurate scalability and performance predictions. For most of the method steps and processes that are included in these steps, system architect role defines the granularity of the model that is used. Systems that are represented by models included in Method steps are defined as composition of different components and granularity defines the used composition level for each of these components. During the analysis and prediction activities we can regulate the granularity of the model, i.e., to what level are the components going to be detected as an individual component and specify the level of merging the components and composing them into more complex components.

During our validation activities (for case studies and showcase) we discovered that there is a trade-off between the costs of modelling (e.g. time effort, development hours, system analysis, development/configuration details) and the accuracy of the results that predict scalability when using CloudScale tools. One of the examples for granularity changes are represented by ENT validation activities during System Construction and Analysis phase and usage of Extractor tool. More detailed granularity impact on scalability prediction and different strategies for component selection are described in D2.2 under the ENT case study description.

The major concern the during model definition is to determine the appropriate level of details when designing new component as one of the main building blocks of each model. First, it is extremely important to determine the impact of the component to the overall behaviour of the system under analysis and create modelling plan based on this input. Components that do not have major impact on the overall system behaviour could be simply defined as abstract components. These types of coarse grain model elements are maximally abstracted and consist of a set of black-boxes. By using this type of strategy many lower level functionalities and interactions are ignored and the focus is put only on most important functionalities and interactions. In that why, without specifying their internal construction, we can produce model much faster and reduce model development cost. Other, more significant components (in the scope of scalability impact) cloud require to be modelled in more detail than others. Fine grain model elements are minimally abstracted or not abstracted at all. Modelling activities for this type of components require significant amount of time because they need to reflect the structure of the source code.

5.7 Future Work on CloudScale Method

This deliverable presents CloudScale Method refinement considering the work presented in the previous deliverable version. The following topics will be covered in the future work on CloudScale Method:

**Additional CloudScale Method validation:** CloudScale Method and CloudScale Method steps will be validated in more detailed way using industrial use cases and showcase.

**CloudScale Method process steps refinement:** Based on the validation results and outputs we will produce new version of the method that will include all validation observations and recommendations. When validating the method we will also use the evaluation methods in Section 3 and in Section 4.

**CloudScale Method process steps effort estimations:** Each CloudScale Method step will be pre-
sented in more detailed way and effort estimation will be defined based on the validation results.
6 ScaleDL

The Scalability Description Language (ScaleDL) is a language to characterize cloud-based systems, with a focus on scalability properties. ScaleDL consists of five sub-languages: three new languages (ScaleDL Usage Evolution, ScaleDL Architectural Template, and ScaleDL Overview) and two reused language (Palladio’s PCM extended by SimuLizar’s self-adaption language and Descartes Load Intensity Model (DLIM)). For each of these sub-languages, we briefly describe its purpose and provide a reference to a detailed description:

**ScaleDL Usage Evolution** allows service providers to specify scalability requirements, e.g., using cost metrics, of their offered services (cf., Chap. 7)

**ScaleDL Architectural Template** allows architects to model systems based on best practices as well as to reuse scalability models specified by architectural template engineers (cf., Chap. 9)

**ScaleDL Overview** allows architects to model the structure of cloud-based architectures and its deployment at a high and user-friendly level of abstraction (cf., Chap. 4 of deliverable D3.1)

**Extended PCM** allows architects to model the internals of the services: components, components’ assembly to a system, hardware resources, and components’ allocation to these resources (cf., Sec. 10.2); the extension allows additionally to model self-adaptation: monitoring specifications and adaptation rules (cf., Sec. 10.3)

**Descartes Load Intensity Model (DLIM)** allows modelling of load intensity in terms of arrival rates over time, but can also be used for modelling of the evolution of other attributes (cf., Chap. 7.3 and [vKHK14])

Figure 7.1 shows an overview of how the languages relate to each other, and the transformations and other components they are input to and output from. Components and relations in gray are planned for the next release of ScaleDL languages and tools.

---

**Legend**
- Component
- Model
- Planned component
- Report

Figure 6.1: Overview of ScaleDL sub-languages and their relationships.
Architects can use the **ANALYSER** to automatically analyze a modeled system with **SCALEDL**. This analysis allows architects to check whether a system meets its scalability requirements stated via a **SCALEDL USAGE EVOLUTION** instance. We detail the **ANALYSER** in Section 10.
7 ScaleDL Usage Evolution

The overall task for ScaleDL Usage Evolution is to enable the characterisation of scalability requirements for a service. Using ScaleDL Usage Evolution, the System Architect can specify all the relevant information about the expected usage and quality thresholds. It will then be possible to perform scalability analyses using Analyser (or any other tools). Looking at Figure 2.5, ScaleDL Usage Evolution will characterise the usage interfaces between XaaS and YaaS.

This chapter first describes an overall scenario for ScaleDL Usage Evolution. We will then describe the basic building stones of the ScaleDL Usage Evolution, Deployment is characterised in Section 2.2. Based on workload and deployment characterisation, we define scalability in Sec. 2.3. We will then describe a first sketch of a meta model for ScaleDL Usage Evolution. Finally, we present some future work. We will use the CloudStore application described in D5.2 as a running example.

7.1 Overall Scenario for Usage Evolution

CloudScale will primarily assist the scalability engineering of projected or existing SaaS services. This scenario presents the basic question which shall be answered by the ScaleDL Usage Evolution. Input from the industrial partners ENT and SAP was invaluable in the formulation of this scenario. The overall usage evolution scenario is:

I anticipate a change in work, load or quality threshold somewhere in the future. What will the consequences be?

We will now detail what this means, and we in addition to CloudStore from D5.2 we will use the EHR (Electronic Health Record) case study from ENT in D4.2 for illustration.

Concerning the change itself this may have many forms:

Peaks in load because of bird flu or migration to a larger market, for example moving an EHR application from Croatia to a larger market in Germany or even in Pakistan.

A gradual increase in load resulting from a steady increase in the number of consumers.

Lower load because of summer vacation or week ends.

Increase in data sizes representing more work as described in Section 2.1.2. In the context of CloudStore, typical data is books and customers, so this will then mean an increasing number of either books, customers or both. For EHR the average size of health records may increase for example because of self administered heart measurements.

Harder quality thresholds for example because of increased competition.

New services which are either designed from scratch or redesigned.

Changes in key components inside the service.

Changes in the PaaS environment for example a new DBMS.
All of these except the latter three concerns the usage evolution. New services will be considered in the same way as existing services. Changes in the service itself has to do with variability described in Section 2.1.4. Changes in the PaaS layer concerns deployment and is described in Section 2.2.

When considering how to represent usage evolution it is of course also relevant to know what we want to know as a result of the scalability analysis. The most basic question to answer in the context of scalability is the following:

Are we able to handle this (given our SLA)?

This question is of course only relevant for an increase in load or work, or for tougher quality thresholds. Then next natural question is:

How much will it cost in terms cost of invoked underlaying services?

This question is also relevant for a decrease in load or work or for softer quality thresholds, because then it is important to know how much the cost will be reduced as a result of this decrease.

If we are not able to handle an increasing demand it is relevant when this point of SLA violation and under provisioning is reached. We may also formulate several questions concerning migration to the cloud, but this will not be considered now. Using case studies we are able to find out how detailed the usage evolution needs to the modelled to answer these questions appropriately.

### 7.2 Characterisation of Usage Evolution

Usage evolution specify scalability requirements by characterising how the workload of a service changes over time.

This section describes different aspects of how usage evolution can be characterised, before the we proceed to the meta-model for how usage evolution is modelled in ScaleDL in section 7.3 We first describe the starting point, which we simply term initial usage in Section 7.2.1. Afterwards, we elaborate the evolution in time for these profiles in Sec. 7.2.2 and more detailed characterisation of evolution in Sec. 7.2.3.

#### 7.2.1 Initial Usage

An application provides one or more operations as a service. For each of these operations, the concept of initial usage specifies:

**Operation parameters:** Operation parameters characterise the amount of data to be processed, stored or communicated. This amount of data may vary for each invocation of the operation, but it may also be constant for many invocations. Operation parameters can also provide an input for the process in which we define the work of each operation. If we apply this concept to the CloudStore, we can specify two operation parameters: (1) "number of books in the database" that stays constant and (2) "number of customers in the database" that varies during the service life cycle.

**Load:** For an open systems, load can be measured in terms of the arrival rate. An arrival rate is valid during a specified time interval. For a closed system, load can be characterised by the number of simultaneous customers. As it is a closed system, this number does not change.
over time, i.e., it stays constant. We model CloudStore as an open system and use a monthly time interval.

**Quality metric:** Each operation must have quality metrics that describe how it is evaluated. Average response time is a common quality metric.

**Quality threshold:** For each operation with a certain quality metric, one or more quality thresholds can be specified such that they identify the border between acceptable and non-acceptable quality, e.g., 3 seconds for an average response time quality metric.

Quality metrics and quality thresholds have been described in more detail in Section 2.1.3.

The initial usage of an SaaS application is specified by the domain expert (of the SaaS customer domain) together with the architect of the SaaS application. For the CloudStore operations there are three different initial usage scenarios: (1) "Browsing" where customers mainly rummage through the book shop, (2) "Ordering" where customers heavily order books, and (3) "Shopping" as a combination of the previous configurations.

### 7.2.2 Characterisation of Evolution

In this section we are describing how to capture the evolution of the initial usage in Section 7.2.1. This has parallels in teletraffic forecasting [Ive10]. To generalise from the scenario in Section 7.1, we are interested in two basic forms of usage evolution:

- **Stable** corresponding to fairly constant load, work and quality thresholds.
- **Gradual change** corresponding to a book-selling company that slowly but steadily becomes more and more popular and where both the load in terms of arrival rate as well as the number of books grow at a regular, but slow pace. We may also the a gradual decrease.

To model a gradual change we first need to know the starting point. A gradual change may basically be formulated in two ways:

- **Linear** where we need to know the rate of increase with respect to the starting point. As an example we may start with 1 000 consumers and then get 100 more consumers every month. After one year we then have 2 200 consumers.

- **Exponential** where the rate of increase is compared to the previous period and not to the starting point. For example so that we always increase the number of users by 10 % compared to the previous month. If we then start with 1 000 consumers, we will after one year reach more than 3 100 consumes.

While the simple case has only one type of gradual change, we may of course also have several types of gradual changes, for example first a fairly slow increase, but then a faster rate of increase, before we enter a mature marked which is stable (a typical S curve).

In both these evolution patterns (stable and gradual change), we may also have sudden changes or spikes, for example representing a book-selling service that must cope with a high load before Christmas. It may also represent a sudden increase in load for a service which is placed in a new, larger market. For these two cases the duration of the spike varies considerably. The duration of the Christmas shopping spike may be measured in weeks, whereas the change in marked in theory may last forever. In the case of the Christmas shopping spike we also have the frequency of once
every year, whereas frequency is not relevant for the new market case. Even if the concept of spike is vital for scalability, the importance of the concepts of duration and especially frequency is more unclear at the moment. This will be clearer when we get feedback from our case studies.

One may be interested in more than one spike, but this is a simple generalisation. While spikes are by nature a sudden increase in demand, we may also have a sudden decrease in demand. We have now discussed spikes in load, but we may of course also have sudden changes in work or in quality thresholds. While the notion of time is implicitly present in the discussion of spikes above (frequency and duration), we believe the absolute time of events happening is not relevant.

Especially for load, but also for work we face more or less variance. This variance may be modelled using statistical distributions. For modelling the typical case we may use the average value and/or the median value. A simple representation for variance capturers the following metrics in addition to the average and/or median value: typical minimum and typical maximum. Using the word "typical" we exclude exceptions which are modelled as spikes. More sophisticated ways of modelling variance are of course possible. On the other hand, we can also do this simpler, by only focusing on typical maximum, and simply ignore both average/median and typical minimum. Case studies will tell us what is the best abstraction level.

When modelling the future we may base ourself on historical data or educated guessing. Using historical data we can easily see evolution during a day, a week or during a year. For example, load for an internet book store may always be highest between 20 and 23 in the evening and there may be more load during Christmas. We may also see differences in the daily load on weekdays compared to the daily load during weekends. However, since we are interested in the general trend in addition to spikes, this detailed evolution is most likely not so interesting in our context.

7.2.3 Detailed Characterisation of Evolution

A more detailed characterisation of usage evolution considers repeated variations. There can be variations in different time scales like daily, weekly, monthly, and yearly:

**Daily** variation during a single day, for example more load when people come to work, or more load during the lunch break, or just before they go to bed.

**Weekly** variations during a typical week, where for example there is less load or more load during week ends, or where the load on Monday is higher than all other days, but where the load from Tuesday onwards are gradually increasing.

**Monthly** variations, for example because of pay days.

**Yearly** variations for the weeks of a typical year where there may be less load during the summer vacation and gradually increasing load before Christmas.

Load for a typical day will then be characterised by the sum of all these variations on different time scales, so that the load late in the Sunday evening just before Christmas will be high for a typical Internet book store.

In addition to the time scales above you also have two other variations:

**Trend** variations across many years, but also for days, weeks or months.

**Noise** variation which do not fit into any of the patterns above, and which is not understood.
The specification of usage evolution must be flexible. In some cases it is relevant with several time scales to really understand the usage evolution. In this case the level of noise, may be quite low. However, a simpler characterisation may only focus on trend variations, but then with a higher level of noise.

### 7.3 Meta-model

To realize the concepts that have been described in the previous sub-sections of this chapter, we have selected an approach where we combine reuse of existing meta-models with a thin ScaleDL Usage Evolution meta-model that focuses on the integration. The meta-models we reuse are:

- The Palladio Component Model (PCM) including its usage model. The PCM is the foundation of the Palladio software architecture simulator.
- The Descartes Load Intensity Meta-Model (DLIM) which is used by the load intensity modelling tool LIMBO [vKHK14].

In more detail, the different concepts from the previous sub-sections of this chapter map to the different meta-models in the following way:

**Operational parameters and load of initial usage** are described in terms of the regular Palladio usage model.

**Quality metrics and quality threshold of initial usage** are described using the new MetricSpecification meta-model and the new ServiceLevelObjective meta-model of Palladio.

**Evolution of individual attributes** are described using DLIM [vKHK14] models. DLIM covers description of stable and gradual change, and supports linear and exponential trends. In addition, DLIM also support logarithmic trends and sine curves. Furthermore, DLIM supports peaks, noise, and different time scales.

**ScaleDL Usage Evolution** provides a small meta-model that maps how a set of DLIM models describe the evolution of load and work in the Palladio models.

Figure 7.1 shows the current meta-model for ScaleDL Usage Evolution. Elements imported from the Palladio Component Model (PCM) and DLIM are shown in light gray in the figure. The meta-model is defined to allow specification of how the usage evolves over time. The root element of a ScaleDL Usage Evolution model is the UsageEvolution element. A UsageEvolution can contain an ordered list of one or more Usage elements. Each Usage contains a reference to one UsageScenario from a PCM model, and this can be seen as the initial configuration. In addition, each Usage can contain a reference to a Sequence element from a DLIM model that describes the evolution of the arrival rates in the case of open workload, and population in the case of closed workload. A Usage can also contain a set of WorkParameterEvolution objects that each describe how a work parameter of the Palladio model evolves in term of a DLIM model.

The focus of Descartes Load Intensity Meta-Model (DLIM) is to support modelling of the intensity profile of open workloads, in terms of request arrival rates over time. We have, however, found that the same meta-model is well suited to also describe the evolution of other attributes, such as the population in closed workload and operational parameters such as data size. The root element of a DLIM model is a Sequence, which can hold one or more TimeDependentFunctionContainers. Each such container holds a Function, and there are many different realizations of Function including
Trend, Seasonal, Burst and Noise. Functions can also be combined with other functions through its list of combinators that can have a add, mult, or sub operator, and a TimeDependentFunctionContainer can in this way be the root of a tree of functions that are combined. For further details about the DLIM meta-model, see [vKHK14].

7.4 Relation to CloudML

The FP7 projects MODAClouds’ and PaaSage’s Cloud Modelling Framework (CloudMF) [FRC+13, FCR+13] consists of the Cloud Modelling Language (CloudML) for specifying the provisioning and deployment of multi-cloud systems at design-time, as well as a models@run-time engine for enacting the provisioning, deployment, and adaptation of these systems at run-time. CloudMF facilitates handling short-term workload evolution in a reactive way.

CloudSCALE usage evolution can be used for long-term workload evolution. To improve the management of workload evolution, short-term and long-term evolution must be combined. In a joint paper by MODAClouds, PaaSage and CloudSCALE, [FBR+14] describes an intermediary layer for handling mid-term adaptations that bridges the gap between the other two layers. A motivation example based on SensApp [sen] is provided. SensAPP is an open-source, service-oriented application for registering sensors, storing their data, and notifying clients when new data are pushed. The motivation example focuses on the following unpredictable variations:

Snow storm and cold wave The workload has an irregular peak due to a snow storm and cold wave, such as the one in Europe on 27 January 2012, which led to the disruption of European air and surface traffic for two weeks. The short-term layer reports the handling of the unpredictable workload to the mid-term layer, which analyses it and concludes it is an acceptable deviation from the prediction and hence does not report it to the long-term layer. This is because during winter one can anticipate that snow storms may occur and their average impact.

Volcanic eruption The workload has an irregular peak due to a volcanic eruption, such as the one of the Eyjafjallajökull volcano on 14 April 2010, which led to the stop of all European air traffic and increase of surface traffic for one week. The short-term layer reports the handling...
of the unpredictable workload to the mid-term layer, which analyses it and concludes it is an unacceptable deviation from the prediction and hence reports it to the long-term layer, which in turn alters the prediction. This is because, for instance, the predicted load in the usage evolution profile is two requests per second, while the current load is four. In addition, the load has been twice the predicted also during the two previous hours. Therefore, it may be reasonable to assume that the load will be twice the predicted also during the next hours, so the usage evolution profile for the next hours may be changed accordingly.

7.5 Relation to MARTE

MARTE (Modeling and Analysis of Real-time and Embedded systems) is a UML profile for modelling of real time and embedded systems. The first version of the MARTE standard was released by OMG in 2009, and was last revised to version 1.1 [OMG11] in June 2011.

As MARTE is defined as a UML profile, it is based on the UML meta-model and it is meant to be used by stereotyping UML modelling elements. For ScaleDL, we have chosen to build on the Palladio tool because it gives a very good foundation for performing performance analysis of architecture models and a good foundation for extending this with scalability analysis. The Palladio component model (PCM) which is the foundation of Palladio, is built directly on the eCore meta-model of EMF and not on UML, and the same is true for additional meta-models to be used with Palladio (such as ScaleDL). This means that UML profiles can not directly be applied to Palladio models, and Palladio and ScaleDL modelling elements cannot directly be applied to UML models nor directly be combined with UML profiles such as MARTE.

Despite these restrictions, MARTE can be of interest to CloudScale at the conceptual level, and also mappings (for manual use or automatic transformations) could be defined between MARTE and Palladio. MARTE defines a set of conceptual models, and maps these models to different sub-packages of the UML profile. For ScaleDL Usage Evolution, the Non-functional Properties Modeling (NFPs) (chapter 8 of [OMG11]) is the most relevant part. This part defines UML stereotypes for concepts such as Unit, Dimension, Nfp (Non Functional Property), and NfpConstraint. For the most corresponding parts in ScaleDL, we are using the Metrics and ServiceLevelObjective packages that have recently been defined for Palladio. One of the modelling concepts defined in the Metrics package is EJSUnit, which is defined by means of the javax.measure.unit.Unit class from JScience. JScience is a comprehensive Java library for scientific computing (available at http://jscience.org). The definitions of Unit and Dimension from this library is similar to the definitions of MARTE. The ServiceLevelObjective (from the corresponding package) can contain an upper and lower Threshold which plays a role similar to (but more limited than) the NfpConstraints in MARTE.

To our understanding, MARTE does not currently have any support for modelling load, work and evolution aspects as supported by ScaleDL Usage Evolution and DLIM. A possible mapping of these elements from ScaleDL to UML and MARTE will be considered in future work (see Section 7.7).

7.6 Relation to SLA@SOI

SLA@SOI (2008-2011) was an FP7 IP that aimed at delivering a comprehensive framework for SLA management (including performance and reliability characteristics) for the whole life-cycle of service-oriented software systems. The main results of SLA@SOI were:

- SLA*, a model of SLA which provides a domain independent, abstract syntax for describing service level agreements in a machine-readable form.
• an reference SLA management architecture

Of these, the SLA* is most relevant for our work. The top level concept in the SLA* model is an SLA template that describes the agreement parties involved, the functional interface offered and required, and agreement terms. The latter includes preconditions under which the terms are effective, and includes guarantied states and actions that will be performed under specific circumstances.

For UsageEvolution, our focus is on the Service Level Objectives (SLO) rather than the full SLA, and thus much of the SLA* model is not relevant for our purpose. Also, declarations of interfaces, operations and properties are already provided by the PCM. The concepts of guarantied state is similar to what have been realised with the new ServiceLevelObjective meta-model that is now in incubation status for Palladio.

The site publishing the open source implementation of SLA@SOI has not posted any updated versions since 2011, so with respect to building on existing communities it is more relevant for Cloud-Scale to focus on Palladio and build on the foundation that it provides.

In conclusion, we do plan to directly reuse the development artefacts from SLA@SOI or the SLA* model, but we will continue to consult the SLA* model as related work when extending our meta-models.

7.7 Future Work on ScaleDL Usage Evolution

Work on ScaleDL Usage Evolution will continue in the third and final project year, where we will look into the following issues:

Use of quality thresholds: Currently, the ScaleDL Usage Evolution meta-model does not include any use of quality thresholds. A metrics meta-model has been developed, and is currently used to specify (in a separate model) what measurements should be taken when Analyser runs an experiment based on a ScaleDL Usage Evolution model. This approach will be extended to also include quality thresholds in the experiments, either by adding the thresholds to the ScaleDL Usage Evolution model or to an additional model used by Analyser. An open question that determines which solution to choose, is whether we need support for thresholds that evolve over time, or to compare two experiment runs using different thresholds.

Use of ScaleDL Usage Evolution to describe evolution of ScaleDL Overview: Currently the ScaleDL Usage Evolution meta-model was designed primarily to make it easy to map how evolution of attributes described in DLIM models map to PCM models. It may in some cases be useful to describe evolution aspects related to ScaleDL Overview, and then to also include this evolution information when a PCM model are derived from the ScaleDL Overview. This may require some adjustments on ScaleDL Usage Evolution.

Relation to CloudML: Currently the integration with CloudML is only concepts, but when usage evolution now is more mature we may of course explore this also in practice resulting in a closer cooperation between the three EU-projects CloudScale, MODAClouds and PaaSage.

Relation to MARTE: We will in the final period of CloudScale consider mapping of parts of ScaleDL Usage Evolution to MARTE. The modelling support for this is most relevant if tools exist that can be extended to utilise the models in analysis of scalability. Possible mappings could consist of defining a UML profile for ScaleDL, and defining transformations between the Palladio and UML-based models. The costs vs benefits of developing such mappings will be considered when deciding on priorities of future work.
8 HowTos: Design Recommendations and Patterns

One of CloudScale’s goals is to provide developers and architects a set of HowTos (design recommendations and/or design patterns), e.g., for building cloud scaling applications. Since project year two, we list and describe several concrete HowTos in the CloudScale wiki. Moreover, we now provide this new chapter to present the rationale behind the the selection process of particular HowTos within our wiki. For multi-tenancy HowTos, we further summaries our contributions going beyond state of the art.

Table 8.1 overviews the HowTos in the wiki and gives some further context information. The table gives the main quality goal of a HowTo as well as the name of the HowTo. It further describes whether a HowTo was a development of CloudScale or whether it was already described in literature before. The publication state describes if the HowTo was already published from CloudScale in any context of a peer reviewed conference (only feasible for HowTos developed by CloudScale). The field AT is used to show which HowTos we formalized using ScaleDL’s AT language. The latter is especially important to make direct use of HowTos in ScaleDL editors as well as for the Analyzer. Chapter 9 about Architectural Templates describes these issues in detail.

The remainder of this chapter is structured along the main quality goals of our HowTos (first column of Table 8.1). We provide a dedicated subsection for each quality goal describing the rationale for the selection of associated HowTos. Because we developed new HowTos for multi-tenancy, we also describe results that go beyond state of the art in the corresponding subsection.

8.1 Scalability HowTos

Based on initial literature surveys, we identified important HowTos for scalability. A generally rich set of literature provides, classifies, and surveys sets of HowTos in the form of architectural styles (e.g., [BMR+96], [SC97]). However, these styles lack an explicit consideration of cloud computing environments, thus, we did not consider them further.

In the context of cloud computing and scalability, typically applied styles are REST [FT00] for HTTP and SPIAR [MvD08] for AJAX. These styles are included within the SPOSAD style, therefore, we did not explicitly add them to our list of HowTos.

In their book, Erl et al. [EPM13] describe a set of cloud computing patterns that foster scalability. A similar catalogue of patterns is provided in the book of Fehling et al. [FLR’14]. A core concept in both of these books is the load balancing pattern as it is a prerequisite for several other patterns (e.g., for the “dynamic horizontal scaling” pattern for elasticity). To acknowledge this relevance, we explicitly added the loadbalancing HowTo to our list.

Amazon’s AWS cloud design patterns catalogue¹ also provides a good source for designing cloud computing applications. We investigated which of these patterns foster scalability, identified “static content” and “sharding” as such patterns, and added these to our HowTos list.

The MapReduce programming model [DG08] involves a design recommendation that is particularly important for processing a huge amount of data in a scalable manner. We added MapReduce as a HowTo to our list as it is highly practically relevant and often used to achieve scalability.

¹http://en.clouddesigntool.org/
### Table 8.1: Overview of Identified HowTos

<table>
<thead>
<tr>
<th>Main Goal</th>
<th>Name</th>
<th>CS Dev.</th>
<th>Publication State</th>
<th>AT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>Loadbalancing</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>Architecture utilizing load balancers.</td>
</tr>
<tr>
<td>Scalability</td>
<td>Static Content</td>
<td>No</td>
<td>-</td>
<td>No</td>
<td>Specific purpose deployment to handle static content</td>
</tr>
<tr>
<td>Scalability</td>
<td>Sharding</td>
<td>No</td>
<td>-</td>
<td>No</td>
<td>Separating data into storage partitions.</td>
</tr>
<tr>
<td>Scalability</td>
<td>MapReduce</td>
<td>No</td>
<td>-</td>
<td>No</td>
<td>Process large data sets by distributing processing nodes.</td>
</tr>
<tr>
<td>Elasticity</td>
<td>Dynamic Horizontal Scaling</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>Architecture utilizing horizontal scaling.</td>
</tr>
<tr>
<td>Elasticity</td>
<td>Dynamic Vertical Scaling</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>Architecture utilizing vertical scaling.</td>
</tr>
<tr>
<td>Elasticity</td>
<td>Simplified SPOSAD</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>3-layer architecture that requires scalable middle and data layers.</td>
</tr>
<tr>
<td>Multi-tenancy</td>
<td>SPOSAD</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>3-layer architecture that requires scalable middle, data layers, and multi-tenancy.</td>
</tr>
<tr>
<td>Multi-tenancy</td>
<td>Request Admission</td>
<td>Yes</td>
<td>conference</td>
<td>No</td>
<td>A method to enable tenant individual scalability.</td>
</tr>
<tr>
<td>Multi-tenancy</td>
<td>Request Resource Control</td>
<td>Yes</td>
<td>conference</td>
<td>No</td>
<td>A method to guarantee each tenant a defined portion or resources.</td>
</tr>
<tr>
<td>Multi-tenancy</td>
<td>Isolation Architecture</td>
<td>Yes</td>
<td>conference</td>
<td>No</td>
<td>Architecture to build tenant specific scalability.</td>
</tr>
<tr>
<td>Multi-tenancy</td>
<td>One Table/Schema/DB per Tenant</td>
<td>No</td>
<td>-</td>
<td>No</td>
<td>For each tenant a coarse grained persistence entity is used.</td>
</tr>
</tbody>
</table>
8.2 Elasticity HowTos

The books of Erl et al. [EPM13] and Fehling et al. [FLR+14] also go beyond scalability by describing elasticity patterns (we see scalability as a prerequisite of elasticity). In many of their elasticity patterns, the scalability pattern "loadbalancing" is integrated. However, instead of using a static load balancing strategy, i.e., with a fixed set of loadbalanced components, elasticity patterns dynamically vary this set of components based on current workload. Being the most prominent pattern of this category, we explicitly added the elasticity HowTo "dynamic horizontal scaling" to our list. We also added the elasticity HowTo "dynamic vertical scaling" that represents the main alternative to loadbalancing.

The SPOSAD [Koz11] architectural style itself targets multi-tenancy and is, therefore, categorized as multi-tenancy HowTo. However, we also added a simplified version of SPOSAD to our list of HowTos. This simplified version of SPOSAD only covers the elasticity aspects of SPOSAD, not the multi-tenancy part. Moreover, it is based on the "dynamic horizontal scaling" HowTo but additionally covers the data layer of software applications. For a smooth transition between "SPOSAD" and "dynamic horizontal scaling", we therefore explicitly added the "simplified SPOSAD" HowTo for elasticity to our list. This HowTo is especially useful to showcase some of our results in a simplified manner (for instance, we use simplified SPOSAD to describe some aspects of Architectural Templates in Chap. 9).

8.3 Multi-Tenancy HowTos

In this section, we describe the rationale behind our efforts in the field of multi-tenancy related HowTos and the contributions already made.

8.3.1 Multi-tenant Applications

Several publications discuss how a Multi-tenant application can be implemented. These papers usually focus on security, privacy and safety aspects. In a first step CloudScale identified existing patterns describing how to build the application layer and the overall architecture of a multi-tenant application. In a second step the identification of patterns for the persistence were evaluated. As aforementioned there are already several publications discussing these issues. Therefore, our primary focus and efforts are not related to this area. Until now, the project identified relevant patterns. The SPOSAD [Koz11] architectural style including multi-tenancy aspects and the shared table approach [WGG+08] were identified as best practices and they are described in the HowTo on the CloudScale wiki to increase the public visibility.

8.3.2 Scalability for Individual Tenants

Every tenant of the application has its own isolated view onto the application. This view includes the data the tenant can access, the configuration of the application and non-functional aspects like the performance behavior. A tenants demand might increase over time. Furthermore, the willingness of tenants to pay for a better performance to compensate the changing demand might be different. Consequently, a provider of a multi-tenant application must be able to provide different tenants different performance and adapt the performance a tenant observes, if the tenant is willed to pay for the additional resources he consumes. If a tenant is not willing to pay an additional fee, for its increasing
demand, the provider has to ensure the performance of the other tenants is not influenced. Compared to the work discussing how to build a multi-tenant application the work regarding to a tenant individual scalability of the application is still an open research question. As a precondition for a tenant oriented scalability is the opportunity to performance isolate tenants from each other. Once this goal is achieved, it should be easily possible to provide different tenants a different share of the system.

This is a challenge, because the layers of the execution environment, responsible for controlling resource usage (e.g., operating system), normally do not have knowledge about entities defined at the application level and thus they cannot distinguish between different tenants. This problem is called layer discrepancy. The tight coupling of tenant's results in strong interference, and the layer discrepancy makes scaling on a tenants basis a challenge. Isolation considering non-functional system properties is a major open research issue in the area of SaaS (e.g., by Bezemer [BZP+10] and Fehling [FLM10]). Therefore, CloudScales primary focus is on this topic.

Various approaches to ensure performance isolation in MTAs were already discussed in the literature. CloudScaled focused on approaches which apply a request based admission control. These methods delay, or reject requests from a certain tenant before they are processed by the application server. Thus, it is possible to influence the performance for each tenant and to reduce the impact of one tenant's requests. By less delaying a tenants it could use a larger share of the application and thus scale up. CloudScale found 5 generic methods which can be used to establish such an admission control and estimate their capabilities to ensure isolation. Further, we described the concrete information requirements that have to be fulfilled to realize the different approaches. This helps developers who aim for performance isolation to find the most suitable approach for their scenario. These results were published in [KL14] to a brought community and will become part of the CloudScalesproject wiki to increase visibility of these methods.

To guarantee a tenant an individual performance it is essential to control the resources used by a tenant. It is hard to predict how tenant requests propagate through the multiple layers of the execution environment down to the physical resource layer. The intended abstraction of the application from the resource controlling layers does not allow to solely solving this problem in the application. Thus a real control of the amount of shared resources is difficult and most existing approaches cannot do this. In CloudScale an approach was developed which applies resource demand estimation techniques in combination with a request based admission control. The resource demand estimation is used to determine resource consumption information for individual requests. The admission control mechanism uses this knowledge to delay requests originating from tenants that exceed their allocated resource share. The proposed method was validated with a multi-tenant version of the CloudStore showcase based on the TPC-W benchmark [KSAK14]. In case the Multi-tenant Application is hosted on a Platform-as-a-Service (PaaS) and the developer wants to control performance-related issues according to individual tenant, the available information is often limited and the control of the request flow is restricted. Thus, it is difficult to control the performance of different tenants to keep them isolated. To overcome this issue, the previous results were used to propose a concrete PaaS enhancement which enables application developers to realize methods to control a tenants performance for their hosted SaaS application. In a case study we evaluated the applicability and effectiveness of the enhancement in different environments [KLK14].

8.4 Future Work

More HowTos and ScaleDL Architectural Templates will be added and the summary about the existing one will be adjusted according to the new state. We especially want to add HowTos and ScaleDL Architectural Templates for engineering elastic and efficient cloud computing applications.
The chronologically next steps are the more detailed presentation of this years identified patterns and results at the CloudScale project wiki to make the results public available for a larger community. This also includes a summary of the various isolation methods pros and cons.
9 ScaleDL Architectural Template

In this chapter, we describe the concepts and the realization of the ScaleDL Architectural Template (AT) language – a sublanguage of ScaleDL. Since our previous deliverable (D1.1), we did not alter these conceptual descriptions. However, note that we worked on integrating ATs into the Analysers (see D3.2) and now provide a rationale and overview of selected HowTos in Chap. 8. For some of these HowTos we now also provide concrete AT implementations as indicated in Table 8.1 (which were not yet realized in project year 1). The remainder of this chapter now reports on AT concepts as where already present in project year two; there where only minor updates.

We define the AT language as follows: the ScaleDL Architectural Template (AT) language is a language to formalize architectural styles on component models. In particular, this formalization allows to enrich styles by quality annotations and completions for model-driven quality analyses. Quality annotations characterize a concrete quality property of interest. Quality completions utilize these annotations to derive quality models that can be used as input to quality analysis tools.

Because we focus on SaaS applications, we enrich ATs by scalability, elasticity, and efficiency annotations and completions that enable architects to conduct according analyses. These analyses are supported by our Analysers tool (cf., Chap. 10).

The AT language distinguishes between AT types and AT instances. With AT types, AT engineers specify the formalization of an architectural style as well as its annotations and completions, i.e., they create an AT. In contrast, AT instances allow software architects to apply an AT to their component-based architecture. This application allows software architects to design and analyze their architecture for scalability.

This chapter is structured as follows. First, Sec. 9.1 introduces a brief example scenario to motivate and illustrate the AT language. Based on this scenario, we describe the processes of AT creation and AT application in Sec. 9.2. Section 9.3 describes the abstract syntax of the AT language, i.e., its metamodel for AT types and AT instances, respectively. Afterwards, we give the concrete, graphical syntax of the AT language (for both, types and instances) in Sec. 9.4. In Sec. 9.5, we briefly list concrete ATs we created and link to the location in the CloudScale Wiki where these ATs are described in detail. Section 9.6 explains the naming “architectural template” based on related terms in software architecture. In particular, the latter section explains why we renamed the proposed “scalability best practices” and “scalability patterns” of our DoW to ATs. Finally, we describe open issues and corresponding future plans regarding the AT language in Sec. 9.7.

9.1 Example Scenario for Architectural Templates

As an example scenario, we consider a simplified book shop that shall be newly designed for running in a cloud computing environment. In this scenario, an enterprise assigns a software architect to design the book shop. The enterprise has the following requirements for this shop:

R1: Functionality In the shop, customers can (1) browse and (2) order books.

R2: Handling of Environmental Changes The enterprise expects that the environment for the book shop changes over time. For example, it expects that books sell better around Christmas while they sell worse around the holiday season in summer. Therefore, the response times of the shop shall stay within 3 seconds even if the customer arrival rates increase or decrease by 1,000
customers per hour (at maximum).

**R3: Linear Cost Scalability** The costs for operating the book shop are only allowed to increase (decrease) by $0.01 per hour when the number of customers using the shop increases (decreases) by 1 per hour. *Costs per hour* is a metric to measure the amount of additional services (cf., the scalability definition in Sec. 2.3).

Requirements R2 and R3 are typical reasons to operate a system in an elastic cloud computing environment, i.e., an environment where application servers automatically provision the required amount of services to cope with environmental changes. Therefore, the software architect will design the shop as an SaaS application operating in a rented cloud computing environment. To provide a *scalable* SaaS application (R3), the architect considers using (1) architectural styles as well as (2) scalability analyses.

The first option (architectural styles) requires that the architect is aware of appropriate cloud-based architectural styles as well as their application and assumptions. Currently, only a few architectural styles for SaaS applications in the cloud exist [EPM13]. One example for such a style is SPOSAD [Koz11].

SPOSAD suggests a 3-tier system with stateless middle tier, but leaves open the decision for a concrete database paradigm on the data tier [Koz11]. Therefore, the architect may design the book shop as shown in Fig. 9.1: he assigns the three SPOSAD tiers (presentation, middle, data) to the corresponding components (Book Shop Frontend, Book Management, Book Database), respectively. The tiers of SPOSAD can be seen as its roles, i.e., as set of constraints for associated components. For example, the data tier may only allow connections from the middle tier. As SPOSAD does not constrain the data tier further, the architect is unsure whether to design the system with a relational or a NoSQL database that both promise to scale differently. Because of this variability point (or parameter), he needs further guidance; SPOSAD alone is not enough. For achieving particular scalability requirements, there is, hence, the need to refine SPOSAD. Also the few other SaaS styles lack detailed suggestions for selecting an appropriate database paradigm.

![Figure 9.1: Model of the designed book shop scenario according to a simplified SPOSAD variant](image)

The second option (scalability analyses) would allow the architect to model both database alternatives and to compare their scalability (what-if analysis). Scalability analyses require simulation or analytical models allowing architects to predict an SaaS applications’ scalability in cloud computing environments. However, according to Becker et al. [BLB12], current analysis models lack (1) support for comprehensive design-time analyses based on architectural models, (2) realistic case studies in cloud computing environments, and (3) explicit support for scalability because of their focus on performance analysis. These lacks hinder the architect to use and trust existing approaches in order to analyze the scalability of the modeled book shop. For instance, these approaches are unable to analyze whether a NoSQL or a relational database suits the enterprises’ scalability requirements best. Hence, these approaches need to be extended and improved.
The lack of guidance (regarding styles and scalability analyses) for the architect leads to the high risk of realizing a cost-inefficient SaaS application and of an expensive re-implementation. For example, it may be expensive to refactor the book shop with an established but non-scalable relational database implementation to an implementation using a NoSQL database. This risk becomes even more crucial in case the enterprise discovers scalability issues when high costs for hosting their application have already incurred, e.g., during system operation.

9.2 Architectural Template Processes

To cope with the lack of guidance for software architects (cf., Sec. 9.1), we propose the AT language. Software architects can apply ATs (specified in the AT language) to design systems as well as to analyze their scalability. AT engineers, on the other hand, create these ATs via the AT language. We guide through these processes of AT application (Sec. 9.2.1) and AT creation (Sec. 9.2.2) by using the book shop scenario of Sec. 9.1.

9.2.1 Applying a SPOSAD Architectural Template

Software architects apply ATs to design SaaS applications and to analyze their scalability. Fig. 9.2 illustrates the process steps for applying ATs. In step 1, architects select an AT from a repository of ATs. For example, the architect of the book shop scenario selects a SPOSAD AT. Afterwards, the architect assigns all roles the AT requires to the components of his architecture (step 2).

As illustrated in Fig. 9.1, he may assign the roles of the SPOSAD architectural style (presentation, application, and data tier role) to book shop components as well as connects these components appropriately. Firstly, the AT formalism assures that no constraints are violated, e.g., it forbids connecting presentation and data tier components. Secondly, the AT formalism allows architects to specify whether a data tier component corresponds to a relational or a NoSQL database (a parameter characterizing a variation point).

Because this design is based on an AT, the architect can run scalability analyses afterwards (step 3). The key idea is that the AT includes scalability annotations and completions for this purpose. For the book shop, the specification of the database parameter corresponds to a concrete scalability annotation. SPOSAD AT’s scalability completion then ensures that the architect obtains results that accurately reflect the scalability of the selected database kind. Therefore, the SPOSAD AT allows the architect to analyze which kind of database better fits his scalability requirements at design time.

9.2.2 Creating the Architectural Template for SPOSAD

To create ATs, AT engineers apply the process illustrated in Fig. 9.3. In the first step, they formalize AT roles (e.g., SPOSAD’s data tier) based on architectural styles (e.g., known from architecture handbooks).

Besides formalizing architectural styles, AT engineers enrich the AT with scalability annotations and completions to enable an automated scalability analysis. The basis for this analysis is an accurate
A scalability model that can be used by scalability analysis tools. Therefore, AT engineers have to specify and validate such a scalability model based on identified and quantified scalability parameters. Once AT engineers have created an accurate scalability model, they can enrich the AT with scalability annotations and completions allowing to automatically derive the scalability model from component-based architectures specified with the AT.

Accordingly, AT engineers first have to identify scalability-relevant parameters and quantify them (step 2). For example, the kind of database (e.g., a relational or a NoSQL database) on the data tier could be a scalability-relevant parameter because NoSQL databases are often designed for scaling horizontally. To confirm and quantify the influence on the scalability of this parameter empirically, an AT engineer (1) implements a series of automated test-drivers (focusing on the database parameter) that systematically collect the necessary data based on scalability metrics and (2) runs these test-drivers in a cloud computing environment. The AT engineer can subsequently check and quantify the influence of the "kind of database" parameter by means of a regression analysis.

In case AT engineers successfully identified a scalability parameter, they specify a scalability model for this parameter (step 3). This scalability model particularly integrates the quantified results for the considered cloud computing environment. For the book shop example, the AT engineer has, e.g., to specify a scalability model for NoSQL databases. Because NoSQL databases can scale horizontally, an accurate scalability model should model this NoSQL database as a "simple database" component that is attached to a "load balancer" component. As soon as load exceeds a certain threshold, an additional adaptation rule assures that another "simple database" component is served by an additional service and added to load balancing. To determine concrete thresholds of the load balancer as well as processing times of the "simple database" component, the AT engineer integrates the results of the regression analysis. The scalability of this model can, finally, be analyzed by ordinary analysis tools supporting simple components annotated with processing times as well as adaptation rules, e.g., the PCM.

In the fourth step of Fig. 9.3, AT engineers validate the scalability model. For this validation, they use a set of case studies that particularly include identified scalability parameters. The book shop scenario can, for instance, be seen a possible case study. For each case study, AT engineers (1) measure its scalability in the considered cloud computing environment and (2) predict its scalability based on the scalability model. In case the predictions accurately reflect the scalability of the measurements, the AT engineers successfully validated the model. Otherwise, they have to iterate the process by refining the automated test-drivers or by identifying and integrating further scalability parameters.

In the final step of Fig. 9.3, AT engineers enrich the AT with scalability annotations and completions. For example, the "NoSQL database" component assigned to the "data tier" role could be transformed to the "simple database" and "load balancer" components as described above. Accordingly, the specification of a "NoSQL database" for the "kind of database" parameter corresponds to a scalability annotation. The transformation to the "simple database" and "load balancer" components corresponds to a scalability completion. Therefore, the scalability of this model can, finally, be analyzed by the Analyser (which lacks support for NoSQL databases but can cope with the PCM models resulting from the scalability completion).

We based the process of Fig. 9.3 on established processes in performance engineering [HFBR08].
The basic idea is that concepts for performance engineering (e.g., performance model specification and validation) apply similarly on scalability as well.

9.3 Abstract Syntax: AT Metamodel

In this section, we describe the metamodel of the AT language, i.e., its abstract syntax. The main AT language package is a subpackage of ScaleDL and includes two subpackages on its own: a type and an instance package. The type package (Sec. 9.3.1) allows AT engineers to specify a new or alter an existing AT. The instance package (Sec. 9.3.2) allows software architects to apply a specified AT on a component-based architecture, e.g., on a PCM model instance.

Note that we describe the latter two processes (AT creation and application) in Sec. 9.2. Further note that the metamodel presented in this section is a preliminary version. We plan to refine and extend this metamodel within project year two.

9.3.1 Type Package

Fig. 9.4 illustrates the type package of the AT language. This package formalizes ATs, thus, defining their syntax and semantics. AT engineers can use this formalization to specify ATs and to organize them in repositories.

The meta class Entity describes elements that have (1) a name and (2) a unique identifier. As both attributes are relevant for several meta classes, these meta classes inherit from Entity.

The meta class Repository represents a set of ATs. For instance, our AT repository (Sec. 9.5) provides the set of ATs we collected and specified for engineering scalable cloud computing applications. The description meta attribute allows AT engineers to document details about the repository (in natural language).

The meta class ArchitecturalTemplate represents an AT. The description meta attribute allows AT engineers to document details about the AT (in natural language). Most notably, ArchitecturalTemplate inherits from the Role meta class and can, therefore, include a set of sub roles. The reason for modeling ATs like this is that we want to allow engineers to compose ATs out of other ATs, similar to roles as described next.

The Role meta class is the central meta class for AT types. A role can include a set of sub roles. This allows AT engineers to create role hierarchies, e.g., a layered architecture within an application tier of a 3-tier architecture. Because ArchitecturalTemplate inherits from Role, it is, in particular, also possible to reuse, e.g., a pipes-and-filters AT on the middle tier of a 3-tier AT.

Furthermore, AT engineers can characterize a role by means of a set of constraints (abstract meta class Constraint). Engineers specify these constraints via OCL invariants (meta class OCLExpression). For example, a constraint for a 3-tier architecture is that presentation tier components cannot directly communicate with data tier components. The latter example is a constraint of type ConnectorConstraint, i.e., it constraints connectors that have their source and/or target at a component associated to the considered role. Similarly, a ComponentConstraint directly constrains role’s associated components. For example, a component may only provide services and does not require other components’ services. An AllocationConstraint could, e.g., require that all components associated to a role have to be by the same service.

Besides constraints, a role can have a set of parameters (meta class Parameter). Parameters characterize variability points of a role. For example, a “data tier” role might have a parameter for the
database kind that has to be supported by at least one component. Two possible kinds of databases are "relation databases" and "NoSQL databases". These two possibilities can be manifested in an enumeration, thus, specifying the data type (meta class EDataType) of the parameter. Role parameters are especially interesting when they have an influence on quality properties of an application. For instance, the database kind (relational vs. NoSQL database) can influence the scalability of an application: typically, NoSQL databases are designed to scale horizontally while relational databases rely on single-node implementations.

The outstanding property of ATs is that they allow software architects to analyze the scalability of their applications at design time. To enable this analysis, AT engineers enrich their roles by scalability completions (meta class Completion). Scalability completions are model-to-model transformations allowing to transform a PCM model with ATs into a PCM model without ATs, thus, enabling an ordinary PCM analysis. In particular, this transformation completes the output PCM model with scalability-relevant information (hence the name "completion"; cf. Sec.9.2.1). The scalability of this model can, finally, be analyzed by the Analyzer (which lacks support for a NoSQL database but can cope with the PCM model resulting from the scalability completion).

### 9.3.2 Instance Package

Fig. 9.5 illustrates the instance package of the AT language. This package allows software architects to instantiate ATs (i.e., to apply them). For this instantiation, architects assign component assemblies the roles required by an AT.

The meta class ArchitecturalTemplateInstance contains mappings between assembled components (meta class Assembly Context) and roles (meta class Role). These mappings allow software architects to assign roles to components. There are two types of mappings: (1) from one component to its assigned roles (meta class Component2Role) and (2) from one role to its assigned components (meta class Role2Component). Given one of these types for all components or roles, respectively, the other mapping type can be derived. Furthermore, ArchitecturalTemplateInstance holds a link to the role acting as the type of the instantiated AT.
Figure 9.5: AT Language Metamodel: Instance Package
To assign AT parameters a value (i.e., to specify a scalability annotation), an ArchitecturalTemplate-Instance can also contain a set of parameterValues of type ParameterValue. These parameter values can be of type enumeration (EnumParameter), Integer (IntegerParameter), String (StringParameter), and Float (FloatParameter).

9.4 Concrete Syntax: AT Graphics

In this section, we describe a graphical, concrete syntax for the AT language. This syntax allows to specify type as well as instance models of the AT metamodel (cf., Sec. 9.3).

We describe the graphical syntax for types (type graphics) in Sec. 9.4.1. Afterwards, we continue with the graphical syntax for instances (instance graphics) in Sec. 9.4.2

9.4.1 Type Graphics

Type graphics visualize elements of the type package of the AT metamodel. In this section, we describe the notation for these elements. We describe the notations for (1) descriptions for repositories and ATs, (2) repositories, (3) ATs, and (4) roles.

Description Notation

ATs as well as their repositories can have a description attached for their documentation. As shown in Tab. 9.1, we follow UML comment syntax to visualize this description.

<table>
<thead>
<tr>
<th>Meta Class</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArchitecturalTemplate, Repository (Type Package)</td>
<td></td>
<td>Attaching an optional description (analogously to the UML comment syntax).</td>
</tr>
</tbody>
</table>

Table 9.1: Description Notation

Repository Notation

To visualize AT repositories, we follow UML package syntax (Tab. 9.2). The at symbol (@) is the logo of ATs, thus, making the difference to UML packages explicit. In addition, repositories usually have a description attached for their documentation (cf., Tab. 9.1).

<table>
<thead>
<tr>
<th>Meta Class</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository (Type Package)</td>
<td></td>
<td>A repository with name &quot;Repository Name&quot;. The at sign (@) is the logo for ATs.</td>
</tr>
</tbody>
</table>

Table 9.2: Repository Notation
Architectural Template Notation

As the ArchitecturalTemplate meta class inherits from the Role meta class, ATs can also be represented like roles (see the next paragraph). In addition, ATs usually have a description attached for their documentation (cf., Tab. 9.1).

Role Notation

In Tab. 9.3, we provide the notation for roles. To visualize these roles, we use a dashed box. The dashed box can include (1) the constraints and/or (2) the parameters of the role. Furthermore, roles can include sub roles. For clarity, sub roles may use a different background color.

<table>
<thead>
<tr>
<th>Meta Class</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role (Type Package)</td>
<td><img src="image" alt="Role" /></td>
</tr>
<tr>
<td></td>
<td>A role with name &quot;Role&quot;.</td>
</tr>
<tr>
<td>Constraint (Type Package)</td>
<td><img src="image" alt="Role" /></td>
</tr>
<tr>
<td></td>
<td>A role with constraints. Each constraint is a separate invariant (&quot;rule1&quot;, &quot;rule2&quot;, etc.). Each invariant is expressed via OCL.</td>
</tr>
<tr>
<td>Parameter (Type Package)</td>
<td><img src="image" alt="Role" /></td>
</tr>
<tr>
<td></td>
<td>A role with parameters &quot;Parameter Name&quot; of type &quot;Type&quot;, &quot;Parameter Name 2&quot; of type &quot;Type 2&quot;, etc.</td>
</tr>
<tr>
<td>Role (Type Package)</td>
<td><img src="image" alt="Role" /></td>
</tr>
<tr>
<td></td>
<td>A role with a sub role. Sub roles optionally use a different background color for clarity.</td>
</tr>
</tbody>
</table>

Table 9.3: Role Notation

9.4.2 Instance Graphics

Instance graphics visualize ATs similar to type graphics. Therefore, we only describe the differences that come with instance graphics. These differences relate to (1) the component and connector notation and (2) parameter instantiation. We describe both types of differences next.
Component and Connector Notation

In Tab. 9.4, we provide the notation for components and connectors within instance graphics. Components of instance graphics can get a role assigned to them. For this assignment, software architects put the assembled components of their architecture directly into the dashed role box. The dashed box can optionally show the constraints of the role.

Furthermore, architects can connect these component assemblies via connectors. The PCM allows for synchronous as well as asynchronous communication between components. For synchronous communication, components can provide and require a set of interfaces. Providing and requiring roles are connected via assembly connectors. For asynchronous communication, components can have a set of sinks and sources. Sinks and sources are connected via event channels that transfer messages send by sinks to sources.

<table>
<thead>
<tr>
<th>Meta Class</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Context (PCM)</td>
<td><img src="image" alt="Component" /></td>
<td>Component of an architecture.</td>
</tr>
<tr>
<td>Role2Component, Component2Role (Instance Package)</td>
<td><img src="image" alt="Role" /></td>
<td>Assigning a role to a component with hidden role constraints.</td>
</tr>
<tr>
<td>Role2Component, Component2Role (Instance Package)</td>
<td><img src="image" alt="Role" /></td>
<td>Assigning a role to a component with visible role constraints.</td>
</tr>
<tr>
<td>Operation Provided Role (PCM)</td>
<td><img src="image" alt="Providing Interface" /></td>
<td>Synchronous communication: providing an interface.</td>
</tr>
<tr>
<td>Operation Required Role (PCM)</td>
<td><img src="image" alt="Requiring Interface" /></td>
<td>Synchronous communication: requiring an interface.</td>
</tr>
<tr>
<td>Assembly Connector (PCM)</td>
<td><img src="image" alt="Assembly Connector" /></td>
<td>Synchronous communication: connecting requiring and providing roles.</td>
</tr>
<tr>
<td>Source Role (PCM)</td>
<td><img src="image" alt="Source Role" /></td>
<td>Asynchronous communication: message sender.</td>
</tr>
<tr>
<td>Sink Role (PCM)</td>
<td><img src="image" alt="Sink Role" /></td>
<td>Asynchronous communication: message receiver.</td>
</tr>
<tr>
<td>Event Channel Source Connector, Event Channel Sink Connector (PCM)</td>
<td><img src="image" alt="Event Channel Connector" /></td>
<td>Asynchronous communication: connecting source and sink roles.</td>
</tr>
</tbody>
</table>

Table 9.4: Component and Connector Notation

---

1. We follow the PCM notation to visualize components and connectors.
Parameter Instantiation

As AT roles can have parameters, their instances need to provide a means to specify a value for parameters. In Tab. 9.5, we show how such a parameter instantiation can be expressed.

<table>
<thead>
<tr>
<th>Meta Class</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterValue (Instance Package)</td>
<td>Role ( Parameter Name = Value 1, Parameter Name 2 = Value 2, ... )</td>
<td>Instantiating a parameter: the parameters &quot;Parameter Name&quot; and &quot;Parameter Name 2&quot; of the role are instantiated with the values &quot;Value 1&quot;, &quot;Value 2&quot;, etc.</td>
</tr>
<tr>
<td>ParameterValue (Instance Package)</td>
<td>Role ( Value 1, Value 2, ... )</td>
<td>Short notation of the previous notation.</td>
</tr>
</tbody>
</table>

Table 9.5: Parameter Instantiation Notation

9.5 Architectural Templates Repository

As illustrated in Fig. 9.6, our CloudScale AT Repository provides ATs for designing and analyzing scalable SaaS applications.

![Figure 9.6: CloudScale's Architectural Templates Repository](http://cloudscale.xlab.si/wiki/index.php/CloudScale_Architectural_Templates)

In Table 8.1, we list the CloudScale ATs we collected in this repository and briefly describe their application context. We provide a detailed description for each of these ATs in our CloudScale Wiki².

9.6 Naming of Architectural Templates

This section explains the rationale for renaming our CloudScale best-practices to ScaleDL Architectural Templates. We used the former term when we proposed to manifest best practices for scalability in the DoW.

ATs can be compared and contrasted to terms used in software architecture. For instance, in their software architecture handbook, Reussner and Hasselbring [BBJ"08] define and differentiate the terms "architectural style", "architectural pattern", and "reference architecture".

Accordingly, architectural styles are conceptual solution structures applied continuously and widely without exceptions on one element of an architecture [BBJ*08]. This also holds for ATs, however, "exceptions” are completely nonexistent (instead of "widely") due to formalization.

**Architectural patterns** are concrete solutions to common problems [BBJ*08]. Therefore, ATs that restrict system design to only one possible solution (by highly restrictive constraints) can also be seen as patterns.

Finally, reference architectures are architectural styles that (1) are specified at a high level of abstraction and (2) provide a framework for the architecture of a whole system or a family of systems [BBJ*08]. Therefore, the AT for SPOSAD may also be seen as a reference architecture specifying the whole architecture of cloud application families.

We renamed the "patterns" proposed in our DoW to SCALEDL Architectural Templates to make these differences regarding architectural terms explicit. A further reason for this renaming is that ATs allow scalability analyses. These analyses are not supported by classical architectural patterns.

### 9.7 Future Work on the Architectural Template Language

In this deliverable, we present the first integrated version of the AT language; fully supported by the Analyser. Still, this version leaves some issues left as a future work. Therefore, we plan this future work in this section. We will tackle this future work in project year three.

Our plans for project year three include the following issues:

- **Constraints** It is an open issue how to formalize concrete architectural constraints. Technically, one approach to formalization is, for instance, using languages based on first-order predicate logic (e.g., OCL like currently suggested) to specify sets of constraints. However, alternatives like using description logic, set theory, or temporal logic exist as well. Therefore, we are planning to evaluate each alternative and implement the most appropriate one.

- **Explicit consideration of connectors** At the moment, the metamodel of the AT language lacks an explicit consideration of connectors. We plan to investigate the possibilities to explicitly consider these connectors in the next metamodel version, similar to what has been done in the MUSIC project [MUS10].
10 **Analyser**

Software architects can use the Analyser to automatically analyze a modeled system with SCALEDL. In project year 1, our Analyser implementation only had support for classical performance metrics. In project year 2, we extended the Analyser by novel metrics for scalability, elasticity, and efficiency and integrated support for architectural templates. We updated especially Sec. 10.4 to report on these new capabilities (it references a new, dedicated document giving an architectural overview of the new Analyser implementation). The rest of this chapter is similar to deliverable D1.1.

The Analyser process is illustrated in Fig. 10.1. This analysis allows architects to check whether a system meets its scalability, elasticity, and efficiency requirements stated via a SCALEDL USAGE EVOLUTION (cf., Chap. 7) instance. For this check, the Analyser first analyzes the scalability, elasticity, and/or efficiency of the modeled system. Based on comparing the analysis results with the SCALEDL USAGE EVOLUTION instance, Analyser can further output whether SCALEDL USAGE EVOLUTION requirements were met.

As an example, consider the architect who models the CloudStore system and specifies requirements via a SCALEDL USAGE EVOLUTION instance. The architect may use the Analyser by taking the CloudStore SCALEDL USAGE EVOLUTION instance as input. The Analyser checks the elasticity requirement specified in the requirement specification (the system should be capable of responding in less than 3 seconds for user arrival rates between 1 and 60 users per hour). Subsequently, Analyser analyses whether the requirement can be met.

![Analyser Process Diagram](image)

**Figure 10.1: The Analyser Process**

10.1 **Relations to Other Chapters**

The Analyser depends on two other chapters: Firstly, the Analyser process illustrated in Fig. 10.1 refines the Analyser-relevant part of the CloudScale Method (described in Chap. 5) – the Analyser process is part of the CloudScale Method process (2), i.e., the System Construction and Analysis process. Secondly, Analyser uses a SCALEDL instance as input. Therefore, Chap. 6 about SCALEDL and its sublanguages (as illustrated in Fig. 10.1) are relevant for the Analyser.
Besides these dependencies, Analyser depends on Palladio and SimuLizar as well. We describe both approaches next.

### 10.2 Palladio

Palladio [BKR09] approach comes with a model to specify component-based systems, the Palladio Component Model (PCM). The PCM allows architects to model components, assemblies of components into systems, hardware resources, the allocation of components to these resources, and static usage scenarios. The Palladio approach comes also with a tool, the Palladio-Bench, allowing to analyze a system (i.e., a PCM instance) regarding quality properties. Currently, the Palladio-Bench supports performance (response time, utilization, throughput) as well as safety (mean time to failure) quality properties.

In the context of CloudScale, we reuse and extend both, the PCM as well as the Palladio-Bench as described in Sec. 10.4.

### 10.3 SimuLizar

SimuLizar [BLB12, BBM13] extends Palladio to support modeling and analysis of self-adaptations. For this, SimuLizar enriches the PCM to specify (1) monitoring annotations and (2) adaptation rules. Monitoring annotations allow to mark PCM elements, e.g., an operation of a component, to be monitored during analysis using a metric such as response time. Adaptation rules can react on changes of monitored values. For example, when a certain response time threshold is exceeded, an adaptation rule could trigger a scaling out of bottleneck components. SimuLizar allows to consider these adaptation rules during system analysis.

The concept of self-adaptation is important for CloudScale because we want to analyze elastic cloud computing environments, e.g., systems that can scale-out and scale-in based on currently monitored load. SimuLizar's concepts of self-adaptation allow to model and analyze such systems.

In the context of CloudScale, we reuse and extend SimuLizar as described in Sec. 10.4.

### 10.4 Technical Realization

Technically, the Analyser is based on Palladio and SimuLizar (cf., Sections 10.2 and 10.3). Analyser adds the capabilities to (1) analyze usage evolution specified using SCALEDL USAGE EVOLUTION, (2) automatically check whether system scalability requirements are fulfilled, and (3) support SCALEDL ARCHITECTURAL TEMPLATES for modeling systems (cf., Sec. 9.2.1). In the example from the introduction of this chapter, the usage evolution is characterized by arrival rates between 1 and 60 users per hour. The quality threshold for this usage evolution is that the system should be capable of responding in less than 3 seconds.

In order to document the full technical realization of the Analyser, we created a separate document – called the "Quality Analysis Lab (QuAL): Software Design Description Version 0.1". This document serves not only the purpose of describing the technical realization of the Analyser but also to disseminate CloudScale results to the Palladio community. Accordingly, the QuAL document can directly be used by Palladio developers that are interested in taking advantages of the described, novel features that were introduced by CloudScale. The QuAL document also includes a brief user guide, illustrat-
ing how to use the Analyser. The software design description of QuAL is available within the Palladio SVN (User: anonymous, Password: anonymous; https://svnserver.informatik.kit.edu/i43/svn/code/QualityAnalysisLab/Documentation/trunk/org.palladiosimulator.qual.docs/QualityAnalysisLab.pdf).

In our deliverable D3.2 (Chapter 6), we provide some additional, CloudScale-specific details about the technical realization of the Analyser. These details particularly include Analyser’s current implementation status, implementation requirements for the next version of the Analyser, and its integration into the CloudScale tool suite.

10.5 Future Work on the Analyser

See deliverable D3.2 (Chapter 6) for future work on the Analyser.
11 Future Work

This deliverable has presented the conceptual foundation for CloudScale as well as the artefacts for scalability and elasticity evaluation method, SCALEDL USAGE EVOLUTION, CLOUDSCALE METHOD, HowTos, SCALEDLARCHITECTURAL TEMPLATES language and ANALYSER. We have specified future work in the following sections:

- Concepts: Sec. 2.5
- Scalability evaluation method: Sec.3.6
- Elasticity evaluation method: Sec.4.3
- CLOUDSCALE METHOD: Sec. 5.7
- SCALEDL USAGE EVOLUTION: Sec. 7.7
- HowTos: Sec. 8.4
- SCALEDLARCHITECTURAL TEMPLATES: Sec. 9.7
- ANALYSER: Sec. 10.5

In summary, we will work more with the following issues in the final period, and be documented in D1.3:

Composition of services will be addressed more extensively.

Sharper definitions of scalability, elasticity and efficiency based on feedback from our metrics and from our case studies.

Metrics refined based on validation with our case studies. New metrics may also be developed.

SCALEDL USAGE EVOLUTION will be validated, and if required, the meta-model will be revised. Work and quality threshold usage evolution will also be included.

Standardisation Work on standardisation will be pursued further.

More mature CLOUDSCALE METHOD based on the validation results. Future version of this document will also contain full range information regarding the usage effort for each method step based on the validation results from industrial partners and the showcase. When validating the method we will also use the evaluation methods in Section 3 and in Section 4.

More HowTos and SCALEDLARCHITECTURAL TEMPLATES will be added. We plan to add HowTos and SCALEDLARCHITECTURAL TEMPLATES for engineering elastic and efficient cloud computing applications.
A Analyser Test Plan

This chapter describes a test plan for the implementation of the new metrics in the Analyser. Because the implementation of the Analyser proved to be hard, we have made this test plan to make it easier to deliver sufficient Analyser implementation quality. We want to test the quality of the answers given by the implementations. The testing is not aimed at assessing the stability of the implemented functionality, even if such problems should of coarse be described in the test report.

A.1 Prerequisites

A.1.1 State of the Analyser

This test plan assumes that Usage Evolution is part of the Analyser, except for some simpler tests that only rely on the Analyser itself. We further assume that we can trust the Analyser for the functionality previously implemented in Palladio, so that it is for example not required to use another tool for solving models with software contention.

A.1.2 Three Important Parts of a Test Case

Apart from the test procedure, an Analyser test case will generally consist of three parts:

1. A model of a service to test. The following models can be used:
   - CloudStore: which has several operations and three scenarios: browsing, shopping and mixed.
   - Make simple models for example with a few components and a simple allocation model.

   A model will also include the resources used (termed resource environment in Palladio/Analyser), and in some cases we will explicitly modify these resources.

2. The workload to use for testing the model
   - Stochastic load variations that is already implemented in Palladio.
   - Generated load using Usage Evolution and Limbo. The following load evolutions are relevant: (1) Stable: where the current load level and the duration are two parameters; (2) Linear, which in addition is characterised also by the slope.

   We will rely on generated load, because this makes it simpler to see if the output is correct. Figure A.1 shows an arch typical load evolution, with a stable load, then a linear increase (or a decrease), followed by a new stable load. The values $A$, $D$, $e$, and $h$ must be determined.

3. Derive expected results. We can have the following cases:
   - Answer is obviously known.
   - Compare model with real service, for example an implementation (and not a model) of CloudStore. This introduces several error sources apart from the implementation of the metrics.
• Compare with a model where the solution is known. This may for example be a published LQN solution if software resources are required. QPME (Queueing Petri Net Modelling Environment) can be used for testing some of the results.

• Compare within each metric: (1) Compare with different inputs and see if the outputs match. (2) Compare same inputs and see if the results match within statistical boundaries.

• Compare different metrics: Compare the results of different metrics and see if they match.

In particular we will rely on comparing outputs from different metrics and will also compare different inputs for similar metrics. For simple cases we will consider cases where the answer is obviously known. Comparing with real services and comparing with (another) model will not be done.

### A.1.3 Equivalence Classes

As part of a test plan it is always good to describe equivalence classes. A test of a representative value of each equivalence class will fairly well represent the test of any other value in the same class [MBS12, p. 50]. We have identified the equivalence classes in Table A.1.

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid Equivalence Class</th>
<th>Invalid Equivalence Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute load level</td>
<td>$[0, \infty)$</td>
<td>Negative values</td>
</tr>
<tr>
<td>Linear load slope increase</td>
<td>All values</td>
<td>None</td>
</tr>
<tr>
<td>Duration of test</td>
<td>$(0, \infty)$</td>
<td>$0$</td>
</tr>
<tr>
<td>Number of software resources</td>
<td>$(0, \infty)$</td>
<td>None</td>
</tr>
</tbody>
</table>

Within the equivalence classes, we must at least test with the boundaries.
A.1.4 Boundary Cases

Experience shows that testing boundary cases has a high payoff [MBS12, p. 55]. Looking at the equivalence classes, we identify the following boundaries cases:

- No load
- No load increase
- Sudden load increase, i.e. for Figure A.1 this means $a = b$.
- No duration of the test
- No software resources

A.1.5 Work and Load and Quality Thresholds

This test plan focuses on testing different load. This is more important than testing different work. However, when this test plan is finished for load, also work shall be tested. All the test cases can then be reused. The difference is that while we previously held work constant and varied the load, we will now do the opposite.

In general, there will often be one load parameter for a scenario, but many work parameters. CloudStore has two work parameters (1) number of books, which can be varied freely and (2) number of customers, which may or may not depend on the load.

For all the tests we assume fixed quality thresholds for the operations in the metric. This restriction may of course be loosened.

A.2 Simple Tests

These tests use simple models and simple workloads which are made directly in the Analyser tool.

A.2.1 Simple Test of Scalability Range

Make a very simple queueing network model with only one CPU and with no software resources.

Prerequisites

None

Test Model

This simple model is a queueing network model with one service centre which use one CPU. Note that there are no software resources in this model. This model is embarrassingly scalable and should therefore not have any scalability range, i.e. it shall be able to scale until infinity.
Test Workload

Is not applicable for testing capacity.

Test Procedure

Test the Scalability Range metric on the simple test model.

Expected Results

The Scalability Range metric shall give an infinite range for this model.

A.2.2 Simple Test with No Load

In this test we will test with no load, but also with constant load and with no duration of the test.

Prerequisites

None

Test Model

Make the same simple model as in the previous test, described in Section A.2.1.

Test Workload

The workload will be both an open class with 0 as load intensity and a closed class with 0 customers. We may also change this slightly to be a constant load, i.e. with a constant intensity (open) or with a constant number of customers (closed).

Test Procedure

Using the specified empty or constant workload run these four metrics

- MTTQR (Mean Time to Quality Repair)
- NSLOV (Number of SLO Violations)
- RPE (Resource Provisioning Efficiency)
- MC (Marginal Costs)

Also do the same with the same, constant load, but with no duration of the test.
Expected Results

The expected results both with 0, with constant load or with no duration of the test, are also follows:

- MTTQR: 0
- NSLOV: 0
- RPE: 0
- MC: 0

A.3 Usage Evolution Tests

These tests builds on the usage evolution described in Figure A.1. We organise these tests according to the order in which they are performed.

A.3.1 Scalability Range (SR)

This metric captures the absolute maximum amount of load for a given system (the capacity of the system), with an infinite amount of resources.

Prerequisites

None

Test Model

We will use the CloudStore model and will test with a configuration with very many CPUs. In the Analyser resource environment, we can for example set the number of replicas to 1 000 000, which means we have one million CPUs.

Test Workload

Is not applicable for testing capacity.

Test Procedure

We first use the CloudStore model and get the Scalability Range. Afterwards we use the same model with very many CPUs and then look at the capacity for this configuration.

Expected Results

We test this by comparing (1) the output of the Scalability Range metric with (2) giving very many resources to the system. We then see if the resulting capacities are equal. If not, we report both numbers and their difference in per cent.
A.3.2 Scalability Capacity (SC)

For configurations of interest this metric finds the capacity. In this test we make four different deployments with increasingly more resources for the bottleneck device. For CloudStore this will most likely be the database CPU. Adding more database CPU resources can be done either by increasing the number of CPUs, the number of cores in each CPU or by increasing the frequency of the cores in each CPU. A combination of all these three options is of course also possible.

Prerequisites

None

Test Model

CloudStore model will be used and we will have four configurations with noticeable differences in bottleneck hardware resources, for example with one, two, three or four CPUs. We set numbers of CPUs in the Analyser resource environment.

Test Workload

Is not applicable for testing capacity.

Test Procedure

For all these four configurations, one by one:

1. Run the Capacity Metric and record the result

2. Test with the load as specified by the Capacity Metric

3. Test with a slightly higher load than what is predicted by the Capacity Metric. By slightly higher we mean for example 450 if the capacity was predicted to be 440. If will of course also be interesting to test both with 445, and then with 443 and 441 to test the accuracy of the Capacity Metric.

Term these four different capacities A, B, C and D, and use them in the following testing.

Expected Results

For all four configurations, when the load is equal to the load specified by the Capacity Metric, the quality threshold should be obeyed, whereas with a slightly higher load, the system cannot fulfil its quality thresholds. As described above, it is also interesting to capture the accuracy of the Capacity Metric, and this can be tested by successively testing with load closer and closer to the predicted capacity.
A.3.3 Resource Provisioning Efficiency (RPE), Internal Comparison

This benchmark captures the mismatch to optimal provisioning. In this test we use the usage evolution specified in Figure A.1, but with different slopes and durations, and we compare these numbers. This is therefore an internal comparison with the same metric.

Prerequisites

None

Test Model

CloudStore model.

Test Workload

As workload, we will use different usage evolutions all corresponding to Figure A.1, but with varying degrees of steepness of the slope and also with varying durations of the slope (which together also implicitly defines the end level, i.e. capacity $D_i$).

Concerning which slopes to test with, we would at least recommend using both a steep slope (fast increase) and a steep slope (slow increase) and then something in between. Concerning the duration, one possibility is to test with 10, 20, 30, 40, 50 and 60 seconds, but longer times may also be interesting.

Test Procedure

Consider Figure A.1 and run several times with slightly different input values for start level, slope and duration of the slope.

Expected Results

Assuming the same start level, one would expect the mismatch to be higher with a steeper slope. Further, one would expect the mismatch as predicted by RPE, to be higher for a longer duration, given a the same start level and the same slope.

A.3.4 Resource Provisioning Efficiency (RPE), Comparing with Capacity Metric

This benchmark captures the mismatch to optimal provisioning, but then by comparing with the Capacity Metric.

Prerequisites

The same four configurations as in the Capacity Metric testing.
Test Model

CloudStore model.

Test Workload

Usage evolution corresponding to the four capacities specified by the Capacity Metric and reflected in Figure A.1.

Test Procedure

Get the RPE Metric for the usage evolution corresponding to the four capacities as specified by the Capacity Metric.

Expected Results

Compare with the values predicted by the Capacity Metric and manually compare against Figure A.1. Use the four capacities established: A, B, C and D. These four capacities then describe optimal provisioning.

1. In particular, until time e, optimal provisioning is given by the initial provisioning with capacity A,
2. and then change to the optimal provisioning leading to capacity B which is valid until time f,
3. where we get the provisioning leading to capacity C, which is valid until we possibly get a new optimal provisioning leading to capacity D at time g.

In addition, we also have to know the actual provisioning, and we can compare this with the benchmark given by the capacity metric, described above, and manually compute the amount of over provisioning. These manually computed values are then compared with the values computed by the RPE Metric.

A.3.5 Number of SLO Violations (NSLOV), Internal Comparison

We investigate how often SLOs are violated given a workload delta. This is the same as for comparing with RPE internally in Section A.3.3, except that we are now comparing SLO violations. We expect a higher delta to result in more violations. Similarly, we expect a shorter time from e to h to result in more SLO violations. If these expectations are not fulfilled, this should be described in the measurement report.

A.3.6 Mean Time to Quality Repair (MTTQR)

Captures the mean time to repair a certain workload delta. We investigate how often SLOs are violated given a workload delta. This is the same as for comparing with RPE internally in Section A.3.3, except that we are now comparing mean time to quality repair. We expect a higher delta to give a longer mean time to. Similarly, we expect a shorter time from e to h to result in a higher time to repair. If these expectations are not fulfilled, this should be described in the measurement report.
A.3.7 Marginal Cost (MC), Internal Comparison

Marginal costs answers the question: How much does it cost to serve a small increment in load? We compare slopes and see if we get a higher marginal cost with a higher delta, i.e.; a larger difference between capacity A and D in Figure A.1. This is the same as for comparing with RPE internally in Section A.3.3, except that we are now comparing marginal costs. We expect a higher delta to give a larger marginal cost, in the same way as we also expect a shorter time from e to h to also give a higher marginal cost. If these expectations are not fulfilled, this should be described in the measurement report.

A.3.8 Marginal Cost (MC), Comparing with Capacity Metric

When compared with the Capacity Metric we should see a zero marginal cost until we need a new configuration, which will then lead to a jump in the marginal costs. This of course assumes that we only pay for the instances and not for anything else, i.e. we pay for IaaS and not for PaaS and SaaS. Since pricing for PaaS and SaaS is harder to model, this seems like a valid assumption.

Prerequisites

The same four configurations as in the Capacity Metric testing.

Test Model

CloudStore model.

Test Workload

Experiment with different usage evolutions corresponding to Figure A.1, some with a small load increment and others with larger load increments. Remember that a load increment will lead to a marginal cost increase of zero unless it corresponds to two different configurations.

Test Procedure

Experiment with load increments corresponding to a shift in configurations and increments that do not lead to shifts.

Expected Results

If we know based on the capacity metric that the increment requires a new configuration, then the marginal cost will be non-zero, otherwise it will be zero.

A.3.9 Scalability Speed (SS)

What is the maximum change rate for the system? Check that for the predicted scalability speed, the SLO is obeyed. For a higher speed the SLO should be violated.
Prerequisites

The same four configurations as in the Capacity Metric testing.

Test Model

CloudStore model.

Test Workload

The same four configuration as specified for the Capacity Metric.

Test Procedure

For all four configurations corresponding to capacities A, B, C and D:

1. Get the maximum speed from Scalability Speed Metric
2. Test with this speed and observe the results
3. Test with a higher speed and observe the results

Expected Results

For all four configurations, when the speed is equal to the speed specified by the Capacity Metric, the quality threshold should be obeyed, whereas with a slightly higher speed, the system cannot fulfil its quality thresholds.
B Glossary

A glossary specifying key CloudScale terms is a live document [Clo14]. A snap shot of the glossary is for completeness also included here.

**Actor** "An actor specifies a role played by a user or any other system that interacts with the subject." [Obj11]

**Architectural Styles** Architectural styles are Conceptual solution structures applied continuously and widely without exceptions on one element of an architecture (based on [BBJ+08]). Examples are the 3-Tier style, Pipes&Filters, and SPOSAD.

**Architectural Templates** The Architectural Template (AT) language is a language to formalize architectural styles on component models. In particular, this formalization allows to enrich styles by quality annotations and completions for model-driven quality analyses. Quality annotations characterize a concrete quality property of interest. Quality completions utilize these annotations to derive quality models that can be used as input to quality analysis tools. The language is formalized as "ScaleDL Architectural Template".

Whenever we refer to "an Architectural Template (AT)", we talk about a concrete architectural style formalized by the AT language. Accordingly, the plural form, Architectural Templates (ATs), refers to a set of these formalizations.

**CloudScale Architectural Templates** CloudScale Architectural Templates (CATs) are the set of architectural templates (ATs) we evaluated for CloudScale. This set of ATs (1) representing best practices for designing SaaS applications and (2) allows to analyse the scalability of SaaS applications.

**CloudScale Environment** CloudScale’s Eclipse application. It is a front end to all CloudScale tools.

**Costs** Costs are the valued consumption of resources necessary to provide a certain service (per time unit). (Our definition is based on two German statements: "Kosten bezeichnen in der Regel den mit Marktpreisen bewerteten Einsatz von Produktionsfaktoren bei der Herstellung von Gütern und Dienstleistungen" [Wik13, Kosten] and "Kosten stehen betriebswirtschaftlich gesehen für den bewerteten Verbrauch an Produktionsfaktoren in Geldeinheiten (GE), welche zur Erstellung der betrieblichen Leistung in einer Abrechnungsperiode notwendig sind" [Wik13, Kosten].) Typically, this valuation is monetary. Also see economic costs [Wik13, Economic cost] and operating cost [Wik13, Operating cost].

**Customer** An actor consuming services of a service provider. The term "service customer" can synonymously be used. Refinements of a "service customer" include "SaaS customer", "PaaS customer", and "IaaS customer". Other typical synonyms are "user" and "consumer", however, these synonyms should be avoided for clarity.

**Deployer** An actor making a layer’s services ready to use, potentially by releasing a software system on top of lower-layer services. The term "service deployer" can synonymously be used. Refinements of a "service deployer" include "SaaS deployer", "PaaS deployer", and "IaaS deployer".

**Dynamic Spotter** The Dynamic Spotter is a performance problem diagnosis that finds scalability issues in code by measurements.
Elasticity For an as-a-Service layer, elasticity is the degree to which the layer is able to adapt to workload changes by (de)provisioning services of its underlying layers in an autonomic manner such that at each point in time the utilised services fulfill the SLOs of the layer as closely as possible. (based on [HKR13])

Load Load is the characterisation of the quantity of customer’s service requests at a given time, e.g., by characterising the request rate.

Metric A metric is a "precisely defined method which is used to associate an element of an (ordered) set V to a system S" [EFR08]. Typically, we use metrics to determine a quantity (V is the set of natural or real numbers). An example SaaS quantity metric is the sum of consumed IaaS and PaaS services. Example IaaS quantity metrics include the number of consumed CPU services as well as the number of CPU, disk and network invocations by customers.

Ontology "In computer science and information science, an ontology formally represents knowledge as a set of concepts within a domain, and the relationships between pairs of concepts. It can be used to model a domain and support reasoning about concepts." [Wik13, Information Science] We use directed graphs to represent ontologies where nodes represent concepts and edges represent relationships. An example for reasoning is the identification of transitive relationships. Nodes can represent types as well as instances. "OWL is a language for making ontological statements, developed as a follow-on from RDF and RDFS" [Wik13, Information Science]. Furthermore, OWL ontologies can be queried by SPARQL. The most popular OWL reasoner is Pellet.

PCM The Palladio Component Model (PCM) is a component-based architecture description language, with a focus on performance properties (response time, throughput, utilization).

Performance For an as-a-Service layer, the performance for a given workload is measured in terms of a quality metric.

Price The price is the valuation for a quantity of a certain service, typically realised in monetary units. (Based on the following German statement: "Der Preis [...] ist der üblicherweise in Geldeinheiten realisierte Wert eines Gutes oder einer Dienstleistung." [Wik13, Preis (Wirtschaft)])

Price Model A price model is the concrete specification of prices for consuming services. For example, the service provider Amazon provides its price model for the EC2 IaaS solution on its web page [Ama]. A typical synonym is "cost model", however, this synonym should be avoided as we want to strictly distinguish between costs and prices.

Product Manager The product manager is a final decision maker for the solution and negotiation with customers about service/system acceptance and approval of system evolution during the runtime phase. Main interests of the product manager include overall system behavior, architecture compliance, and price of the final solution.

Provider An actor offering services. The term "service provider" can synonymously be used. Refinements of a "service provider" include "SaaS provider", "PaaS provider", and "IaaS provider".

Quantity a property that can exist as a magnitude or multitude." [Wik13, Quantity]
representing quality of experience. Quality metrics may also reflect system properties like utilisation of some hardware resources.

**Quality threshold** is the border between acceptable and non-acceptable quality, when applying a quality metric. With under-provisioning, the quality crosses this border, and we get an SLA violation. With over-provisioning, the quality is better than the threshold. As an example where the quality metric is response time, the threshold may be 1 second, meaning that users are satisfied with less than 1 second response time. More than one second will be an SLA violation. On the other hand, less than 1 second means that we have over-provisioning.

Part of the quality threshold specification must also be an agreement on how this quality shall actually be measured. In the case of response time, the threshold can, e.g., be the same for all service invocations, only apply to the 90 percentile, or only for the average.

**Resource** In computer science, a “resource, or system resource, is any physical or virtual component of limited availability within a computer system.” [Wik13, Computer science] In economics, resources (or ‘factors of production’) “are the inputs to the production process.” [Wik13, Resource (economics)] Furthermore, resources “are any commodities or services used to produce goods or services” [Wik13, Resource (economics)], which provides a link between the computer science and the economic definition.

**Scalability** For an as-a-Service layer, scalability is the ability of the layer to sustain changing workloads while fulfilling its SLA, potentially by consuming a higher/lower quantity of lower layer services. (based on [HKR13])

For the SaaS layer, scalability is the ability of the software to sustain changing workloads while fulfilling its SLA, potentially by consuming a higher/lower quantity of PaaS or IaaS services.

**Scalability Analyst** The scalability analyst is an actor who is monitoring the scalability of the system. Based on the analysis, the scalability analyst can detect scalability issues potentially resulting with system evolution.

**ScaleDL** The Scalability Description Language (ScaleDL) is a language to characterize cloud-based systems, with a focus on scalability properties. ScaleDL includes three sublanguages: ScaleDL Usage Evolution, ScaleDL Architectural Template, and ScaleDL Overview.

**ScaleDL Architectural Template** is the formalization of the Architectural Template (AT) language. ScaleDL Architectural Template is realized via an EMF metamodel. Its syntax and semantics are described in deliverable D1.1.

**ScaleDL Overview** is a formalization of the deployment and architectural decisions language. Its syntax and semantics are described in deliverable D3.1.

**ScaleDL Usage Evolution** is the formalization of the usage evolution language. ScaleDL Usage Evolution is realized via an EMF metamodel. Its syntax and semantics are described in deliverable D1.1.

**Service** Software deployed by the deployer, i.e., software running on a layer waiting for requests (by service customers) to be executed. Typically, this execution is charged by service providers. Because services can be consumed by service customers, they are a special kind of a resource. The term “software service” can synonymously be used.

**Service Description** The characterization of a service, e.g., by operation interfaces, their protocols, or by an SLA.
SLA  "A service-level agreement (SLA) is a contractual agreement outlining a specific service commitment made between contract parties – a service provider and its customer. The SLA includes language describing the overall service, financial aspects of service delivery, including fees, penalties, bonuses, contract terms and conditions, and specific performance metrics governing compliant service delivery. These individual performance metrics are called service-level objectives (SLOs)." [Sea]

SLO  Service-level objectives (SLOs) are "[...] performance metrics governing compliant service delivery. [...] Each SLO corresponds with a single performance characteristic relevant to the delivery of an overall service. Some examples of SLOs would include: system availability, help desk incident resolution time and application response time." [Sea]

Spotter  With spotter, we refer to both, the static or the dynamic spotter.

Stakeholders  Stakeholders are actors who have roles associated that are specific for a concrete as-a-Service (XaaS) layer. Typical XaaS stakeholders are XaaS customers, XaaS providers, XaaS architects, XaaS developers, XaaS deployers, and XaaS maintainers.

Static Spotter  The Static Spotter is a static code and model analysis that finds scalability issues by analyzing code and/or system models.

System Engineer  The system engineer is an actor responsible for the runtime system monitoring process and identification of the system critical elements. This actor also drives the system evolution phase.

Usage Evolution  A description of how the workload and the quality thresholds of a service evolves. Described in more detail in D1.1.

Work  Work is the characterisation of the data to be processed by a certain layer.

Workload  Workload is the combined characterisation of work and load.
Bibliography


