Steffen Becker
Gunnar Brataas
Sebastian Lehrig *Editors*

Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications

The CloudScale Method

Springer

Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications

Steffen Becker • Gunnar Brataas • Sebastian Lehrig
Editors

Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications

The CloudScale Method

Springer

Editors
Steffen Becker
Reliable Software Systems Group
University of Stuttgart
Stuttgart, Germany

Gunnar Brataas
Software Engineering, Safety & Security
SINTEF Digital
Trondheim, Norway

Sebastian Lehrig
IBM Research
Dublin, Ireland

ISBN 978-3-319-54285-0 ISBN 978-3-319-54286-7 (eBook)
DOI 10.1007/978-3-319-54286-7

Library of Congress Control Number: 2017937065

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

The digitization of the world around us also impacts IT systems themselves. The increased connectivity of everything and everybody with IT services creates challenges in software design. Systems are becoming increasingly distributed, open, adaptive, dynamic, and mobile.

As a consequence for software quality engineering, run-time factors are increasingly determining the quality of a system. Formerly, assumptions on the run-time context could be taken as static and hence be modeled at design time. Such models were then used in software quality analysis. Nowadays, the context and the system itself are more dynamic. The context changes, and the system, being adaptive, also reflects such changes.

This means, we have no fixed moment during design when system properties are fixed and can be determined by a static analysis. For software quality, this means, instead of proven design-time properties, we need to shift to a means to deal with this dynamicity at run-time. For example, instead of proven security, we talk about highly resilient systems; instead of performance, we talk about scalability.

However, all such measures to increase the resilience of scalability need to be built into the software at design time. Hence, techniques that are able to evaluate at design time the effects of such run-time measures and run-time system properties are of interest. While this may sound impossible, the approach shown in this book demonstrates the feasibility and applicability of this design-time evaluation of quality properties depending on run-time factors in the domain of performance engineering. This book is therefore overdue. After the advent of cloud-based systems in the beginning of the century, this trend of cloud computing is even enforced by the ubiquitous use of several mobile devices, which all need to work with consistent data. Due to the mobility of the devices and the highly fluctuating number of users, the workload of the cloud drastically varies.

This underpins the relevance of the methods presented in this excellent book by outstanding researchers in this field of scalability modeling and analysis. There is no book before this that enlightens us so much about this shift from performance

to scalability! I hope you can read it with the same gain as I did and with the same joy—of seeing Palladio being used and advanced!

Karlsruhe, KIT and FZI
Germany
December 2016

Professor Dr. Ralf Reussner
Chair Software Design and Quality, KIT
Executive Director FZI

Preface

Berlin, 2017: The start-up company SmartService has built an application allowing users to manage subscriptions to business services, magazines, etc. and to send out cancellations automatically and on time. The service is implemented based on a cloud computing environment and uses various third-party services, for example, a service to convert images of contracts uploaded by users into PDF. So far, SmartService has assumed that using a cloud platform will allow it to scale its application as needed without further actions. With this application, SmartService has found a promising niche. The application has spread rapidly, showing a perfect growth of a hockey stick-shaped curve.

Unfortunately, the rising number of users leads to ever-increasing infrastructure costs and, especially during peak loads, to high end-to-end response times, which result in customer losses. The application faces some severe scalability issues. If they cannot fix the problem soon, the whole start-up will be at risk. SmartService needs to address the problem quickly.

Did you ever wonder how to engineer cloud computing services that are scalable, elastic, and cost-efficient—just like the above fictional scenario about the SmartService company?

In this book we describe a detailed method—the CloudScale method—for ensuring that services running on the cloud achieve exactly these properties, ideally by design. With the CloudScale method, software architects can analyze both existing and planned IT services. The method allows to answer questions like:

- With an increasing number of users, can my service still deliver an acceptable quality of service?
- What if each user uses the service more intensively, can my service still handle it with an acceptable quality of service?
- What if the number of users suddenly increases, will my service still handle it?
- Will my service be cost-efficient?

Continuing SmartService's scenario, the CloudScale method allows them to analyze the scalability problem in detail and identify scalability anti-patterns and bottlenecks within its application. Using the method, SmartService quickly realizes that its scalability problems are caused by conservative handling of data in the realization of the application. SmartService uses CloudScale's scalability know-how and applies the CloudScale method to find the best scalable architecture. Equipped with this knowledge, the company swiftly restructures its application and continues its path of growth successfully. In the future, CloudScale's method and tools will allow SmartService to avoid any such critical scalability problems right from the beginning.

When we, the CloudScale EU project,[1] began our work about 4 years ago, we pursued the vision to provide assistance to stakeholders involved in engineering scalable, elastic, and cost-efficient cloud computing services. The "Berlin, 2017" scenario about the SmartService company describes our ideas pretty well. You might still consider such a scenario purely fictional. But, in fact, it is realistic!

For example, the initial design of SAP's BusinessByDesign system had severe scalability issues.[2] As a consequence, SAP suffered from significant financial losses and had to reimplement large parts of the system before it was ready to be released. Another recent famous example is the US government's healthcare system.[3] The system broke down under the load caused by users trying to gather information about their health insurance during the first weeks. Officials had not tested the system with the load the system finally had to face. Also online games regularly face scalability issues during the week of a new game or add-on release. For example, in World of Warcraft, Blizzard regularly faces issues on new game releases despite its extensive experience in this business domain.[4]

Motivated by such scenarios, we began our work on the CloudScale method. You, the reader of this book, have the final outcome of our efforts in your hands: This book gives you an overview of the problems involved in engineering scalable, elastic, and cost-efficient cloud computing services and describes the CloudScale method—a description of rescue tools and the required steps to exploit these tools.

This book is meant for all stakeholders interested in achieving our vision: managers, product owners, software architects, developers, testers, operational personnel, etc. With this book, they can both see the overall picture and drill into issues of particular interest.

[1] www.cloudscale-project.eu.

[2] www.v3.co.uk/v3-uk/news/1970547/sap-update-business-bydesign-plans.

[3] www.informationweek.com/healthcare/policy-and-regulation/why-healthcaregov-failed/d/d-id/1112064.

[4] www.ibtimes.com/battlenet-servers-down-world-warcraft-hearthstone-overwatch-players-around-world-2383254.

This book presents results developed by a motivated group of researchers from different companies, originating from various countries. Not all of them participated in writing this book. However, this book would not exist without their contributions. We thank all CloudScale members for this effort.

Chemnitz, Germany — Steffen Becker
Trondheim, Norway — Gunnar Brataas
Dublin, Ireland — Sebastian Lehrig
December 2016

Acknowledgments

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007–2013) under grant number 317704 (CloudScale) and the Norwegian Research Council under grant number 256669 (ScrumScale). Richard Sanders and Gregor Pipan contributed with proofreading.

Gregor Pipan
XLAB d.o.o.
Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: gregor.pipan@xlab.si

Richard Torbjørn Sanders
SINTEF Digital
Strindvegen 4, 7034 Trondheim, Norway
e-mail: Richard.Sanders@sintef.no

Contents

Part I Introduction and Overview

1 Introduction 3
Steffen Becker, Gunnar Brataas, Mariano Cecowski, Darko Huljenić, Sebastian Lehrig, and Ivana Stupar
1.1 Getting It Right 4
1.2 Software in the Cloud Computing Era 5
1.3 Some Useful Definitions to Characterize Services 7
1.3.1 Operations 7
1.3.2 Service-Level Objectives 8
1.3.3 Workload 8
1.3.4 Capacity 9
1.4 Quality Properties of Services 9
1.4.1 Scalability 10
1.4.2 Elasticity 10
1.4.3 Cost-Efficiency 11
1.5 Consequences of Scalability, Elasticity, and Cost-Efficiency Issues 12
1.6 Causes of Scalability, Elasticity, and Cost-Efficiency Issues 13
1.7 How Should You Manage Scalability, Elasticity, and Cost-Efficiency? 13
1.8 Reactive Scalability, Elasticity, and Cost-Efficiency Management 14
1.8.1 Immediate Temporal Solutions 14
1.8.2 Long-Term Solutions 15
1.9 Proactive Scalability, Elasticity, and Cost-Efficiency Management 16
1.10 The CloudScale Method 17
1.11 What Does It Cost? 18

1.12 What Do You Need? 19
1.13 Conclusion 20
References 21

2 CloudScale Method Quick View 23
Gunnar Brataas, Steffen Becker, Mariano Cecowski, Darko Huljenić, Sebastian Lehrig, and Ivana Stupar
2.1 Process Steps of the CloudScale Method 24
2.2 Running Example 26
2.3 Identify Service-Level Objectives, Critical Use Cases, and Key Scenarios 27
2.3.1 Service-Level Objectives 27
2.3.2 Critical Use Cases 28
2.3.3 Key Scenarios 28
2.4 Identify Scalability, Elasticity, and Cost-Efficiency Requirements 29
2.4.1 Scalability Requirements 30
2.4.2 Elasticity Requirements 30
2.4.3 Cost-Efficiency Requirements 31
2.5 Specify ScaleDL Model 32
2.6 Use Analyzer 32
2.6.1 Scalability Analysis 32
2.6.2 Elasticity Analysis 33
2.6.3 Cost-Efficiency Analysis 34
2.7 Use Spotters 34
2.8 Realize, Deploy, and Operate System 35
2.9 Cloud Computing HowTos 35
2.10 Cloud Computing HowNotTos 37
2.11 The CloudScale Method in the Unified Process 41
2.11.1 Unified Processes 41
2.11.2 Relating the CloudScale Method 42
2.12 Conclusion 43
References 43

Part II Modeling Cloud Computing Applications

3 Cloud Computing Applications 47
Mariano Cecowski, Steffen Becker, and Sebastian Lehrig
3.1 Introduction 48
3.2 Web Applications 49
3.3 Cloud Computing Characteristics 51
3.4 From Web to Cloud Computing Applications 52
3.5 Requirements of Cloud Computing Applications 53
3.6 Modeling Cloud Computing Applications 54
3.6.1 Common View Types for Applications 54
3.6.2 Cloud-Specific View Types for Applications 55

3.7 CloudStore Running Example 56
3.8 Modeling Hints 58
3.9 Conclusion 59
References 60

4 ScaleDL 61
Gunnar Brataas, Steffen Becker, Mariano Cecowski, Vito Čuček, and Sebastian Lehrig
4.1 Introduction 62
4.2 Overview Model 62
4.2.1 Concepts of Overview Model 63
4.2.2 Example of Overview Model 64
4.2.3 Tool Support for Overview Model 65
4.3 Usage Evolution 65
4.3.1 Concepts for Usage Evolution 66
4.3.2 Example of Usage Evolution 67
4.3.3 Tool Support for Usage Evolution 69
4.4 Architectural Templates 69
4.4.1 Concepts of Architectural Templates 70
4.4.2 Example for Architectural Templates 70
4.4.3 Catalog of Architectural Templates 72
4.4.4 Tool Support for Architectural Templates 73
4.5 The Extended Palladio Component Model 73
4.5.1 Concepts of the Extended Palladio Component Model 74
4.5.2 Example for the Extended Palladio Component Model 76
4.5.3 Tool Support for the Extended Palladio Component Model 78
4.6 Conclusion 81
References 81

Part III The CloudScale Method for Software Architects

5 The CloudScale Method 85
Gunnar Brataas and Steffen Becker
5.1 Introduction 86
5.2 Granularity 86
5.3 Method Notation 88
5.4 Roles in the Method 89
5.5 Method Steps 90
5.6 Identify Service-Level Objectives, Critical Use Cases, and Key Scenarios 92
5.7 Identify Scalability, Elasticity, and Cost-Efficiency Requirements 94
5.8 Use-Case I: Analyzing a Modeled System 96
5.9 Use-Case II: Analyzing and Migrating an Implemented System 97
5.10 Realize, Deploy, and Operate 98

5.11 Conclusion 99
References 99

6 Analyzing a Modeled System 101
Sebastian Lehrig, Gunnar Brataas, Mariano Cecowski, and Vito Čuček
6.1 Introduction 102
6.2 Step I: Specify ScaleDL Model 102
6.2.1 Determine Granularity 104
6.2.2 Specify Usage Evolution 106
6.2.3 Specify Overview Model and Generate Extended Palladio Component Model 109
6.2.4 Complete Extended Palladio Component Model 111
6.2.5 Summary for the Specification of ScaleDL Models 115
6.3 Step II: Use Analyzer 115
6.3.1 Set Configuration Parameters 117
6.3.2 Run Analyzer and Assess Requirements 118
6.4 Analyzer Running Example 119
6.4.1 Step I: Specifying a CloudStore Model via ScaleDL 119
6.4.2 Step II: Using the Analyzer with the CloudStore Model 123
6.5 Conclusion 127
References 128

7 Analyzing and Migrating an Implemented System 131
Steffen Becker and Sebastian Lehrig
7.1 Introduction 132
7.2 Spotting HowNotTos 133
7.3 Statically Detecting HowNotTos 136
7.4 Dynamically Detecting HowNotTos 138
7.5 Resolving HowNotTos with HowTos 140
7.6 Spotter Running Example 141
7.6.1 Static Spotter 141
7.6.2 Dynamic Spotter 143
7.7 Conclusion 145
References 146

Part IV Making the CloudScale Method Happen

8 The CloudScale Method for Managers 149
Steffen Becker, Gunnar Brataas, Mariano Cecowski, Darko Huljenić, Sebastian Lehrig, and Ivana Stupar
8.1 Introduction 150
8.2 Key Considerations 150
8.3 Relation to Other Engineering Methods 152
8.4 Pros and Cons of the CloudScale Method 154
8.4.1 Critical Success Factors for Method Adoption and Use 154

8.4.2 Organizational Issues 156
8.4.3 Costs 156
8.4.4 Covering the Cost of the CloudScale Method Adoption ... 157
8.4.5 Risks 158
8.4.6 Critical Factors for Successful Projects 158
8.5 A Pilot Project 159
8.6 Setting Up the CloudScale Environment 161
8.7 Complementing Tools 162
8.8 Following the CloudScale Method for the Pilot Project 163
8.9 Conclusion 164
References 165

9 Case Studies 167
Darko Huljenić, Ivana Stupar, and Mariano Cecowski
9.1 Case Study: Electronic Health Record 167
9.1.1 Electronic Health Record 168
9.1.2 Applying the CloudScale Method and Tools to Electronic Health Record 170
9.1.3 Discussion of the Electronic Health Record Case 176
9.2 Case Study: Kantega's Flyt CMS 177
9.2.1 Flyt CMS 177
9.2.2 Applying the CloudScale Method and Tools to Flyt CMS 179
9.2.3 Discussion of the Flyt CMS Case 180
9.3 Additional Case Studies for the CloudScale Method 181
9.4 Conclusion 182
References 183

Glossary 185

Index 187

Contributors

Steffen Becker University of Stuttgart, Stuttgart, Germany

Gunnar Brataas SINTEF Digital, Trondheim, Norway

Mariano Cecowski XLAB d.o.o., Ljubljana, Slovenia

Vito Čuček XLAB d.o.o., Ljubljana, Slovenia

Darko Huljenić Ericsson Nikola Tesla, Zagreb, Croatia

Sebastian Lehrig IBM Research, Dublin, Ireland

Ivana Stupar Ericsson Nikola Tesla, Zagreb, Croatia

Part I
Introduction and Overview

Cloud computing applications operate in environments that tailor the number of computing resources to current requests—we refer to this dynamic tailoring as elasticity. For example, if an application is heavily requested, the environment can allocate further CPUs to the application. Such additional resources are only beneficial if the application makes actual use of the resources, e.g., by integrating additional resources into its load-balancing strategy—we refer to this ability as scalability. For varying numbers of requests, scalable and elastic applications neither suffer from under-provisioning (too few resources available) nor over-provisioning (too many resources available). Avoiding over-provisioning saves application providers costs for operating the application—we refer to such savings as an improved cost-efficiency.

The CloudScale method allows software architects to engineer applications for scalability, elasticity, and cost-efficiency. This engineering is supported by various best practices for engineering cloud computing applications and dedicated analyses, e.g., based on models of an application and its environment, static code analyses, and a dynamic instrumentation of an application. Analysis results provide software architects feedback for improving their applications.

In Part I, we motivate the importance of scalability, elasticity, and cost-efficiency as driving quality attributes of modern cloud computing applications (Chap. 1). Moreover, to analyze and optimize these quality attributes, we motivate and outline the CloudScale method by giving a brief overview (Chap. 2).

Chapter 1
Introduction

Steffen Becker, Gunnar Brataas, Mariano Cecowski, Darko Huljenić, Sebastian Lehrig, and Ivana Stupar

Abstract When building IT systems today, developers face a set of challenges unknown a few years ago. Systems have to operate in a much more dynamic world, with users coming and going in an unpredictable manner. User counts have exceeded the limit of billions of users, and the Internet of Things will even increase those numbers significantly. Hence, building scalable systems which can cope with their dynamic environment has become a major success factor for most IT service providers. Those systems are run on a vast amount of hardware and software resources offered by cloud providers. Therefore, this chapter gives an introduction into the world of cloud computing applications, the terminology and concepts used in this world, and the challenges developers face when building scalable cloud applications. Afterward, we outline our solution for engineering cloud computing applications on a very high level to give the reader a jump-start into the topic.

This chapter is structured as follows. In Sect. 1.1 we sketch the world of cloud applications and motivate the need for engineering their scalability. For those who have not worked on a cloud system, we outline its characteristics in Sect. 1.2 and define its essential concepts in Sect. 1.3. As this book is about building scalable

S. Becker (✉)
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
e-mail: steffen.becker@informatik.uni-stuttgart.de

G. Brataas
SINTEF Digital, Strindvegen 4, 7034 Trondheim, Norway
e-mail: gunnar.brataas@sintef.no

M. Cecowski
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: mariano.cecowski@xlab.si

D. Huljenić • I. Stupar
Ericsson Nikola Tesla, Krapinska 45, 10000 Zagreb, Croatia
e-mail: darko.huljenic@ericsson.com; ivana.stupar@ericsson.com

S. Lehrig
IBM Research, Technology Campus, Damastown Industrial Estate, Dublin 15, Ireland
e-mail: sebastian.lehrig@ibm.com

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_1

cloud computing applications, we introduce the most important quality properties (scalability, elasticity, and cost-efficiency) in Sect. 1.4. In Sect. 1.5 we describe what consequences you will face when failing to get the quality of your application right, while Sect. 1.6 explains reasons why developers fail to provide sufficient quality when not following specialized methods. As development efforts and methods need to be managed, Sect. 1.7 describes what it takes to manage scalability, elasticity, and cost-efficiency. Management can be done in two ways, i.e., reactively (cf. Sect. 1.8) or proactively (cf. Sect. 1.9). Once you have understood why you should care about scalability, we sketch our solution approach, the CloudScale method (cf. Sect. 1.10). We explain what it costs to introduce it in Sect. 1.11 and what is required to be successful (cf. Sect. 1.12).

1.1 Getting It Right

Building properly scalable systems has been a challenge for software developers for decades already. Here, scalable systems means systems which either had some spare resources left (i.e., over-provisioned resources), or which were expected to process further workload when moved to faster hardware. Why then did the cloud computing area lead to an even more increased attention on scalability topics? The reason is the shift in the type of applications we are building and operating today. Providers like Facebook, Amazon, Twitter, and Netflix operate on a so-called WebScale [1]. WebScale refers to the fact that any person connected via some device to the Internet is a potential customer and might generate requests for the service. As a consequence, we see both high and very unpredictable loads as well as huge amounts of data being worked upon.

Today, most of these systems are less critical and may determine only the fate of its providing company. But in the near future, many of these systems will be or will become mission critical, i.e., the fate of people's life, economic welfare, or of whole societies will depend on these systems. For example, consider systems which form the backend of connected devices like smart distribution grids for electricity, or water; traffic control systems for autonomous connected cars, trains, or airplanes; high-frequency stock trading; etc.

The more critical these systems become, the more crucial it becomes to get them right? Getting them right means to implement systems in a way that they comply to their requirements right from the beginning. In particular, they have to comply to their extra-functional requirements, including scalability, elasticity, and efficiency. A non-scalable smart grid system will break in case of an unpredicted event, causing a burst of unprocessed control messages; a non-scalable traffic control system might route cars and trains right into traffic, worsening the situation and causing a huge loss of time and money. But even on smaller scales, non-scalable systems cause problems. For example, the initial design of SAP's BusinessByDesign system had severe scalability issues. As a consequence, SAP suffered from significant financial losses and had to reimplement large parts of the system [2] before it was ready

to be released. Another recent famous example is the US government's health-care system. The system broke down under the load caused by users trying to gather information about their health insurance during the first weeks. Officials had not tested the system with the load the system finally had to face. Also online games or games with online authorization components regularly face scalability issues during the week of a new game or add-on release. For example, in World of Warcraft, Blizzard faces regularly issues on new game releases despite their extensive experience in this business domain [3].

Looking at all these examples, one might ask what to do to get it right. When revisiting classical software development, the most often practiced technique to ensure that systems comply to their requirements is testing. However, testing focuses frequently on testing the functional correctness of systems, while extra-functional properties are often ignored. To address extra-functional concerns, special types of tests are needed. For instance, performance and scalability is tested in so-called load or staging tests. However, these tests are often complicated in practice. To test systems under high load or heavy work conditions, those conditions have to be created first. The larger the load or work situation is, the more infeasible this becomes as it requires a huge amount of resources to generate that load or work.

Often, the requirements specifications themselves are part of the problem. While functional requirements are typically specified in detail, extra-functional properties are either lacking or are expressed in vague specifications which cannot be systematically engineered nor tested.

New engineering methods are needed to tackle the problem. The CloudScale method introduced in this book provides such a method. By explicitly taking scalability, elasticity, and efficiency requirements into account and by providing approaches to verify their fulfillment either using models of the system under development or their actual implementation, it provides a way to implement cloud computing applications right from the beginning in a way that they fulfill their requirements. This is mainly achieved by introducing engineering principles in the software development process which have not been integrated before. In particular, by analyzing the system's design already early on in the development process and later checking for this, it provides the means to forecast the success of the development endeavor and avoid surprises.

1.2 Software in the Cloud Computing Era

With the rapid development of computing hardware, high-speed networks, web programming, distributed and parallel computing, and other technologies, cloud computing has recently emerged as a commercial reality. Cloud computing is rapidly emerging as the new computing paradigm of the coming decade.

Its main idea is to perceive computing power, storage capacities, and networking resources as resources which can be utilized like we utilize energy. These resources

are offered by specialized companies, while the resource users neither know nor care how or where this actually takes place.

In practice, providers offer resources by virtualizing them. They make revenue due to the fact that not all resources sold will be used at all times. Providers sell not just hardware (Infrastructure as a Service, IaaS), but software resources as well (Platform as a Service, PaaS, or Software as a Service, SaaS). This has attracted the attention of industry developers as well as researchers across the world.

Cloud computing has five major properties: on-demand self-service, broad network access, resource pooling, rapid elasticity or expansion, and measured service [4]. With on-demand self-service, cloud customers request their needed resources themselves, typically using fully automated provisioning application programming interfaces (APIs). Broad network access gives them the ability to access their services. Resource pooling is used by the cloud providers to share their resources among all their customers efficiently. Rapid elasticity allows cloud customers to provision and deprovision their resources in small time frames, often in the range of minutes or even seconds. Exploiting this elasticity, cloud customers can match the resource demand of their application. This leads to a cost-efficient delivery. Finally, measured services refers to the fact that cloud providers meter the resource by their cloud customers and charge them based on usage, i.e., cloud customers only pay for resources they have actually used.

Cloud computing offers a viable solution to the challenges of addressing scalability and availability concerns for large-scale applications. This is partly due to the rapid elasticity property mentioned, but also thanks to the expertise of the cloud providers to operate their resources without failures. Cloud computing has evolved during the years starting from data centers to present-day infrastructure virtualization. Although cloud computing is maturing, there are still many unsolved challenges: formal models to analyze properties of cloud computing applications, immature infrastructures and their configuration options, a need for novel architectures (like microservice-based architectures), rapid and timely provision of services, and development of applications utilizing the new opportunities best.

In the “classical” IT environment, software is binary code installed onto a local computer. Software in a cloud becomes a service. The cloud deployment model for software (SaaS) delivers code and data remotely. Cloud software is designed to be loosely coupled services, to be encapsulated, and to be available through the web. In traditional software systems, the most important characteristics are: reliability, configurability, and usability. In cloud based systems the most important characteristics are: scalability, security of the processed data, and performance.

Cloud computing is not a completely novel paradigm but evolved from service-oriented computing (SOC), and shares most of its properties. SOC is a computing paradigm that exploits both web services and service-oriented architecture (SOA) as fundamental elements for developing software systems that use remote services. This paradigm changes the way software systems are designed, delivered, and consumed. The service-oriented paradigm is emerging as a new way to engineer systems that are composed of and exposed as remote services for use through standard protocols.

One of the important advantages of for cloud computing is its encouragement to reuse assets. Asset reuse can constitute reuse of computing models, reuse of architecture and infrastructure, and reuse of platforms and services.

Cloud computing provides a new criterion for service provisions from scratch, with reduced upfront investment, expected performance, high availability, fault tolerance capability, and "infinite" scalability, and opens a new era for freshly spawned companies to provide services at humble initial cost investment.

1.3 Some Useful Definitions to Characterize Services

In cloud computing, services are often characterized along their functionality, i.e., their set of provided operations. Moreover, service providers and consumers negotiate the quality-of-service targets that these operations need to achieve—so-called service-level objectives (SLOs). SLOs define a system's capacity: the maximum workload a system (with all of its operations) can handle, e.g., in terms of the number of customers.

Operations, SLOs, workload, and capacity are at the core of the CloudScale method: these concepts characterize a service and, thus, need to be taken into account during its engineering. In this section, we define and exemplify operations, SLOs, workload, and capacity. Subsequent chapters detail how software architects can engineer—along the CloudScale method—services with these issues in mind.

1.3.1 Operations

A service typically offers several operations (see Definition 1.1). Each operation represents a unique way of interacting with the service. Several words are more or less synonyms for operation: activity, function, process, and even transaction. On a higher level, a use case will often be implemented by one or more operations.

DEFINITION 1.1: OPERATION [5, BASED ON]

An operation specifies the name, type, parameters, and constraints for invoking an associated behavior.

An electronic book store service will, for example, have operations for accessing the home page as well as other operations for searching for books to buy. In the search operation, the name of the author or the title may be parameters. The operations in a service may be more and less demanding, and also the popularity of operations may differ drastically. An operation for buying books may depend on many complex underlying services and will therefore be demanding. On the other hand, retrieving information about a particular book is simpler, especially since this information normally does not change and therefore is easy to cache.

1.3.2 Service-Level Objectives

SLOs are the quality-of-service targets of a service's operations (see Definition 1.2). To specify an SLO, service providers and consumers need to agree on a suitable metric and a threshold for this metric. The metric should be "a defined measurement method and measurement scale" [6], and the threshold the lower limit for which a metric measurement violates the SLO. Violations of SLOs potentially lead to contractually specified penalties, e.g., the consumer may get discounts for service usage.

Definition 1.2: Service-level objective (SLO) [7, based on]

The quality-of-service target that must be achieved for each of a service's operation.

Let us exemplify an SLO negotiation. A service provider and a consumer negotiate an SLO for performance of a particular service operation. The consumer suggests the performance metric "maximum response time" and a concrete threshold of "2 s". The service provider disagrees to guarantee 2 s response time in 100% of the time. Therefore, the provider suggests a refined threshold that asks for "2 s for 99% of monthly requests". The consumer agrees, thus resulting in the performance SLO: "the offered service responds with a maximum response time of 2 s for 99% of requests in a month". Analogously to this negotiation, each service provider and consumer determine SLOs for each of a service's operations.

1.3.3 Workload

Given a service, its quality of service may differ depending how many consumers are using it concurrently. Therefore, the behavior of customers is a context factor that needs to be taken into account, e.g., when negotiating SLOs and when engineering a service. The concept of workload (see Definition 1.3) allows software architects to characterize this context factor.

Definition 1.3: Workload [8]

Workload is the combined characterization of work and load where

- work is the characterization of the data that is processed by a service's operations and
- load is the characterization of the quantity of consumer requests to a service's operations at a given time.

Workload characterizes the usage context of a service regarding two distinct aspects—work and load—over its operations. Work characterizes the data to be processed by a service's operations. For example, a typical characteristic is the array size when an operation processes an array, e.g., for sorting its elements. Load characterizes the number of consumers served by a service's operations. For example, typical characteristics are the frequency and probability with which consumers request an operation.

1.3.4 Capacity

Given the concepts of SLOs and workload, the question arises whether a service can sustain an unlimited amount of workload or whether there is a limit. The concept of capacity (see Definition 1.4) captures this limit.

DEFINITION 1.4: CAPACITY [9]

Capacity is the maximum workload a service can handle as bound by its SLOs.

For example, a web service for providing information about books may have a capacity of 100 consumers per second with a constant work. The limiting factor that determines this capacity may be a CPU with too low processing rate or a too strict SLO. Therefore, both increasing CPU processing rate and agreeing on less restrictive SLOs can be options to increase capacity.

1.4 Quality Properties of Services

Quality properties characterize "how well" a service operates. From the view of service consumers, externally observable properties are important—i.e., the degree to which SLOs are fulfilled and the involved usage costs. From the view of service providers, internal properties are important—i.e., the degree to which SLOs can be fulfilled and the involved operation costs.

Scalability, elasticity, and cost-efficiency are the internal quality properties that service providers need to consider to minimize operation costs while fulfilling SLOs as best as possible. They are the only quality properties focused in this book. This section describes and exemplifies these properties. Subsequent chapters explain how software engineers follow the CloudScale method to engineer their services for these properties.

1.4.1 Scalability

Scalability is a quality property that tells whether a service can increase its capacity by consuming more services of its underlying layers or not (see Definition 1.5). Here, only this ability is important—not the degree to which it does.

Definition 1.5: Scalability [9]

Scalability is the ability of a service to increase its capacity by expanding its quantity of consumed lower-layer services.

Examples for underlying services are third-party services (e.g., a payment service for web shops) and directly consumed resources (e.g., servers, CPUs, and hard disk drives). Given a service that consumes all of these lower-level services, it is scalable if an increased consumption of at least one underlying service leads to an increased capacity. That is, consuming either more third-party services (e.g., by issuing more parallel requests to the payment service) or more direct resources (e.g., by using more servers) leads to an increased capacity.

An example of an unscalable service is a service where many consumers share items of the same database. Whenever a change of a single item is made by a consumer, this part of the database is locked. Changes made by other consumers on this item cannot be processed until the first consumer is finished. The capacity of such a service stays at one customer per request, independent of the number of additional database servers, CPUs, etc. To make such a service scalable, alternative means to manage database items have to be found, e.g., by using alternative databases with less restrictive constraints on data consistency or by assigning database items to dedicated users only.

1.4.2 Elasticity

Elasticity describes to which extent a service can adapt its capacity to changes in workload (see Definition 1.6). Elasticity needs to be considered over time for changing workloads, e.g., for sudden workload peaks that require an adaptation of capacity. Such an adaptation needs to be timely, i.e., such that potential SLO violations are minimized. Timeliness entails an adaptation process that is autonomous, i.e., it is either automated or guaranteed to be manually realized in time.

Definition 1.6: Elasticity [9]

Elasticity is the degree to which a service autonomously adapts capacity to workload over time.

Based on this definition, a service needs to be able to adapt its capacity to be elastic. Because exactly this property is captured in scalability, scalability is a prerequisite for elasticity. For example, a service is elastic if it dynamically boots a dedicated virtual machine (VM) that copes with work-intensive requests (and shuts it down once the request has been served).

The benefit of an elastic service is that, at each point in time, only the minimal amount of underlying services are used (in the example above: a minimal amount of virtual machines). This minimization improves the cost-efficiency of the service as discussed in the next section.

1.4.3 Cost-Efficiency

Cost-efficiency for a service describes the relation between the amount of capacity demanded and the amount of consumed lower-layer services (see Definition 1.7). A cost-efficient service uses cost-efficient lower-layer services to deliver the required capacity.

DEFINITION 1.7: COST-EFFICIENCY [9]

Cost-efficiency is a measure relating demanded capacity to consumed services over time.

Cost-efficiency is closely related to optimal provisioning. Optimal provisioning strikes a balance between over-provisioning and under-provisioning. Under-provisioning results in SLO violations like high response times or low throughput, resulting in dissatisfied customers. Over-provisioning leads to low utilization of lower-level services [10]. However, since lower-level services also have a cost, optimal provisioning is not enough. For a given service with a given workload, two lower-level services may both provide optimal provisioning, but their cost may differ. In addition to optimal provisioning, cost-efficiency therefore also is about selecting cheap lower-level resources.

With a variable workload, the cost-efficient amount of lower-level services will vary. A service which cannot adjust its amount of lower-level services as the capacity demand varies will have poor cost-efficiency. Therefore, elasticity is a prerequisite for cost-efficiency. However, also with a constant workload, a poor match between a service and its demanded lower-level services can result in the cost being too high, and far from optimal. As a result, cost-efficiency will be poor.

An example of a service which is not cost-efficient is a service that relies on a lower-level costly database service. Redesigning this service so that it uses a less expensive database service may considerably improve the cost-efficiency. This saving must be traded off against the cost of redesigning the service.

1.5 Consequences of Scalability, Elasticity, and Cost-Efficiency Issues

As described in Sect. 1.4.1, a scalable system can handle an increasing workload by consuming more lower-layer services. Therefore, it is important to know the maximum workload our service must handle and design, implement, and tune the service accordingly. A service which is not able to scale up to the required workload will result in:

Unsatisfied end-users End-users will experience long response times, and ultimately, no response at all. If there are alternatives, disappointed end-users will go to your competitor.

Frustrated employees To handle a service with poor scalability will be challenging for all personnel involved: from the help desk answering frustrated or even angry end-users, to service engineers responsible for handling operations as best as possible, and, finally, to architects and developers working overtime trying to fix the service. When they come home late at night, their spouses may wait with divorce papers, before they die due to work-related stress.

Financial loss Revenue may plummet if customers go elsewhere because of poor SLOs. The organization responsible for such a service will get a bad reputation and may find it hard to get new customers or to recruit qualified personnel. To redesign a non-scalable service will take time. During this time, quality will still be bad, and thus you lose even more customers. Redesigning services is costly and you may end up using personnel which was actually meant to design new, profitable functionality instead of redesigning old functionality. A service with poor scalability may also require costly lower-level services. You may actually end up paying more for the lower-level services than what you earn from your customers.

Loss of lives In the case where critical public infrastructure stops working because of missing scalability, lives may eventually be lost, because of overloaded services for telecom, hospitals, air traffic management, etc.

We now assume that our service is scalable, but that it is not elastic enough. When the number of users grows quickly or if the amount of work they perform also increases fast, this challenges the elasticity of the service. A service which is not elastic enough results in poor SLOs. We will face the same consequences as for bad scalability. The consequences will not last for so long, since eventually we will be able to provision the required lower-level services.

A service which is not cost-efficient may still offer acceptable quality to its end-users. The problem is the large bill from the lower-level services. Consequences of this may still be severe, but limited to the organization offering the service. Costly redesign may also be required.

1.6 Causes of Scalability, Elasticity, and Cost-Efficiency Issues

A very scalable service can scale up to infinite work and load, provided we supply enough lower-level services. Why not engineer all services with high scalability? The answer is simply that designing scalable software has several consequences:

Cost A software design organization has a typical way of making services which gives a certain scalability. If more scalability is required, then this will not only cost twice as much. It may cost ten times more, and will also require more time, since new design solution must be explored.

Non-standard solutions New solutions may be required, contributing to more complex development, testing, maintenance, and operations.

Other qualities suffers It is well-known that scalable services may be more complex to maintain, but if a scalable service is required, this may also limit the amount of security which is possible to implement. With a fixed budget, a costly scalable service may also miss functionality compared to a less scalable service.

In summary, scalability is a trade-off between other qualities. A good trade-off gives services with balanced qualities, but if scalability is neglected during architecture, design, testing, and operations, this will have consequences as outlined in Sect. 1.5.

Scalability is holistic and will be affected by the weakest part of the system. Obviously, a bad architecture will lead to bad scalability, but also sloppy coding or uniformed choice of configuration parameters may lead to scalability problems. The large difference is that the latter is much cheaper and faster to fix. However, for a solution which simply has to work on a specific occasion, the damage is already done.

The causes of inferior elasticity and cost-efficiency may resemble the causes of bad scalability.

1.7 How Should You Manage Scalability, Elasticity, and Cost-Efficiency?

As we saw in the previous sections, it is important to address scalability, elasticity, and efficiency requirements. Failure to manage these requirements properly may lead to severe losses of business opportunities, money, or even human life. Hence, it is important to manage them early on and throughout the system lifecycle in the same way as is necessary to manage other (quality) requirements.

Due to this necessity, we have to talk about managing scalability engineering in software development projects. In the beginning, the management tasks start with either training existing personnel or employing skilled new personnel. Required

skills include handling of extra-functional properties, gathering extra-functional requirements correctly, analyzing systems and their designs for requirement fulfillment, etc.

It is necessary to integrate steps dealing with scalability, elasticity, and efficiency requirements into the software development process. This does not imply only technical activities like system design and implementation, but also business-related activities like finding, evaluating, and selecting an appropriate cloud computing provider. As cloud providers differ significantly in the offered price model as well as in technical parameters like startup times of provisioned virtual machines, this has a direct impact on properties like elasticity and cost-efficiency.

In most cases, managing scalability, elasticity, and efficiency requirements requires good communication between the customer, the developers, and the operations team. Stakeholders need to understand concepts sufficiently so they can discuss and decide on the requirements in an educated way. It also may include establishing a new development process or culture, as is currently propagated in the DevOps movement [11]. Here, developers and operations people work closely together to achieve high-quality services. In particular, implementing sophisticated elasticity requirements requires a close cooperation between developers and operators; the reason is that developers need to implement monitoring and reporting of the current work and load situation faced by the running system. The information is used at runtime by operators, who then, based on the delivered data, provision or deprovision resources.

Finally, managing scalability, elasticity, and efficiency requirements also has an impact on time and cost estimations for development projects of cloud computing applications. The more complex and critical these requirements are, the more effort will be needed.

1.8 Reactive Scalability, Elasticity, and Cost-Efficiency Management

Solving problems on running systems is much more complicated and expensive than tackling the scalability, elasticity, and efficiency from the start. Time spent in a good architectural design and careful implementation will save several times more effort in solving problems once a system is implemented and in production.

Nevertheless, it is not seldom that we find ourselves in a situation in which we need to find a solution for a running system that is struggling to process an existing load, or has problems when peaks occur, fails to respond quickly enough to peaks, or simply becomes too expensive to make commercial sense.

1.8.1 Immediate Temporal Solutions

There are a few things that we can do to alleviate the system stress to cope with the load if horizontal scaling, i.e., introducing more instances of the service, is not

helping due to lack of scalability. The most obvious being vertical scaling: having instances with more RAM, CPUs, or disk space can bring the system to normal functioning without any other change (provided we understand what produces the bottleneck). This action can help to keep the system operational while more permanent solutions are worked out.

Elasticity problems can be reduced by over-provisioning, though damaging the cost-effectiveness of the system. This is particular common when there are replicable components that are very expensive to replicate (e.g., huge databases).

In multi-tenant systems it is sometimes possible to divide the information that is stored in independent copies of the system, routing the requests to the appropriate deployment. This is only possible if the information from different tenants is independent of each other.

Other solutions include making use of an external service implementing a particular component (e.g., a static storage from a public PaaS provider) or throttling the requests while reducing the SLOs.

1.8.2 Long-Term Solutions

Any permanent solution to a system that does not scale beyond a certain point, or does not do it quickly enough, will require an analysis of the problem, and probably a redesign, reimplementation, and redeployment of some or several of the system's components. The first step in solving these problems is that of the measurement and analysis of the system's behavior by means of dynamic measurements (e.g., with the Dynamic Spotter introduced in this book and an appropriate measurements methodology; cf. Sect. 7.4), the static code (e.g., the Static Spotter introduced later; cf. Sect. 7.3) and model simulation (the Analyzer introduced later; cf. Sect. 6.3). This will help us understand when and, more importantly, where do the bottlenecks occur, in order to plan for a course of action to solve them.

Often, these issues arise due to deployment problems and misconfigurations, which can be solved relatively easy. A bad gateway, a low connection pool size, or a poor load-balancing policy can greatly affect the performance, scalability, and efficiency of a system, but can be solved without the need of costly refactoring.

Poor technology choices (e.g., the usage of an embedded database) can also have a great impact on the system, but replacing them (e.g., to a distributed database) can require costly changes in the code (e.g., SQL flavor) and, in some cases, complete redesign of the entire solution and rewrite of a large part of the system (e.g., if moving from a relational database to a key-value store).

Finally, a bad design and architecture can result in an inherently poor system that will suffer efficiency and scalability issues sooner than later, and might require a mostly new system from scratch. Common pitfalls in the design and development of cloud systems can be found in Sect. 2.10.

Any solution that requires a revision of the architectural design can be validated prior to its actual implementation, e.g., by means of analyzing its blueprint and

simulation of foreseeable scenarios, to then start its actual implementation, and its implementation be tested and measured before being deployed in production to dephase the temporary solution that has hopefully been able to cope until now.

1.9 Proactive Scalability, Elasticity, and Cost-Efficiency Management

Because solving scalability problems on a running system is costly and complicated, it is essential to try to minimize them already during the design and implementation phases. There are several things that we have to keep in mind that can help us avoid common mistakes and minimize future scalability problems.

During the design of the system architecture, it is very useful to make use of existing architectural models (i.e., Architectural Templates) and follow best practices (i.e., HowTos) with proven practical scalability properties, while avoiding common pitfalls and bad models (i.e., HowNotTos) that will eventually translate into run-time problems. This is particularly helpful because it allows us to calculate the cost-efficiency of very different workloads, different architectures and deployments, and even different infrastructure providers in order to find the best combination of those, without incurring in performing many expensive test runs and measurements.

Perhaps the most general advice is to decouple independent units or sub-systems that can be scale independent, which helps not only in detecting scalability and contention problems if they arise, but will also help improve efficiency (by using just the right amount of each) and the scalability, by isolating inherently hard-to-scale parts, such as a database. This is also one of the essential ideas of the current movement to microservice-based architectures [12].

Additionally, it is useful to make use, whenever possible, of an eventually consistent approach to data instead of a transactional model, since it has a much better scaling expectation since it avoids the contention problems present in transactional models, which require a distributed commit in order to assure constant coherence of the information present in a database cluster.

Once we have a design, we can then analyze it for different work and loads in order to evaluate the scalability properties of our future system (detailed in Sect. 6.3). Even if certain properties of the infrastructure or external systems might not be yet known, such simulations can help design problems very early on in product development, which can save a lot of headaches, money, and resources later, when the system is fully developed, deployed, and running. When one such problem is detected, it is necessary to update the design and repeat the cycle.

It is also important to make measurements during the implementation of the system, which will help detect efficiency and scalability problems as early as possible. This should be part of the continuous deployment and testing process that validates the quality of the entire system.

These tests should be performed in infrastructures and platforms analogous to the final product infrastructure, in order to prevent unpleasant surprises later on. This includes the expected range of work and loads, underlying static storages, database clusters and any other such PaaS elements, computing nodes, virtual disks and other IaaS items, and elasticity settings. Likewise, it is important to stress-test any necessary external service to confirm its ability to cope with the expected load and defined SLOs.

The earlier problems are found in the design and development phases, the easier and cheaper it is to tackle them. A problem found during design can be hundreds or even thousands of times cheaper to treat than in a system already running in production, without taking into account even opportunity losses.

1.10 The CloudScale Method

To engineer scalable, elastic, and cost-efficient systems is a complex undertaking.

The CloudScale method guides stakeholders by describing the steps to follow when engineering scalable, elastic, and cost-efficient systems, and also describes when different stakeholders are involved. The basis for the CloudScale method is the CloudScale tools, which automate some parts of the method. The CloudScale Method describes the input as well as the output of the CloudScale tools. In some cases, inputs to a CloudScale tool are produced with another CloudScale tool, but manual steps may also be required to produce the required inputs. The CloudScale method describes the sequence of manual and tool-driven steps required when engineering scalable, elastic, and cost-efficient systems using the CloudScale tools.

In a typical cloud-based system, scalability, elasticity, and cost-efficiency are only some of the requirements to take care of. The CloudScale method must therefore be embedded in a wider method where the functions of the system are established and detailed. This wider method will take care of the functionality as well as other extra-functional requirements like security, usability, and maintainability. The CloudScale method is flexible concerning this wider method.

The CloudScale method covers two distinct use cases. These use cases are distinguished by the key artifact, being either a *model* of a system or an *implemented* system. A model typically represents unimplemented parts of a system. When a model is sufficiently detailed, it will be possible to project scalability, elasticity, and cost-efficiency. A model must be built, feed with inputs, and interpreted. The CloudScale method describes these steps.

An implemented system can be tested using different workloads, and it may then be possible to identify scalability problems in the system. Moreover, it may also be possible to find the weak spots in the system that are the root cause of these scalability problems. When investigating an implemented system, a mixture of tool-driven and manual steps are described in the CloudScale method so that root causes are identified. A fully detailed description of the CloudScale method follows in Chap. 2.

1.11 What Does It Cost?

When considering to introduce the CloudScale method, managers typically want to know how much it costs to introduce it into their development processes. They then have to compare these costs to the expected benefits to make an educated decision. To better judge the cost/benefit ratio of the CloudScale method, the following paragraphs detail the two main cost categories.

Costs come as one-time costs for the initial introduction of the CloudScale method and as per-execution cost for each project using the CloudScale method. For introducing the CloudScale method, the major cost driver is education of the people who are to use the CloudScale method. In principle, there are two options: either train already existing personnel or employ/hire persons who already have appropriate skills. Training the necessary skills on a very basic level during a guided workshop takes about 2–3 days. More details with respect to these costs are described in Sect. 8.4.3.

After appropriate personnel has been trained, costs for executing the CloudScale method occur on each application of the method during a particular software development endeavor. These costs depend on the skills of the personnel (as discussed before), the experience they have, and the situation the software development project is in.

The first and important aspect is to gather the systems requirements. In practice, for many systems, requirements for non-functional properties have not been sufficiently detailed in a quantitative manner. However, this is a precondition for the CloudScale method. Therefore, you need to plan for costs to collect and define these requirements in a more detailed way. In particular, requirements for scalability, elasticity, or cost-efficiency do not exist for classical software systems, as their importance increased only when cloud computing applications became mainstream. In our experience, gathering such requirements can take a couple of months as it involves quite a number of stakeholders who have to meet and discuss. This incurs some costs. However, you should have gathered these requirements anyhow, no matter whether you plan to use the CloudScale method or not.

Benefits of knowing the extra-functional requirements in a detailed and quantitative way allows for systematic designing for them and testing them. In so doing, software architects can avoid risks which exist if the software is underperforming during operation. In particular, for mission-critical systems, i.e., systems which have to be available all the time, avoiding such risks is essential.

For model creation, an existing software architecture description which is up to date is an important speed-up factor. Otherwise, software architecture reverse engineering approaches like the Extractor tool are needed (cf. Sect. 4.5.3.2). They often require specially prepared inputs (e.g., code in a certain format) as well as configuration by their users—often identified in a trial-and-error procedure.

Measurement data from preexisting software is useful. It can help in identifying the system's current and expected workload, its current and future resource demands, etc. In addition, software architects can use measurements to characterize

their infrastructure layer (e.g., VM provisioning delay and basic performance measures like the system speed of processing). Again, the real costs are situation dependent. Overall, the data gathering phase might take up to half a person-year—depending on the situation and the system's complexity.

The benefits of having a model in the end are manifold. First, having a model has its use as documentation, even without doing any kind of analyses. Having such a documentation eases any kind of maintenance task. In addition, models allow for analyses and predictions, e.g., via simulations. In particular, models enable software architects to perform what-if analyses and evaluate the system under development in different environments or settings, as explained in Chap. 6.

When validating implemented systems via CloudScale's Spotter tools (cf. Chap. 7), the system has to be readily implemented and executed. In addition, a workload driver has to stress-test the system in various settings. Therefore, costs arise from installing and operating the system. Installation costs originate mainly from the effort spent by operations personal. System operational costs are costs you have to pay for used cloud resources (virtual machines, network resources, storage resources). These costs can be significant as the Spotter tools drive the system to its scalability limits, which means also to a state in which it should consume the most resources. As spotting problems may take quite some time (even in the magnitude of days), the sum of operational costs can quickly become a significant factor.

1.12 What Do You Need?

There are a bunch of questions to clarify, before you can begin to introduce the CloudScale method into your own development processes. First of all, introducing the CloudScale method is not a matter of downloading, installing, and using the CloudScale tools bundled in its integrated development environment (IDE). As outlined in Sect. 1.11, one of the main factors is to have appropriate personnel that has the right skill set, i.e., trained in modeling and system analyses.

No matter, whether you hire suitable candidates or train them yourself, all of this requires management commitment. Your management has to be convinced of the benefits the CloudScale method has to offer and be supportive of it. This support covers providing all necessary resources: the time and money to train or establish required skills and the time and money to model or otherwise analyze systems. In addition, management should foster and encourage the use of the CloudScale method. This is of major importance as the CloudScale method promotes activities which are not directly related to common development activities like programming. In addition, it shifts efforts from late development phases, in particular system staging tests, to early phases. The reason for this is that application of the CloudScale method aims at avoiding surprises due to late discovery of insufficient scalability. However, late discovery of such issues may lead to very expensive reengineering of the system under development. In this light, you may consider

the CloudScale method to be a risk mitigation technique: it adds additional efforts in early development phases to avoid costly and dangerous risks in late phases.

Besides management commitment, you also need to integrate the CloudScale method into your existing development methods and culture. This requires a good understanding of your current development process and sufficient control of it. Only then can you understand what a combined process could look like for your organization. Once defined, you also need to implement this process in your organization. When doing this, you may encounter resistance from your developers, who will now be forced to use methods and techniques unfamiliar to them. Again, management commitment in this phase is very important to overcome such issues. Also, a good organizational internal communication is needed to explain the new method and its benefits.

To summarize, the benefits of the CloudScale method do not come for free, and not just using a set of novel tools. It requires rethinking your development process and integrating new steps with appropriate management backup and sufficient resources. In the end, it is a decision you have to take: risk late project failures due to insufficient scalability or invest in your processes to avoid these risks to a large extent. In the web-scale economy of many of today's systems, the severe impact of failing to meet scalability often justifies introducing the CloudScale method.

1.13 Conclusion

As industry adoption of cloud computing is rising, building properly scalable cloud systems will become more and more critical. In order to achieve such systems, one must have a holistic approach regarding the systems requirements, i.e., scalability, elasticity, and cost-efficiency requirements must be considered in the system engineering process. An example of such engineering approach is offered through the CloudScale method, thoroughly described throughout this book.

As an increasing number of both service and infrastructure providers are ready to invest in proper service engineering, they recognize that achieving quality of service in cloud environments has become an imperative. In order to track the service quality, a service needs to be characterized in terms of operations it offers, SLOs specified as quality-of-service targets, and workload. Service can then be described using their internal quality properties—scalability, elasticity, and cost-efficiency—defining if the service can increase its capacity, to which extent can the capacity be adapted to the changes in the workload, and if it can be done in a cost-efficient way. Lack of service scalability can lead to financial loss, disappointed end-users, and even some fatal circumstances, depending on the service type. Hence, service providers examine these properties in order to achieve specific level of quality of service while minimizing service operational cost.

In order to avoid consequences of bad system scalability, risking poor SLOs, and costly solutions, the scalability, elasticity, and cost-efficiency requirements should be managed early on and during the whole system lifecycle. By using new

engineering methods that promote and focus on proactive scalability, elasticity, and cost-efficiency management during the service design and implementation phases, service providers can significantly reduce the need of complex and costly solutions of scalability problems that need to be applied on the running system. However, although they offer significant benefits and decrease the risk of late project failures, applying methods such as the CloudScale method requires additional efforts and their introduction into the current development process has to be carefully considered and planned.

References

1. Web-scale IT: http://www.webopedia.com/TERM/W/web-scale-it.html (2016) [Visited on 11/13/2016]
2. Marshall, R.: Sap gives update on business bydesign plans. http://www.v3.co.uk/v3-uk/news/1970547/sap-update-business-bydesign-plans (2016) [Visited on 11/13/2016]
3. Battle.net Servers Down for World of Warcraft, Hearthstone and Overwatch Players Around the World: http://www.ibtimes.com/battlenet-servers-down-world-warcraft-hearthstone-overwatch-players-around-world-2383254 (2016) [Visited 11/13/16]
4. The NIST Definition of Cloud Computing: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (2016) [Visited on 04/18/2016]
5. UML Operation: http://www.uml-diagrams.org/operation.html (2016) [Visited on 10/18/2016]
6. Cloud Select Industry Group on Service Level Agreements Subgroup (C-SIG SLA): Cloud service level agreement standardisation guidelines. Cloud Select Industry Group (C-SIG), Technical Report (2014)
7. Gartner, Inc. and/or its Affiliates, The gartner glossary of information technology acronyms and terms. Gartner Inc., Technical Report (2003)
8. CloudScale Wiki: Glossary: http://wiki.cloudscale-project.eu/Glossary (2016) [Visited on 12/19/2016]
9. Lehrig, S., Eikerling, H., Becker, S.: Scalability, elasticity, and efficiency in cloud computing: a systematic literature review of definitions and metrics. In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, ser QoSA '15, Montreal, QC, pp. 83–92. ACM, New York (2015) [Online]. Available: http://doi.acm.org/10.1145/2737182.2737185
10. Brataas, G., Stav, E., Lehrig, S., Becker, S., Kopcak, G., Huljenić, D.: CloudScale: scalability management for cloud systems. In: Proceedings of International Conference on Performance Engineering (ICPE). ACM, New York (2013)
11. Bass, L., Weber, I., Zhu, L.: Devops: A Software Architect's Perspective, 1st edn. Addison-Wesley Professional, New York (2015)
12. Fowler, M.: http://www.martinfowler.com/articles/microservices.html (2016) [Visited on 11/13/2016]

Chapter 2
CloudScale Method Quick View

Gunnar Brataas, Steffen Becker, Mariano Cecowski, Darko Huljenić, Sebastian Lehrig, and Ivana Stupar

Abstract In this chapter, we overview the complete CloudScale method and show how the CloudScale method relates to existing development processes. Our overview is accompanied by a running example termed CloudStore—a simple online bookshop to be operated in a cloud computing environment. In a fictional scenario, we exemplify how a software architect follows the CloudScale method to realize CloudStore. The architect finally realizes CloudStore such that all of its scalability, elasticity, and cost-efficiency requirements are fulfilled. After having exemplified the CloudScale method, additional guidelines for software architects are given in the form of best practices (HowTos) and common pitfalls (HowNotTos). The chapter closes with a discussion on how the CloudScale method can be integrated into existing development processes such as the Unified Process.

This chapter is structured as follows. Section 2.1 overviews the process steps of the CloudScale method, and Sect. 2.2 introduces CloudStore as a running example. Afterward, the fictional scenario starts in which a software architect identifies critical use cases and key scenarios (Sect. 2.3), derives appropriate service-level objectives (SLOs) (Sect. 2.4), creates an architectural model of CloudStore

G. Brataas (✉)
SINTEF Digital, Strindvegen 4, 7034 Trondheim, Norway
e-mail: gunnar.brataas@sintef.no

S. Becker
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
e-mail: steffen.becker@informatik.uni-stuttgart.de

M. Cecowski
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: mariano.cecowski@xlab.si

D. Huljenić • I. Stupar
Ericsson Nikola Tesla, Krapinska 45, 10000 Zagreb, Croatia
e-mail: darko.huljenic@ericsson.com; ivana.stupar@ericsson.com

S. Lehrig
IBM Research, Technology Campus, Damastown Industrial Estate, Dublin 15, Ireland
e-mail: sebastian.lehrig@ibm.com

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_2

(Sect. 2.5), improves this model based on analyses (Sect. 2.6), implements CloudStore and resolves implementation issues (Sect. 2.7), and deploys and operates CloudStore in a cloud computing environment (Sect. 2.8). Afterward, Sect. 2.9 describes HowTos, and Sect. 2.10, HowNotTos. Finally, the CloudScale method is related to the Unified Process in Sect. 2.11.

2.1 Process Steps of the CloudScale Method

After briefly motivating the CloudScale method in Sect. 1.10, this section gives a high-level overview of the CloudScale method based on Fig. 2.1.[1] Figure 2.1 illustrates the control and data flow (denoted as arrows) for software architects who want to follow the CloudScale method. Various nodes denote a flow's start/end (rounded rectangles), processes supported by tools (rectangles with double-struck vertical lines), decisions for software architects (diamonds), and manual tasks (trapezoids).

The CloudScale method deals with scalability, elasticity, and cost-efficiency of a system and must therefore be embedded in a wider method where the functionality of the system is specified. The functionality of a service is expressed as one or more operations offered by the service. Therefore, the first step of Fig. 2.1 is to look at these operations and identify the most important operations from a performance point of view—these are the critical use cases. As a prerequisite for finding the most critical operations, software architects have to establish rough service-level objectives (SLOs) for these operations. Later, software architects must also estimate where, in the planning horizon, these critical use cases are most likely to be toughest—these are the key scenarios.

In the second step of Fig. 2.1, critical use cases and key scenarios allow software architects to define coarse requirements for scalability, elasticity, and cost-efficiency. Software architects formulate their requirements via SLOs (as described in Sect. 1.3).

The first decision node of Fig. 2.1 branches based on the type of artifacts to be studied. If the analysis shall be based on modeling, software architects go to the left, and if the analysis shall be based on implementation artifacts, software architects go to the right.

A model in the CloudScale method is an architectural model expressed in the Scalability Description Language (ScaleDL)—software architects have to specify such a model when following the left path in Fig. 2.1. ScaleDL is comparable to a Unified Modeling Language (UML) model with dedicated annotations for quality analyses, similar to UML's MARTE profile [4]. ScaleDL represents the structure and behavior of the system's architecture, the workload by the users, as

[1] The CloudScale method extends the Q-ImPrESS method [1] and builds on the Palladio performance modeling tool [2]. A first draft of the CloudScale method was introduced in [3].

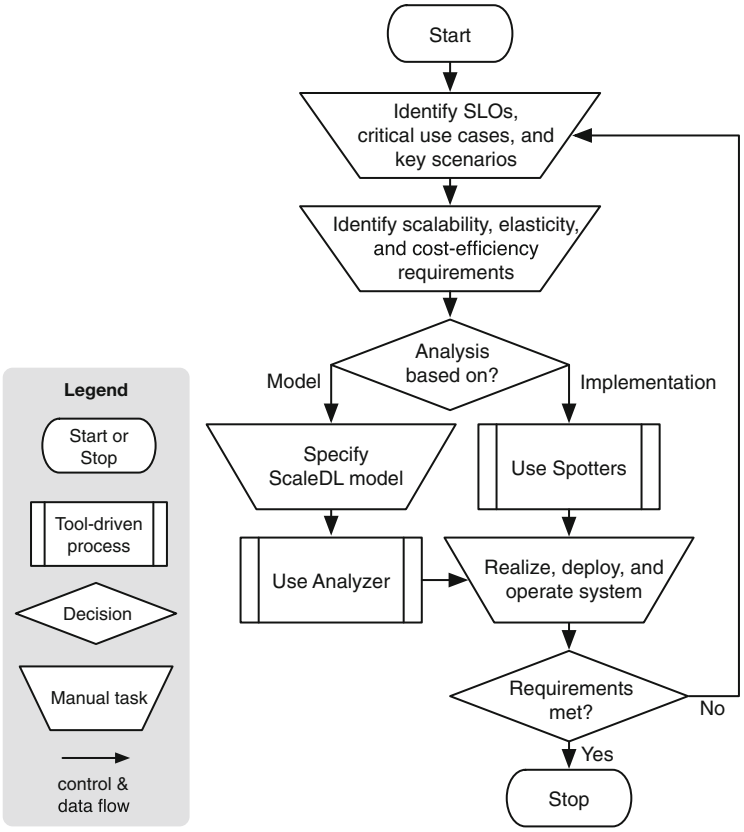

Fig. 2.1 High-level process steps of the CloudScale method

well as hardware and software resource demands. In addition, ScaleDL specifies how workload changes over time and how autonomous elasticity managers behave. ScaleDL is described in more detail in Chap. 4.

The ScaleDL model can either be a complete new design or a refinement of an earlier architectural model. Moreover, a model can be partially extracted from existing implementations via the so-called Extractor tool for reverse engineering. Partially extracted ScaleDL models must be completed manually inside the "Specify ScaleDL model" process step.

Modeled systems are analyzed in the "Use Analyzer" process step in Fig. 2.1. This analysis may reveal a perfect system, or one or more weaknesses. When these weaknesses have been corrected on the model level, the model can be analyzed again. Moreover, analyses may reveal that some of the requirements are hard to fulfill. Based on negotiations with the relevant stakeholders, alleviated requirements may then be derived and analyzed. In this way, important trade-offs between cost

and quality can be resolved before the system is put in operation where users potentially react furiously as a result of unsatisfying user experience.

Alternatively, if the analysis shall be based on implementation, software architects continue with the "Use Spotters" process step of Fig. 2.1, where anti-patterns are detected and handled using the so-called Spotter tool. The Spotter tool has two parts: the Static Spotter examines static code, while the Dynamic Spotter comprises instrumentation and load generation of a running service. In both cases, software architects reengineer the code for spotted anti-patterns by fixing their root causes.

The second decision node of Fig. 2.1 checks (based on analysis result of Spotter or Analyzer) whether scalability, elasticity, and cost-efficiency requirements can be sufficiently met. If met, the realization of the system can be completed and the CloudScale method stops. For new systems, software architects realize the service based on the architectural representation in the ScaleDL model. After realizing a service, the Static Spotter may be used for static spotting of anti-patterns in the code. For existing systems, software architects semi-automatically reengineer detected issues based on either Spotter or Analyzer suggestions. A realized system will be deployed. After deployment, you may use Dynamic Spotter on the deployed service, which may spot further anti-patterns.

The CloudScale Method has one feedback loop. If, during operations, new requirement violations arise, a new iteration of the method needs to be executed, again supported by dedicated detection tools. A result of the analysis may also be a relaxation of some of the requirements and the subsequent identification of refined critical use cases and key scenarios. If the system meets its requirements, software architects can stop the CloudScale method, and only have to reenter the method in case requirements or the system's environment change.

In later chapters, we refine some of the core steps in the method. These refinements will further introduce feedback loops.

2.2 Running Example

This section introduces a running example, which will—later in this book—be reused, revisited, and extended. The running example is termed CloudStore—an online bookshop to be deployed in a cloud computing environment. In that sense, one can think of CloudStore as a simplified variant of Amazon.

CloudStore's book-selling services have four core operations:

Home Page Provide the home page of CloudStore.
Search Look for a suitable book.
Shopping Cart Put a book in the shopping cart.
Pay Check-out the books in the shopping cart and get them shipped.

These core operations represent the basic functionality of CloudStore. Other operations for registering customers, order inquiry, etc. are also required but will not be considered now.

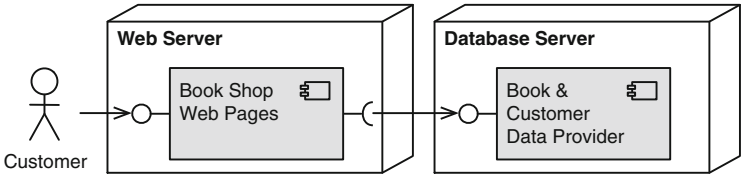

Fig. 2.2 Conceptual CloudStore architecture

CloudStore has two work parameters: number of customers and number of books. With more customers and more books, each operation in CloudStore will be heavier.

The overall structure of CloudStore is depicted in Fig. 2.2. A customer makes use of the CloudStore service via a web browser and most of the functionality is handled by a `Web Server`. This `Web Server` connects to a `Database Server` storing information about customers and the book items of the shop. For now, it is assumed that payment is handled internally by the `Web Server`.

2.3 Identify Service-Level Objectives, Critical Use Cases, and Key Scenarios

A software architect plans to realize CloudStore as introduced in Sect. 2.2. This section outlines how the software architect can start following the CloudScale method by identifying CloudStore's business-related requirements.

The software architect first establishes the SLOs and, afterward, the critical use cases or operations from a scalability, elasticity, and cost-efficiency point of view. The software architect also identifies the key scenarios where the load, as well as work, on this critical operation becomes highest.

2.3.1 Service-Level Objectives

As a first step, the software architect has to estimate rough SLOs for the four CloudStore operations introduced in Sect. 2.2. As described in Sect. 1.3, an SLO consists of a quality metric and a quality threshold for this metric. The software architect selects 90 percentile response times as a suitable metric. As a first approximation, the software architect wants the `Home Page` and the `Shopping Cart` operations to respond in 1 s for 90% of requests to these operations. The `Search` operation has to complete in 2 s, and the `Pay` operation has to complete in 5 s.

2.3.2 Critical Use Cases

Critical use cases correspond to the critical operations in CloudStore. The software architect inspects the four CloudStore operations described in Sect. 2.2 and assesses their risks:

Home Page Providing the home page will essentially be a read operation and does not require any writes to the database. Moreover, this read operation can easily be cached, and it is also not critical if the content is not completely up to date.

Search Searching for a suitable book is also a read operation, but the stock level must reflect the actual situation. Realizing this requirement should be trivial in a well-indexed database.

Shopping Cart Putting a book in the shopping cart involves some writing to the database.

Pay To pay for books in a secure way is clearly the most complex operation. This is a transaction that involves several different payment systems. Moreover, the `Pay` operation is mission critical. If the `Pay` operation goes wrong, involved stakeholders potentially lose money. Customers also get very upset if there are problems with payments, much more than with the other functionality of a website. The `Pay` operation thus has to work.

When considering these four operations, the `Pay` operation is clearly the most challenging. Even if the SLO for this operation, as described in Sect. 2.3.2, is most flexible, the software architect still considers this operation to be the most challenging. To reduce the complexity in later analyses, the software architect focuses on this operation.

2.3.3 Key Scenarios

To establish key scenarios for scalability, the software architect first establishes the length of the planning horizon. The software architect uses a 3-year planning horizon for CloudStore's services.

Subsequently, the software architect must find the most critical point in the planning horizon. The software architect inspects when in the 3-year planning horizon CloudStore's requirements become toughest. As already established in Sect. 1.3, work, load, and quality thresholds for CloudStore's quality metrics are the key ways of characterizing a service in terms of scalability, elasticity, and cost-efficiency.

When it comes to load, the load of an operation often has seasonal variations. These variations typically span several time scales. The software architect identifies where in these seasonal variations CloudStore will have the highest load:

Yearly variation Most customers approach CloudStore just before Christmas.

Monthly variation Most customers approach CloudStore at salary, social security, and pension days, which is often the 20th of every month.

Weekly variation Most books are bought on Mondays. Later in the week, the sale gradually drops until it reaches a new peak on Mondays.

Daily variation Requests peak at noon, i.e., during lunch break.

In addition, the software architect identifies an exponential, yearly increase trend in the 3-year planning horizon. Therefore, based on the seasonal variation as well as the trend, the highest load on the Pay operation is expected to happen at noon on a Monday just before Christmas in the third year. The software architect assumes that this situation will also be a critical scenario for work and regarding CloudStore's SLOs.

Elasticity concerns the ability of CloudStore's services to handle sudden increases in workload while still fulfilling CloudStore's SLOs. When establishing key scenarios for elasticity, the software architect therefore considers sudden changes in load and work. The software architect first looks at sudden changes in load. The arrival of a new best-seller book typically leads to a peak in load.

When it comes to the two work parameters described in Sect. 2.2, the number of customers is connected to the number of payment operations. The software architect consequently inspects key scenarios regarding the number of books. The number of books can suddenly grow when customers buy many new books at once. Realistically, the software architect expects this to happen when CloudStore buys books from a new publisher.

SLOs must also be handled at an acceptable operational cost. The software architect therefore considers key scenarios for cost-efficiency. To be on the safe side, the software architect could use the same large deployment configuration throughout the day. However, CloudStore will not take advantage of scaling down during the night or at other times when CloudStore has less than peak load. In our example, the cost-efficiency requirements will be as follows: we take the cost required for handling the maximum daily load and divide it by 2. On average, the cost shall only be half as much as the cost required to satisfy the maximum daily load. With this cost-efficiency requirement, the simple option of using the same configuration at all times is no longer feasible.

2.4 Identify Scalability, Elasticity, and Cost-Efficiency Requirements

The software architect next derives technical requirements from the business-related requirements of Sect. 2.3. Of course, the software architect must ensure that the functionality and non-functional requirements like security and maintainability are fulfilled. However, motivated by the focus of this book, the software architect focuses on scalability, elasticity, and cost-efficiency requirements.

As described in Sect. 1.3, load and work are key service properties from the point of view of scalability, elasticity, and cost-efficiency. In this section, the software architect therefore determines the expected load and work for CloudStore.

2.4.1 Scalability Requirements

In Sect. 2.3.3, the software architect has decided that the planning horizon is 3 years long. For this planning horizon, the software architect estimates the largest load and work and the toughest quality thresholds for CloudStore's quality metrics. The software architect's estimates then allow to specify scalability requirements.

The software architect has already identified that

- the `Pay` operation is the most complex (cf. Sect. 2.3.2),
- the 90 percentile metric with a threshold of 5 s is used for the `Pay` operation (cf. Sect. 2.3.1), and
- the highest load on the `Pay` operation is expected to happen at noon on a Monday just before Christmas in the third year (cf. Sect. 2.3.3).

Based on experience and available data, the software architect next estimates the maximum load at this particular time. Historical data and the software architect's best business-forecasting tools indicate that the load will be 1000 simultaneous customers for the `Pay` operation.

CloudStore has two *work parameters*: number of books and number of customers. The software architect expects the total number of customers to be proportional to the load. Based on experience, this number is 1000 times the number of simultaneous `Pay` operations, which gives 1,000,000 customers in total. When it comes to the work parameter "number of books", the software architect simply expects a gradual growth. This means that CloudStore has the highest number of books at the end of the planning horizon. Here, the software architect expects 1,000,000 books indexed in CloudStore's database.

In summary, the software architect's scalability requirement can be formulated as the ability to handle 1000 simultaneous users on the `Pay` operation with 1,000,000 customers and 1,000,000 books. Here, handling refers to CloudStore's SLO—i.e., `Pay` fulfills the less-than-5 s 90 percentile response time SLO. The software architect will later investigate this scalability requirement via modeling.

2.4.2 Elasticity Requirements

Elasticity concerns the ability of CloudStore's services to handle sudden increases in workload while still fulfilling CloudStore's SLOs. The software architect first investigates elasticity requirements for sudden changes in load.

The arrival of a new best-seller book typically leads to a peak in load (cf. Sect. 2.3.3). During such a scenario, the software architect expects that CloudStore must handle 100% more payment operations compared with what it must normally handle. In Sect. 2.4.1, the software architect identified a maximum of 1000 simultaneous customers on the `Pay` operation. Based on this maximum, CloudStore clearly must handle a peak of 1000 simultaneous customers on the `Pay` operation. However, the software architect also needs some estimation of how fast this peak will build up. The software architect guesses that, during a period of 1 min, the number of simultaneous `Pay` operations can grow from 500 to 1000. The software architect therefore formulates the elasticity requirement that CloudStore must be able to handle such load variations.

When it comes to the work parameter "number of customers", this requirement is connected to the number of payment operations by the factor 100. That is, CloudStore must be able to go from 500,000 customers to 1,000,000 customers within 1 min.

The number of books can suddenly grow when CloudStore provisions many new books at once. As outlined in Sect. 2.3.3, the software architect expects such a situation when CloudStore starts to buy books from a new publisher. CloudStore will then provision 100,000 new books at once. The software architect particularly expects that this number of books will be added during 1 week. Compared with the increase in terms of both load and customers, the increase in the number of books is so slow that the software architect can simply ignore this increase when considering elasticity requirements.

In summary, the software architect formulates the elasticity requirement that CloudStore must be able to handle both an increase from 500 to 1000 simultaneous `Buy` operations and an increase from 500,000 customers to 1,000,000 customers within 1 min.

2.4.3 Cost-Efficiency Requirements

Cost-efficiency covers the cost for handling CloudStore's workload while fulfilling its SLOs. In Sect. 2.3.3, the software architect formulated the cost-efficiency requirement to pay only half the cost required for handling the maximum daily load (assuming we used the same configuration throughout the day). This requirement is detailed enough from a technical point of view, so the software architect adds no new details here.

The software architect will next inspect the scalability, elasticity, and cost-efficiency requirements, given the expected load, work, and SLOs.

2.5 Specify ScaleDL Model

The software architect next checks whether CloudStore can handle the scalability, elasticity, and cost-efficiency requirements as formulated in Sect. 2.4. The software architect may use benchmarking by testing an implemented CloudStore that uses a wide selection of cloud computing resources. In the software architect's case, CloudStore is not fully developed. Therefore, the software architect decides to based the analysis of SLOs on a ScaleDL model.

The ScaleDL model basically contains the following information:

Usage Evolution It describes the anticipated usage of CloudStore. It has already been described at an overall level in Sect. 2.3.3, but in this step, the software architect will go deeper into the expected evolution of work, load, and potentially also SLOs.

Component Model It describes the relations between the software components inside of CloudStore.

Resource Model It describes the hardware and software resources in the underlying cloud resources for the Web Server and the Database Server, as well as how the components relate to these hardware resources. It must reflect the properties of the specific cloud computing resources used for both the `Web Server` and the `Database Server`.

This understanding of the ScaleDL model is extended in Chap. 4.

As described in Sect. 2.1, software architects can create a ScaleDL model based on educated guessing, based on extracting relevant parts of the source code, or from monitoring data. Because the software architect only has a partial implementation of CloudStore, the software architect will base the model on a combination. Available artifacts of CloudStore will be extracted into a model, which the software architect will then complete based on educated guessing. The software architect must also estimate resource demands; i.e., how components inside of CloudStore's model use cloud computing resources.

2.6 Use Analyzer

The software architect uses the CloudStore's ScaleDL model created in Sect. 2.5 for performing scalability, elasticity, and cost-efficiency analyses using CloudScale's Analyzer tool. In the following, each analysis is reported in a separate subsection.

2.6.1 Scalability Analysis

In a scalability analysis, the software architect experiments with several cloud resource configurations for the `Web Server` and for the `Database Server`.

The software architect's goal is to find a configuration that delivers a 90 percentile response time of less than 5 s with 1000 simultaneous customers for the `Pay` operation. At the same time, CloudStore's database has to hold 1,000,000 customer items and 1,000,000 book items (cf. Sect. 2.4.1).

Unfortunately, the software architect cannot find a suitable cloud resource configuration—CloudScale's Analyzer predicts SLO violations for inspected variants. Even if the software architect assumes that a more suitable cloud resource will appear on the market in the next 3 years, Analyzer shows that SLOs cannot be fulfilled.

Therefore, the software architect uses CloudScale's Spotter tool to find the root cause of the problem. Spotter points to the locking of the transactional database as the root cause. The software architect therefore decides to use a NoSQL database implementation instead.

After remodeling CloudStore's ScaleDL model with the new database type, the software architect runs the Analyzer again. The software architect finds that CloudStore is able to fulfill its scalability requirements now. However, the total cost for the envisioned cloud resource configuration is quite high, i.e., $ 100 per hour. Even if the software architect expects prices to drop during the next 3 years, this amount is more than what the software architect expected.

2.6.2 Elasticity Analysis

During the scalability analysis in the preceding step, the software architect used a steady-state version of the load. Now, the software architect conducts an elasticity analysis for transient phases. The software architect looks closer at the best-seller book scenario in Sect. 2.4.2 to see if CloudStore is able to fulfill its SLOs.

Again, the software architect focuses on the toughest workload described in the scalability analysis. As described in Sect. 2.6.1, the software architect has decided to use a NoSQL database. Because this database offers autoscaling, the CloudStore model also reflects this important aspect of the Database Server. Particularly, the `Web Server` also offers autoscaling.

The software architect defines scaling-out when average CPU utilization exceeds 70% and scaling-in when average CPU utilization is below 50%. Using this autoscaling rules, the software architect finds that CloudStore is able to fulfill its SLOs throughout the day. This fulfillment holds for the elasticity requirement to handle an increase, within 1 min, from 500 to 1000 simultaneous requests to the `Buy` operation and from 500,000 customer items to 1,000,000 customer items within CloudStore's database.

2.6.3 *Cost-Efficiency Analysis*

In Sect. 2.4.3, the software architect has defined the cost-efficiency requirement that operational cost should be half the cost of the expensive cloud resource configuration. The software architect runs a cost-efficiency analysis with the Analyzer to check this requirement.

The software architect finds that this requirement is violated—no matter which adjustments the software architect makes in the auto-scaling configuration. However, the software architect finds that CloudStore is able to achieve an overall operational cost of 60% of the operational cost of the most expensive cloud resource configuration (assuming we used this configuration the whole time). As a result, the software architect adjusts the cost-efficiency requirement accordingly.

In this cost-efficiency analysis, the software architect has found that the requirements were too optimistic. Adjusting requirements before investing in development efforts is wise since adjustments allow to resolve trade-offs with different stakeholders before continuing the development effort. An alternative to an adjustment of infeasible requirements is to stop the development project, which can, by all means, be a viable option because of unexpected high development efforts and unrealistic expectations.

2.7 Use Spotters

After the software architect has realized CloudStore, the software architect may use the two parts of the Spotter tool to identify anti-patterns. The Static Spotter examines static code, while the Dynamic Spotter comprises instrumentation and load generation of a running service. In both cases, the code is reengineered if anti-patterns or root causes are spotted. Spotter comes with a catalog of supported anti-patterns, which is detailed in Sect. 2.10.

Within a test environment, the software architect has now set CloudStore in operation but observes that response times are too high. In other words, CloudStore's SLOs are violated.

Using the Spotter, the software architect first finds potential weak spots in the CloudStore code and fixes them. Afterward, the software architect experiments with different loads derived from the key scenarios. The software architect finds out that connection pooling is a bottleneck and a simple solution is to make the connection pool larger. This solves all problems, and CloudStore's customers are again happy. The software architect is also happy, being the one responsible for the operation of CloudStore.

2.8 Realize, Deploy, and Operate System

After having analyzed CloudStore's ScaleDL architectural model and its implementation, the software architect realizes CloudStore as a service. At this point, the software architect is more confident than without analyses that the service will satisfy its requirements. The analysis also revealed information that assists this realization, especially because automatic code generation can be employed.

After the software architect has set up the operating environment, the software architect deploys the CloudStore. As an extra precaution, the software architect may use the Dynamic Spotter to identify potential issues in the final production environment.

After deployment, the software architect puts the system in operation with real customers. Monitoring is active during the system operation and enables control of system behavior. Monitoring includes collecting measurements for scalability, elasticity, and cost-efficiency parameters. Measurements allow the software architect to revise SLOs, which potentially triggers a rerun through the CloudScale method.

In this step, the software architect may also discover new anti-patterns. These new anti-patterns can then be put into the anti-pattern catalog (cf. Sect. 2.10) used by the Spotter for its automatic detections.

2.9 Cloud Computing HowTos

HowTos describe reusable best practices for software architects to design systems in recurring situations. Such situations typically appear in specific application domains and with respect to particular quality properties. In this section, we shed light on the respective HowTos directly or indirectly related to the cloud computing domain, especially with a focus on scalability and elasticity as cloud computing's defining characteristics. By following these HowTos, software architects can effectively and efficiently create ScaleDL models and realize cloud-aware systems.

Table 2.1 lists the HowTos so far collected in CloudScale's catalog of HowTos.[2] This catalog covers the application domains of business information systems, cloud computing, and big data (first column). For each of these domains, the table gives the name of the HowTo (second column), its type (third column), and the fostered quality properties (fourth column).

HowTos in cloud computing focus on elasticity that particularly improves cost-efficiency and depends on scalability. Because of this dependency, scalability HowTos from related domains, i.e., business information systems and big data, are required as well.

[2] An up-to-date description of the catalog is available at CloudScale's Wiki page [5].

Table 2.1 CloudScale's catalog of HowTos

Application domain	HowTo name	HowTo type	Quality properties
Business information systems	3-Layer	Architectural style	Maintainability
	Loadbalancing (Container)	Architectural pattern	Scalability
	Loadbalancing (Component instance)	Architectural pattern	Scalability
	Static content	Architectural pattern	Scalability
	Sharding	Architectural pattern	Scalability
Cloud computing	SPOSAD	Architectural style	Elasticity
	Horizontal scaling (Container)	Architectural pattern	Elasticity
	Horizontal scaling (Component instance)	Architectural pattern	Elasticity
	Vertical scaling	Architectural pattern	Elasticity
Big data	MapReduce	Architectural style	Scalability
	Hadoop MapReduce	Reference architecture	Scalability

In the domain of **business information systems**, the identified HowTos lay the foundations for cloud computing HowTos. The 3-layer HowTo represents a common architectural style to structure a system into three different logical layers: a presentation layer, an application layer, and a data access layer. Each of these layers can only access the respective lower-level layer. Because of this restriction, the system becomes loosely coupled and therefore more maintainable and easier to scale. Other HowTos can be particularly applied on each layer in separation.

The load-balancing HowTo represents an architectural pattern that requires the existence of a proxy (i.e., a load balancer) that distributes workload for improving scalability. Depending on the variant of this HowTo, the load balancer distributes workload either over a replica of a stateless container, e.g., a virtual machine, or a stateless component instance. For example, the load-balancing pattern can be applied on component instances of the application layer if these are implemented without state.

The static content HowTo represents an architectural pattern that separates static content, e.g., images and static HTML files, from dynamically created content. This separation improves scalability because static content requires no state, and thus, it can be easily load balanced and cached.

The sharding HowTo represents an architectural pattern similar to the load-balancing HowTo: it also requires a load balancer to improve scalability. However, instead of requiring load-balanced elements to be stateless, the sharding HowTo separates workload based on the requested data. The data is divided into partitions (so-called shards) such that each load-balanced element is responsible for only

a particular set of shards. Requests to the same data are then processed by the responsible element.

In **cloud computing**, the SPOSAD [6] HowTo is an architectural style to promote elastic and multi-tenant software applications. SPOSAD describes a variation of the 3-layer HowTo that additionally requires stateless component instances on the middle layer [6]. Because they are stateless, these component instances can be safely replicated (scaled out) and load balanced. In SPOSAD, the load-balancing HowTo can therefore be applied on component instances of the middle layer. For multi-tenancy, SPOSAD introduces a metadata manager on the application layer that provides different tenants with tenant-specific information. This information includes tenant-specific user interface elements, business logic, and data from multi-tenant databases.

The horizontal scaling HowTo [7, 8] depends on SPOSAD's architectural constraints and extends the load-balancing HowTo such that the load balancer dynamically adapts the required number of component instances or container replicas to the current workload. While the load-balancing HowTo improves scalability only, the horizontal scaling HowTo improves elasticity as well.

The vertical scaling HowTo [7, 8] requires that computing resources of a single computing node be dynamically (de-)provisioned. For example, a virtual machine allows to dynamically provision higher CPU processing rates and more main memory. This HowTo therefore improves scalability and elasticity without requiring stateless component instances; however, it can only be applied if sufficient computing resources are available.

In the **big data** domain, the MapReduce HowTo is a common architectural style to foster scalability. MapReduce requires that data be processed independently from each other within mapper and reducer functions. Mapper functions filter and sort such data, and reducer functions summarize collected data. Based on data independence, multiple mapper and reducer functions can run in parallel, thus improving scalability.

Hadoop MapReduce represents a technical HowTo because it suggests using Apache's Hadoop, an open-source MapReduce framework. This HowTo can, however, also be seen as a reference architecture [9] that illustrates how to implement the MapReduce HowTo. The reference architecture essentially describes a processing pipeline for the control and data flow of the MapReduce HowTo.

2.10 Cloud Computing HowNotTos

Similar to HowTos that provide with best practices and suggested approaches to common problems, we can also identify the most common *pitfalls* for the design and development of cloud computing systems—so-called HowNotTos. Once deployed in production, these bad practices can results in problems related to security, performance, scalability, and others. In the CloudScale method, software architects

can detect these HowTos on the model level and within the realization of the system. This detection can be both manual and automatic using CloudScale's Spotter.

On the conceptual level, it is worth mentioning that our work differentiates from the current state of the art in performance engineering in an important aspect: we distinguish between performance anti-patterns and scalability anti-patterns. Performance anti-patterns are classical issues which hinder a system to perform as expected under a given constant workload. Scalability anti-patterns are those anti-patterns which prevent the system from scaling when the system has to face higher workloads because of an evolving usage profile (cf. Sect. 4.3). An example for a HowNotTo is *Excessive Dynamic Allocation*, as described in Fig. 2.3.

Focusing here on scalability (and thus indirectly also on performance), we can categorize the most usual symptoms and their most common root causes. In Fig. 2.1, we show a taxonomical representation of the symptoms and causes that are available at the CloudScale Wiki [11] documentation, and which we will outline in the following (Fig. 2.4).

Application Hiccups refer to a temporary degradation of the system performance, mostly related to the implementation within internal processes of the system such as a garbage collector, online reindexing of a database, or any such maintenance task that has a system-wide impact due to its complexity and/or the need to work on a big portion of the system's information.

The Ramp is the progressive degradation of a system's performance associated with implementation problems, most commonly related to unreleased resources (locks, memory objects, temporary database objects, etc.) or related to unfinished processes that prevent the system to regain access to certain resources.

The Blob is a design bad practice in which most of the complexity of the system's logic is enclosed in a single functional unit (e.g., a class), not only decreasing the maintainability of the code, but also resulting in many cases of unnecessary traffic (Excessive Messaging), of massive amounts of data (e.g., sending an entire object instead of only the changes) and requests (due to lack of local reference), and often preventable lock contentions due to poor granularity.

The **Empty Semi-Truck** refers to the usage of messages to send very small pieces of information, which results in Excessive Messaging that would be preventable by clubbing similar or related requests into one single request, greatly reducing messaging overhead, latency, and throughput. The most common cases are related to single-row requests instead of a query to obtain a result set, or the individual requests of different attributes of an object instead of the entire object.

In **Excessive Dynamic Allocation**, chunks of memory or other resources are reserved very often, taking a considerable part of the system's working time. It occurs most commonly in object-oriented-language designs in which objects are created and destroyed very often, overloading the system with the costly task of allocating memory and instantiating new objects from scratch. Having a pool of resources marked as obsolete (objects that are not needed any more, but kept in memory to be reinstantiated when needed) can greatly alleviate this problem.

The **One-Lane Bridge** is a common design in which a passive resource (a connection pool, database, lock, mutex, etc.) is needed by different process at the

HowNotTo Example: "Excessive Dynamic Allocation"

Name: Excessive Dynamic Allocation
Abstract: The Excessive Dynamic Allocation occurs in object-oriented software systems when dynamic allocations are needed. Excessive Dynamic Allocation is where creation and destruction of the objects of the same class are frequent and unnecessary.
Example: An example in cloud computing is when the same resource (an object composed of several child objects) offered by cloud vendors in a certain service is regularly needed by most cloud users. Rather than creating and destroying this set of objects dynamically on a per-user basis, a set of objects can be pre-created and shared among the users who need the service.
Context: In object-oriented software systems, where many dynamic allocations are used for objects. Typically, those allocations occur in frequent behaviors like loop bodies or event handlers for requests.
Problem: When a new object is created in object-oriented software systems, the memory used to contain the object must be allocated from the heap (i.e., a sufficiently large chunk of free heap memory needs to be found and reserved) and any code used to initialize all the objects contained in this memory location must be executed. Memory leaks can be avoided by doing memory clean-up and returning the reclaimed the memory to the heap, when objects are no longer needed. Performance can be significantly improved by removing the overheads, which are caused by frequent and unnecessary creation and destruction of objects.
Detection: One way of detecting this HowNotTo is based on observing memory allocations and identifying reoccurring patters with high amounts of allocations of objects of the same class.
Solution: There are two possible ways to solve the problem of Excessive Dynamic Allocation:

- The first is to recycle old objects rather than create new ones every time they are needed. This means allocating and storing a large amount of objects (a pool of objects) in a collection. When new objects are needed, they can be fetched from the pool, and when old objects are no longer in use, they can be returned to the pool. This can be very useful when new objects with relatively short lifespans are created all the time like the objects in the example above. This means we spent a little more time in the initialization of the system caused by allocating objects in the pool, but also reduce the overheads for creating and destroying the same objects all the time. This can help to reduce the problem of memory leaks and the overheads caused by garbage collection.
- The second way is to use sharing instead of creating new objects. An example of this is the use of the Flyweight design pattern, which allows all clients to share a single object.

Spotter Implementation: Statically, occurrences of memory allocations ("new") are detected by identifying parts (a) in which larger memory blocks are allocated (e.g., Arrays) and (b) which are executed often (e.g., in a loop). Dynamically, Excessive Dynamic Allocation can be detected indirectly by leveraging the Application Hiccups detection where the user has to manually analyze whether the hiccups were caused by garbage collection or memory defragmentation.
See also: C. Smith [10]

Fig. 2.3 Excessive dynamic allocation HowNotTo example

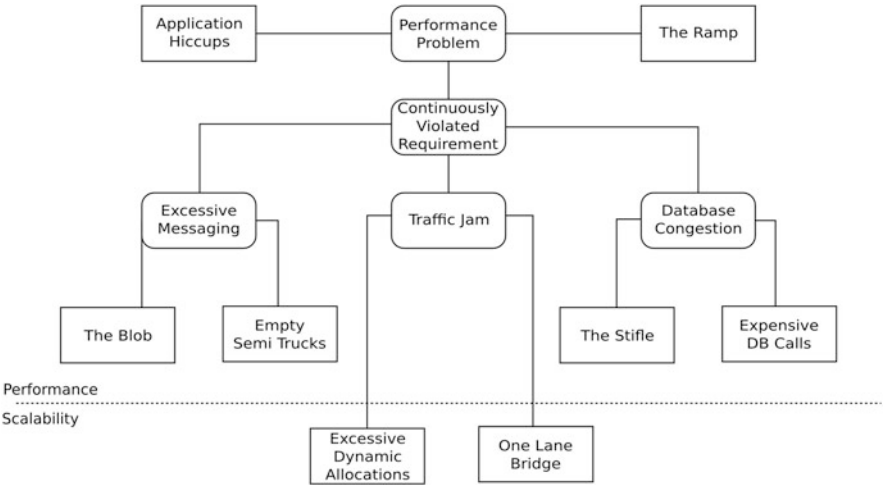

Fig. 2.4 Main synthoms (inner nodes) and their most common root causes (leaves)

same time, becoming the bottleneck for the most important operations of the system by limiting its concurrent processing. A concurrent contention can be alleviated by adding additional passive resources, but the contention point is only moved higher and not resolved. A completely different design approach is needed in order to obtain a really scalable system, though solutions are often impractical, and thus, a good-enough approach is followed.

The Stifle is basically an **Empty Semi-Truck** that updates data within the database. For example, if calculating the current age of each employee, we retrieve the birth date, calculate the age, and insert it in that row one at a time instead of using a single SQL statement to update all the rows at the same time.

Expensive Database Calls is another design problem that affects the database, this time related to **The Blob**. Complex data queries such as those needed for data warehousing and analytics, or back-up systems, can bring a database's performance close to zero. This is particularly true for queries that hold locks for longer times (e.g., if consistency of relationships must be preserved), basically preventing operation on most of the database. Night batch processing and parallel warehouse databases are common approaches to solve this problem.

These are, by no means, all the existing symptoms and root causes of cloud computing. A thorough study of these bad practices and common pitfalls can be found in [12].

2.11 The CloudScale Method in the Unified Process

After exemplifying the CloudScale method in the preceding sections, this section describes how the CloudScale method fits into existing development processes. In particular, this section focuses on relating the CloudScale method to the Unified Process, as described next.

2.11.1 Unified Processes

Any kind of software development follows some more or less explicit process steps that transform an initial idea to a working solution. Best practices have been accumulated during the last decades, and currently, there are development processes covering parts of or the whole lifecycle for software products. Some development processes are thin and solve a partial problem, and some of them are more complete. Complete development processes are often termed unified processes. The unified process (UP) is a generic name for a family of process models that are iterative, incremental, architecture centric, driven by use cases, and focus on addressing risks early on. In general, UP defines four project phases: inception, elaboration, construction, and transition. The most recognized unified processes are the rational unified process (RUP) [13] and OpenUP. RUP was created by Rational Software (now owned by IBM) and is supported by commercial tools. OpenUP is promoted by the Eclipse Foundation and is supported by free tools.

Agile development processes have gained traction in recent years. Agility development processes are even more important in cloud-aware development, where the creation of new services is expected to be fast. The Agile Unified Process, as simplification of RUP, has prepared for agile and lean development.

RUP consists of nine disciplines, from which six are related to technical software development: business modeling, requirements, analysis and design, implementation, test, and deployment. Three RUP disciplines are related to executive support: configuration and change management, project management, and environment. In the further analysis, the focus will be on the inception and elaboration phase and the first three disciplines: business modeling, requirements, and analysis and design.

The inception phase is focused on establishing the business case for the system and delimits the project scope. In this phase, all external entities and system roles (actors) must be recognized, and the nature of their interactions described. The business case includes success criteria, risk assessment, and estimates of required resources. The main output of the inception phase is a vision document that contains a general vision of the core project requirements, key features, and main constraints. Additional key outputs are initial use-case models and prototypes.

The elaboration phase focuses on analyzing the problem domain and establishes a sound architectural foundation. To fulfill such an expectation, a wide-enough and deep-enough view of the system is required. Architectural decisions have to

be made with an understanding of the whole system based on the scope, major functionality, and non-functional requirements, such as performance requirements. The main outputs from the elaboration phase are executable architectural prototypes, supplementary requirements capturing non-functional ones, finished use-case models, software architecture descriptions, and potential business models.

2.11.2 Relating the CloudScale Method

The essential objective of the CloudScale method is to guide software architects by means of an engineering method to develop scalable, elastic, and cost-efficient applications. The CloudScale method provides a framework to build new systems and to analyze deployed systems in operation. Building a new system is focused on system analysis based on a system model and can be done from scratch or by using some existing components. Looking at the described phases of UP, we see that the CloudScale method fits into some elements of the UP, where the CloudScale method performs a deeper analysis of key elements of the system for cloud deployment.

The CloudScale method starts with requirements' collection, with an additional focus on the scalability, elasticity, and cost-efficiency requirements. In the UP, this means that core requirements are extended, with clear concepts for scalability, elasticity, and cost-efficiency requirements. Additionally, system constraints are also clarified. Based on initial requirements and definition of actors, we define the foundation for basic decisions in the CloudScale method: should we develop a system from the scratch, reuse some components, or evolve an existing system into the cloud environment by reusing the existing artifacts of the software system?

When we decide what will be our starting point, we proceed to the elaboration phase in the UP. This phase is correlated with the specification of a ScaleDL model that can be based on the existing software or a completely new model. When applicable, CloudScale's HowTos (cf. Sect. 2.9) provide good assistance for model creation. By constructing the model, we prepare an architecture that can be further analyzed by using Static Spotter and it is the first architectural prototype that can be tested according to scalability, elasticity, and cost-efficiency requirements. An additional possibility is to use some additional external tools and test dynamic behavior of the provided system model. The key supporting elements in the CloudScale method are tools and the provided HowTos (cf. Sect. 2.9) and HowNotTos (cf. Sect. 2.9) that guide system architects in selection of optimal architecture decision and behavior according to requirements. Selecting a good architectural prototype with the CloudScale method creates a very good foundation for the construction phase of the UP.

Analysis of a running system is done in the transition phase in the UP by testing and analyzing system behavior and by collecting critical information about system behavior, with points for potential improvements. The Dynamic Spotter from the CloudScale method supports software architects in this task.

2.12 Conclusion

This chapter gives a first overview of the CloudScale method by guiding a software architect through the development of CloudStore. Best practices (HowTos) and common pitfalls (HowNotTos) are also described to assist in the work of the software architect. The overview of the CloudScale method has particularly allowed to relate it with existing development processes such as the Unified Process.

Throughout this chapter, the main benefits of the CloudScale method are exemplified. CloudStore's software architect has been able to analyze scalability, elasticity, and cost-efficiency on the model level, i.e., even before implementing CloudStore. This analysis enabled the architect to resolve any issues early on, which reduced risks of violating SLOs during CloudStore's operation in a production environment. The Spotter tool, HowTos, and HowNotTos additionally helped the software architect in engineering a bullet-proof online shop.

While this chapter only gives a quick high-level overview of the CloudScale method, subsequent chapters provide further details. Particularly, Part III provides detailed step-by-step instructions that explain how software architects can follow the CloudScale method.

References

1. Q-ImPrESS: Project Deliverable D6.1: Method and Abstract Workflow. www.q-impress.eu/wordpress/wp-content/uploads/2011/03/D6.1-method_and_abstract_workflow-v2.0.pdf. Visited: 2 November 2015
2. Becker, S., Busch, A., Brosig, F., Burger, E., Durdik, Z., Heger, C., Happe, J., Happe, L., Heinrich, R., Henss, J., Huber, N., Hummel, O., Klatt, B., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K., Küster, M., Langhammer, M., Lehrig, S., Merkle, P., Meyerer, F., Noorshams, Q., Reussner, R.H., Rostami, K., Spinner, S., Stier, C., Strittmatter, M., Wert, A.: In: Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K. (eds.) Modeling and Simulating Software Architectures – The Palladio Approach, 408 pp. MIT Press, Cambridge, MA (2016) [Online]. Available: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures
3. Brataas, G., Stav, E., Lehrig, S., Becker, S., Kopcak, G., Huljenić, D.: CloudScale: scalability management for cloud systems. In: Seetharami S. (ed.) Proceedings of International Conference on Performance Engineering (ICPE), pp. 335–338. ACM, New York (2013). http://dx.doi.org/10.1145/2479871.2479920
4. OMG: UML Profile for MARTE. http://www.omg.org/spec/MARTE/. Version 1.1, Inspected 11 November 2016, June 2011
5. CloudScale Wiki: HowTos: (2016) [Visited on 12/19/2016]
6. Koziolek, H.: The SPOSAD architectural style for multi-tenant software applications. In: Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 320–327. IEEE Computer Society, Washington (2011). http://dx.doi.org/10.1109/WICSA.2011.50
7. Erl, T., Puttini, R., Mahmood, Z.: Cloud Computing: Concepts, Technology and Architecture, 1st edn. Prentice Hall Press, Upper Saddle River (2013)
8. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer Vienna, Vienna (2014). http://dx.doi.org/10.1007/978-3-7091-1568-8_1

9. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Boston, MA (1998)
10. Smith, C.U., Williams, L.G.: Software performance antipatterns. In: Proceedings of the 2nd International Workshop on Software and Performance (WOSP), pp. 127–136. ACM, New York (2000). http://dx.doi.org/10.1145/350391.350420
11. CloudScale Wiki: HowNotTos: wiki.cloudscale-project.eu/HowNotTos:_Anti-Patterns (2016) [Visited on 12/19/2016]
12. Wert, A.: Performance problem diagnostics by systematic experimentation. Dissertation, Fakultät für Informatik (INFORMATIK). Karlsruhe Institute of Technology, Karlsruhe (2015)
13. Kruchten, P.: The Rational Unified Process: An Introduction (The Addison-Wesley Object Technology Series). Addison-Wesley, Boston (2003)

Part II
Modeling Cloud Computing Applications

After having introduced the topic of this book in Part I, we address modeling cloud computing systems in this part. The use of various kinds of models is standard in software development. Software developers use models to analyze the problem domain and to specify how the system under development should be structured.

Since the late 1990s, software architects have used models to describe the architecture of their systems in terms of components (or services), connectors, associated hardware and software environments, etc. Since the early 2000s, software architects have also used models of their software architecture to analyze and predict the quality attributes of their systems, e.g., performance, reliability, and maintainability.

In the last years, the shift toward web-scale systems has raised the demand for analyses that focus on quality properties important in the cloud computing era: scalability, elasticity, and cost-efficiency. Modeling languages are needed to capture the essential behavior of cloud computing applications, their scaling behavior, and their resource consumptions. These languages enable software architects to model and analyze their systems. In this part, we introduce the modeling language used by the CloudScale method.

In Chap. 3 we introduce cloud computing applications in detail and discuss the aspects that need to be captured in models of such applications. In the CloudScale method, these aspects are captured in instances of the ScaleDL modeling language. This language is introduced in Chap. 4.

Chapter 3
Cloud Computing Applications

Mariano Cecowski, Steffen Becker, and Sebastian Lehrig

Abstract Cloud computing focuses on elasticity, i.e., providing constant quality of service independent of workload. For achieving elasticity, cloud computing applications utilize virtualized infrastructures, distributed platforms, and other software-as-a-service offerings. The surge of cloud computing applications responds to the ability of cloud computing environments to only pay for utilized resources while saving upfront costs (e.g., buying and setting up infrastructure) and allowing for dynamic allocation of resources even in public-private hybrid scenarios.
This chapter investigates the shift from classical three-tier web applications to such elastic cloud computing applications. After characterizing web applications, we describe cloud computing characteristics and derive how web applications can exploit these characteristics. Our results motivate novel requirements that have to be engineered and modeled, as further described in this chapter.

The remainder of this chapter is structured as follows. Section 3.1 gives a high-level overview of views and aspects important for cloud computing applications. Afterward, Sect. 3.2 details characteristics of web applications, and Sect. 3.3, those of cloud computing. Section 3.4 outlines how web applications can be moved to cloud computing applications by exploiting cloud computing characteristics. Based on these insights, we derive novel requirements for cloud computing applications in Sect. 3.5 and show how to appropriately model cloud computing applications to analyze such requirements in Sect. 3.6. Section 3.7 refines our running example—CloudStore—to a cloud computing application. Subsequently, Sect. 3.8 highlights some hints for modeling such applications.

M. Cecowski (✉)
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: mariano.cecowski@xlab.si

S. Becker
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
e-mail: steffen.becker@informatik.uni-stuttgart.de

S. Lehrig
IBM Research, Technology Campus, Damastown Industrial Estate, Dublin 15, Ireland
e-mail: sebastian.lehrig@ibm.com

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_3

3.1 Introduction

This introduction outlines views and aspects of cloud applications to give engineers a broad high-level understanding of such applications. These views and aspects are detailed in subsequent sections.

From a historical view, advancements in hardware virtualization have enabled a novel infrastructure management that, together with the adoption of standards for the definition of HTTP interfaces or application programming interfaces (APIs), has shifted the focus of distributed computing from grid computing toward cloud computing environments. Cloud computing is not limited to applications from the scientific or high performance computing (HPC) world since even applications that have been historically considered desktop applications—from e-mail to office suits—appear as online services for the general public. Cloud computing is particularly attractive for building new applications while outsourcing the necessary infrastructure and expertise needed for its management.

In general, cloud computing [1] refers to the ability of an on-demand self-service of capabilities, infrastructure resource pooling, and an elastic (auto-scalable) resource allocation for infrastructure, platform services, and software services. Cloud applications, i.e., software applications operating in a cloud computing environment, share some general properties: they provide a service available simultaneously to multiple consumers (either users or composed services), they can be replicated in order to increase their capacity, they can be migrated from one infrastructure to another, and they are reachable through an API.

As with other distributed applications, cloud applications require a great degree of decoupling between its different logical components and as little contention as possible. These requirements allow for increasing the applications' capacity by allocating more instances of each needed component. The most important characteristics we need to keep in mind when analyzing a cloud application therefore are capacity, scalability, elasticity, and cost-efficiency, as defined in Sect. 1.3.

Nevertheless, in order to analyze these characteristics, we first need to clearly define the system's qualities with service-level objectives (SLOs) (cf. Definition 1.2). SLOs often focus on the user experience for the application, e.g., by monitoring the response time needed to perform a certain task or the success rate of such tasks. It is only then that we can obtain, for a given setup, the capacity or any other such indicator by measuring the point after which the system cannot guarantee the fulfillment of the SLO.

Other aspects that describe a cloud application are related to its internal structure and technical architecture. The first web applications have only made use of static HTML to present the information of the distributed application. Scripts made it possible to generate web pages on demand based on a session context, and are still used in some applications. Dynamic pages, remote objects, servlets, and other techniques have been and are still used for implementing cloud applications [2].

Moreover, in the last years, the model-view-controller (MVC) paradigm has become very popular. Following the MVC paradigm and since the arrival of HTML5, more and more of the presentation logic has been moved to the browser, e.g., via single-page web applications (a.k.a. service-oriented front-end applications) [3].

Finally, an important aspect of cloud applications centers around the security and privacy concerns related to having publicly reachable APIs, computing and storage in public clouds, and a relative lack of control over both the execution of the processes and the physical access to the data.

The subsequent sections detail the shift from classical web applications to cloud computing applications.

3.2 Web Applications

Technically speaking, web applications are programs that have their front-end in a user's browser, which in turn communicates with its back-end services. The code that runs at the user's computer is the presentation or view part of the layered or MVC architecture, while the code running in the service represents the other layers containing the control, model, and data parts of the system (see Fig. 3.1 for a typical example).

Examples of such applications are web-mail implementations (from Gmail[1] or Yahoo mail[2] to open-source solutions such as SquirrelMail[3] or Roundcube[4]), content management systems (OpenCms,[5] Umbraco,[6] WordPress[7]), and e-commerce

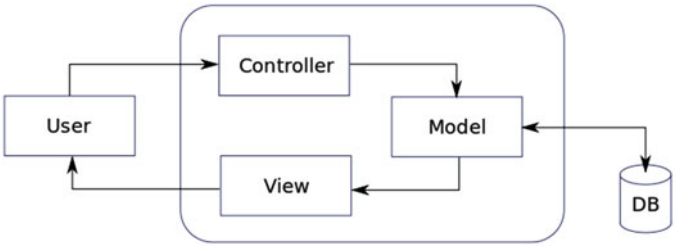

Fig. 3.1 Typical model-view-controller with user interaction and database

[1] https://gmail.com.
[2] https://mail.yahoo.com.
[3] https://squirrelmail.org/.
[4] https://roundcube.net/.
[5] http://www.opencms.org.
[6] https://umbraco.com/.
[7] https://wordpress.com//.

systems (OpenCart,[8] osCommerce,[9] simpleCart[10]), which provide a typical layout with more or less dynamic behavior.

Frameworks for creating desktop-like web applications are very popular (from Angular,[11] jQuery[12] to Flash[13] or Silverlight[14]), allowing for applications to achieve high standards in terms of usability and responsiveness, while more complex web applications that mimic complex desktop counterparts also exist (Pixlr[15] image editor, Plex[16] for media streaming, Google Hangouts[17] for video conferencing).

At the time of writing this book, web applications often make use of JavaScript and dynamic content to provide a close-to-desktop experience, while relaying most of the logic to the remote service. These remote services can maintain a session identifier that is passed with each call in order to follow the state of the user's interaction, retrieve information from storage—e.g., a database, a key-value store, or static files—or connect to further services to obtain other information and functionality, such as a weather gateway or an external payment system. It is becoming more common to make use of the approach of composite services to break down a web application's back-end into smaller pieces, so-called micro-services. Microservices communicate through an API—e.g., a RESTFull API—and use the much more decoupled pieces to provide a per-service scalability and elasticity.

Decoupling the service's components has several further advantages besides scalability. For example, each component can be developed in the most appropriate technology for that task, be deployed on the hardware configuration that provides the best cost-efficiency, or make use of an external service (see example in Fig. 3.2). In recent years, a small fragmentation of components has been supported in the form of Docker containers—virtual environments within the same operating system—to run different component instances within the same instance of the operating system and programming libraries, reducing the overhead of virtualization.

Web applications are thus designed to serve several users with the same running service, and are expected to manage an increasing number of customers by replicating instances of those component types that are being flooded by work. Nevertheless, scalability not always comes naturally, and adding more resources to certain components can only marginally improve the number of serviceable customers, or even undermine it.

[8] https://www.opencart.com/.

[9] https://www.oscommerce.com/.

[10] http://simplecartjs.com/.

[11] https://angularjs.org/.

[12] https://jquery.com/.

[13] http://www.adobe.com/software/flash/about/.

[14] https://www.microsoft.com/silverlight/.

[15] https://pixlr.com/.

[16] https://www.plex.tv/.

[17] https://hangouts.google.com/.

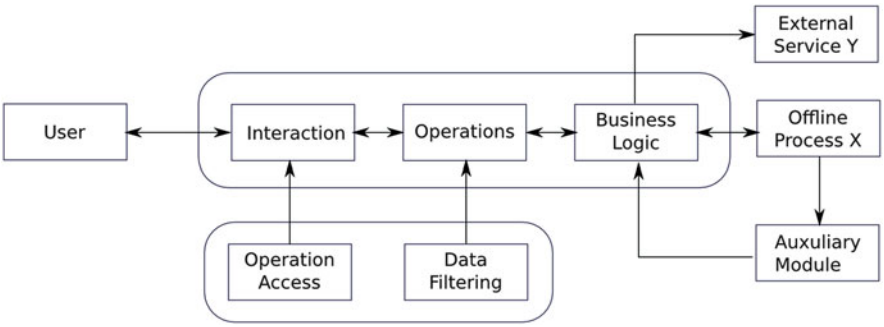

Fig. 3.2 Simplified example of composed back-end services of loosely coupled components, including external services

A common example of such scalability issues is that of relational databases. Once clustered, relational databases can provide good scalability for reading operations but suffer from the nature of transactional queries that require all versions of the data to be synchronized. In such clusters, a transaction has to be propagated to all nodes before any further operation can be done on that data. Modern approaches follow an eventually-consistent approach that relaxes the transactional paradigm, but which in turn brings its own issues and trade-offs. This paradigm renounces the idea of having a consistent state throughout all instances of a replicated dataset and focuses on eventually having all the instances updated. This relaxation of the consistency can be useful in systems where it is not imperative to use the latest up-to-date value in computations or displaying it to the user. For example, a limited delay in showing the actual number of "Likes" in an entry within a social media element is mostly irrelevant.

Cloud computing characteristics are inspected next and, subsequently, compared to these technical considerations of web applications for investigating the shift from web to cloud computing applications.

3.3 Cloud Computing Characteristics

For engineering cloud computing applications, software architects need to consider the essential characteristics of the cloud computing domain. Therefore, this section gives a quick summary of these characteristics.

This summary is based on the well-accepted and standardized NIST definition of cloud computing [1], which finds the following characteristics to be essential:

On-demand self-service: A cloud consumer can request additional resources on demand, without requiring human interaction on the cloud provider side.

Broad network access: Cloud providers provide access to cloud services through standardized network interfaces, thus supporting both thin and thick clients on the cloud consumer side.

Resource pooling: Cloud providers can group resources, e.g., storage, processing, and memory resources, into pools from which multiple cloud consumers (tenants) can be served. In such a multi-tenant setup, each tenant is unaware of the activities of other tenants and actual physical resources, so the number of available resources appears to be unlimited.

Rapid elasticity: Services of cloud providers can autonomously scale in and scale out, depending on cloud consumer demand, through an elasticity management.

Measured service: Cloud providers measure the usage of resources by cloud consumers. Cloud consumers typically only pay for the resources they have used or reserved (pay per use).

There is a strong relation between these characteristics and the internal quality properties of services as described in Sect. 1.4. The *rapid elasticity* characteristic, for example, demands for the cloud computing service to be both *scalable* and *elastic*. To achieve these properties, such a service needs to make use of an *on-demand self-service* that accesses *resource pools* of cloud providers, in order to cope with varying workloads. Such an approach becomes *cost-efficient* because of the *measured service* and the involved pay-per-use paradigm: cloud consumers only have to pay for the resources needed to cope with varying workloads.

3.4 From Web to Cloud Computing Applications

We have described web applications as web pages that attempt to provide a user experience similar to that of desktop applications. However, web application logic and data reside on a remote—and often cloud-deployed—set of services.

Cloud computing applications are any kind of service that is deployed to run on a cloud computing platform. In that sense, cloud computing applications include web applications but they include other services and applications for which a web interface is secondary, or at least just another means of interaction with it.

The following applications are examples of web applications based on cloud computing platforms, i.e., they are not classical web applications. A back-up service that stores incremental information on remote servers by means of a small daemon application running on a user's machine provides a web interface to navigate and download those files through a browser. However, the core service is not web oriented, and in any case, the web interface uses an API that is available to different front-ends, including even a mobile application. File storage services like DropBox, GoogleDrive, or Microsoft OneDrive are typical examples of such cloud applications.

There are, nevertheless, less clear examples that can blur the line separating the two. Most social networks can be seen as web applications or more general cloud applications, especially those which make their APIs available to third parties. But as soon as a web application allows its back-end services to be used by means other than a web browser, we generally consider it more as a cloud application rather than a web application.

3.5 Requirements of Cloud Computing Applications

As discussed in the previous sections, web applications based on cloud computing platforms present some characteristics that differentiate them from classical web applications. Therefore, cloud computing applications need new approaches for requirements elicitation.

Despite the classical requirements for web applications—for which we assume that it is known how to gather them—new approaches are needed to elicit the requirements of cloud computing applications. This includes new aspects to be discussed with the system's stakeholders, and new approaches and languages to persist the collected set of requirements.

Revisiting the NIST definition of cloud computing (see Sect. 3.3), we can identify that requirements elicitation needs to address the following system aspects:

On-demand self-service: Requirements need to be captured for conditions under which the system should use the on-demand self-service APIs and provision or deprovision necessary resources. This also includes the degree of autonomy the system should have; i.e., whether there are any boundaries where human interaction is needed. For example, the system may not be allowed to provision more than 100 servers, with an administrator authorizing this.

Broad network access: The requirements for the amount and speed of data sent to and from the cloud computing application need to be identified and elicited in order to verify against the network access service level agreement (SLA) of the cloud offering.

Resource pooling: Cloud computing uses shared resources to fulfill customer's demands; i.e., the same physical resources are used by multiple tenants. Besides requirements toward security (protection of the data of different shared resources against each other's access), this also imposes requirements toward performance isolation. A tenant using a lot of compute power at any given point in time should not impact other customers in their compute tasks. A level of acceptable performance interaction should be identified and compared to the cloud provider's performance isolation SLAs.

Rapid elasticity: For the CloudScale method, rapid elasticity is the most important characteristic of cloud computing applications. This importance is caused by the primary focus of elasticity: reducing total cost of ownership (TOC). Requirements toward rapid elasticity need to identify, e.g., the amount of acceptable SLA violations due to elasticity management reacting too slowly. A main metric

impacting the speed of the elastic behavior of the cloud computing application is the time to provision resources in lower application layers. Requirements toward these reaction times should be derived from requirements for the overall application elasticity.

Measured service: Cloud computing applications need to be measured for various reasons. On the one hand, measurements need to be taken to charge customers of the application for their service usage. On the other hand, measurements are also needed within the application itself to inform the rapid elasticity management when it is time to provision or deprovision resources. There exists a trade-off between monitoring frequency (directly influencing reacting time) and monitoring overhead. Therefore, requirements need to be elicited on this reaction time and acceptable monitoring overheads; i.e., what percentage of the used resources may be utilized by the monitoring subsystem.

The best way to express most of the cloud computing application requirements—as for most other quality-of-service requirements—is to characterize them quantitatively, i.e., with concrete figures specifying thresholds for the application's behavior.

However, as with performance requirements, standardized languages usable in practice to persist the elicited requirements are not established. In today's practice, as a consequence, software architects need to invent their own way of persisting the aspects described in the list above.

The CloudScale method mitigates the issue of lacking requirement specification languages to some extent. In particular, it allows to capture the usage evolution in a quantified mode to characterize the system's work and load context over time. In addition, it provides a model to model the system's SLOs in a quantitative mean. Both models are required when using the Analyzer, but they provide additional value even if the Analyzer is not used.

3.6 Modeling Cloud Computing Applications

When designing cloud computing applications, the question arises on how software architects can model these applications to document and analyze their designs. When looking at the de-facto software modeling standard UML2, several lacking features become visible. In the following, we will discuss which types of modeling views are needed to describe cloud computing applications, including the ones that already exist and those that need to be considered anew.

3.6.1 Common View Types for Applications

In the beginning, for each application, there should be a model showing its use-cases to motivate its development and give an overview. In UML2, this is done in use-case diagrams. Use- cases show the main actors and their most important interactions

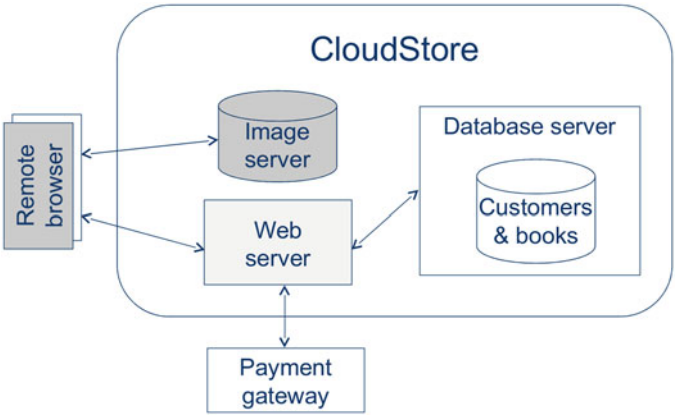

Fig. 3.3 Diagram of the internal and external components of CloudStore

with the modeled system. Use cases should be accompanied by scenarios. In turn, scenarios model concrete interactions of the system's actors with the modeled system, including the frequency of these interactions, the data sent to and received from the system, etc.

Internally, cloud computing applications—as regular applications—have static and dynamic aspects to be modeled. Statically, they are composed on a coarse-grained level of components that implement the primary functional blocks, and which collaborate via communication channels over so-called connectors. This aspect is typically modeled using languages similar to UML2 component diagrams or box-and-line diagrams (see Fig. 3.3 for an example), which often have only loosely defined semantics.

Components offer operations that internally perform a certain dynamic behavior. This behavior is commonly modeled using UML2 activity diagrams or state charts, where the latter is however uncommon for web and cloud applications. Activities depict the behavior of operations as a sequence of actions which are connected by control and/or data-flow arcs.

When the implementing components are modeled, they are allocated on resources of lower application layers, like an (elastic) Infrastructure as a Service (IaaS) layer. In UML2, deployment diagrams show both the allocation of components to layers and the stacking of different vertical layers on top of each other.

3.6.2 Cloud-Specific View Types for Applications

All of the view types discussed so far are used for non-cloud computing applications, too. The following view types are special to cloud computing applications.

As cloud computing applications are targeted toward highly dynamic environments with strong variations in the workload, it is important to model workload

changes over time, particularly if the model is used to analyze and predict the behavior of the system. There are no standard modeling languages available today to represent this aspect of the system. However, the LIMBO language [4] is a research proposal that can be used to model such changes. LIMBO basically models a function of the varying aspect of the system, e.g., the system's load, over time. LIMBO can model different types of variational patterns, including trends or repetitive change over time, e.g., different load situations over the hours of a day.

Finally, cloud computing applications are elastic applications; i.e., they change the amount of used resources and the allocation of components on these resources at particular points in time when certain environmental conditions are met. For example, a cloud application provisions additional resources if it notices that the load has increased and, consequently, system response times go down. Such actions are often called self-adaptations.Again, there is no standard modeling language to model self-adaptations today. In research literature, there are proposed languages that are based on graphical model transformation languages, in particular Story-Diagrams [5], that have been adapted to particular domains like component-based software development. In these languages, software architects model the changes a self-adaptation performs on the components, their connectors, or allocations as a model transformation.

To summarize, we can model cloud computing applications using the traditional use-case, static, dynamic, and allocation views. In addition, novel views are needed to describe the changing environment and the cloud computing application's self-adaptations. This can be done via LIMBO and graphical model transformation languages. We describe our modeling approach, which is based on these views, in Chap. 4.

3.7 CloudStore Running Example

CloudStore is a simple open-source e-commerce web application that represents an online bookshop. We introduced CloudStore on a high level in Sect. 2.2, and in this section we refine it slightly with concepts that we have introduced in this chapter.

CloudStore is based on the functional and non-functional requirements defined by the TPC-W standard [6] for the implementation and provision of an online bookshop. Figure 3.3 gives an overview of CloudStore's components.

Customers access CloudStore via a web browser that has direct IP communication to components that provide the user experience. Most of the functionality is handled by a web server that receives HTTP requests from the customer's browsers, and for information retrieval purposes connects to a database server storing information about the customers and the items of the shop (books) and transactions (purchases). The HTTP responses from the back-end to the customer's browser contain references to static elements that are contained in an image server (to efficiently display pictures of books and other graphics). In the case of payment transactions, the HTTP response contains references to a payment gateway, which

Table 3.1 The 14 CloudStore operations and their default probabilities and quality thresholds

Operation	Probability (%)	Quality threshold (s)
Home	29.00	3
New products	11.00	5
Best sellers	11.00	5
Product detail	21.00	3
Search request	12.00	3
Search results	11.00	10
Shopping cart	2.00	3
Customer registration	0.82	3
Buy request	0.75	3
Buy confirm	0.69	5
Order inquiry	0.30	3
Order display	0.25	3
Admin request	0.10	3
Admin confirm	0.09	20

is expected to perform the money exchange independently of the system, but which is currently mocked-up within CloudStore for testing purposes.

CloudStore, following TPC-W's definitions, specifies that the system should obtain credit card authorization from a payment gateway emulator, and, if successful, present the customer with an order confirmation. The standard also defines other aspects such as data element's size and expected response times and error rates, as well as expected customer behavior and workload (i.e., the probability of each of the API's calls and session durations). These non-functional requirements were used to specify a set of SLOs that should be respected by the deployed service.

Table 3.1 shows all the 14 operations and their probability values defined for the "Browsing Mix", as well as 90 percentile response time quality thresholds defined for its SLO.

Within the CloudScale project, CloudStore served as a showcase application for the usage of the projects' tools and methodology. As such, a first implementation of CloudStore was subject to a scalability analysis, code extraction, static and runtime spotting of scalability bottlenecks, and different metrics (capacity, scalability, elasticity, cost-efficiency) while following the CloudScale method.

This process has resulted in three different CloudStore versions, each with a different implementation and deployments that had as a final goal the reduction in the total cost of ownership for a projected usage load. The first version is a rewrite of the old TPC-W implementation using the more modern Spring framework, but in a monolithic way. The second version separates different components based on the MVC paradigm. The third and final version makes use of elastic deployments for each component (elastic computing and elastic storage and database services), as well as an external payment gateway.

3.8 Modeling Hints

Even for experienced software architects, modeling can be a complex task. Fortunately, the same modeling tasks and challenges often recur. Such recurring situations allow for providing modeling hints that suggest software architects what to do and what to avoid, given certain situations. This sections gives a list of such hints based on the lessons learned during the CloudScale project.

Identify the critical use cases and key scenarios! Modeling is a lot about abstraction. Without abstraction, a model would be a one-to-one representation of the realized system. However, such a representation is bad because it involves high modeling effort, negatively impacts analyzability (the larger the model, the longer the analysis takes), and obfuscates the main quality-of-service issues of an application. Maybe the most important task for software architects is therefore to get the abstraction level right. But how should a software architect know whether the abstraction level is indeed the most convenient?

Acknowledging the importance of abstraction, the CloudScale method provides a dedicated process step as answer to this question: software architects need to identify the critical use cases and key scenarios for their application (cf. Sect. 2.3). Models must allow to assess these cases and scenarios—yet nothing more! If this constraint holds, the abstraction level is the right one, and the effort for creating models is kept minimal.

For example, to create a model of CloudStore, CloudScale has spent high efforts to cover every one of CloudStore's 14 operations. However, analysis results indicate that only three of these operations (Home, Shopping Cart, and Buy Request) are crucial when it comes to scalability, elasticity, and cost-efficiency issues. That is, to get the same insights, CloudScale could have spent nearly 80% less time by focusing only on the three critical cases (assuming each operation involves the same modeling effort). While CloudScale can justify the incurred effort for CloudStore—it serves as a running example and for learning—a real-life use case can typically not afford such "unnecessary" efforts.

Iterate! Software architects should iteratively create models. This hint is related to the previous suggestion about abstraction levels: often, it helps to find the right abstraction level by starting as coarse grained as possible. For example, we may start modeling CloudStore with only a single component. Only when realizing that, e.g., web and database components should be distinguished, the initial component should be split into two.

Reuse! Modeling tasks are often repetitive. For example, load balancers and caches appear over several applications and different contexts. If possible, software architects should reuse such recurring situations, e.g., by copying or by using dedicated reuse mechanism.

CloudScale has collected HowTos for these recurring situations (cf. Sect. 2.9). CloudScale's tooling particularly provides dedicated reuse mechanisms for applying these HowTos (cf. Sect. 4.4). Evaluations based on CloudStore have

shown that over 80% of time spent on modeling can be saved when using these mechanisms.

Learn modeling frameworks! In model-driven development, a certain amount of technical expertise is required. There is a huge community of modeling advocates that use frameworks which integrate into the Eclipse platform integrated development environment (IDE) [7]. It is generally a good advice to get familiar with these frameworks, especially if existing tools need to be extended.

Practice in small scale! As in programming languages, creating a minimal "hello world!" example helps in getting a starting point. Do the same for modeling. Start with a minimal project for which also a small model suffices and extend it with more advanced features to get the big picture. Our pilot project described in Sect. 8.5 provides an example for such a minimal project.

Share and discuss! Nothing is more disappointing than spending huge efforts in creating a model to then just throw it away and start all over again. Such a waste of effort can actually happen when the model created does not represent what is needed.

The good practice against this issue is to share and discuss created models with other stakeholders—as early and often as possible. This approach especially helps in getting shared knowledge about the application under investigation. Typically, everyone involved learns something new, which eventually leads to a better overall design of applications. When done early and often, the model converges to what is really useful (an accurate reflection of critical use cases and key scenarios) while not deviating from the actual application.

3.9 Conclusion

Web applications are systems that run on the browser while interacting and presenting information to the user. Most of its business logic makes use of cloud computing applications at its back-end. These and other cloud computing applications can be used by multiple users and applications at the same time. Cloud computing applications are limited by the resources available to them and their ability to scale these resources with workload. To that end, cloud computing applications require a number of characteristics, namely on-demand self-service of resources, broad network access, and resource pooling, as well as rapid elasticity to automatically cope with the change in workload. These characteristics particularly call for novel types of requirements and modeling approaches—e.g., as covered by the CloudScale method.

The CloudScale method enables software architects to model the architecture of a cloud computing application. Models can be analyzed to predict the dynamic behavior of the application in different conditions and the application's elastic responsiveness under different loads. With the CloudScale method, such predictions are possible even before an application is actually implemented and without testing the application within a potentially expensive infrastructure. In particular, modeling

allows to identify the architecture, implementation, and configuration with the most suitable trade-offs.

However, modeling can be a difficult and costly process. Fortunately, some good practices can make the process simpler. For example, identifying the most important components to model, improving existing models, and sharing experiences and models between colleagues can quickly improve knowledge and reduce modeling efforts while increasing their benefits. The remainder of this book details these initial hints and the CloudScale method.

References

1. The NIST Definition of Cloud Computing: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (2016) [Visited on 04/18/2016]
2. Tim, O.: What is web 2.0? design patterns and business models for the next generation of software (2005) [Online]. Available: oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html
3. Bass, L., Clements, P., Kazman, R.: Single Page Web Applications. Manning, Shelter Island, NY (2013)
4. von Kistowski, J., Herbst, N.R., Kounev, S.: Using and extending LIMBO for the descriptive modeling of arrival behaviors. In: Proceedings of the Symposium on Software Performance 2014, pp. 131–140. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Best Poster Award, Stuttgart (2014)
5. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: a new graph rewrite language based on the unified modeling language and java. In: Selected Papers from the 6th International Workshop on Theory and Application of Graph Transformations, ser TAGT'98, pp. 296–309. Springer, London (2000) [Online]. Available: http://dl.acm.org/citation.cfm?id=645872.668867
6. García, D.F., García, J.: TPC-W E-Commerce benchmark evaluation. Computer **36**(2), 42–48 (2003) [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1178045
7. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language Toolkit. The Eclipse Series. Addison-Wesley (2009) [Online]. Available: http://books.google.de/books?id=8CrCXVZXLjcC

Chapter 4
ScaleDL

Gunnar Brataas, Steffen Becker, Mariano Cecowski, Vito Čuček, and Sebastian Lehrig

Abstract This chapter describes the family of languages required to analyze the scalability, elasticity, and cost-efficiency of services deployed in the cloud. First, the ScaleDL Overview Model describes the overall structure of a cloud-based architecture. Second, ScaleDL Usage Evolution specifies how load and work vary as a function of time. Third, ScaleDL Architectural Templates save time by reusing best practices. Fourth, the Extended Palladio Component Model is used for modeling software components and their mapping to underlying software services. The first three languages are new in CloudScale, while the fourth, Extended Palladio Component Model, is reused. For each language, we describe the basic concepts before we give an example. Tool support is then outlined. We list our catalog of Architectural Templates.

This chapter is structured as follows: Sect. 4.1 outlines the relation between the ScaleDL languages. For each language, we describe the basic concepts before we give an example. Tool support is also outlined. The ScaleDL Overview Model is described in Sect. 4.2. ScaleDL Usage Evolution is explained in Sect. 4.3. In Sect. 4.4 ScaleDL Architectural Templates are introduced in detail. Section 4.5 describes the Extended Palladio Component Model.

G. Brataas (✉)
SINTEF Digital, Strindvegen 4, 7034 Trondheim, Norway
e-mail: gunnar.brataas@sintef.no

S. Becker
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
e-mail: steffen.becker@informatik.uni-stuttgart.de

M. Cecowski • V. Čuček
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: mariano.cecowski@xlab.si; vito.cucek@xlab.si

S. Lehrig
IBM Research, Technology Campus, Damastown Industrial Estate, Dublin 15, Ireland
e-mail: sebastian.lehrig@ibm.com

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_4

4.1 Introduction

The Scalability Description Language (ScaleDL) is a collection of languages to characterize scalability, elasticity, and cost-efficiency aspects of cloud-based systems. For other aspects like system behavior, data models, etc., complementary languages like Unified Modeling Language (UML) must be used. ScaleDL consists of five languages: three new languages (ScaleDL Usage Evolution, ScaleDL Architectural Templates (ATs), and ScaleDL Overview Model) and two reused language (Palladio's Palladio Component Model (PCM) extended by SimuLizar's self-adaption language and Descartes Load Intensity Model (DLIM)). For each of these, we briefly describe their purpose and provide a reference to a detailed description later in this chapter:

ScaleDL Overview Model (developed in CloudScale) allows architects to model the structure of cloud-based architectures and cloud deployments at a high level of abstraction (cf., Sect. 4.2).

ScaleDL Usage Evolution (developed in CloudScale) allows service providers to specify scalability requirements, e.g., using evolution of work and load of their offered services (cf., Sect. 4.3).

Descartes Load Intensity Model (DLIM) (reused; see [1]) was originally designed to model load intensity in terms of evolution of arrival rates over time, but can also be used for modeling the evolution of work and load in general (cf. Sect. 4.3).

ScaleDL Architectural Templates (developed in CloudScale) allows architects to model systems based on best practices as well as to reuse scalability models specified by architectural template engineers (cf., Sect. 4.4).

Extended Palladio Component Model (reused; see [2]) allows architects to model the internals of the services: components, components' assembly to a system, hardware resources, and components' allocation to these resources; the extension allows, additionally, to model self-adaptation: monitoring specifications and adaptation rules (cf., Sect. 4.5).

Figure 4.1 shows an overview of how the languages relate to each other, and the transformations and other components they are input to and output from. We will detail this in the next sections, for one language at a time.

4.2 Overview Model

Important issues while modeling cloud architectures and their deployments are their replicability and the necessity of high-level descriptions that can be easily understood and shared. Common approaches for sharing such models are diagrams and descriptions that are not useful as formal definitions of architectures or deployment strategies that can be used automatically with different tools, and formal

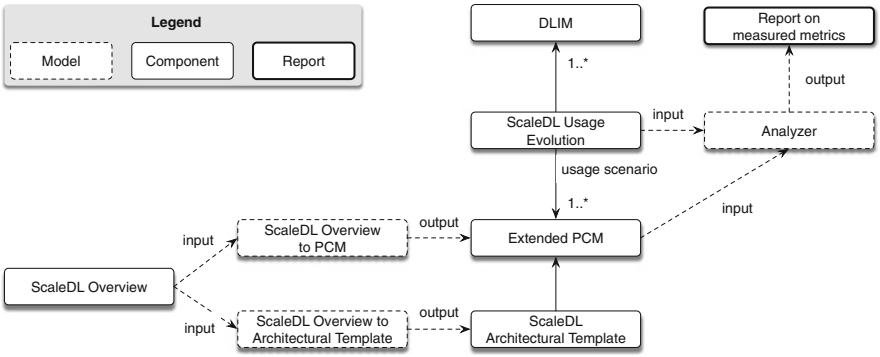

Fig. 4.1 Overview of ScaleDL languages and their relationships

descriptions, such as deployment scripts or recipes, provide little utility for the high-level study of the defined systems.

DEFINITION 4.1: OVERVIEW MODEL

The Overview model is a meta-model that provides a design-oriented modeling language and allows architects to describe the structure of cloud-based systems. It provides the possibility of representing private, public, and hybrid cloud solution, as well as systems running on a private infrastructure.

We will first describe concepts in the Overview Model in Sect. 4.2.1, before we sketch an example in Sect. 4.2.2. In Sect. 4.2.3 we detail the tool support.

4.2.1 Concepts of Overview Model

The Overview model consists of Architecture, Deployment, and Specification models. The Architecture model provides a descriptive abstraction of the system's architecture without any deployment or performance information, which is defined in Deployment and Specification models and referenced to the Architecture model.

The Architecture model was designed as a base model for describing and visualizing components in a cloud environment. It contains different cloud environments and external connections, for linking operations with a user interface or an external black-box service, or to interconnect cloud environments in a hyper-cloud configuration. The cloud environment component contains basic information about data centers and performance in different regions, using descriptors defined in the Specification model. From the architecture point of view, the most important components in the Architecture model are internal connections, defined between two components, and a layered tree structure of services, defining a deployment

hierarchy. The latter is separated into three categories: the infrastructure layer, the platform layer, and the software layer.

The infrastructure layer contains provided Infrastructure services. Services mentioned as provided in the Overview model context do not contain further information about the implementation of a lower-level mechanics. This can be substituted and described with independent components, capable of executing higher-level routines, according to measured performance limitations. A set of aforementioned components defines the Deployment. In practice, every service, except for the physical hardware, needs a lower-level service on which it operates, but sometimes the exact specification is not known. To make the Overview model flexible for such cases, the Provided service interface can be applied on any service inside the Architecture service layer stack to obscure or simplify the complexity of the underlying layers. Infrastructure services are the lowest in the service layer hierarchy, so they must provide the Deployment model. Practical implementation of the Infrastructure service is the Computing infrastructure service, which reference a Deployment and a Computing resource descriptor.

The platform and software layers contain provided or deployable platform and software services. The platform services can act as a placeholder for the software services or provide a full description of a software by describing the application's inner-working with the PCM language in Sect. 4.5.

The descriptions of cloud components inside the Architecture and the Deployment package are defined inside the Specification model to allow easy extensibility or migration and performance testing between different cloud providers.

4.2.2 Example of Overview Model

An example model can illustrate more clearly the Overview Model's capabilities. Figure 4.2 shows a visual representation of an Overview Model of a simple system composed of two Tomcat applications running on Amazon EC instances, and which make use of the Amazon DynamoDB service, a MySQL RDS service, and an e-mail service.

The model includes networking details such as average latency and bandwidth, which can be used for the behavior analysis and simulation of the overall system. Several other data can be defined. For example, we can define the expected statistical distribution of response time for an external service, or the expected capacity of a computing unit.

The general Overview Model can thus give a quick understanding of the overall architecture and deployment strategy, but contains also detailed information that can be used at different stages of the evaluation of the solution.

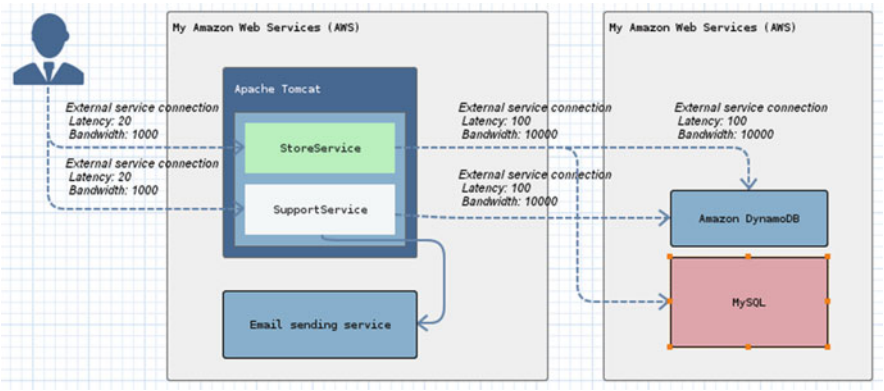

Fig. 4.2 Hybrid cloud architecture example

4.2.3 Tool Support for Overview Model

To simplify the creation and editing of the Overview model, a specially designed diagram editor with a components palette, properties view, and a number of supporting wizards has been created. The graphical diagram offers an organized view of the cloud solution architecture, and the supporting editors, together with a properties view, provides the ability to alter service descriptions.

The creation of the Overview model starts by choosing the desired cloud environment. Currently supported environments in the CloudScale Environment are Amazon web services (AWS), OpenStack, SAP Hana Cloud, and generic. The latter one contains services that are environment independent. The system architect has the ability to model hybrid cloud architectures (see Fig. 4.2) by creating multiple cloud environments in a single Overview model. When the environment is created, the architect can stack infrastructure and platform services. If the implementation of a service is described as Partial PCM model, it can simply be imported into the service component of the Architecture model. A lot of options and settings inside the properties view of the diagram are selection dependent to speed up the modeling and configuring process. More demanding, in terms of configuration, software services have dedicated editors for specifying operation interfaces, data types, and required connections.

When the Overview model is finished, it can be used for performance and cost analyses, because it contains the complete description of a cloud solution.

4.3 Usage Evolution

Existing modeling environments like Palladio [2] have usage scenarios with a fixed value for arrival rate (open) or for population (closed). Work is also fixed. If you want to analyze what happens with your service during evolution of work and load,

the current approach would be to run several simulations and manually change load and work. This manual process is time-consuming as well as error prone, as new values need to be entered in several locations of the model for each run.

Here, we propose a more direct approach using usage evolution that particularly accounts for transient phases, i.e., phases in which the system is subject to contextual changes during simulation. By *usage evolution* we mean how usage-oriented concepts like work, load, and quality thresholds vary as a function of time (Definition 4.2).

DEFINITION 4.2: USAGE EVOLUTION [3]

Usage evolution describes how usage-oriented concepts like work, load, and quality thresholds vary as a function of time.

In this section, we will first describe concepts for usage evolution in Sect. 4.3.1, then the usage evolution on an example in Sect. 4.3.2, before we describe tool support for usage evolution in Sect. 4.3.3.

4.3.1 Concepts for Usage Evolution

Figure 4.3 illustrates the meta-model for usage evolution in CloudScale. Elements imported from the PCM and DLIM are shown in light gray in the figure. The meta-model is defined to allow the specification of how the usage evolves over time. The root element of a Usage Evolution model is the *UsageEvolution* element. A *UsageEvolution* contains an ordered list of one or more *Usage* elements.

A *Usage* defines how one *UsageScenario* from a PCM model evolves over time. The referenced *UsageScenario* defines the initial values for work and load. Evolution of load for the *Usage* is described in a DLIM model (shown as a relation to the *Sequence* element from DLIM in the figure). The output values of the

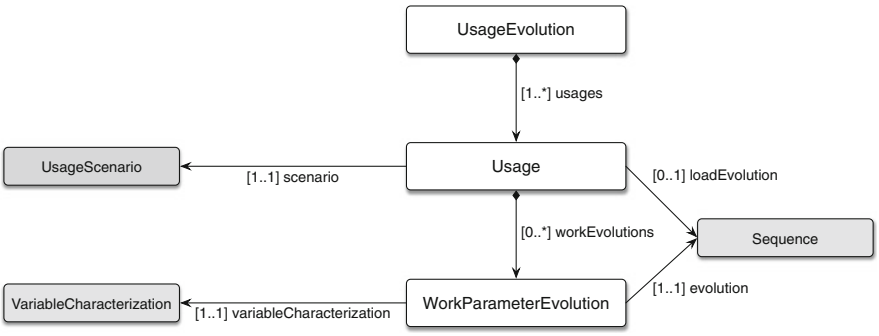

Fig. 4.3 Meta-model for Usage Evolution

DLIM model determine the evolved arrival rates in the case of open workload, and population in the case of closed workload. A *Usage* can also contain zero or more *WorkParameterEvolution* objects that each describes how a work parameter of the PCM model evolves in terms of a DLIM model. Which work parameter is to evolve is determined by a reference to a *VariableCharacterisation* defined in the PCM model.

The root element of a DLIM model is a *Sequence*, which can hold one or more function containers. Each such container holds a function for characterizing seasons and trends. Seasonal variation can be daily (peaks during lunch breaks), weekly (peaks at weekends), monthly (peaks at pay days), and yearly (peaks before Christmas). Trends describe a gradual increase or decrease and may be linear or exponential. Functions can also be combined with other functions through a list of combinators that can have addition, multiplication, or subtraction semantics. For further details about the DLIM meta-model, see [1].

4.3.2 Example of Usage Evolution

In this section, we will describe examples of usage evolution for load as well as for work. In Sect. 6.2.2 more examples will be shown.

Since CloudStore has many different operations, each of these operations could have had different load evolutions, but a natural simplification is to have one load parameter, representing the evolution of the average operation, instead of several operations evolving independently. Figure 4.4 shows how the number of users vary during a 3-min period for CloudStore. Initially, there are 2000 simultaneous users. In the first half-minute, this figure illustrates a linearly increasing trend, followed by a stable period with 5000 users for 1 min, and then a new increase up to a new stable period in the last minute. In this last stable period, there are 10,000 users.

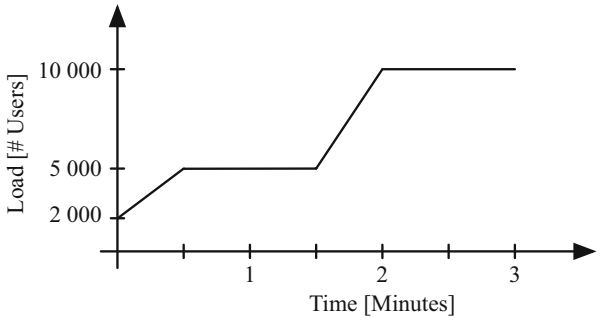

Fig. 4.4 Load evolution

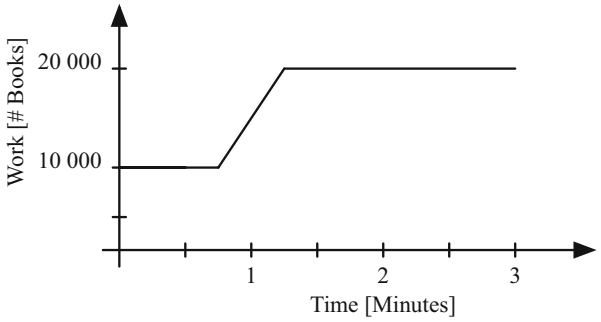

Fig. 4.5 Work evolution

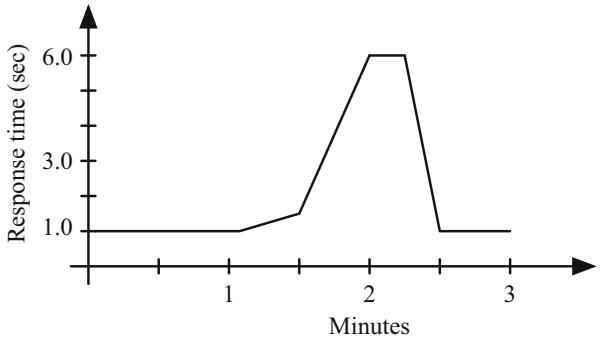

Fig. 4.6 Simulation results—response time

Figure 4.5 depicts evolution of the work parameter describing the number of books. For the first three-quarters of a minute, the number of books is stable at 10,000 books. Then, during the next half-minute, we have a linear increase up to 20,000 books, which defines the stable load during the remaining period. The reason for this sudden increase in the number of books can, for example, be the inclusion of several new publishers in the book store.

The response time for the Product Detail operation of this example is shown in Fig. 4.6 and is calculated by running the simulation based on the usage evolution model. Assume that the service-level objective (SLO) for this operation is a 90% response time of 3 s. The *x*-axis on this figure is again minutes, and the *y*-axis is the 90% response time in seconds. From the figure, we can determine that the initial increase in load is handled by the system without any increase in response time. The increase in the number of books just after 1 min results in a small increase in the response time. The further increase in the arrival rate between 1.5 and 2 min leads to a sharp increase in the response times. However, after less than 0.5 min, the response time drops again, even if neither the load nor the work on the system drops. The reason is of course that because of auto-provisioning, we now use more cloud resources. Since this auto-provisioning takes some time, we experience high

response times, before the system eventually returns to normal operation again and SLOs are no longer violated.

4.3.3 Tool Support for Usage Evolution

The load of the system is described as part of the usage scenarios in the Analyzer [2], either as a closed load, based on a fixed population and a waiting time, or as an open load described by the inter-arrival rate of new users. Work is modeled as a characterization of input and output parameters of operations, and is included in the service effect specifications (SEFFs), with some initial values defined in the usage scenario.

Palladio's usage scenarios define static values for load and work. To support evolution in terms of *variations* in load and work over time, we have extended the modeling support of Palladio by introducing a usage evolution model based on DLIM, which is used by the load intensity modeling tool LIMBO [1]. While work evolution and load evolution are explicitly modeled, other evolutions require a new simulation: change in quality thresholds, new or deprecated operations, or change in the implementation of operations. See [3] for more details.

We have added support to Palladio's simulator SimuLizar [4] such that it can run simulations following the characteristics of usage evolution models. At simulation time, SimuLizar updates workload parameters according to *Load* (as specified by *Usage* elements) and *Work Evolutions* (as specified by *WorkParameterEvolution* elements). For these updates, SimuLizar samples the linked DLIM models once per simulated time unit.

4.4 Architectural Templates

The creation of architectural models—especially with analysis capabilities—can involve huge efforts by software architects. During creation, architects may have to manually use architectural knowledge in the form of CloudScale's HowTos (cf. Sect. 2.9). Common design-time analysis approaches unfortunately lack support for directly reusing such HowTos. This lack makes the design space for software architects unnecessarily large; architects potentially consider designs that violate the constraints of HowTos. Moreover, this lack makes an automatic processing of HowTos impossible; architects have to manually model elements described in HowTos over and over again, even in recurring situations. These issues point to an unused potential to make the work of software architects more efficient.

To use this potential, the CloudScale method introduces so-called Architectural Templates (ATs) [5] for efficiently modeling and analyzing quality-of-service (QoS) properties of software architectures. With ATs, software architects can quantify

such quality properties based on reusable analysis templates (Definition 4.3) of recurring architectural knowledge such as documented in CloudScale's HowTos. Architects only have to customize such templates with parts specific to their software application, thus reducing effort and leading to a more efficient engineering approach.

DEFINITION 4.3: TEMPLATE

A template is a reusable model blueprint from which (parts of) concrete models can be instantiated.

In this section, we will first describe AT concepts in Sect. 4.4.1. An example of an AT is described in Sect. 4.4.2. Based on our catalog of HowTos in Table 2.1, we derive a catalog of ATs in Sect. 4.4.3. Tool support for ATs is outlined in Sect. 4.4.4.

4.4.1 Concepts of Architectural Templates

The AT language is a language to specify and apply templates of architectural models for model-driven design-time analyses [6]. Such templates are called Architectural Templates (ATs).

At the core, the AT language distinguishes between ATs, i.e., template types and their instances. *ATs* consist of (1) *roles* (Definition 4.4), with *parameters* and *constraints* to extend and restrict elements of architectural models; (2) a *mapping* of such roles to a semantically equivalent architectural model construct (translational semantics [7]); (3) a *documentation* that references the HowTo to be modeled, e.g., to point to the SLOs potentially impacted by the AT; and (4) an optional *default AT instance* to be used as a starting point for modeling. These constituents allow to formalize HowTos as ATs and to collect them in *AT catalogs*.

AT instances refer both to an AT and to an architectural model, e.g., a ScaleDL model, into which the AT is instantiated. AT instances particularly include a set of bindings that instantiate AT roles with bound architectural elements and actual parameters.

4.4.2 Example for Architectural Templates

Let us have a look at a concrete application of the so-called "loadbalancing" AT for component instances. The loadbalancing AT specifies a template for the loadbalancing HowTo of component instances (see Sect. 2.9). Next, we are going to apply the loadbalancing AT to our running example (CloudStore), as introduced in Sect. 2.2.

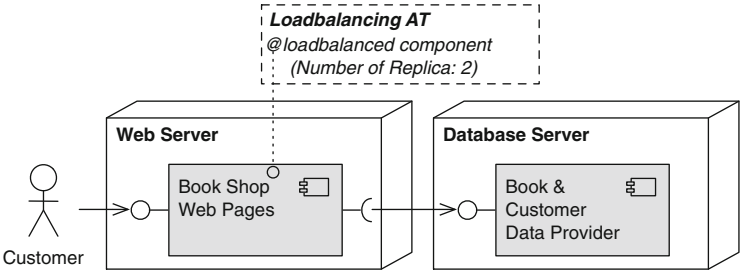

Fig. 4.7 CloudStore's architectural model annotated with the "loadbalanced component" role of the "loadbalancing" AT for component instances

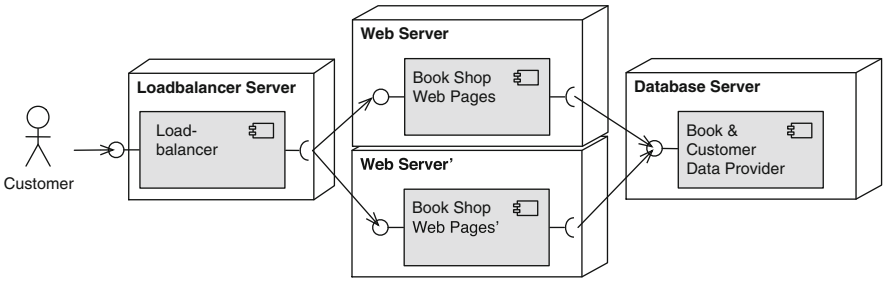

Fig. 4.8 CloudStore's architectural model after the execution of the mapping

Figure 4.7 illustrates the modified architectural model of CloudStore. In the modified version, the `Book Shop Web Pages` component instance is annotated with the `loadbalanced component` role (Definition 4.4). Moreover, we set "2" as an actual parameter value for the formal `Number of Replica` parameter.

DEFINITION 4.4: ROLE [8]

A role is the responsibility of a design element within a context of related elements.

Semantically, this model can then be mapped to the architectural model illustrated in Fig. 4.8. The new model includes a load balancer in front of the `Book Shop Web Pages` component instance. Moreover, based on the actual parameter of `Number of Replica`, the load balancer distributes workload over two copies of this component instance. The annotation of the original model is not needed anymore because its semantics have now been equivalently expressed with common elements of architectural models (such a semantic definition is called "translational semantics" [7]).

This example application of an AT shows that ATs can simplify recurring modeling tasks: instead of manually modeling a load balancer and creating two replicas, a simple, declarative annotation in the form of a role and an actual parameter is

sufficient. ATs group such recurring modeling constructs within parametrizable AT roles that can be semantically mapped back to the original constructs.

Additionally, AT roles can include constraints. These constraints allow architects to receive direct feedback during modeling. For example, whenever they annotate component instances that are stateful, the AT application may be aborted (the load balancer HowTo demands stateless component instances). Such direct feedback and restrictions reasonably limit the design space for architects, thus reducing potential modeling mistakes by architects.

4.4.3 Catalog of Architectural Templates

Based on our catalog of HowTos (see Table 2.1 in Sect. 2.9), we also derived an AT catalog with carefully engineered ATs. Table 4.1 illustrates CloudScale's AT catalog: the table provides for each AT its application domain (first column); its name, which also points to the realized HowTo (second column); and AT roles, with their parameters in parentheses (third column).

Table 4.1 CloudScale's catalog of ATs

Application domain	AT (and HowTo) name	AT Roles and parameters
Business information systems	3-Layer	Presentation layer, middle layer, Data layer
	Loadbalancing (container)	Loadbalanced container (Integer: number of replica)
	Loadbalancing (component instance)	Loadbalanced component (Integer: number of replica)
Cloud computing	SPOSAD	Presentation layer, middle layer, data layer, replicable tier
	Horizontal scaling (container)	Horizontal scaling container (Integer: number of initial replica, double: scale-in threshold, double: scale-out threshold)
	Horizontal scaling (component instance)	Horizontal scaling component (Integer: number of initial replica, double: scale-in threshold, double: scale-out threshold)
	Vertical scaling	Vertical scaling container (double: scale-up threshold, double: scale-down threshold, double: rate step size, double: minimal rate, double: maximal rate)
Big data	Hadoop MapReduce	Map component, Reduce component

The ATs of the AT catalog directly correspond to the equally named HowTos of the HowTo catalog. We therefore refer to the section about the HowTos (Sect. 2.9) for detailed descriptions of the concepts realized in these ATs. The AT roles and parameters given in the third column of Table 4.1 are directly derived from these descriptions and realize corresponding concepts.

Section 4.4.2 provides an example for the loadbalancing AT for component instances. Another and similar example is the "loadbalancing" AT for containers. This AT introduces the role of a "loadbalanced container" with a formal parameter "number of replica" of type "Integer". Software architects can accordingly attach this role to a resource container, e.g., a virtual machine. Semantically, this container is then load balanced; i.e., a load balancer is introduced that distributes workload over replicas of the container. The actual parameter that architects set for "number of replica" determines how many of these replicas exist. These semantics correspond to the according descriptions of the loadbalancing HowTo.

Analogously to these examples, the CloudScale Wiki [9] documents each AT of CloudScale's AT catalog. This documentation includes a detailed description of the related HowTo, "before mapping" and "after mapping" descriptions with according figures, and a list of concrete constraints of AT roles.

4.4.4 Tool Support for Architectural Templates

Software architects can use the graphical editors of the CloudScale integrated development environment (IDE) to apply ATs. Architects select ATs from existing AT catalogs, e.g., from CloudScale's catalog (Sect. 4.4.3). When software architects start an analysis of an architectural model with applied ATs, the mappings of ATs are automatically (and transparently to the software architect) executed.

To specify additional ATs, the CloudScale IDE provides a graphical editor to specify the elements of ATs. The mapping of an AT is specified in a separate model transformation file. As model transformation language, QVT-O [10] is currently supported.

4.5 The Extended Palladio Component Model

A unique selling point of ScaleDL is that it not only documents cloud-based systems but also allows for (semi-)automated analyses of scalability, elasticity, and cost-efficiency. ScaleDL's key ingredient for these analyses is the "Extended Palladio Component Model" (Extended PCM), an architectural description language for elastic (i.e., cloud-based) systems. Models specified with the Extended PCM can be automatically analyzed by CloudScale's Analyzer tool. The other ScaleDL parts (Overview Model, Usage Evolutions, and ATs) can be mapped to the Extended PCM to enable their analysis.

In this section, we describe concepts of the Extended Palladio Component Model in Sect. 4.5.1. Section 4.5.2 shows an example of an Extended Palladio Component Model. Tool support for the Extended Palladio Component Model is discussed in Sect. 4.5.3.

4.5.1 Concepts of the Extended Palladio Component Model

In this section, we discuss the core of the Extended PCM—the PCM itself—to understand its basic paradigms for architectural modeling. Afterward, we describe PCM extensions for elastic environments that constitute the Extended PCM.

4.5.1.1 The Palladio Component Model

The PCM [2] is an architecture description language that particularly covers performance-relevant attributes. Instances of the PCM can therefore be analyzed with respect to performance metrics like response times, utilization, and throughput.

PCM instances are constituted of partial models. Each of these partial models is inspired by the UML and covers performance-relevant attributes of the system to be modeled:

Component Specifications. Models a repository of software components. Components provide and optionally require a set of interfaces. Components can be reused whenever their provided interface is required, or exchanged whenever other components provide the same interface.

For each operation of a provided interface, components include behavior descriptions, e.g., modeling requests to operations of required interfaces, demands to resources like CPUs and hard disk drives, and acquiring and releasing connections from resource pools. These behavior descriptions are called *service effect specifications* (SEFFs).

System Model. Models a system that instantiates and assembles the software components. The system provides interfaces on its own such that users can externally access them. For implementing its provided interfaces, the system delegates requests to appropriate component instances. If these instances require further interfaces, the system includes assembly connectors that delegate requests to appropriate providing interfaces of further component instances.

Resource Environment Model. Models the resource environment (e.g., in terms of hardware) in which the system is allocated. The environment consists of containers connected via networks. Containers can, for instance, represent bare-metal or virtualized servers. Containers particularly include a set of active resources like CPUs and hard disk drives. Each of these resources comes with different processing rates and scheduling strategies.

Allocation Model. Models the allocation from component instances (system) to containers (resource environment). Therefore, the allocation specifies which container component instances demand resources.

Usage Model. Models the workload to a system in terms of its users. The usage model consists of different usage scenarios, each being either a closed workload (fixed number of users) or an open workload (users enter based on inter-arrival rates). In each usage scenario, users can access operations provided by the system. Users access such operations with a certain probability and with specific work parameters, e.g., characterizing the size of input data.

4.5.1.2 Extensions for Elastic Environments

The PCM initially was designed for static environments, i.e., for resource environments that do not change the amount of their computing resource over time. However, the usage of information systems shifted from a static to a highly dynamic behavior that challenged such static environments. For example, online shops often observe workload increases before Christmas. In such scenarios, static environments demand that resources be aligned to the maximum workload to be expected (over-provisioning). Otherwise, customers will remain unserved, which eventually leads to business losses. The disadvantage of this solution is that such an over-provisioning is expensive during non-peak times.

Cloud computing, therefore, revised the assumption that resource environments are static: to minimize expenses for resources, their amount is now elastically adapted to changing workloads. CloudScale provides PCM extensions for modeling and analyzing these elastic resource environments. CloudScale's modeling extensions cover workloads that change over time (dynamic usage environments), self-adaptation rules that react on these changes by adapting the amount of resources, and monitors to trigger self-adaptation rules:

Usage Evolution Model. Usage Evolutions specify how workload parameters of PCM usage models change over time. For example, steadily increasing and periodically varying arrivals of users can be modeled. Section 4.3 details and exemplifies Usage Evolutions.

Self-Adaptation Rules. Self-adaptation rules react on changes of the monitored usage or resource environment. For example, when a certain response time threshold is exceeded, a self-adaptation rule could trigger a scaling-out of bottleneck components. These rules, therefore, consist of two parts, a trigger and an action that can be activated by the trigger. The trigger relates monitored values to pre-specified thresholds to determine whether to activate the action. The action describes the change in the system to be applied. Actions are formulated in terms of model-to-model transformation languages like QVT-O [10], StoryDiagram [11], and Henshin [12].

Monitors. Monitors describe which metrics should be recorded at specific measuring points. Monitors can, for instance, measure metrics like utilization of a specific CPU. The resulting measurements are used as input to the trigger of self-adaptation rules, which then potentially activates an adaptation action.

4.5.2 Example for the Extended Palladio Component Model

In this section, we describe the elements of the Extended PCM and Architectural Templates used by a model of CloudStore. Figure 4.9 gives a simplified high-level overview of these elements.

With this overview, software architects can easily follow the control and data flow (arrows) from customers through CloudStore's components (UML component symbols) allocated on various resource containers (UML node symbols). In Fig. 4.9, customers enter CloudStore via the `Book Shop Web Pages` component to browse and order books. To provide its functionality, `Book Shop Web Pages` requests information from the `Book Shop Business Rules` component. `Book Shop Business Rules` can in turn request payment services from an externally hosted `Payment Gateway`. Additionally, it can request data about books and customers from the `Book & Customer Data Provider` component. If a web page returned by `Book Shop Web Pages` references images, a customer's browser subsequently fetches these references via the `Book Image`

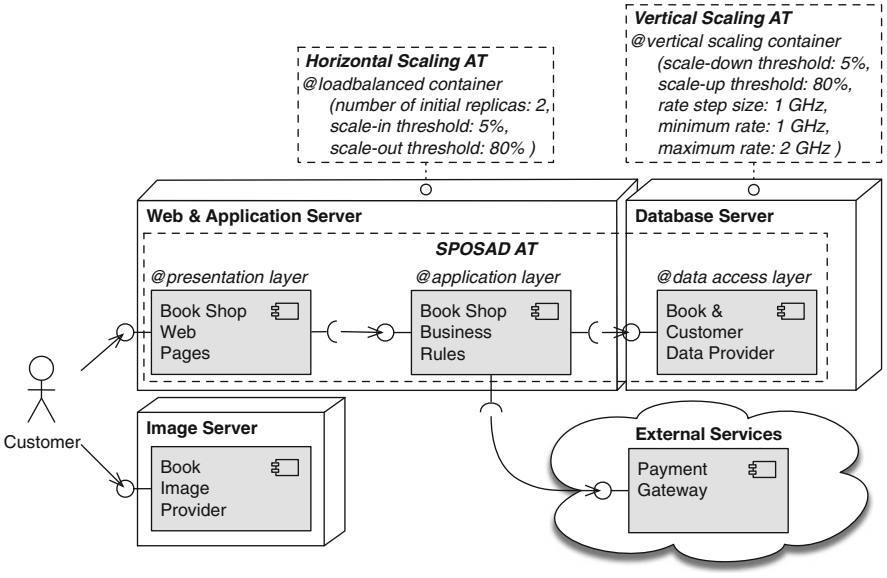

Fig. 4.9 Simplified ScaleDL model of the CloudStore online bookshop with annotated AT roles

`Provider` component. As illustrated in Fig. 4.9, `Book Shop Web Pages` and `Business Rules` are allocated on a `Web & Application Server`, `Book & Customer Data Provider` on a `Database Server`, and `Book Image Provider` on a dedicated `Image Server`.

All of these elements come from the Extended PCM: customers entering the system (Usage Model), components (Component Specifications) instantiated and assembled to CloudStore (System Model), and the allocation of these instances (Allocation Model) to different resource containers (Resource Model). As software architects, we are interested in analyzing the impact on CloudStore's QoS properties when applying architectural knowledge. Therefore, Fig. 4.9 additionally illustrates elements from ScaleDL's AT language: applications of the ATs *SPOSAD*, *Horizontal Scaling*, and *Vertical Scaling* are annotated (bold italic text in dashed boxes).

The *SPOSAD* AT (middle of Fig. 4.9) introduces roles to structure CloudStore into a *presentation layer* (bound to `Book Shop Web Pages`), an *application layer* (bound to `Book Shop Business Rules`), and a *data access layer* (bound to `Book & Customer Data Provider`). These roles constrain the bound components to only access the respective lower-level layer (in Fig. 4.9 shown from left to right). Moreover, the *SPOSAD* AT requires components on the presentation and application layers to be stateless. Because ATs formalize such constraints, an architecture tool with AT support (like the CloudScale IDE; cf. Sect. 8.7) can ensure their fulfillment, e.g., by forbidding direct connections from `Book Shop Web Pages` to `Book & Customer Data Provider`.

The *Horizontal Scaling* AT (top middle of Fig. 4.9) introduces a *loadbalanced container* role bound to the `Web & Application Server`. In a preprocessing step of a design-time analysis, a template engine will reflect the performance impact of this binding by creating a load balancer in front of the container that distributes workload. According to the parameters given in Fig. 4.9, the load balancer initially distributes workload over two container replicas and dynamically decreased or increased this number if the CPU utilization of the container drops below 5% or exceeds 80%, respectively.

The *Vertical Scaling* AT (top right of Fig. 4.9) introduces a *vertical scaling container* bound to the `Database Server`. A template engine will create adaptation rules that dynamically increase or decrease the processing rate of this container's CPU. The rate is decreased if CPU utilization drops below 5% and increased if it exceeds 80% (see the role's parameters in Fig. 4.9). These adaptations come in steps of 1 GHz within a range of 1–2 GHz (hence, the rate is either 1 or 2 GHz).

4.5.3 *Tool Support for the Extended Palladio Component Model*

Software architects can specify instances of the Extended PCM with the graphical editors of the CloudScale IDE. Once specified, architects can use various analysis tools to inspect the QoS properties of a modeled system. Moreover, if the source code of a system is already available, CloudStore's Extractor can be used to automatically create partial instances of the Extended PCM. This section briefly describes the analysis tools and the Extractor for the Extended PCM.

4.5.3.1 Analysis Tools

PCM instances serve as input (Fig. 4.10) (left) to various analysis tools (Fig. 4.10) (right):

Analyzer. CloudScale's Analyzer is a simulation of the modeled system. The simulation interprets the input PCM instance to provide measurements for performance metrics like response times. Because the Analyzer interprets PCM instances, it can also acknowledge changes of these instances during simulation time. This feature, therefore, allows to model self-adaptive systems: the execution of a self-adaption action transforms a current PCM instance into an adapted version. The Analyzer subsequently continues by simulating the adapted version. Moreover, the Analyzer supports ScaleDL's Usage Evolution models: at simulation time, the Analyzer updates workload parameters according to an input Usage Evolution model. For these updates, the Analyzer samples the Usage Evolution

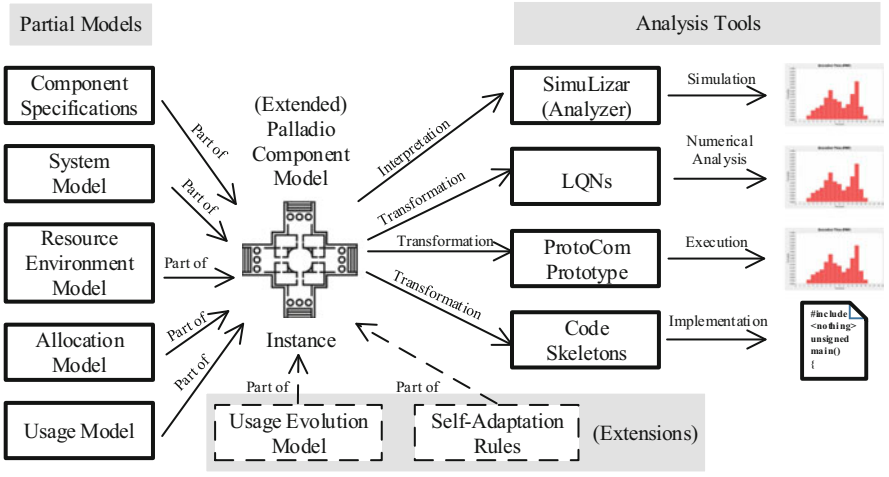

Fig. 4.10 Instances of the Extended PCM serve as input to various analysis tools

model once per simulated time unit to receive the concrete workload parameter for the current simulation time.

LQNs. Layered queuing networks (LQNs) extend queuing networks with layered structures and related elements, e.g., to fork/join different layers. Based on input PCM instances, transformations can create LQN models. These models can then be solved with numerical mean-value approximation methods, e.g., to provide mean response times as output. In contrast to simulations, these methods require less time for analyses; however, they provide only information about mean values.

ProtoCom Prototype. ProtoCom transforms PCM instances into runnable performance prototypes. Such performance prototypes can execute in different target environments and mimic demands to different types of hardware resources. Their execution, therefore, allows to take performance measurements for an early assessment of the modeled software system within a real environment.

Code Skeletons. Based on a PCM instance, a transformation generates appropriate code skeletons. These skeletons serve developers as a starting point for implementing the modeled system. Code skeletons are therefore especially important in forward engineering (see Chap. 6).

4.5.3.2 Extractor

ScaleDL models can be created manually. This is described in more detail in Chap. 6. However, CloudScale's Extractor tool can assist in potentially tedious manual tasks, given that source code is available.

The Extractor is a reverse engineering tool for the automatic extraction of partial Extended PCM models, thus lowering modeling effort for system engineers if source code already exists. The Extractor is based on the Archimetrix approach [13] that combines different reverse engineering approaches to iteratively recover and reengineer component-based software architectures.

The inputs to the Extractor are source code and configuration parameters for reverse engineering, e.g., thresholds that specify when to cluster classes into components. Software architects particularly have to decide which part of the system should be extracted. In a large system, architects may only be interested in a few critical services.

Once configured, software architect can start the Extractor. After parsing the source code, the Extractor clusters relevant elements based on these parameters into software components. The output is a partial Extended PCM model, i.e., a model that covers the component-based structure of the extracted source code as well as its control and data flow. The model is partial because it misses context information like system usage and hardware specifications.

While the Extractor relieves software architects from potentially tedious modeling tasks, software architects need to spend some effort finding the right configuration parameters.

In general, software architects start with the Extractor's default configuration and assess whether the resulting Extended PCM model is satisfying for their system. Software architects are typically unsatisfied if the result is too abstract (e.g., the Extractor clustered the whole system into one component) or too fine-grained (e.g., the Extractor clustered each class into a dedicated component). In that case, software architects alter configuration parameters and rerun the Extractor until satisfied.

The main parameters for the Extractor are (default values included):

Clustering Merge Threshold Max (End Value) (100) Start threshold between 0 and 100 for deciding whether to merge the elements of a component candidate into a single component by melting the component candidates in a single component. The lower the value is the fewer components are merged into single components.

Clustering Merge Threshold Min (Start Value) (45) End threshold between 0 and 100 for deciding whether to merge the elements of a component candidate into a single component. The lower this value is the more likely less connected component candidates will be merged into a single component.

Clustering Merge Threshold Increment (10) The increment between 0 and 100 for the merging components. The Extractor will merge components using a threshold starting at the start value and ending at the end value using this increment.

Clustering Composition Threshold Max (Start Value) (100) The start threshold between 0 and 100 for deciding whether to compose the elements of a component candidate into a new composed component by linking the component candidates using connectors. The lower the value is the fewer components are composed into a new composed component.

Clustering Composition Threshold Min (End Value) (25) The end threshold between 0 and 100 for deciding whether to compose the elements of a component candidate into a composed component. The lower this value is the more likely less connected component candidates will be composed into a composed component.

Clustering Composition Threshold Decrement (10) The increment between 0 and 100 for the composing components. The Extractor will compose components using a threshold starting at the start value and ending at the end value using this increment.

The Extractor has additional parameters which characterize the coupling of component candidates which need to be adjusted for each project to be extracted. A description of these parameters can be found in the CloudScale user manual [14].

4.6 Conclusion

This chapter presents ScaleDL as of a family of related languages. The ScaleDL Overview Model describes the overall structure of a cloud-based architecture. ScaleDL Usage Evolution specifies how load and work vary as a function of time. ScaleDL ATs save modeling efforts by reusing formally captured best practices. The Extended Palladio Component Model is used for modeling software components and their mapping to underlying elastic software services.

With the ScaleDL family of languages, software architects can specify critical aspects of a software system to enable analysis of scalability, elasticity, and cost-efficiency. Software architects may model the complete software system using all the languages, but selectively using a subset (or only fragments) of languages is also possible.

Subsequent chapters show how ScaleDL is integrated in the CloudScale method and how CloudScale's tools utilize ScaleDL. In particular, Chap. 8 describes how ScaleDL may be extended in the future.

References

1. Kistowski, J.V., Herbst, N., Zoller, D., Kounev, S., Hotho, A.: Modeling and extracting load intensity profiles. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '15, pp. 109–119. IEEE, New York (2015)
2. Becker, S., Busch, A., Brosig, F., Burger, E., Durdik, Z., Heger, C., Happe, J., Happe, L., Heinrich, R., Henss, J., Huber, N., Hummel, O., Klatt, B., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K., Küster, M., Langhammer, M., Lehrig, S., Merkle, P., Meyerer, F., Noorshams, Q., Reussner, R.H., Rostami, K., Spinner, S., Stier, C., Strittmatter, M., Wert, A.: In: Reussner R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K. (eds.) Modeling and Simulating Software Architectures – The Palladio Approach, 408 pp. MIT, Cambridge, MA (2016). [Online] http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures
3. Brataas, G., Stav, E., Lehrig, S.: Analysing evolution of work and load. In: QoSA: Conference on the Quality of Software Architectures. ACM, New York (2016)
4. Becker, M., Becker, S., Meyer, J., SimuLizar: design-time modelling and performance analysis of self-adaptive systems. In: Proceedings of Software Engineering 2013 (SE2013), Aachen (2013)
5. Lehrig, S.: Architectural templates: engineering scalable saas applications based on architectural styles. In: Gogolla, M. (ed.) Proceedings of the MODELS 2013 Doctoral Symposium Co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, October 1, 2013. CEUR Workshop Proceedings, vol. 1071, pp. 48–55 (2013). CEUR-WS.org [Online]. http://ceur-ws.org/Vol-1071/lehrig.pdf
6. Lehrig, S.: Applying architectural templates for design-time scalability and elasticity analyses of saas applications. In: Proceedings of the 2Nd International Workshop on Hot Topics in Cloud Service Scalability. HotTopiCS '14, Dublin, pp. 2:1–2:8. ACM, New York (2014). [Online] http://doi.acm.org/10.1145/2649563.2649573

7. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 1st edn. Addison-Wesley, Upper Saddle River, NJ (2008)
8. Buschmann, F., Henney, K., Schmidt, D.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages. Wiley, New York (2007)
9. CloudScale Wiki: HowTos. http://wiki.cloudscale-project.eu/HowTos [Visited on 12/19/2016]
10. Object Management Group (OMG), Meta Object Facility (MOF) 2.0 — Query/View/Transformation Specification. Technical Report OMG Document Number: formal/2011-01-01 (2011). http://www.omg.org/spec/QVT/-1.1/
11. von Detten, M., Heinzemann, C., Platenius, M., Rieke, J., Travkin, D., Hildebrandt, S.: Story diagrams - syntax and semantics, Software Engineering Group, Technical Report, Heinz Nixdorf Institute, University of Paderborn (2012)
12. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I. MODELS'10, Oslo, pp. 121–135. Springer, Berlin (2010). [Online] http://dl.acm.org/citation.cfm?id=1926458.1926471
13. Platenius, M.C., Von Detten, M., Becker, S.: Archimetrix: improved software architecture recovery in the presence of design deficiencies. In: Software Maintenance and Reengineering (CSMR 2012), pp. 255–264. IEEE, New York (2012)
14. CloudScale: User Manual of the CloudScale Environment. http://www.cloudscale-project.eu/media/filer_public/2016/02/01/cloudscaleenvironment-userguide_1_1.pdf [Visited on 12/19/2016]

Part III
The CloudScale Method for Software Architects

In this part, we detail the CloudScale method introduced in Part I. We describe the manual steps required when using the ScaleDL tool suite introduced in Part II.

To describe the manual and tool-supported method steps, this part gives an overview of the complexity and the amount of manual work involved in using the CloudScale method. We explain the significant trade-offs between the amount of manual work and the prediction accuracy for scalability, elasticity, and cost-efficiency. This is useful for anyone dealing with systems where these qualities are important. The method steps are flexible and can be merged with a variety of commonly used development methods.

Chapter 5 describes the CloudScale method in detail. The forward engineering part of the CloudScale method, using modeling tools, is then detailed in Chap. 6. This is applicable when developing a new service. The reverse engineering and reengineering parts of the CloudScale method are outlined in Chap. 7. These are applicable when an existing (legacy) service is modified.

Chapter 5
The CloudScale Method

Gunnar Brataas and Steffen Becker

Abstract This chapter details the CloudScale method. We describe its high-level process with the most important steps. We look more closely at the CloudScale method from Sect. 2.1 and detail it with respect to the developer roles executing it. We also introduce the two major method use cases. Method use case I is about analyzing a modeled system; method use case II deals with analyzing and migrating an implemented system. All discussions in this chapter are guided by the granularity of the analysis you want to perform, hence; this chapter also introduces granularity as a key concept and discusses how to find the right one.

As granularity is important for all steps of the CloudScale method, it is introduced in Sect. 5.2. As a second basis, our graphical notation is described in Sect. 5.3. The method description starts with an introduction into the CloudScale method roles in Sect. 5.4. As a core section in this chapter, Sect. 5.5 gives a detailed overall overview on the CloudScale method. Afterward, the following sections give details on all method steps: Sect. 5.6 outlines how to identify service-level objectives (SLOs), critical use cases, and their associated key scenarios from business needs; Sect. 5.7 then describes how to transform the SLOs and critical use cases into scalability, elasticity, and cost-efficiency requirements. Afterward, the two main use cases of the CloudScale method are introduced: Sect. 5.8 outlines how to use models to analyze a system's properties, while Sect. 5.9 sketches how to analyze implemented and executable systems. Finally, Sect. 5.10 briefly describes how to realize and operate the system.

G. Brataas (✉)
SINTEF Digital, Strindvegen 4, 7034 Trondheim, Norway
e-mail: gunnar.brataas@sintef.no

S. Becker
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
e-mail: steffen.becker@informatik.uni-stuttgart.de

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_5

5.1 Introduction

As already described in Sect. 1.10, the CloudScale method guides stakeholders by describing the steps to follow when engineering scalable, elastic, and cost-efficient systems. It also describes the set of different stakeholders involved. The CloudScale method as presented in this book is based on initial ideas published in [1] and further refined in [2]. In addition, it was inspired by the Q-ImPrESS method to engineer evolving service-oriented systems [3].

The basis for the CloudScale method is CloudScale's tools, which automate some parts of the method. The CloudScale method describes the input as well as the output of the CloudScale tools. In some cases, the input to a CloudScale tool is produced by another CloudScale tool, but manual steps may also be required to produce the required inputs.

Before starting the CloudScale method, you must have a clear idea of what you want to learn. Possible overall objectives are:

- Trade-off between cost, functionality, and quality during development. You may, for example, compare two different architectures.
- Trade-off between cost, functionality, and quality during modification. The new parts can be a new operation, or a new architecture. A new architecture may simply mean to compose existing services and components differently, but it may also mean replacing some of them.
- Compare scalability, elasticity, and cost-efficiency of competing services.
- Compare competing deployments for an existing service.

Common for all these objectives is that you need to do some sort of scalability, elasticity, or cost-efficiency analysis. The CloudScale method is a method for performing such analysis. When you do this analysis, you have a granularity trade-off. Generally, answering a more detailed question, like finding an optimal deployment, requires more details, compared to finding the best of two competing architectures. An important part of this method is the necessary manual steps, like setting objectives for scope and accuracy, setting configuration parameters, instrument source code, or finding out if the defined quality objectives are met. Which guidance is available for performing them and how do you know if you have to adjust something, and then what shall you adjust?

5.2 Granularity

By granularity, we refer to the level of detail. A coarse-grained model has few details compared to a fine-grained model, for example. We will describe this in more details for each method step, but more generally, the level of detail will have consequences both for what you put into the analysis as well as what you get out of the analysis and the time an analysis will take.

For what you put into the analysis, two aspects related to granularity, or level of detail, are important:

- Amount of manual work, or effort used to follow the method. Making a scalability model of a service consists of several manual steps, and generally the effort is related to the sophistication of the model. The level of detail of a model increases when you model more operations, more components, more resources, and more complex relation between them.
- Run time to do the analysis on the computer. Most often this time will be negligible, but for some analyses you may spend several hours, and then this time will also influence the amount of manual work. The amount of instrumentation will often be related to run time, because the more instrumentation you have, the higher the run time will be.

When it comes to what you get out of the analysis, three aspects of the result are important and are also related to granularity, or level of detail:

- Precision, relating to the repeatability of the results, normally quantified by a confidence interval [4]. To increase precision (i.e. reducing confidence intervals), you can run a simulation several times (with different seeds).
- Accuracy, the difference between the reported value and the "real" value [4]. Generally, a more complex model may be more accurate, since it is then easier to match reality. Validation where you compare the "actual" service with the modeled service is the key to increase accuracy [4].
- Scope, relating to coverage. You may look for scalability problems in the complete service as well as all its underlying services, a broad scope, but you may also confine the scalability investigation to one of the many classes inside of the service, a narrow scope. Similarly, you may focus on the use of processing resources and ignore the use of storage resources. You may also focus on one critical operation and ignore the remaining operations.

Generally, the more you put into the method in terms of effort and run time, the more detailed your model will be and the more you get out in terms of precision, accuracy, and scope. A comprehensive approach gives good precision but with a high effort, whereas a coarse approach gives low precision with a low effort. An important part of the method is to shed light on this granularity trade-off. The level of granularity should be sufficient for meeting this objective, but not more detailed, because then it will also be too costly in terms of manual effort.

This granularity trade-off is important, because the manual effort involved using our tools as well as our methods is *the* showstopper for their widespread use. Using more coarse-grained models, this manual work may be reduced, but then at the cost of precision, accuracy, and scope. The question afterward becomes: which precision, accuracy, and scope are required for spotting scalability/elasticity/cost-efficiency problems in a software service? The answer may not be simple. For example, more accurate instrumentation/models may be required to spot elasticity problems

than what is required to spot scalability problems. Probably a baseline scalability model will be required first, also with validation, before it is meaningful to analyze elasticity and cost-efficiency. We recommend a coarse approach first and afterward to increase the granularity of key parts so that you reach the given precision/accuracy and scope. We advise to avoid small increments as this leads to too many iterations of the method.

5.3 Method Notation

The notation used in this chapter as well as its decomposed method steps in later chapters is shown in Fig. 5.1 and explained below:

- Start or stop: describes the start and stop of these method steps. For a decomposed process, you start and stop where the high-level process starts and stops.
- Tool-driven process: a task which is supported by a tool. This process may be later decomposed.
- Decision: the flow of control depends upon a decision. This decision may involve complex manual tasks.
- Artifact: a file used to store text or models.
- External artifact: an artifact which is external to the process shown.
- Manual tasks: manual, complex tasks which may also be assisted by tools outside of CloudScale like text editors, compilers, monitoring tools, etc.
- External manual task: a manual task which is external to the process shown.
- Role: one of CloudScale's roles.
- Data flow: data flow in the specified direction.
- Data & control flow: data flow as well as flow of control with manual work in the specified direction.
- Parallel tasks: synchronization points between or after parallel tasks.

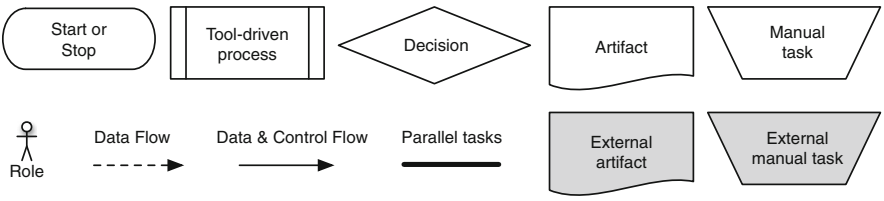

Fig. 5.1 Notation used in the CloudScale Method

5.4 Roles in the Method

In this section, we identify major roles required to (re)design scalable services in a cloud environment. We define the following six roles:

Service consumer: a person or enterprise entity that uses the service and its resources and therefore has a service-level objective (SLO) for this service. The service consumer is part of a larger business process. As this business process has business needs, there are resulting SLOs the service consumer needs to have fulfilled by a supporting IT system. Since there are different delivery models of cloud services (Infrastructure as a Service [Iaas], Platform as a Service [PaaS], Software as a Service [SaaS], and all of their subgroups), you can distinguish different service consumer roles for each of these delivery models. Depending on the type of service and their role, the consumer works with different user interfaces and programming interfaces.

Product manager: responsible for identifying system requirements and defining development goals, especially from the business perspective. The product manager is engaged in making decisions regarding the requirements fulfillment and business potential of the solution. The main interests of the product manager are the overall system behavior, architecture compliance, and price of the final solution. He is also a final decision-maker for the solution and negotiation with the customer about service/system acceptance and approval of the system evolution during operation.

System engineer: responsible for deploying the service and for the monitoring of the system in operation. Based on monitoring results, the system engineer optimizes the system's operation parameters. If required, the system engineer initiates the system evolution process steps so that further redesign and reimplementation of the system may be triggered if it is impossible to fix the system by fine tuning. The system engineer cooperates with all other roles during the system lifecycle.

System architect: is the architect of a certain layer in the cloud stack and the main system modeler. He is responsible for selecting the system components on a certain cloud layer and their interaction. The responsibility of the system architect is to find an optimal cloud service organization and deployment for all used cloud services. The system architect cooperates with the product manager and the service developer. He is also the main user of the tools and methods during design. During the system design phase, the system architect needs to provide an optimal evolution scenario and include optimal system components into the system architecture.

Service developer: develops a cloud service for a deployment model on a certain layer in the cloud stack. The service developer is responsible for both development and testing during service realization, and for preparing the system deployment process. The service developer cooperates with the system architect for checking realized services and with the system engineer when preparing system deployments. Most of the current deployed cloud services are developed

for SaaS cloud deployment models. Services developed for the IaaS and PaaS deployment models will subsequently be used by SaaS developers and cloud providers. The service developer uses infrastructure, as well as all accompanying mechanisms, provided by the cloud service provider on certain cloud stack layers.

Service provider: delivers the service to the consumer. The service provider is responsible for fulfilling SLOs and other requirements toward service consumers. She prepares service requirements and interacts with system engineers to enable an appropriate service, operates the system during the whole system lifecycle, and identifies the needs for system evolution.

The cloud computing definition published by the NIST [5] defines five basic cloud computing actors: cloud consumer, cloud provider, cloud auditor, cloud broker, and cloud carrier. The NIST actor cloud consumer is similar to the service consumer role in CloudScale. However, we focus more on the service itself and less on the cloud infrastructure. The latter three roles may also be relevant to scalability analysis, but are not considered further in this book.

The NIST actor for cloud providers is further decomposed into five NIST actors for service deployment, service orchestration, cloud service management, security, and privacy. The CloudScale role of system engineer aggregates the NIST actors for service deployment, service orchestration, cloud service management, but with a focus on scalability engineering. The NIST actors for security and privacy are complementary to the CloudScale roles.

The CloudScale roles for product manager, system architect, service developer, and service provider are not explicitly mentioned by the NIST, but they are relevant to CloudScale with our focus on (re)design.

5.5 Method Steps

The CloudScale method was introduced in Sect. 2.1. In this chapter, we show which roles are relevant to each method step. We also point out two main method use cases: one for modeling projected services and the other for reengineering existing services.

The product manager has tight cooperation with the service customer, and as a result, the product manager identifies the need for a new service. He or she elicits more or less vague business-oriented scalability, elasticity, and cost-efficiency requirements for this new service. In cooperation with the service provider, the product manager identifies SLOs, critical use cases, and key scenarios. In our context, critical use cases are the riskiest operations. Key scenarios describe when these operations have the highest workload.

Based on the business-oriented scalability, elasticity, and cost-efficiency requirements defined in the previous step, the system architect joins the discussion with the product manager and the service provider, and, together with them, agrees

on projected workloads. In this way, the technical requirements for scalability, elasticity, and cost-efficiency requirements are established.

In this scenario, the system architect has to design a new system. Therefore, she uses the CloudScale method in method use case I: the analyses will be based on a model as no useable source code exists so far. This method use case is described in more detail in Sect. 5.8. In this method use case, the system architect, together with the service developer, specifies a Scalability Description Language (ScaleDL) model. This is a complex task with several manual steps, where the system architect provides high-level information concerning the overall architecture describing how the components fit together and where the service developer fills in all the details about the components. Some details may be required from the system engineer concerning the cloud resources used. Moreover, the requirements for scalability, elasticity, and cost-efficiency are detailed as part of working with the ScaleDL model.

When the system architect, together with the service developer, is satisfied with the ScaleDL model, it is fed into the Analyzer. The system architect, together with the service developer, uses the results from the simulation in the Analyzer to improve both the high-level architecture as well as more detailed design choices. When the service developer, together with the system architect, decides that the service model is sufficiently mature, it is realized. After realization, service implementation is later deployed.

When a service is deployed, the system engineer takes over the responsibility. Hopefully, he will discover any poor scalable, elastic, or cost-efficient behavior before the service consumers do, and will then trigger the required reengineering steps in method use case II. Method use case II deals with analyzing and migrating an implemented system, and its steps are performed using the Spotter tools, which can identify scalability root causes based on either code or a running system. This method use case is described in more detail in Sect. 5.9.

There are fundamentally two ways of progress during design: adding details and making implementation decisions. These two will often be linked. As part of the development process, when detailing the architecture or implementing the system, the level of detail will increase:

Decomposition of operations: so that one operation becomes several operations. The opposite process of aggregation of operations, where operations are merged, is less likely.

SLOs may be detailed: where groups of operations sharing an SLO each get individual SLOs.

More work parameters: may be required to cope with the increased level of detail in a more elaborate design.

5.6 Identify Service-Level Objectives, Critical Use Cases, and Key Scenarios

In this section, we look at the requirements from a business point of view. A first step when executing the CloudScale method is to identify which of the system's SLOs imply the most risks from a business point of view. This is the first step of the CloudScale method. You can find it at the top of the actions in Fig. 5.2.

Risks are often functional, i.e., implementing the wrong functionality or implementing it different to customers' expectations. For example, the use case to administer the bookshop is important from a functional point of view but does not intersect with any risks concerning scalability, elasticity, or cost-efficiency. Therefore, for the CloudScale method, we need to identify use cases where those three properties are risky to implement. We call such use cases *critical*. Knowing the critical use cases is important for any kind of analysis—either model based or on the real system—because they are the major input to focus on the analysis steps. The reason is that these steps can be quite time consuming. Accordingly, it is important

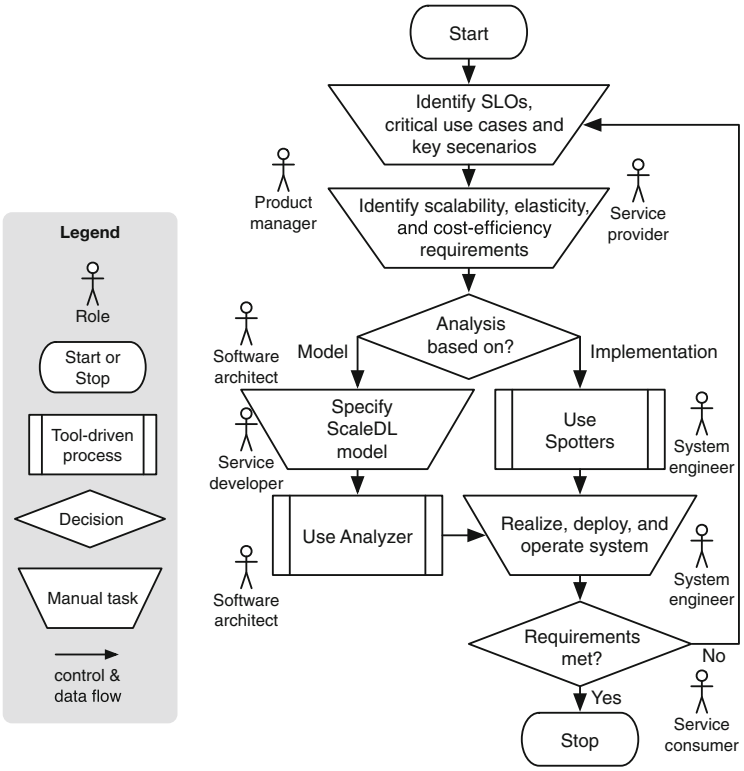

Fig. 5.2 High-level process steps and roles of the CloudScale method

to execute them not on the complete system but only on the identified critical use cases in order to keep the overall task small and manageable. As critical use cases are those which deal with high-risk scenarios, the following outlines the risks for scalability, elasticity, and cost-efficiency.

Let us focus on scalability risks first. As described in Sect. 1.3, an SLO consists of a quality metric and a quality threshold for this metric. In Sect. 2.3.1 we used the 90% response times as metric and the quality thresholds for the four CloudStore operations were between 1 and 5 s.

To identify scalability risks, software architects first need to identify rough SLOs for the most important functionality, i.e., the most important operations. Based on these SLOs, the software architect looks at these operations and finds the most critical operations, making up critical use cases. Based on rough SLOs for the four CloudStore operations in Sect. 2.3.2, as well as some reasoning, we identified the Pay operation to be most critical.

Afterward, the architect has to consider the planning horizon and based on seasonal and trend variations estimate the point in time when the work and load on the most critical operations are likely to pose the biggest threat to the scalability of the system. As this identifies a scenario for the critical use case which stresses its SLOs the most, it is called a *key scenario*. In Sect. 2.3.3, we find that in scenarios for the Pay operation, the highest load is expected to happen at noon on a Monday just before Christmas in the third year. Therefore, we assume that this will also be the key scenario used in analyses.

Once the scalability risks are known, we have to focus on elasticity risks next. Elasticity allows the system to scale according to its actual workload. Typically, risks arise in this area from insufficient adaptations or adaptation speed of the resources available to the system. Hence, we need to identify critical use cases in which rapid provisioning and deprovisioning of resources are required. For example, when a bunch of customers decide to buy books at almost the same point in time, this might need swift provisioning of additional resources and can therefore serve as a critical use case. To identify the risk, we need to figure out how long it will take for a certain rapid increase in load to increase resources as fast as possible, given their provisioning time. As the latter depends upon the capabilities of lower system layers like the IaaS layer, software architects need to identify risks that provisioning of resources in these lower layers might be slow or might happen delayed. The assessment of elasticity risks also depends on how critical it is that customers might leave our web shop, in case they do not get an answer in time due to an ongoing adaptation of the available resources. The more critical this is, the higher the architectural risk of such a scenario becomes. When specifying a key scenario for this critical use case, we need to specify how fast the load increases from the normal level to the level where many customers arrive at the bookshop in a burst.

Finally, we need to identify cost-efficiency risks of the system. These are risks where high losses of money are caused by massive over-provisioning of resources. For example, if there is a critical use case with a dramatic and rapid drop in the number of customers in a web shop but the elasticity management takes a long time

to notice this and the cost for provisioned resources is high, then this is a financial risk. Even worse, adding resources in a situation where the system faces increasing load but has already reached its capacity will cause significant financial losses, too. This setting emphasizes once more the importance of the system's capacity limits. In a key scenario of such a critical use case, you need specifications of the drop in the number of users but also information on the reaction time of the elasticity management of your system (including provisioning and deprovisioning times, the delay needed to detect the new situation, etc.).

As a result of the three identification steps, the software architect has a complete list of SLOs, associated critical use cases, and their scenarios. This is used as input in the next step to identify requirements as outlined in the following subsection.

5.7 Identify Scalability, Elasticity, and Cost-Efficiency Requirements

In Sect. 5.6 we identified business-related risky SLOs with respect to scalability, elasticity, and cost-efficiency; derived critical use cases from them; and selected key scenarios for those critical use cases. In this section, we revisit these business-related SLOs, critical use cases, and key scenarios from the previous step and enrich them with detailed workload information. As a result, we get *technical requirements* for scalability, elasticity, and cost-efficiency. In addition, we also sort these requirements according to their priority. This section, therefore, describes the second step on top of Fig. 5.2.

A requirement can be formulated on several levels of detail. For example, a generic, business-driven requirement may be that the system should be able to respond within 1 s. Here, the metric, the operations, the maximum load, as well as specifications of the work parameters are missing. However, one quality threshold is already specified. Such a requirement will therefore be open to interpretation. Nevertheless, it might make sense from a business perspective to specify them in the first place. To continue with the CloudScale method, such business-related requirements need to be revisited and detailed to reach a precise technical requirement. In this process we may differentiate between the quality thresholds among the operations. In addition, we may add information about the precise quality metric used, the operations, work parameters, and load. In practice, this will be an iterative progress between the first step and this second step in the CloudScale method.

Together with SLOs and the expected maximum workload under a specific scenario, these more detailed technical requirements guide the selection of architecture as well as implementation in the case of a new system. For an existing system, the technical requirements will guide our analysis efforts. Armed by these technical requirements, it will also be possible to test a system: is it able to manage the expected workload while adhering to the SLOs?

To identify scalability requirements, software architects need to determine the maximum workload the system should handle as constrained by the SLOs, critical

use cases, and key scenarios from the previous step. Workload covers both aspects, work and load.

There is a fundamental difference between the user point of view and the actual implementation. The requirements of a system should be formulated without considering the implementation of the system. This distinction between requirements and their realization might not be so easy in practice, because the operations constrained by SLOs can be already seen as part of the implementation. In practice, this is handled by iterations. In addition, to fully characterize the scalability, elasticity, and cost-efficiency of a service, we must also specify its configuration parameters as well as its deployments. These details are often fixed late in the development process. Therefore, feedback from these cycles might have an impact on requirements fulfillment and might require iterating them.

For dynamic properties like elasticity, it is furthermore important to know the change of the load and work over time. This is called *usage evolution* in the CloudScale method. This information needs to be captured quantitatively. In particular, for each key scenario (Sect. 5.6), the architect should capture information about the work and load evolution quantitatively.

Similarly, we also need to identify more details for work and load evolution also for the key cost-efficiency scenarios to transform them into technical requirements. Here, we are interested, in particular, in situations where the load and work decrease so that resources can be released.

After specifying the technical requirements, finally, the software architect should sort them with respect to risks for scalability, elasticity, and cost-efficiency according to priority, e.g., based on the severity of their business impact. In this way, the software architect starts with the most critical ones and first analyzes them. Note that this allocation of priorities should be done by the architecture board (i.e., architects, managers, the shop operator, and other relevant stakeholders).

When prioritizing, we also have some trade-offs concerning the granularity of the information:

Operations Some operations are much more important than other operations because of their product of work and load. Even a rare operation may be critical with a high work, and an operation with a low work may also be significant with a high load.

SLOs (service-level objectives) describing both a quality metric and a threshold for this metric. To fully specify a service, we need an SLO for all operations, but a first simplification may be to say that all operations share the same quality metric and also that they share the same quality threshold.

Load We can specify the load for all the operations individually. Another possibility is to specify the probability between the operations, together with the load on the average operation. In this case, we also need the operation mix describing the probability of each of these operations. Note that for a scalability analysis, load is often an output of the analysis.

Work The granularity of the work specification can be adjusted by putting the focus on a few work parameters only in contrast to modeling all work parameters.

For each of the identified and prioritized requirements, we run through an analysis as described in the next steps. Once the most critical use cases have been analyzed, the next critical use case might be selected from the remaining list. It then will also get analyzed. This repeats as long as critical use cases are remaining, and further analyses should be performed.

Depending on the scenario, the type of information available, in particular the availability of executable source code and usable resources, the architect makes a decision for each identified requirement: Should it be analyzed using a model of the system or should it be analyzed by inspecting the system code and executing it? Situations in which the architect wants to model a system and the steps needed in this case are detailed in Sect. 5.8. In Sect. 5.9 we look at situations in which the system's implementation is used as basis for analysis, i.e., either by source code analysis or by executing and tracing the system.

5.8 Use-Case I: Analyzing a Modeled System

One alternative to check the identified critical use cases and key scenarios for fulfillment of the risky requirements is to use a *model* of the system under study (see left branch in Fig. 5.2). There are different situations in which software architects like to gain the benefits of analyzing a system based on a model. First, in a greenfield development situation, i.e., a situation in which no preexisting implementation of the system under development exists, a model allows to analyze the critical use cases. Such a model will most likely be on an abstract level, which is often good enough for assessing the identified risky requirements. Second, a model is beneficial in cases where an implementation exists; however, the critical use case contains key scenarios, which are difficult and costly to execute, as they may require a lot of effort and resources. For example, executing a system in a high-load situation is a challenge for load generators due to the needed hardware or operating system resources. In addition, executing scenarios, in particular, elasticity or cost-efficiency scenarios, might take a long time. Third, software architects may want to use a model for quick analyses of a variety of what-if scenarios, often in order to optimize their system's behavior. Due to the large amount of slightly varied scenarios, the time it takes to analyze a running system is even multiplied. And finally, a model is also useful in cases of brownfield developments, where software architects face a combination of existing, legacy system components and non-existing, to-be-developed components.

Despite the benefits models provide in the situations outlined in the previous paragraph, there are also drawbacks of models, which might prevent their use. First, creating and parameterizing a model is a non-trivial task and requires skill and effort. The CloudScale method tries to lower these drawbacks by providing the CloudScale integrated development environment (IDE) and the CloudScale Extractor tool (details on ScaleDL extracting in Sect. 4.5.3.2), which aid in the model creation process for brownfield developments.

In case the software architect wants to use a system model, he has to execute two coarse-grained steps: First, he needs to specify the system using the ScaleDL modeling language with the aid of CloudScale's modeling tools (for details on ScaleDL modeling, refer to Chap. 4). As soon as the model is done, the software architect can use the ScaleDL model as input to the Analyzer. The Analyzer simulates whether scalability, elasticity, and cost-efficiency requirements are sufficiently achieved, thus allowing the software architect to iteratively improve the architectural model until satisfied (details in Sect. 6.3). Once satisfied, service developers and system engineers can realize, deploy, and operate the planned system with a lowered risk of violating the identified risky requirements. If system engineers—during testing or operation—still detect requirement violations, they can reiterate through the CloudScale method. Because the system has now been realized, system engineers can also decide to analyze the newly implemented system in the next iteration.

5.9 Use-Case II: Analyzing and Migrating an Implemented System

In contrast to using a model (cf. Sect. 5.8), software architects might want to analyze existing system implementations and check for the fulfillment of the identified risky requirements (see right branch in Fig. 5.2). This might be useful in the following situations. First, the system has been implemented before and faces a change in its environment; in particular, it is exposed to a critical use case which includes a usage evolution for which the system might not be prepared. In this case, the architect wants to know whether the system will still comply with its requirements under the changed load and/or work situation. Second, the system might face a planned migration. One example, which is particularly important for CloudScale, is the migration of an existing, non-cloud legacy application to a cloud computing environment. However, when migrating such a system to a cloud computing environment, the system does not automatically guarantee scalability. Scalability might be limited by a system's capacity due to existing software bottlenecks, like the ones imposed by the One-Lane Bridge HowNotTo (cf. Sect. 2.10). Therefore, when system engineers want to move an existing system to a cloud computing environment, they have to analyze whether their system fulfills scalability requirements or suffers from scalability issues.

The benefit of using a real implementation of the system is that the analyses are representative of the studied scenario, as they do not include abstractions as models do. Therefore, the gained results are often considered more trustworthy by decision-makers. In addition, analyzing the implementation of a system often does not require as much manual effort as creating a model, in particular, in cases where the system is already installed and configured in a representative testing lab.

The drawback of analyzing the implementation of a system is that it often takes much longer until analysis results become available, as executing the system under study and collecting sufficient measurements might be time consuming and resource consuming. In addition, system engineers need to provision a representative infrastructure environment and install the application in it. Furthermore, they need to provide realistic load scripts which implement the identified key scenario under study, e.g., using Apache's JMeter.

With the CloudScale method, system engineers are able to use CloudScale's scalability detection tool called Spotter. Spotter has two subcomponents called Static Spotter and Dynamic Spotter. The Static Spotter can detect potential scalability issues at the code level without executing the code. Instead, it identifies potential scalability issues. These potential scalability issues point the Dynamic Spotter to the code it should instrument. Based on a workload generator, it test drives the application in different work and load situations and identifies whenever the application does not scale in the performed tests. Based on a detailed analysis on which of the tests failed and for what reason they failed, the Dynamic Spotter reports a set of diagnostic statements with found scalability anti-patterns (CloudScale's HowNotTos). After software engineers eliminate detected scalability issues, they can continue to deploy the reengineered system and test for other identified critical use cases, as long as there are some left.

5.10 Realize, Deploy, and Operate

Once the selected analysis (based on Spotters or the Analyzer) indicates that scalability, elasticity, and cost-efficiency requirements are sufficiently met, the realization of the system can be completed (see lower-right corner of Fig. 5.2). This realization depends on whether a new system or an existing system is to be realized.

For new systems, system engineers realize the system based on the architectural representation in the ScaleDL model. For example, for each modeled software component, according source code has to be implemented, reused from existing components, or be externally bought. The Static Spotter may be used for static spotting of anti-patterns in the code.

For existing systems, system engineers semi-automatically reengineer detected issues based on either the Spotter or Analyzer suggestions. The Spotter points to code that requires reengineering and suggests HowTos to solve detected issues. The Analyzer suggests a revised ScaleDL model that software architects have approved to satisfy SLOs. System engineers have to then reengineer the system according to this revised model.

Once realized, the system has to be deployed and operated. The ScaleDL model particularly prescribes how a system's components are assembled and deployed to the target environment. System engineers follow this prescription and set the system in operation.

The Dynamic Spotter may be used on the system in operation to further spot anti-patterns. If new requirement violations arise, a new iteration of the method needs to be executed, again supported by the dedicated detection tools. One result of the analysis may also be a relaxation of some of the requirements and the subsequent identification of refined critical use cases and key scenarios.

If the system meets its requirements, system engineers can stop the CloudScale method and only have to reenter the method in case requirements or the system's context change.

5.11 Conclusion

This chapter introduced the CloudScale method. This method can be used to engineer cloud services, which have critical requirements with respect to scalability, elasticity, and cost-efficiency. The method supports two main usages: analyze non-existing systems on a model basis and analyze systems which have been implemented before and can be executed. The method is introduced step per step starting with requirements elicitation and ranging until system deployment and operation.

The CloudScale method provides benefits to software architects, service deployers, and service customers. It allows to check for requirements fulfillment as early as possible, thus avoiding costly rework activities. In addition, it is a useful method when migrating existing systems to the cloud. In contrast to other method, like software performance engineering (SPE), it focuses specifically on cloud computing applications. In contrast to SPE by Smith [6], for these systems, scalability, elasticity, and cost-efficiency are more important than performance. However, a good performance is the prerequisite to implement scalability, elasticity, and cost-efficiency.

In the following chapters, we outline the two use cases of the CloudScale method in more detail, give hints on how to manage introducing and executing the method, and illustrate it in case studies. In the future, the method can be extended by further steps, or it can be enriched by introducing new tools into the method's steps. For example, tools which reduce the manual effort to create and enrich ScaleDL models would have a significant impact on the method's applicability.

References

1. Brataas, G., Stav, E., Lehrig, S., Becker, S., Kopcak, G., Huljenić, D.: CloudScale: scalability management for cloud systems. In: Proceedings of International Conference on Performance Engineering (ICPE). ACM, New York (2013)
2. Brataas, G., Becker, S., Lehrig, S., Huljenić, D., Kopcak, G., Stupar, I.: The CloudScale method: a white paper (2016). http://www.cloudscale-project.eu/publications/whitepapers

3. Koziolek, H., Schlich, B., Bilich, C., Weiss, R., Becker, S., Krogmann, K., Trifu, M., Mirandola, R., Koziolek, A.: An industrial case study on quality impact prediction for evolving service-oriented software. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) Proceeding of the 33rd International Conference on Software Engineering (ICSE 2011), Software Engineering in Practice Track. Acceptance Rate: 18% (18/100), Waikiki, Honolulu, HI, pp. 776–785 (2011). [Online] http://doi.acm.org/10.1145/1985793.1985902
4. Lilja, D.: Measuring Computer Performance. Cambridge University Press, Cambridge (2000)
5. NIST Cloud Computing Standards Roadmap. National Institute of Standards and Technology (NIST), Technical Report 500-291 (2013). http://www.nist.gov/itl/cloud/upload/NIST_SP-500-291_Version-2_2013_June18_FINAL.pdf [Visited on 06/18/2016]
6. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, Boston, MA (2002)

Chapter 6
Analyzing a Modeled System

Sebastian Lehrig, Gunnar Brataas, Mariano Cecowski, and Vito Čuček

Abstract Architectural analysis describes the activity of discovering important system properties—like functional requirements and service-level objectives (SLOs)—using models of the system. The main benefit of such analysis is that software architects can assess their design and what-if scenarios without having to implement each option. For example, architects can easily elaborate alternatives and variants of HowTos. This chapter describes how software architects follow the CloudScale method to model and analyze systems via ScaleDL. The chapter links the CloudScale Method outlined in Chap. 5 with the ScaleDL language described in Chap. 4. For ScaleDL modeling, the most important manual and automated steps are described. For ScaleDL analysis, the usage of the Analyzer tool is detailed. A running example brings the different pieces together and shows how scalability, elasticity, and cost-efficiency can be projected based on a ScaleDL model.

The chapter starts in Sect. 6.1 with detailing the modeling steps in the CloudScale method. We refine the two steps specific for the model-based analysis of a modeled system: the specification of ScaleDL models (Sect. 6.2) and using CloudScale's Analyzer tool for its analysis (Sect. 6.3). After a detailed description of these steps, we illustrated the whole process based on our running example (Sect. 6.4).

S. Lehrig (✉)
IBM Research, Technology Campus, Damastown Industrial Estate, Dublin 15, Ireland
e-mail: sebastian.lehrig@ibm.com

G. Brataas
SINTEF Digital, Strindvegen 4, 7034 Trondheim, Norway
e-mail: gunnar.brataas@sintef.no

M. Cecowski • V. Čuček
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: mariano.cecowski@xlab.si; vito.cucek@xlab.si

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_6

6.1 Introduction

The analysis of modeled systems paves for software architects the way from functional requirements and service-level objectives (SLOs) to the realization and evolution of a system. Intermediate stages in this way include engineering the general system architecture from requirements and SLOs, refining the system architecture with concrete software components, mapping components to source code, deploying components to target environments, operating the system, and planning and executing system evolutions. For these stages, architectural analyses allow software architects to continuously ensure the required quality of the system under development and to make decisions with predictable, requirement-fostering outcomes throughout the whole process. Because all these goals are non-trivial, software architects need both education and guidance in engineering with architectural analyses, e.g., in the form of training courses, documented guidelines, and tool support.

In the CloudScale Method, the analysis of modeled systems is tailored to cloud computing environments and explicitly addresses scalability, elasticity, and cost-efficiency SLOs. As such, the CloudScale method is more concrete than general architectural analysis methods. This concreteness fortunately allowed us to give software architects more precise guidelines and tools that come with a higher degree of automation (and therefore involve less effort for software architects in the cloud computing domain).

It is the goal of this chapter to guide the reader step by step through this Scalability Description Language (ScaleDL)-driven analysis process, thereby, refining the overview of the CloudScale method from Chap. 5, as illustrated in Fig. 6.1.

6.2 Step I: Specify ScaleDL Model

The specification of ScaleDL models consists of several steps that software architects need to conduct in close cooperation with service developers. This section describes these steps as illustrated in Fig. 6.2.

Figure 6.2 shows that software architects first have to decide whether to refine an existing ScaleDL model (left path through Fig. 6.2) or whether to start modeling from scratch (right path through Fig. 6.2). Quite obviously, software architects choose to refine an existing ScaleDL model only if such a model already exists.

When starting from scratch, software architects proceed as we detail in the following. Based on the pre-specified critical use cases and key scenarios, they can determine the granularity with which ScaleDL models need to be modeled (Sect. 6.2.1). As a result, software architects can specify ScaleDL Usage Evolution models (Sect. 6.2.2) that cover usage evolutions at the right level of abstraction. These usage evolutions particularly allow software architects to derive the needed system interaction in combination with the given critical use cases and key

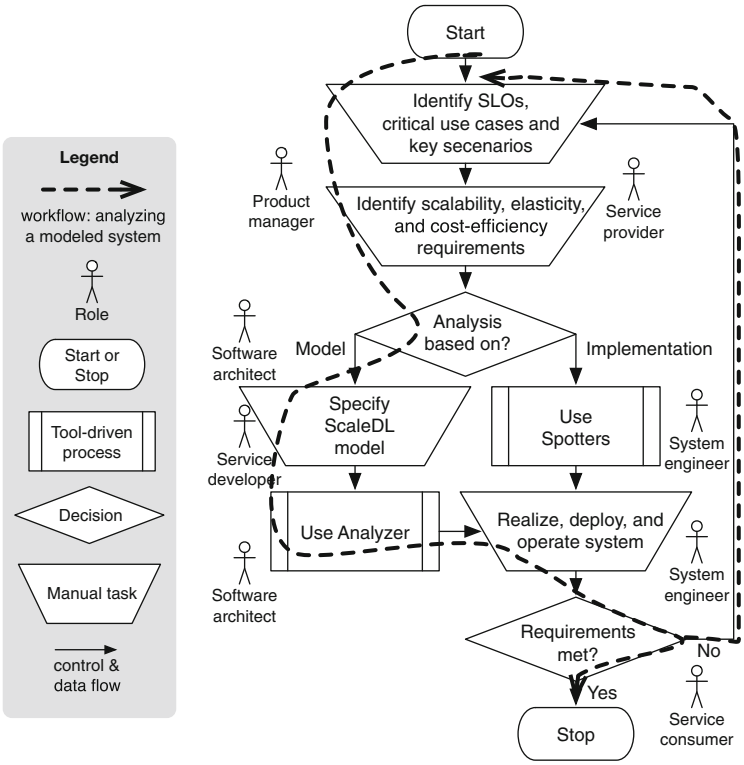

Fig. 6.1 Analyzing a system according to the CloudScale method through ScaleDL

scenarios. Derived system interactions enable architects to specify a ScaleDL Overview Model from which they can eventually generate a ScaleDL Extended Palladio Component Model (Extended PCM) (Sect. 6.2.3). The generated model is finally completed by software architects and service developers. This completion involves adding information that could not be automatically derived from the ScaleDL Overview Model (Sect. 6.2.4). This completion can be guided along HowTos as formalized via ScaleDL Architectural Templates (ATs) and be enriched with information from existing source code. In particular, software architects have to calibrate the Extended PCM model such that modeled resource demands accurately reflect the targeted environment.

When refining existing ScaleDL models, software architects generally proceed analogously. The only difference is that they take the existing models as the basis for the refinement (note: which allows to specify more fine-granular ScaleDL models compared to the initial specification).

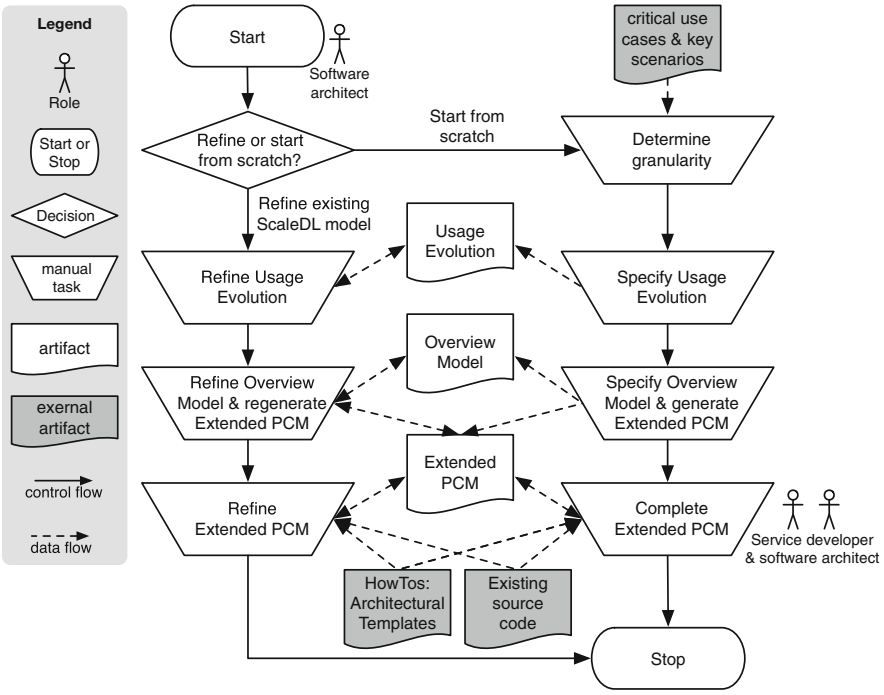

Fig. 6.2 Specify ScaleDL model process steps

6.2.1 Determine Granularity

As described in Sect. 5.2, there is a granularity trade-off between the amount of detail in ScaleDL models and the accuracy of their predictions. We advice to start with a coarse model and then introduce more details as needed. Generally speaking, a more comprehensive PCM model will require more manual work. On the other hand, it is challenging to make good abstractions. An elaborated model will more accurately resemble the actual service. This is also the case when introducing more sophisticated statistical resource modeling. However, a model with no statistical modeling of resources (i.e., where only one value is used for resource demands and few potential values are distinguished by a probability) will have lower confidence intervals; so in this case, accuracy will be improved with more detailed resource modeling, whereas precision will be reduced.

We will focus on the granularity of PCM models. ScaleDL Overview Model basically consists of a high-level PCM diagram; therefore, granularity trade-offs will be similar. Granularity trade-offs when making ScaleDL ATs are also the same as when making PCM models. For a PCM model, granularity is determined by (1) the service itself and its internal components, (2) its usage, and (3) its resources as

well allocation of internal components to resources. We will describe the granularity of each of these levels in turn:

Service granularity For the service itself, granularity is first connected to the number of components. On the one extreme, each method is a component, and on the other extreme, the complete service only consists of one component. In between, you can have one component for each class, or you may put classes together. Second, the number of operations in each component can vary. A detailed characterization of a component has many operations, whereas a coarse characterization has few or only one operation. Third, the number of connections between components is relevant. A coarse characterization ignores less important connections, whereas a detailed characterization includes all possible connections.

Usage granularity Granularity of the usage of a service also consists of several aspects. First, the number of user operations represent the way the users invoke a service. If the user mainly uses a few operations, a coarse characterization may ignore rare operations. However, uncommon operations may be important because they may have large resource demands. Second, the number of work parameters may vary from one to many. Third, the probability of the operations may span from a full Markov matrix to only the probabilities of each operation (a vector). Fourth, there may be one or more user scenarios like the browsing mix or the shopping mix for CloudStore. Fifth, quality thresholds (SLOs) may also be specified more or less accurately.

Resource granularity The lower-level services also have granularity trade-offs. First, some of the lower-level services (tiers) may be ignored. In CloudStore, we may, for example, neglect the payment and the image servers. Second, when it comes to resources, we may focus on processing using CPUs and not consider network, disk, or (primary) memory. Third, we may model more or less of the complexity of each resource. For example, a CPU has several cores, which may or may not be modeled explicitly. Fourth, for the operations of the resources, we may use an average CPU operation. When it comes to storage and network, we may also simplify the number of operations in the resources. Fifth, the work modeling of the operations may be more or less complex. We may model work parameters like message lengths to the amount of disk storage, but we can also ignore this. Sixth, we may use statistical distribution of resource demands or just a constant. The sophistication of the statistical resource demand distribution may also vary, from a coarse model with only two different values where one probability is enough to describe the distribution between them, to many different resource demand values where several probability values are required to characterize their distribution. Furthermore, the allocation of software to resources (or lower-level services) may be more or less sophisticated. There may be many links between components and resources, or fewer links.

In addition, granularity of the usage evolution is determined by several factors. First, the number of operations with independent load evolution. Second, the number of work parameters with independent work evolution. Third, you may also change the usage scenario, but altering the operation mix (probabilities of the operations) as well as the quality thresholds.

Finally, the Analyzer configuration variables determine simulation time as well as precision and accuracy. The run length determines the accuracy of the results. A longer run length will typically give higher confidence intervals, but then also a better match to the "real" service. We may vary the number of times to run the model (with different seeds). More runs may result in a higher confidence interval and, therefore, seemingly worse confidence intervals, but then with improved accuracy.

The following sections detail how software architects create ScaleDL models while considering these trade-offs. The modeling hints from Sect. 3.8 provide further insights as to how software architects can make these trade-offs.

6.2.2 Specify Usage Evolution

In Sect. 4.3 we have described ScaleDL Usage Evolution together with a simple example both for load and for work evolution during a 3 min interval. For the iterative CloudScale method described in Sect. 5.5, we described how requirements are also refined when we analyze scalability, elasticity, or cost-efficiency. We will now describe using the CloudStore example in Sect. 2.2 how we can model its usage evolution.

As described earlier in Sect. 1.3, SLOs, load, and work are key service properties from the point of view of scalability, elasticity, and cost-efficiency. Expected values for these key properties will therefore represent basic requirements, in addition to more detailed requirements for scalability, elasticity, and cost-efficiency. Requirements for these key properties may be elicited with varying level of detail, as explained in the CloudScale method in Sects. 5.6 and 5.7. The simplest solution is to find one value representing each of these requirements. This was done in Sect. 2.4.1. We will now go one step further and coarsely describe how these key characteristics will evolve during a 3-year period.

6.2.2.1 Service-Level Objectives

For our customers, the 90 percentile represents a sound quality metric. For each of the four operations in Sect. 2.2, we must decide the quality threshold which marks the distinction between acceptable SLO and SLO violation. Since the four CloudStore operations differ considerably both in frequency and in complexity, we will distinguish between their quality thresholds. We consider the quality thresholds as indicated in the third column in Sect. 2.3.1 to represent sound quality thresholds.

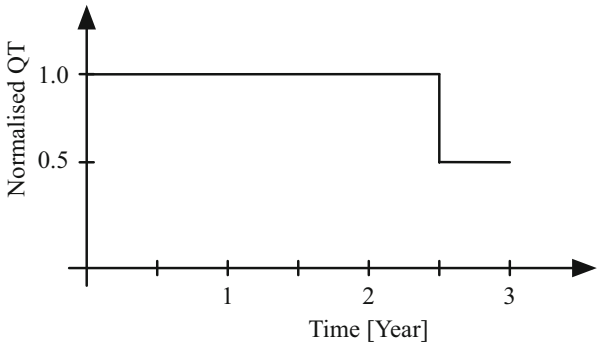

Fig. 6.3 Quality threshold evolution for 3 years

However, in the last half-year of the 3-year period, we expect the competition to become so tough that we will halve all of these quality thresholds. This is represented in Fig. 6.3, where we have one value representing the quality threshold evolution for all operations, with their original values as the start values. This, for example, means that the quality threshold for the Pay operation will become 2.5 s instead of the start value of 5.0 s, in the last 6 months.

6.2.2.2 Load

Since CloudStore has four different operations, each of these operations may have different load evolutions, but a natural simplification is to have one load parameter, representing the evolution of the average operation, instead of several operations evolving independently.

As briefly outlined in Sect. 2.3.3, the load of a typical system will often have several seasonal variations. We will have a general variation for our 3-year period, but in addition, we may also have yearly variations, for example, an increased number of customers before Christmas and a reduced number of customers after Easter. There may be variations during a month as well as during a week and during each day. When it comes to days, then Saturday and Sunday may differ from the other days of the week. To keep our example simple, except for the 3-year trend, we only have a daily variation.

Scalability is the ability to handle a certain workload, and for this metric, we are therefore interested in the maximum for each day. Daily evolution is not relevant to scalability. Figure 6.4 shows how the maximum number of daily users vary during a 3-year period for CloudStore. Originally, there are 2000 maximum simultaneous users. In the first half-year, this figure illustrates a linearly increasing trend, followed by a stable period with 5000 simultaneous users of 1 year, and then a new increase up to a new stable period in the last year. In this last stable period, there are 10,000 simultaneous users.

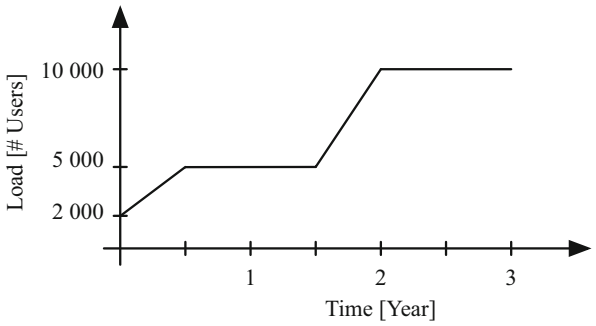

Fig. 6.4 Load evolution for 3 years

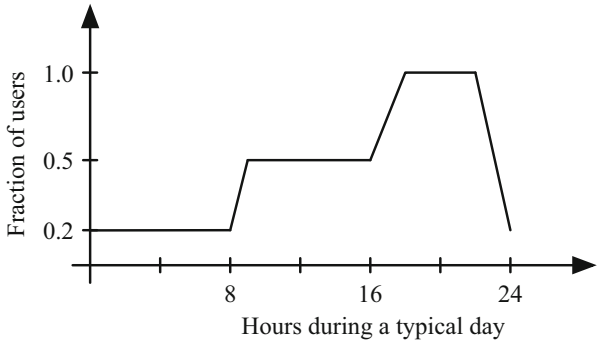

Fig. 6.5 Daily load evolution

For both elasticity and cost-efficiency, we are also interested in the daily evolution. While Fig. 6.4 represents the maximum number of daily users during the 3-year period, Fig. 6.5 describes in more detail the daily variation in users. For simplicity, we state that each day has the same pattern. From Fig. 6.5 we observe that just in the 4 h interval between 18 and 22, the maximum number of daily users are present. During the night, between 0 and 8, only 20% of the maximum users are present, and during the normal working hours, from 9 to 16, half of the maximum number of users are using our online bookstore.

6.2.2.3 Work

CloudStore has two *work parameters*: the number of books and the number of customers. As a rule of thumb, we expect the number of customers to be 100 times the number of simultaneous users. Therefore, the number of customers will grow from 200,000 to 1,000,000 in the 3-year period, as derived from looking at Fig. 6.4.

Figure 6.6 depicts the evolution of the work parameter describing the number of books. For the first 9 months, the work parameter number of books is stable

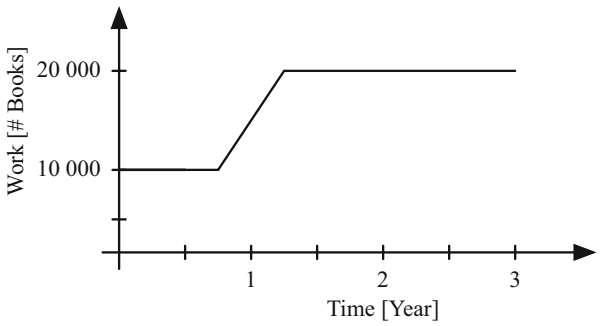

Fig. 6.6 Work evolution for 3 years

at 10,000 books. Then, during the next 6 months, we have a linear increase up to 20,000 books, which defines the stable load during the remaining period.

6.2.3 Specify Overview Model and Generate Extended Palladio Component Model

An Overview Model provides a high-level abstract view of the software architecture, services, deployment, and the infrastructure of cloud-based architectures and its deployment at a high and user-friendly level of abstraction, allowing for a formal definition that can be used to share best practices and potentially automate analysis and deployment actions.

The Overview Model can also encapsulate a software service implementation (represented as a partial PCM model). This separation of the architecture and the implementation into separate models provides a greater flexibility to reuse and analyze existing software components using various software architectures. If we want to analyze an already existing system, the easiest way is to extract logical parts of the existing code, produce a partial PCM model, and attach it to the software service containers in the Overview Model. This process can be automated and guided by the Overview import wizard, which can also create the initial cloud environment and the deployment with the desired properties.

Figure 6.7 shows a visual representation of an Overview Model of a simple system composed of two Tomcat applications running on Amazon EC instances, and which make use of the Amazon DynamoDB service, a MySQL RDS service, and an email service. The model includes networking details such as average latency and bandwidth, which can be used for the behavior analysis and simulation of the overall system. Several other data can be defined. For example, we can define the expected statistical distribution of response time for an external service, or the expected capacity of a computing unit.

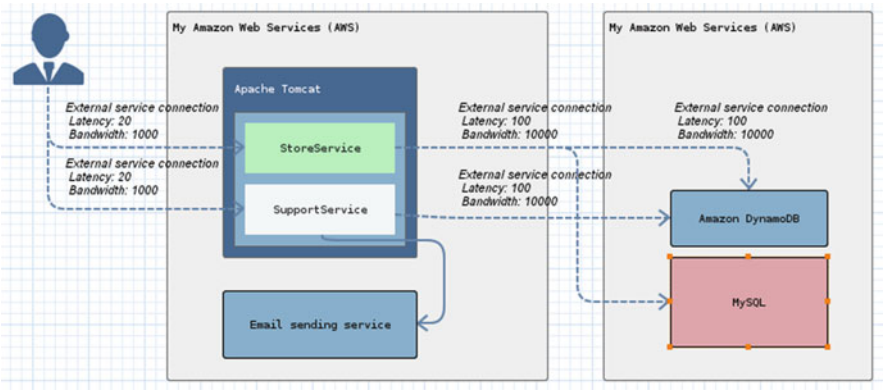

Fig. 6.7 Overview Model of a simple two-tier application

To create the Overview Model from scratch, the system architect can use the Overview diagram editor, which allows the user to drag components (e.g. services, connections, cloud environments. . .) into a canvas and define their main properties.

The Architecture is the root component of the Overview Model. It holds the descriptions of all external black-box services (e.g. external services) and Cloud Environments (e.g. infrastructure services with well-defined properties). Each Cloud Environment contains descriptions about availability zones, regions, internal network, and services. The latter are divided onto three separate layers, the infrastructure, the platform, and the software layer. This separation is based on the type of a functionality it provides. The description of the particular service in the Overview Model is modeled with the composition of interfaces and properties, abstracted by the Descriptor entities. Descriptors are reusable objects with static properties and can be referenced by services and network connections in the Overview Architecture. This offers simplified migration of the described software architectures between different cloud providers.

When the Overview Model is created together with partial PCM models, the system is fully described and ready to be transformed into Extended PCM models, which, besides the definition of the components, resources, and overall architecture of a system, allows for the definition of self-adaptation rules, monitoring specifications, and SLOs.

Elastic properties of the infrastructure and the ability to program its runtime behavior open a whole new horizon of possibilities. The infrastructure adaptation and deployment can be modeled using the ATs language. In addition to this, the role of ATs is to generalize the particular system architecture and provide basic parameters to produce reusable models. Those ATs models can then be used in the Overview-to-Extended PCM transformation process. Depending on the Overview's model deployment specification, the transformation process selects the appropriate ATs and applies them to the Resource Environment and System models. In other words, the Overview Model contains a description of the elastic properties, together

with the deployment, and the ATs provides the implementation for the Analyzer simulation engine.

6.2.4 Complete Extended Palladio Component Model

To unleash the full capabilities of ScaleDL-based analyses, software architects need to provide completely specified Extended PCM models. This section describes how software architects proceed with this completion—the last process from Fig. 6.2.

As a starting point, partial Extended PCM models can be automatically generated from existing artifacts. Existing artifacts are either ScaleDL Overview Models (as described in the previous section) or existing source code (as described in Sect. 4.5.3.2). These models are only partial as, e.g., the Overview Model has no detailed information about control and data flow, and the extracted model from source code misses information about the resource environment and allocation models.

Either way, software architects have to manually complete their Extended PCM models in cooperation with service developers (Sect. 6.2.4.1). HowTos help software architects in this task (Sect. 6.2.4.2).

6.2.4.1 Manually Creating and Adapting Extended Palladio Component Models

The creation and adaption of Extended PCM models can be classified based on the sub-models that need to be specified. Sub-models include the normal PCM models and models for self-adaptation rules as available in the Extended PCM (cf. Sect. 4.5.1). The specification of Usage Evolution models is part of a dedicated, earlier process step (cf. Sect. 6.2.2). This section provides a brief summary of these two categories; on the CloudScale website, we provide a detailed step-by-step workshop [1].

Specifying Palladio Component Models

The Extended PCM allows to specify the same models as the PCM: component specification, system model, resource environment model, and allocation model are part of both languages. Therefore, software architects can simply follow the processes already described for the PCM in the Palladio approach.

The development process according to the Palladio approach is based on the process for component-based development [2]. Koziolek and Happe have refined this process for quality-of-service (QoS) analyses [3]. The Palladio approach follows this process and provides appropriate editors to software architects and service developers [4, 5]. In summary, software architects proceed as follows.

First, software architects derive **system interfaces** based on the pre-specified Usage Evolution models. Each use case specified in these models is covered in a dedicated interface. The operations of each interface can be derived from the needed system interactions to realize the respective use case. The interfaces are attached to the system via PCM's system model.

Second, software architects derive and assemble **components** based on the core business of the service to be developed. For example, for a bookshop like CloudStore, the core business is books. Software architects accordingly need to specify a business component that allows to manage books. The provided interface of these components includes all business-relevant operations, e.g., to add new books and to alter existing books. Software architects also specify the instantiation and assembly of these components within PCM's system model.

Third, software architects request **implementations** for these business components from service developers. Service developers specify the SLO-relevant behavior of these components, e.g., as resource demands. The result is refined component specifications.

Fourth, software architects specify the **allocation** of the assembled components onto **processing resources**. For this specification, they specify PCM's resource environment and allocation models according to the deployment information of ScaleDL's Overview Model.

Specifying Self-adaptation Rules

Typical cloud computing applications are elastic, i.e., they are able to adapt required resources to actual workload demands. For this reason, an accurate ScaleDL model needs to model such adaptations via self-adaptation rules. Software architects specify self-adaptation rules via model-to-model transformations that consist of (1) a trigger and (2) an action that can be activated by the trigger (cf. Sect. 4.5.1.2).

Software architects first specify the trigger as a simple query inside the model-to-model transformation. This query checks whether monitored values exceed a threshold value. For example, software architects may formulate a query that checks whether 90% of response times of the last 10 s were below 2 s.

This query determines whether a transformation rule, which software architects specify in the second step, is triggered. The transformation rule translates a PCM model from a state (a) to a state (b). In the example in Fig. 6.8, state (a) describes a system where two replicas of a `Web & Application Server` are load balanced, whereas state (b) describes a system where three of these replicas are load balanced. A transformation takes care of this adaptation.

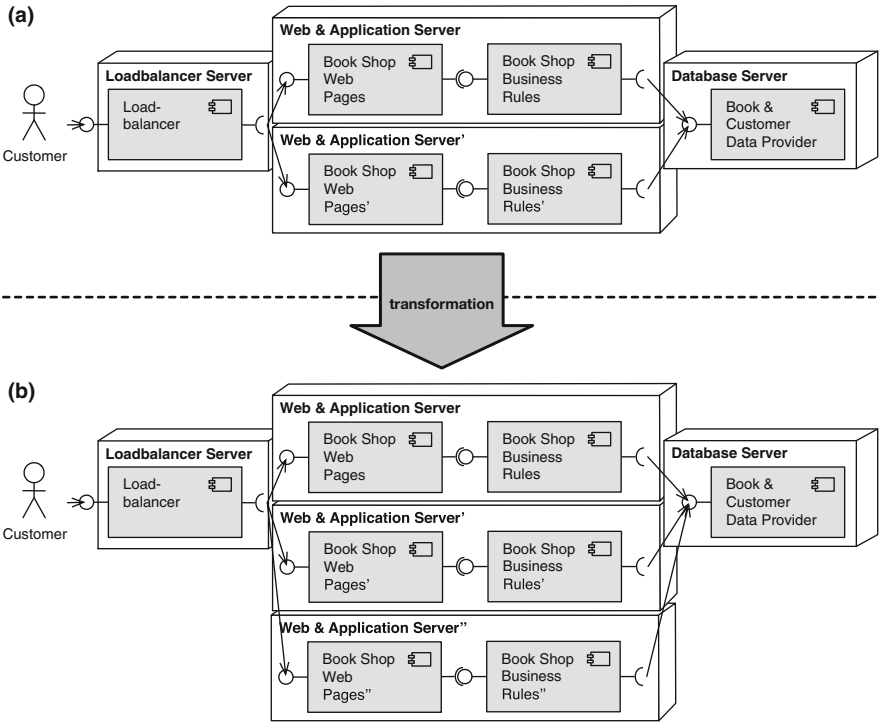

Fig. 6.8 A transformation can adapt a system from a state (**a**) to a state (**b**)

Calibrating Palladio Component Models

PCM models have to be calibrated once they are specified. A calibration enriches a PCM model with concrete resource demands of active resources, e.g., a CPU. Resource demands can either be estimated by experts or measured in the target environment (based on prototype or actual implementations). If the actual implementation is already available, the latter approach should be followed, because the implementation can be used to measure demands directly in the target environment; this is generally more accurate.

6.2.4.2 Using HowTos for Specifying Extended Palladio Component Models

Software architects create architectural models, like Extended PCM models, based on a given set of scalability, elasticity, and cost-efficiency requirements. Applications conforming to these models should optimally meet all pre-specified

requirements. Architects try to meet these requirements by making suitable design decisions during architectural modeling.

Fortunately, decision-making is not a purely creative task but can be governed by HowTos, i.e., best practices to design systems in recurring situations (see Sect. 2.9). In applying HowTos, software architects are less likely to create models that would lead to requirement violations. Software architects only have to find and select suitable HowTos for their particular situation and subsequently apply the HowTos correctly—we describe these steps in the following.

Finding HowTos

When software architects follow the CloudScale method, a natural source for HowTos is the CloudScale's HowTos catalog (see Sect. 2.9 for the catalog). HowTos from this catalog are especially useful if the relevant HowTos are formalized via ATs because it eases their application (see Sect. 4.4.3 for formalized ATs).

However, there are more sources to be considered other than CloudScale's HowTos catalog. A generally good source is books on software architecture, e.g., [6–12]. These books provide several HowTos in the form of architectural styles and patterns for general and distributed software.

For more specific domains, specialized software architecture books report appropriate HowTos. For example, for the domain of cloud computing, several of such books exist [13–16]. These books provide valuable information for variations and additional descriptions to CloudScale's HowTos catalog, which only includes the most common HowTos from these sources, e.g., patterns for horizontal and vertical scaling.

Besides such books, specialized HowTos are often directly provided by providers of services and frameworks. For example, Amazon web services (AWS) provides several HowTos for its cloud computing services in the "AWS Architecture Center" [17].

Selecting HowTos

From the set of found HowTos, software architects need to select the HowTos that they want to apply to their architectural model. Architects select by matching their QoS requirements to the promises of the HowTo. For example, for performance requirements, architects may select a HowTo that promises to improve response times, like the *Horizontal Scaling* HowTo from Sect. 2.9. If software architects cannot find such a suitable HowTo, software architects have to continue without following HowTos (thus following a less engineering and more creative approach).

Applying HowTos

For applying HowTos, software architects follow the best practices described within these HowTos. While software architects can follow these practices manually, the use of HowTos formalized as ATs automatically ensures that architects follow these practices consistently and correctly.

With ATs, software architects create appropriate bindings to AT roles and set actual parameters. For example, they bind the *load-balanced container* role of the *Horizontal Scaling* AT to the `Web & Application Server`, as shown in Fig. 4.9.

Modeling tools prevent architects thereby from violations of constraints captured in the bound AT, e.g., by disabling the creation of illegal connections and by pointing to missing elements or existing but invalid elements. In contrast, such a consistent and correct application of HowTo is not guaranteed when done manually.

6.2.5 Summary for the Specification of ScaleDL Models

With the completion of an Extended PCM model, software architects have finalized the specification process of ScaleDL models. As illustrated in Fig. 6.2, all constituents of a ScaleDL model are now available: Usage Evolution, Overview Model, Extended PCM (including potentially applied ATs). These constituents completely characterize all relevant factors of a system's scalability, elasticity, and cost-efficiency. Therefore, ScaleDL models are suitable for analyses of these properties. The next section describes such analyses using CloudScale's Analyzer.

6.3 Step II: Use Analyzer

As illustrated in Fig. 6.1, the second step for architectural analyses within the CloudScale method is the *Use Analyzer* step. Figure 6.9 details this step to show how to configure, run, and interpret results of the Analyzer—CloudScale's simulator for ScaleDL models.

During simulation, the Analyzer measures metrics for typical cloud computing properties, i.e., scalability, elasticity, and cost-efficiency. Based on these measurements, software architects can decide whether their service requirements are met. If these requirements are violated, software architects can iteratively alter their ScaleDL model and check whether this alteration improves the situation.

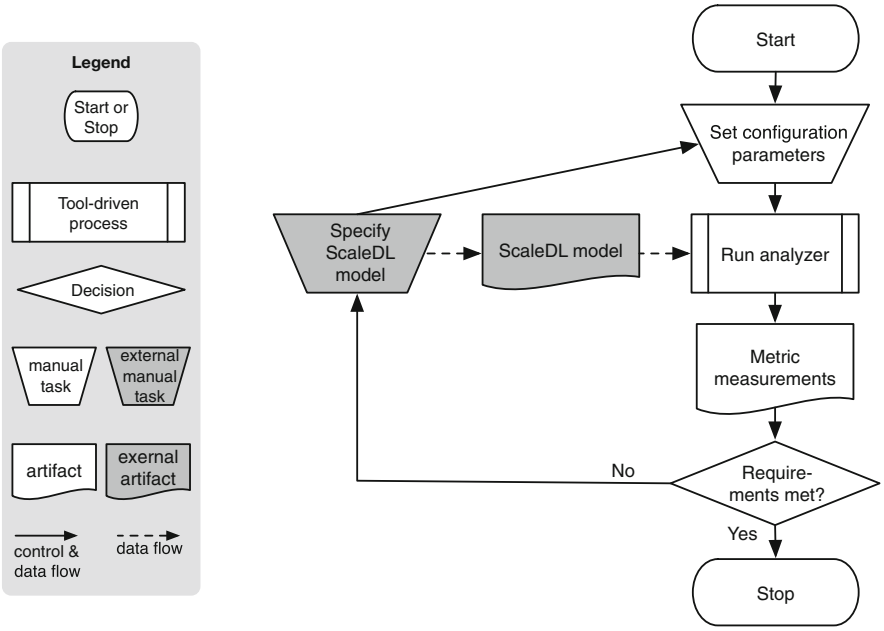

Fig. 6.9 Use Analyzer process steps

Typical elements to be altered are additions and modifications of HowTos (e.g., adding caches and modifying load balancer parameters) and of resources (e.g., adding CPUs and increasing their processing rates). Once requirements are met, software architects, service developers, and system engineers can continue to realize, deploy, and operate their service—with a reduced risk that their system violates requirements.

Figure 6.9 shows that, before starting an analysis, software architects must determine a set of configuration parameters for the Analyzer (Sect. 6.3.1). Once configured, software architects run the analysis (Sect. 6.3.2), which results in a set of metric measurements. Based on whether these measurements indicate that service requirements are met, software architects decide whether (a) to alter the specification of their ScaleDL model and to reiterate the process or (b) to finalize the *Use Analyzer* process. In case (a), software architects return to the previous step described in Sect. 6.2. Moreover, in a later step of the CloudScale method—i.e., the realize, deploy, and operate system step (Sect. 5.10)—software architects can compare analysis results against actual measurements in the target environment. Such comparisons allow software architects to check whether the system has been correctly realized.

6.3.1 Set Configuration Parameters

The first step in Fig. 6.9 is to set the Analyzer's configuration parameters. The Analyzer's configuration parameters determine which metrics are investigated, the confidence in measurement results, the total simulation time of the Analyzer, and how capacity is investigated. Software architects need to determine which metrics are of interest and how to trade off confidence with total simulation time. The suggested strategy is to start with small total simulation times. Subsequently, software architects can increase the confidence where needed.

Besides configuring a ScaleDL model as input, the Analyzer's main parameters are the following:

Monitor specification: Allows software architects to configure which metrics will be measured. A monitor specification includes a set of monitors. Each monitor points (1) to an architectural element for which measurements need to be taken and (2) to a set of concrete metrics that have to be measured for that element. In setting up these monitors, the Analyzer automatically measures the configured metrics.

The Analyzer can simulate typical performance metrics like response times, throughput, and utilization. Moreover, the Analyzer supports metrics for scalability, elasticity, and cost-efficiency [18].

For example, the *number of SLO violations* [19] elasticity metric counts the number of violated SLOs during adaptation. Another example is the *mean time to quality repair* [19] elasticity metric, which measures the time a system needs to move from a state that violates SLOs to a state that satisfies all SLOs. An example for a cost-efficiency metric is the *cost over time* [20] metric, which computes the operation costs incurred for using cloud computing resources per billing interval.

Simulation time stop condition: Limits the total execution time of an Analyzer run. A longer run will typically give a higher confidence interval.

Measurement count stop condition: Limits the maximum number of measurements of an Analyzer run. A higher number will typically give a higher confidence interval but increase the total execution time.

Capacity investigation interval: Sets an interval for the number of users for which the Analyzer investigates whether a system can fulfill SLOs. The Analyzer first checks whether the system fulfills SLOs at interval borders. If the lower border already violates SLOs, the Analyzer reports that the capacity is below this lower threshold. If the upper border fulfills SLOs, the Analyzer reports that the capacity is beyond this upper threshold. Otherwise, the lower border is fulfilled but the upper border is not. The Analyzer then conducts a binary search for determining the concrete capacity for the given interval.

Enable scalability analysis: Determines whether the Analyzer assumes an unlimited amount of available resources (as virtually available in cloud computing environments). If enabled, the Analyzer uses the previously described capacity analysis to determine whether the upper interval bound fulfills all SLOs, given an unlimited amount of resources. If all SLOs are fulfilled, the Analyzer reports that the system is scalable; otherwise, the Analyzer reports that the system includes a scalability bottleneck, along with metric measurements to investigate this bottleneck.

6.3.2 *Run Analyzer and Assess Requirements*

In the *Run Analyzer and Assess Requirements* step from Fig. 6.9, software architects start the Analyzer with a given ScaleDL model and the configuration parameters from the previous step. The analysis results subsequently allow software architects to assess whether QoS requirements were met.

For assessing QoS requirements, software architects have to investigate analysis results and compare them to the QoS requirements. The CloudScale Environment provides a dedicated results view for this purpose. Inside the results view, software architects can select results by choosing the metrics of interest.

For example, Fig. 6.10 illustrates analysis results for the response times of a fictional `DefaultUsageScenario`. The view on the left part of Fig. 6.10 allows

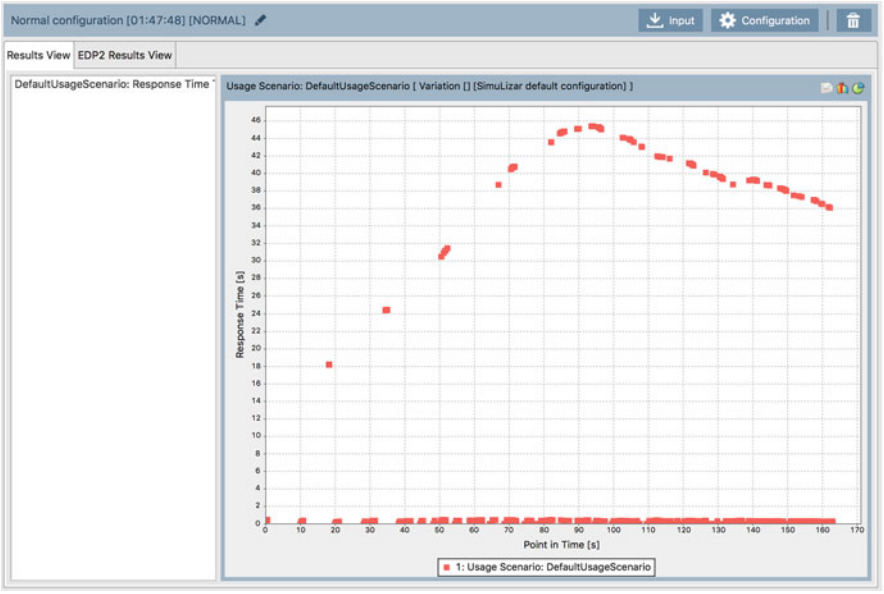

Fig. 6.10 Example analysis results for the response times metric

software architects to select such analysis results. The view on the right part of Fig. 6.10 subsequently visualizes the results, e.g., in an *XY* graph.

The *XY* graph in Fig. 6.10 shows response time measurements (*Y*-axis) over the simulated time (*X*-axis). For example, after 19 s of simulation, the Analyzer measured a response time of approximately 18 s. If a QoS requirement stated that response times are never allowed to exceed the 10 s mark, such a requirement would clearly be broken. In consequence, software architects would have to go back to the previous step of specifying the ScaleDL model to refine and alter the model such that QoS requirements can be finally met. Another Analyzer run can confirm whether this is the case.

For further details, the Palladio workshop [1] includes step-by-step instructions for software architects. These instructions cover the whole *Run Analyzer and Assess Requirements* step.

6.4 Analyzer Running Example

This section illustrates how we created a ScaleDL model for CloudStore (Sect. 6.4.1) and shows how we used this model to evaluate capacity, scalability, elasticity, and cost-efficiency metrics (Sect. 6.4.2). Particularly, we report the efforts we made during these actions in terms of creation and evaluation times. We provide a detailed description on how we created and evaluated the CloudStore model in [21] and as online screencasts [22].

6.4.1 Step I: Specifying a CloudStore Model via ScaleDL

In the first step for analyzing modeled systems (cf. Sect. 6.2), we specify a ScaleDL model for CloudStore. We required a total specification effort of approximately 214 h. This specification involves the specification of Usage Evolutions, an Overview Model, and an Extended PCM model (83 h effort), the calibration of the PCM model (121 h effort), and the evaluation of the model (10 h effort).

The modeling granularity is determined by the critical use cases from Sect. 5.6: the `Sell Single Book` and the `Premium Customer Sale` use cases. These use cases require customers to interact with a total of four different system operations. Each of these operations needs to be covered in one of CloudStore's system interfaces and is modeled in the following.

6.4.1.1 Constructing the Model

To specify an extended PCM model for CloudStore, we first used the Extractor to get an initial overview of CloudStore's structure and behavior. Afterward, we manually refined and completed the extracted CloudStore by inspecting the source code of CloudStore's implementation.

Figure 6.11 depicts the resulting model. It refines the previously described simplified CloudStore model (see Fig. 4.9) and uses the same syntax by illustrating: (1) the static system structure of the CloudStore system via connected component instances; (2) the allocation of these components to system resources (`Web & Application Server`, `Image Server`, `Database Server`, and externally hosted `External Services`); and (3) the system entry point for customers (interfaces for accessing web pages for books, home page, shopping carts, and

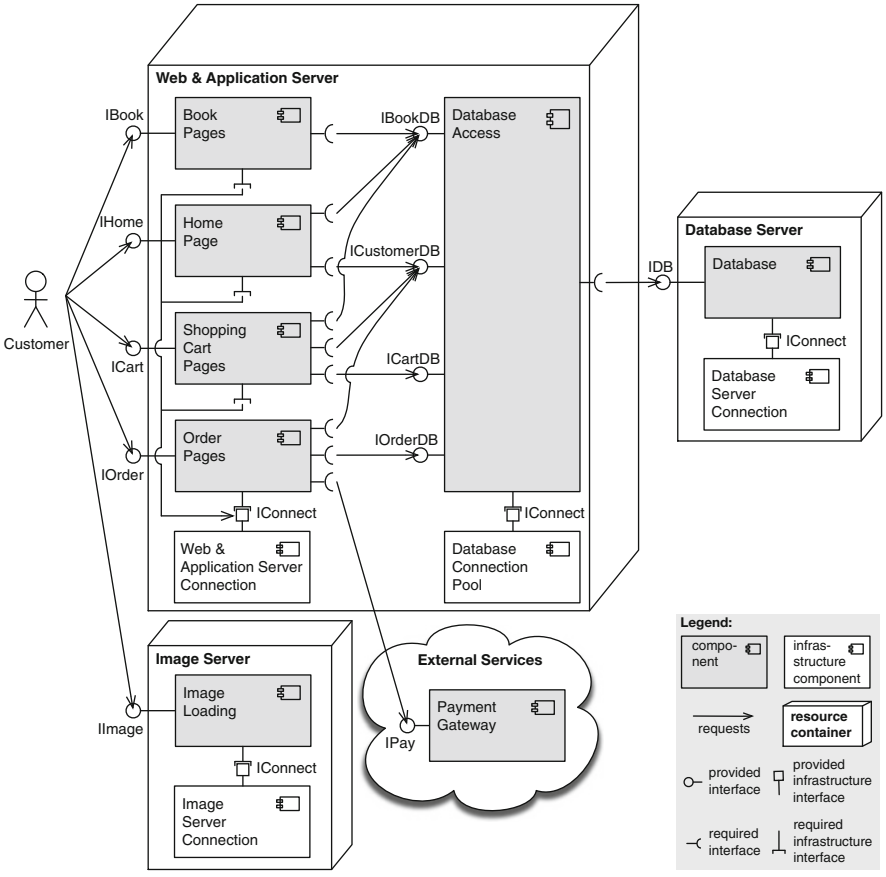

Fig. 6.11 Extended PCM model of the CloudStore online bookshop

orders). We specified each of these views in a dedicated extended PCM model with a behavior, as described next.

Customers enter the system via the web pages provided by the `Book Pages`, `Home Page`, `Shopping Cart Pages`, and `Order Pages` components allocated on the `Web & Application Server`. `Book Pages` provides operations regarding books (e.g., to query book details or search for books). The `Home Page` component shows CloudStore's home page, which welcomes its customers and displays possible book categories for browsing. `Shopping Cart Pages` allows customers to register, add books to a shopping cart, and check-out the shopping cart. Afterward, `Order Pages` allows to follow up on the order. `Order pages` can particularly request payment services from the externally hosted `Payment Gateway`.

The aforementioned components require operations of the `Database` component as allocated on the `Database Server`. CloudStore's database stores entries for books, customers, shopping carts, and orders. Moreover, if a returned web page references images (e.g., book covers), a customer's browser subsequently fetches these references via the `Image Loading` component allocated on the dedicated `Image Server`.

Requests to the `Database` are intercepted by a `Database Access` component that manages database connections. `Database Access` receives/returns such connections from/to the `Database Connection Pool` component. `Also Web & Application`, `Image`, and `Database Server` use pools for handling customer requests (`Web & Application Server Connection`, `Image Server Connection`, `Database Server Connection`). These pools (white-colored infrastructure components in Fig. 6.11) are typical performance factors as their pool size limits the amount of requests that can be processed in parallel.

The extended PCM supports acquiring and releasing connections from these resource pools within service effect specifications (SEFFs; cf. Sect. 4.5.1.1). In our model, every interaction requires the acquisition of connections and their release once the interaction ends.

Figure 6.12 illustrates this pattern for SEFFs of front-end component operations that interact with database and image components. Actions (1)–(3) model the performance impact of creating an HTML page for customers, while action (4) models the performance impact of subsequently resolving image references. These two phases—receiving an HTML page and subsequently its references—reflect the typical behavior of web browsers.

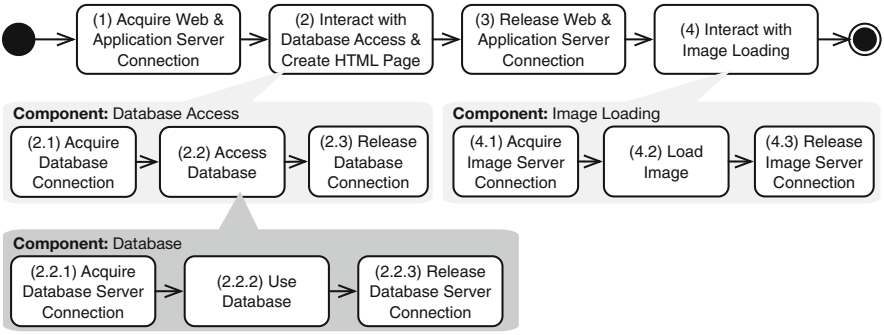

Fig. 6.12 Behavior of front-end components interacting with database and image components

We also used the concept of connection pools to realize transactions; modeled as connection pools with pool size 1. Our transactions include write operations for creating new customers, books, and orders. Such transactions typically have a major performance impact. The complete model is available at [23].

6.4.1.2 Calibrating the Model

For the calibration of the CloudStore model, we measure demands directly in the target environment because we have the actual implementation already available. We accordingly executed the following actions:

Set up on-premise environment. We used three equivalent computers for each of CloudStore's tiers. Each computer operates with Ubuntu 14.04, 2.67 GHz 2-core CPUs, and 2× 4.00 GiB main memory. We synchronized these values with our performance model for CloudStore.

Deploy CloudStore servlets into the environment. For the front-end, we deployed CloudStore's servlets on a TomCat 7.0.30 (`Web & Application Server`). For the back-end, we used an Apache HTTP 2.2.29 server (`Image Server`) and MySQL 5.5.27 (`Database Server`). We limited the size of pools to 250 server connections (for servers)/100 database connections (for the `Database Access` component). We synchronized these values with our performance model.

Generate workload and measure resource demands. For each operation of interest, we generated dedicated workloads to measure the exclusive resource demands for that operation (with a 10-min warm-up phase). We used Kieker 1.9 [24] as the monitoring framework. Kieker sent response time measurements to a home-brew performance engineering tool for visualizing and recording these

measurements in an appropriate format. For requests to `Image` and `Database Server`, we only measured their response times as externally observed by the `Web & Application Server`, i.e., without conducting more fine-grained measurements on these servers. Because our focus is on software applications, we wanted to center on the behavior of the `Web & Application Server` itself and avoided digging deeper in the file management of `Image Server` database details.

Put measured demands into the performance model. Our home-brew performance engineering tool allowed us to live-monitor and store received measurements. Based on this data, we calculated probabilistic distribution functions for response times and enriched our performance model with these functions. Because CloudStore is CPU bound, we enriched our model only with functions for CPU demands.

6.4.1.3 Evaluating the Model

We evaluated the performance model's accuracy with the "Browsing Mix" workload (cf. Sect. 3.7) and a load of 300 users. In contrast to the calibration workloads (which are separated per operation), such a workload is considerably more complex and, thus, suited for evaluation. Because the Palladio-Bench allowed us to generate JMeter workload scripts from our "Browsing Mix" usage scenario model, we used Apache JMeter 2.12 [25] as workload generator and for response time measurements. We had to iterate the model creation process two times as we discovered modeling mistakes during evaluation (e.g., we forgot to return acquired connections, resulting in drastically increasing response times). After these two iterations, we achieved a prediction error of 2.76%. As this value was below our 30% threshold, we considered our model creation process to be successful.

6.4.2 Step II: Using the Analyzer with the CloudStore Model

This section details our analysis results for the CloudScale model. This analysis is the second step for analyzing modeled systems (cf. Fig. 6.1) and requires only minor effort compared to model specification (we required less than 10 min per investigated quality property).

6.4.2.1 Capacity Analysis

By using our CloudStore model as input, the Analyzer can automatically determine capacity values for a given user number interval (see Sect. 6.3.1). For example, when we set the interval to [1 user, 1000 users], the Analyzer determines the concrete capacity for that interval. If the real capacity is beyond the interval borders, the Analyzer returns "1000 users" as a result, thus leaving to the software architect the decision to investigate further intervals.

A run of the Analyzer with our CloudStore model and a range of [1 user, 1000 users] indeed results in this situation: The Analyzer outputs that the user capacity is 1000 users (or more). To get more accurate results, we next configured the Analyzer for the interval [1000 users, 2000 users]. Unfortunately, the Analyzer throws an "unable to create new native thread" exception when started like this. After digging into the cause of this exception, we found that the Analyzer's simulation engine creates two threads per simulated user; however, our operating system does not support the creation of so many threads. We reported this limitation at a Palladio developer meeting and got informed that a new implementation of the simulation's thread handling based on light-weight threads for the Java virtual machine (JVM) (fibers) is currently ongoing, which will fix the issue.[1] At this point, we stopped the investigation of CloudStore's capacity based on our model, while noting that the result for 1000 users is in line with the corresponding benchmark.

6.4.2.2 Scalability Analysis

Given the same inputs as for the capacity analysis, the Analyzer again outputs a value of 1000 users for a scalability analysis (configured as described in Sect. 6.3.1). The interpretation of this result is that there is no scalability issue in the range of 1 to 1000 users. Given that the capacity for this configuration is at least 1000 users, this result is no surprise: it generally holds that "scalability range $\geq$ capacity" for any system configuration.

6.4.2.3 Elasticity Analysis

Elasticity analyses make sense only for elastic systems. Therefore, the CloudStore model needs to include self-adaptation rules that specify how CloudStore minimizes SLO violations when workload changes, e.g., in the case of sudden workload peaks.

[1]The minutes of the developer meeting can be found (in German) at https://sdqweb.ipd.kit.edu/wiki/PCM_Development/Palladio_Concall/Minutes_20150509.

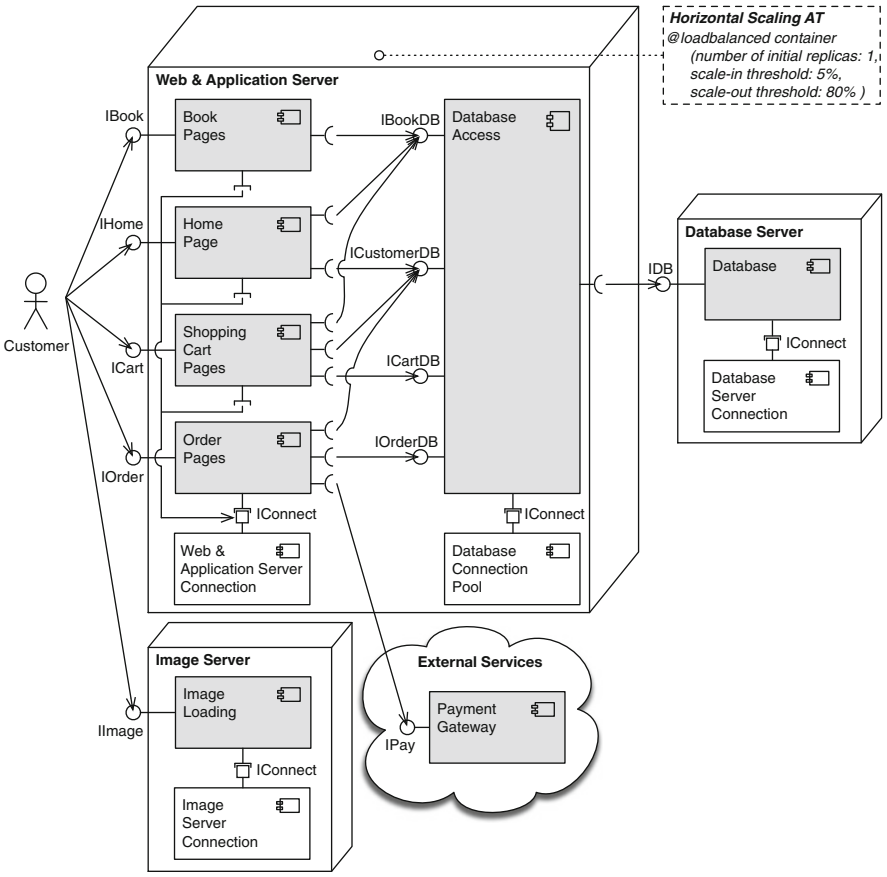

Fig. 6.13 Extended PCM model of the CloudStore online bookshop with an annotated AT role

We have included such a self-adaptation mechanism via the `Horizontal Scaling (Container)` AT (described in Sect. 4.4.3) in our model. Figure 6.13 illustrates that the AT's `load-balanced container` role annotation at the `Web & Application Server` models horizontal scaling: Initially, CloudStore has one replica, scales out if average CPU utilization increases above 80%, and scales in if average CPU utilization decreases below 5%. Such averages are determined every 5 min (like in Amazon EC2).

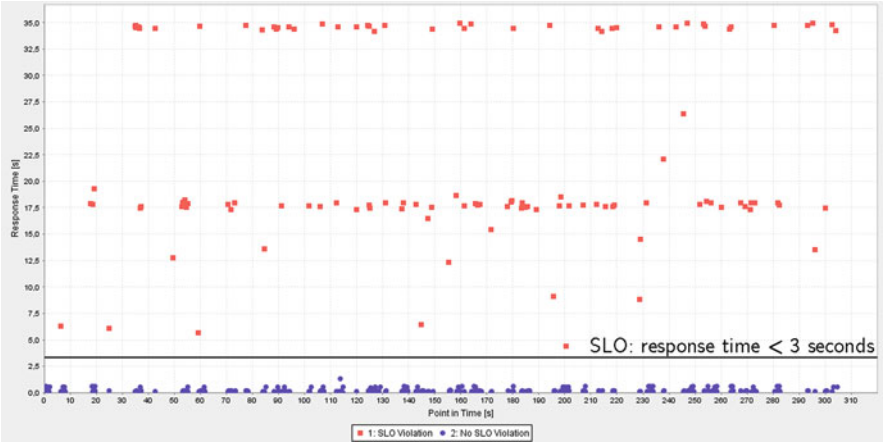

Fig. 6.14 Elasticity result for CloudStore with a horizontal scaling web & application server

The graph in Fig. 6.14 shows the result of the elasticity analysis with regard to the number of SLO violations. Over simulation time (*X*-axis), response times (*Y*-axis) are repeatedly too high and violate the given response times SLO of 3 s. The Analyzer output highlights these violations, as specified in the graph's legend.

The interesting observation is that—despite the applied AT—response times keep violating the SLO. These results therefore point to the fact that we actually reconfigured the wrong server: The `Database Server` is the actual bottleneck for the investigated workload (which can be seen, e.g., in the Analyzer's results for `Database Server` utilization). A way to resolve the issue is to apply the `Vertical Scaling` AT (cf. Sect. 4.4.3) to the `Database Server`, like illustrated in Fig. 4.9. A second elasticity analysis run indeed confirms that then no SLOs are violated anymore.

6.4.2.4 Cost-Efficiency Analysis

Given a system that dynamically scales in and out, different costs incur over time for the demanded resource containers. The Analyzer reports the total costs incurred within a specific time interval over time.

The example result in Fig. 6.15 illustrates such an Analyzer output. Over time (*X*-axis), different costs incur for resource usage (*Y*-axis). Costs vary because the system demands more and more resource containers (i.e., replicas of CloudStore's `Web & Application Server`).

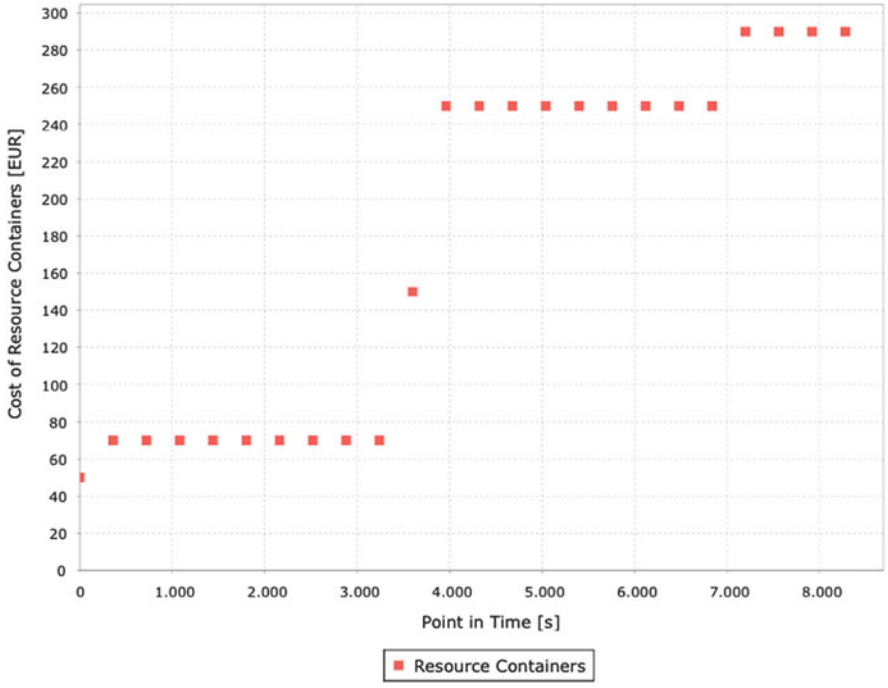

Fig. 6.15 Cost-efficiency result for CloudStore with a horizontal scaling web & application server

6.5 Conclusion

This chapter details the process, using the CloudScale method, that software architects can employ to analyze a modeled system. The two essential steps that software architects need to conduct in this process are (1) to model a system in ScaleDL and (2) to analyze this model using CloudScale's Analyzer. Based on analysis results, software architects can decide whether SLOs need to be renegotiated, the model needs adaptations to fulfill SLOs, or the system can be realized as planned.

The main benefit of such a model-based analysis is that software architects can assess their design and what-if scenarios without having to implement each option. For example, software architects may need to decide whether they should horizontally scale a particular server. As our results in this chapter show, such patterns can be easily applied wrongly, i.e., without being beneficial: In the CloudStore example, it was not beneficial to horizontally scale the web & application server. The analysis results directly pointed to this issue without much effort.

Some modeling effort (and experience) is required to create accurate ScaleDL models. Fortunately, CloudScale's Extractor allows to automatically extract initial models from existing source code. Moreover, CloudScale's HowTos—and ATs in particular—allow software architects to efficiently integrate recurring patterns in their models. Also the power of the Analyzer—with support for capacity, scalability, elasticity, and cost-efficiency analyses—often outweighs modeling efforts.

References

1. CloudScale: Whitepapers, http://www.cloudscale-project.eu/publications/whitepapers [Visited on 12/19/2016]
2. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley Longman, Boston (2000)
3. Koziolek, H., Happe, J.: A QoS driven development process model for component-based software systems. In: Proceedings of the 9th International Conference on Component-Based Software Engineering, ser. CBSE06, Västerås, pp. 336–343. Springer, Lund (2006)
4. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. J. Syst. Softw. **82**, 3–22 (2009). [Online] Available: http://dx.doi.org/10.1016/j.jss.2008.03.066
5. Becker, S., Busch, A., Brosig, F., Burger, E., Durdik, Z., Heger, C., Happe, J., Happe, L., Heinrich, R., Henss, J., Huber, N., Hummel, O., Klatt, B., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K., Küster, M., Langhammer, M., Lehrig, S., Merkle, P., Meyerer, F., Noorshams, Q., Reussner, R.H., Rostami, K., Spinner, S., Stier, C., Strittmatter, M., Wert, A.: In: Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K. (eds.) Modeling and Simulating Software Architectures – The Palladio Approach, p. 408. MIT Press, Cambridge (2016). [Online] Available: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures
6. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman, Boston (1998)
7. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley, New York (2009)
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley, New York (1996)
9. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, vol. 2. Wiley, New York (2000)
10. Kircher, M., Jain, P., Pattern-Oriented Software Architecture: Patterns for Resource Management. Wiley, New York (2004)
11. Buschmann, F., Henney, K., Schmidt, D.: Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing. Wiley, New York (2007)
12. Buschmann, F., Henney, K., Schmidt, D.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages. Wiley, New York (2007)
13. Rhoton, J., Haukioja, R.: Cloud Computing Architected: Solution Design Handbook. Recursive Press, London (2011)
14. Wilder, B.: Cloud Architecture Patterns: Using Microsoft Azure. O'Reilly Media, Sebastopol (2012)
15. Erl, T., Puttini, R., Mahmood, Z.: Cloud Computing: Concepts, Technology and Design. Prentice Hall PTR, Upper Saddle River (2013)
16. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer, Heidelberg (2014)
17. AWS Architecture Center. http://aws.amazon.com/architecture [Visited on 05/20/2016]

18. Lehrig, S., Becker, M.: Approaching the cloud: using Palladio for scalability, elasticity, and efficiency analyses. In: Proceedings of the Symposium on Software Performance 2014, 26–28 November 2014, Stuttgart (2014)
19. Becker, M., Lehrig, S., Becker, S.: Systematically deriving quality metrics for cloud computing systems. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ser. ICPE'15, pp. 169–174. ACM, Austin (2015). [Online] Available: http://doi.acm.org/10.1145/2668930.2688043
20. Lehrig, S., Eikerling, H., Becker, S.: Scalability elasticity, and efficiency in cloud computing: a systematic literature review of definitions and metrics. In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, ser. QoSA'15, pp. 83–92. ACM, Montreal (2015). [Online] Available: http://doi.acm.org/10.1145/2737182.2737185
21. Lehrig, S., Becker, S.: Using performance models for planning the redeployment to infrastructure-as-a-service environments: a case study. In: Proceedings of the 12th International ACM SIGSOFT Conference on Quality of Software Architectures, ser. QoSA'16. ACM, Venice (2016)
22. Screencasts. http://www.cloudscale-project.eu/results/screencasts/ (2016)
23. CloudStore Model. https://github.com/CloudScale-Project/Examples/tree/master/CloudStore/analyser (2016)
24. Kieker. http://kieker-monitoring.net [Visited on 06/20/2016]
25. Apache JMeter. http://jmeter.apache.org [Visited on 06/20/2016]

Chapter 7
Analyzing and Migrating an Implemented System

Steffen Becker and Sebastian Lehrig

Abstract While modern systems are often built in a way that respects cloud computing requirements, the majority of existing systems have been built before dynamically allocatable resources became mainstream. Those systems have been designed and implemented for dedicated hardware, often large-scale servers, which had been sized to the maximum workload the systems were expected to face. This approach led to a high amount of wasted resources, which were operated in a stand-by fashion only to deal with seldom high-load situations. Therefore, it is desirable to migrate legacy systems in a way that they can benefit from the cloud computing approach. However, several issues arise. Legacy systems were often built without upfront modeling or the models became outdated over time. Additionally, while there are several tools that analyze legacy systems to detect insufficient coding style or bad designs, almost no tooling exists that spots defects in the systems. This deficiency hinders systems to smoothly operate in cloud computing environments, i.e., these systems have a limited scalability. In CloudScale, we address this issue by dedicated, built-in method support for system evolution, i.e., for migrating legacy systems to cloud computing environments. In this chapter, we outline CloudScale's evolution support and present tools which help software architects to migrate legacy systems to scalable, cloud computing applications.

Section 7.2 describes the process steps when spotting HowNotTos. Statical detection of HowNotTos is further elaborated in Sect. 7.3, while dynamic detection is discussed in Sect. 7.4. Section 7.5 links HowNotTos with best-practice HowTos. An example is described in Sect. 7.6.

S. Becker (✉)
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
e-mail: steffen.becker@informatik.uni-stuttgart.de

S. Lehrig
IBM Research, Technology Campus, Damastown Industrial Estate, Dublin 15, Ireland
e-mail: sebastian.lehrig@ibm.com

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_7

7.1 Introduction

In the following, we give a brief overview of the evolution support process in the CloudScale method (cf. Fig. 7.1).

When using the CloudScale method in evolution scenarios, we take the route highlighted in Fig. 7.1 using a bold black, dashed arrow. These evolution scenarios cover methods and tools to assist software engineers to evolve an existing software system to enable its deployment in cloud computing environments by checking for and implementing scalability requirements.

Besides the steps already explained in Chap. 5, the relevant step for evolution scenarios is the `Use Spotters` step. It is needed for scenarios where software architects want to know whether their *existing* systems will scale. This is particularly important if legacy systems are migrated to a cloud computing environment, where the underlying platform provisions additional resources to deal with increasing demands. Only scalable software is able to effectively utilize the additional resources and, hence, to cope with the increased demands. The `Use Spotters` step analyzes the source code of existing code-bases or implemented, deployed,

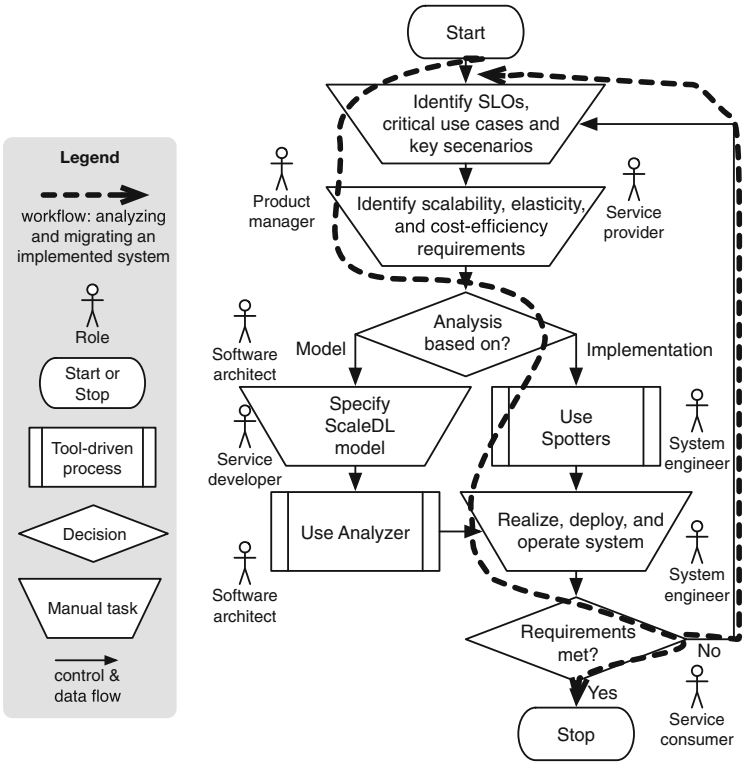

Fig. 7.1 Reverse engineering and reengineering in the CloudScale method

and executing systems. In various substeps (as explained in Sect. 7.2), it diagnoses problems (called HowNotTos, as introduced in Sect. 2.10). These HowNotTos indicate limits in the code's or the deployed system's scalability. In the remainder of this chapter, we explain in detail how this spotting works.

7.2 Spotting HowNotTos

This section details the `Use Spotters` step in the CloudScale method as depicted in Fig. 7.1. Figure 7.2 shows the subactions that have to be executed by an engineer in the `Use Spotters` method step.

Engineers start the `Use Spotters` activity in Fig. 7.2 with a static detection of anti-pattern candidates in the source code of the system's implementation. This is done in the `Use Static Spotter` step and supported by the Static Spotter tool. For this detection, the tool takes three artifacts as input (see top row of Fig. 7.2): the system's source code, an anti-pattern catalog (e.g., the scalability anti-pattern catalog provided by CloudScale), and a Static Spotter configuration defining various detection parameters (as documented in CloudScale's user manual [1]). The main parameters are those which configure the included Extractor run. More details on these parameters can be found in Sect. 4.5.3.2.

The Static Spotter tool first creates a model from the provided source code (by reusing internally the Extractor tool; cf. Sect. 4.5.3.2). Afterward, the Static Spotter tool searches inside the created model for anti-pattern candidates provided in the anti-pattern catalog. For example, it looks for and marks all synchronized methods in an implementation, as they are all potential One-Lane Bridges (OLBs). Details on the Static Spotter and its anti-pattern catalog are given in Sect. 7.3. After successfully executing the Static Spotter, software engineers have a set of anti-pattern candidates. This set might be helpful on its own, as it provides useful insights to developers. If developers can make sense out of the anti-pattern candidates, they can abort executing the spotter method here. However, normally, developers continue with the next steps.

The goal of the next steps is to check whether identified anti-pattern candidates from the static detection manifest in actual anti-patterns. In these steps, software engineers first prepare the desired target environment for a dynamic detection. For this, software engineers conduct two steps in parallel: They set an instrumented service into operation (left hand fork in Fig. 7.2) and they configure the dynamic detection run (right hand fork in Fig. 7.2). Note that there is more configuration and set-up work needed for the Dynamic Spotter, as it is generally more complex to instrument and execute a system in an artificial workload environment than doing static code analysis.

For instrumentation, the system's source code has to be instrumented with probes that measure various performance metrics like response times, passage times, waiting times, or utilizations. This instrumentation is a semi-automatic step for developers and depends on the instrumentation framework used. Typical frame-

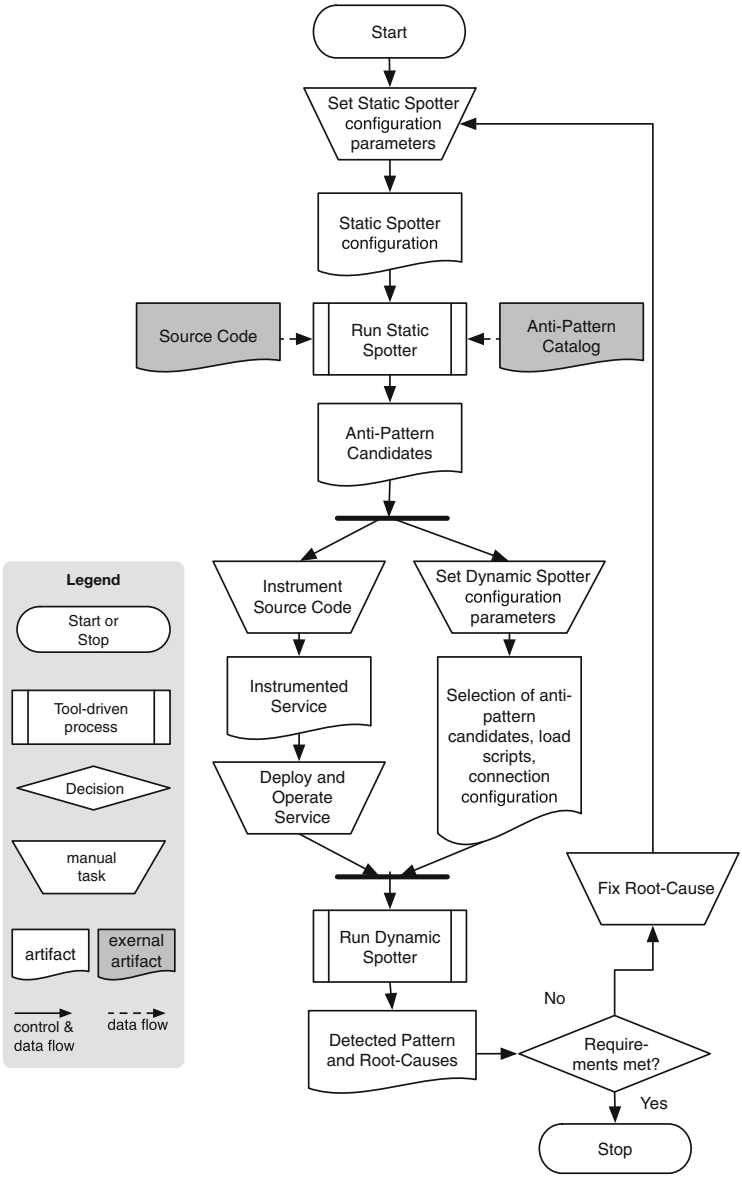

Fig. 7.2 Overview on the Use Spotters process

works used for instrumentation are Kieker,[1] AIM,[2] and DynaTrace.[3] Configuring an instrumentation framework is a semi-automatic step, as developers need to manually define the positions in the system's implementation that they want to monitor. Afterward, the monitoring frameworks can automatically insert probes into the system's implementation. The anti-pattern candidates identified by the Static Spotter serve as starting points for the manual identification of suited monitoring spots. Once the system is instrumented with probes, it is started in its genuine execution environment.

Concurrently, the Dynamic Spotter is configured by selecting the anti-pattern candidates to be verified, load scripts implementing key scenarios, and settings that specify how to connect to the instrumented service (for detailed configuration documentation, consult CloudScale's user manual [1]). In case the Static Spotter provided suitable anti-pattern candidates, the selected candidates allow the Dynamic Spotter to inspect only those candidates. But even in case no anti-pattern candidates have been found (or no suitable candidates have been selected), the Dynamic Spotter operates with a set of default dynamic anti-pattern candidates. These default candidates then allow for a coarse-grained inspection of the system, i.e., by using only probes at the system boundary and not analyzing system internals. For example, such a probe might only inspect the main business logic interface or the entry of the database access layer.

Once these inputs are prepared, the Dynamic Spotter analysis step is conducted. As output, the Dynamic Spotter provides a list of detected anti-patterns as well as their root causes. This step is further detailed in Sect. 7.4.

After running the Dynamic Spotter, the software engineer has a set of HowNotTos that hinder his system to scale. It is up to the engineer now to decide whether any found HowNotTos are severe and need to be fixed or whether they are satisfied with the current state of the system. Depending on whether this list includes crucial anti-patterns (i.e., those that cause violations in the configured service-level objectives (SLOs)), the software engineer can decide as part of the reengineering activity to fix root causes of such crucial anti-patterns. For fixing root causes, we provide a novel concept that links anti-patterns to patterns (cf. Sect. 7.5). Once fixed, the system engineer can then iterate the Spotter process again to check whether he or she successfully removed detected anti-patterns or to fix more anti-patterns.

Once the results are satisfactory, the `Use Spotters` substep in Fig. 7.1 is finished, and engineers can resume executing the CloudScale main method.

[1] http://kieker-monitoring.net.

[2] http://sopeco.github.io/AIM.

[3] https://www.dynatrace.com.

7.3 Statically Detecting HowNotTos

In this section, we describe the Static Spotter's internals, and the methodology used. The Static Spotter is a reverse engineering tool based on Reclipse [2], a tool for the automatic detection of so-called search patterns, which are then interpreted as potential scalability anti-patterns. Search patters formalize certain source code structures, which may appear in existing code, e.g., a block of Java statements encapsulated in a synchronized block. Such a structure becomes a scalability anti-pattern if the synchronized block turns out to become a OLB scalability anti-pattern in the case of increased workload. Here, we use the Static Spotter to detect scalability anti-pattern candidates so that during the evolution phase, software engineers can locate potential or dormant scalability issues with the help of the Static Spotter.

In the following, we describe the Static Spotter's methodology. The Static Spotter takes partial Scalability Description Language (ScaleDL) models as an input. Partial ScaleDL models can be the ones generated by the Extractor (cf. Sect. 4.5.3.2). The output of the Static Spotter is a ranked list of scalability anti-pattern candidates.

The Static Spotter does two things: parse the code and models, and then look for search patterns and interpret them as scalability pattern candidates. There is a pre-configuring step before parsing and searching: to set up the search pattern catalogs. In these catalogs, graph patterns are used to formalize the search patterns. Figure 7.3 shows a catalog used to detect the scalability anti-pattern OLB. For example, it shows an AcquireReleasePair on the top, which is a pattern detecting a code region protected by a mutual exclusive semaphore, or a SynchronizedMethod pattern at the bottom, which detects synchronized Java methods. Other visible patterns are helper patterns.

Since synchronized blocks and synchronized methods are potential OLB candidates, Fig. 7.4 depicts an AcquireReleasePair pattern. A static analysis parses partial ScaleDL models and searches for instances of the defined search patterns. The black rectangles represent objects to be detected in one of the source models (code or extracted ScaleDL model). For the AcquireRelease pair, Fig. 7.4 shows

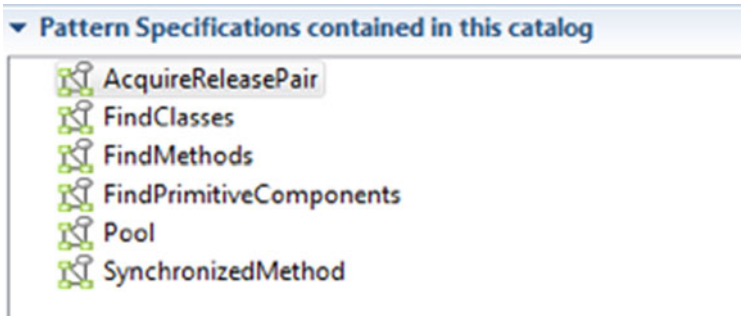

Fig. 7.3 Overview of a pattern catalog

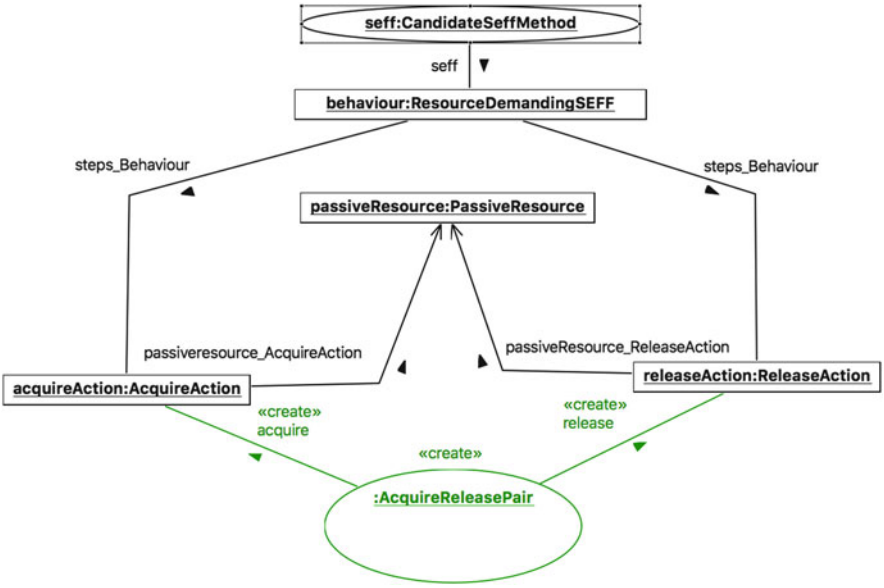

Fig. 7.4 Static Spotter search pattern for AcquireReleasePair

an AcquireAction and a ReleaseAction to be detected. Black lines represent links between objects. For example, the Acquire Action and ReleaseAction have to be linked to the same service effect specification (SEFF) (cf. Sect. 4.5.1.1). Ellipses are annotations used by the Static Spotter tool. Black ellipses are annotations which have been created before, during the detection of a subpattern or helper pattern. A subpattern or helper pattern describes a reusable partial pattern that can be used in the specification of multiple higher-level patterns. Green ellipses specify annotations to be added to the model as marker for found patterns. In our example, the AcquireReleasePair annotation will be used to mark any detected instance of this anti-pattern. The pattern specification language supports more concepts; see [2].

The result of the static analysis is a set of annotations that comply with the search-pattern specifications. We then interpret some of these search patterns as potential scalability anti-patterns (while others just form helper search patterns to structure the search-pattern catalog). The tool provides together with each detected scalability anti-pattern a relevance ranking based on likelihoods. Each likelihood informs the user how certain the tool is that the found scalability anti-pattern indeed is a correct one.

In Fig. 7.5, you see example results after running the Static Spotter. In these results, the Static Spotter finds two instances of AcquireReleasePair pattern and five instances of SynchronizedMethod pattern, as introduced before. Other patterns are helper patterns. The 100% ranking means that the Static Spotter would classify the candidates as certain scalability anti-patterns.

The accuracy of the detection results generated by the Static Spotter obviously depends on the used granularity when configuring the Static Spotter. On a high

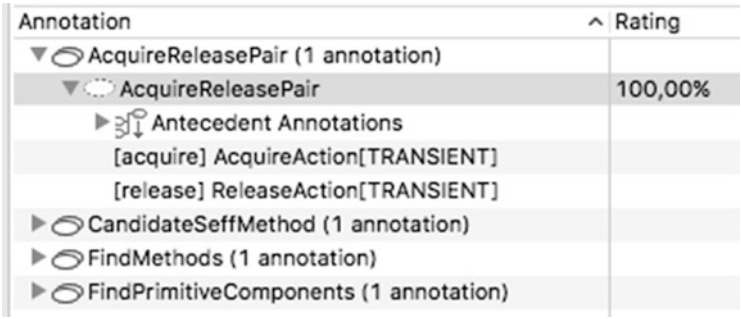

Fig. 7.5 Static Spotter result view

granularity, it will use large components as input. Using large components means abstracting more from the actual component behavior. In these abstract behavior models, scalability issues might not be detectable any more. Using a low granularity, the Static Spotter might detect more anti-pattern candidates, but it takes much longer to complete if it completes at all. Therefore, when using the Static Spotter, system engineers need to pay attention to granularity, as discussed in Sect. 5.2.

7.4 Dynamically Detecting HowNotTos

This section describes the internals of the Dynamic Spotter tool. The tool is an extended version of the method and tool developed by A. Wert and others in A. Wert's PhD thesis and related works [3–5]. In principle, the tool can be compared to the way a doctor diagnoses a disease. By asking for certain symptoms, like high blood pressure, fever, etc., a doctor includes or excludes potential diseases from his final diagnosis. When the list of potential diseases has been reduced to a single disease, she knows what the root cause is and how to treat the patient.

The Dynamic Spotter applies this principle on running software. As symptoms, it uses measurements taken from the instrumented system and interprets these measurements. Such measurements can be classified into two categories: normal measurements and abnormal measurements. For example, if the response times of the critical use case increase the longer the system is running, this is abnormal behavior. Any abnormal behavior is interpreted as a symptom. Based on these symptoms, the Dynamic Spotter maintains a list of "diseases", i.e., a list of HowNotTos which might be a root cause of any of the found symptoms. To improve the Dynamic Spotter's performance, symptoms and root causes are structured in a hierarchy (similar to Fig. 2.4) where deeper levels are refinements of their parent levels (cf. Sect. 2.10). In this way, the Dynamic Spotter can skip inspecting a child layer if no symptoms are found that indicate the existence of the parent level. In our example, for the steadily increasing response times, the Dynamic Spotter includes

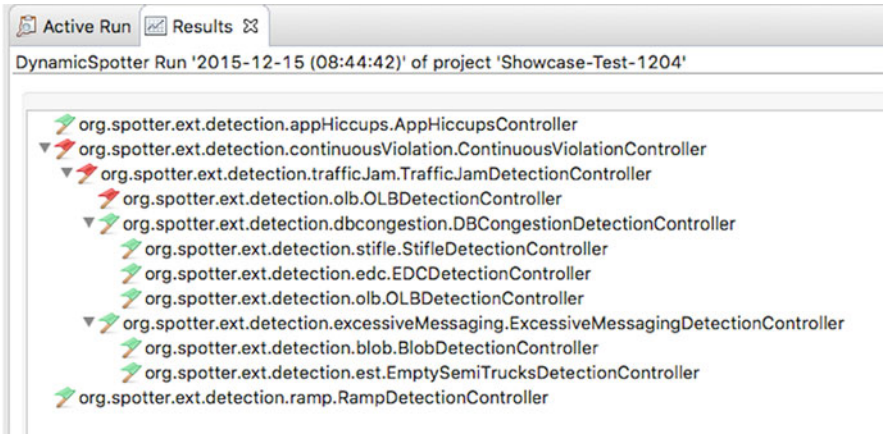

Fig. 7.6 Anti-pattern hierarchy for first iteration: detected anti-patterns are marked in *red*

the "The Ramp" HowNotTo as the possible root cause, which, for example, can be refined into a memory leak when further diagnosed. If none of the symptoms indicate the existence of "The Ramp", we can skip looking for a memory leak. Figure 7.6 shows an example of the final result of running the Dynamic Spotter. It shows the hierarchy of investigated HowNotTos. In front of each HowNotTo, a flag indicates whether this HowNotTo is supported by the identified symptoms. Green flags show HowNotTos that are unsupported by the observed symptoms; red flags mark HowNotTos that provide an explanation of the observed symptoms.

As explained in the general principle used in the Dynamic Spotter, it is important to collect the right measurements to efficiently detect HowNotTos. Collecting the right measurements has two aspects: finding the right measurement point and correctly interpreting the identified key scenarios, in particular, the right work and load situations.

In order to find the right measurement points to instrument, our `Use Spotters` process conceptually integrates static and dynamic analyses. After detecting the pattern candidates in the static analysis step, the software system under analysis is executed and only the candidates' behavior is traced; i.e., the candidates are used as measurement points. A number of measurements are generated for each candidate. The measurements are then analyzed as to whether they are normal or abnormal. In doing so, just parts of the complete program behavior are measured, i.e., only those parts that belong to one of the candidates. This drastically reduces the search space for the dynamic analysis. However, just as the Static Spotter's results depend on the granularity of the extracted model, so do the Dynamic Spotter's results, on the granularity of this model, as it determines the amount of measurement points.

The key scenarios to be tested by the Dynamic Spotter are derived from the key scenarios identified in the beginning of the CloudScale method. They determine the work and load situations for the Dynamic Spotter's measurements. It is a bit of

manual effort to convert the identified key scenarios into the right configuration of the Dynamic Spotter's workload driver. The reason for that is that key scenarios in the CloudScale method often come with a specification of work and load whose realization in the Dynamic Spotter is distributed over multiple configuration parameters. Details can be found in CloudScale's user manual [1].

Overall, the Dynamic Spotter is a long-running tool. Analyses might take hours or even days, as the system under study is measured for each key scenario in all different types of usages as defined by the usage evolution. However, once the system and the Dynamic Spotter are configured, the Dynamic Spotter executes automatically; i.e., it does not need user interaction. The effort for configuring the Dynamic Spotter itself is rather low in cases where the Dynamic Spotter already provides plug-ins for the used development platform. In case such plug-ins are lacking and need to be produced first, adaptation efforts of the tool might be much higher. More effort is also typically spent in configuring and instrumenting the system under study. As companies often have their systems configured in testing labs anyhow, in practice, this effort is often also low. As both efforts are rather low in many real-world settings, the effort in using the Dynamic Spotter in companies is typically rather low, i.e., in the range of a couple of days or weeks. In addition, the tool requires limited training, and developers often trust measurements more than model-based predictions. These arguments have led to our observation that companies often prefer using the Dynamic Spotter over modeling to get started.

7.5 Resolving HowNotTos with HowTos

The "Reengineer" step—illustrated in Fig. 7.2—involves manual and potentially complex actions by reengineers. To guide reengineers, we therefore linked HowNotTos with best-practice HowTos that potentially resolve the respective anti-pattern.

Table 7.1 presents our initial suggestion for this linking. The first column lists some of our scalability HowNotTos (as introduced in Sect. 2.10). The second column lists, per scalability HowNotTo, the scalability HowTos (cf. Sect. 2.9) we associated to these anti-patterns. We associated the patterns we previously formalized as ScaleDL Architectural Templates (ATs) (cf. Sect. 4.4.1) where feasible, i.e., whenever its problem description matched the description of a detected anti-pattern.

As Table 7.1 shows, we associated the Service Load Balancer, Dynamic Horizontal Scaling, Dynamic Vertical Scaling, and Map Reduce to the OLB. A strategy to cope with an OLB is to shorten the time jobs hold a passive resource, e.g., when jobs wait for semaphores to be returned by other jobs. This time can be either shortened by introducing more servers (i.e., multiple lanes) or by lowering service times. More servers are introduced via Service Load Balancer, Dynamic Horizontal Scaling, and Map Reduce. Lower service times are expected via Dynamic Vertical Scaling. Therefore, these patterns should help if an OLB is detected.

Table 7.1 Link between performance or scalability HowNotTos and scalability HowTos

Performance or scalability HowNotTo	Associated HowTo
OLB	(a) Service Load Balancer
	(b) Dynamic Horizontal Scaling
	(c) Dynamic Vertical Scaling
	(d) Map Reduce
The Blob	(a) Service Load Balancer
	(b) Dynamic Horizontal Scaling
	(c) Dynamic Vertical Scaling
	(d) Map Reduce
Empty Semi-Truck	(a) Data Transfer Object
Excessive Dynamic Allocation	(a) Resource Pooling

We suggest the same patterns for The Blob. The rationale is the same: To improve the scalability of a Blob component, we can either distribute it over multiple servers or improve its service time.

For Empty Semi Trucks and Excessive Dynamic Allocation, we follow Smith and William's suggestion to use Data Transfer Objects, respectively Resource Pooling [6].

Our anti-pattern catalog (available at our Wiki [7]) now includes the complete list of suggested solutions to detected anti-patterns.

7.6 Spotter Running Example

In this section, we highlight the results of executing the `Use Spotters` process depicted in Fig. 7.2 on the CloudStore running example. The remainder of this section is structured into two subsections—each one corresponding to one of the main steps: Static Spotting and Dynamic Spotting.

7.6.1 Static Spotter

As first step, we needed to run the Static Spotter. Therefore, we needed to provide a good Static Spotter configuration and the system's source code (the anti-pattern catalog is provided by our CloudScale tool support). In particular, as part of the Static Spotter configuration, we needed a good extractor configuration. For the modernized CloudStore version, we needed eight iterations (2 h effort) with our Static Spotter to find a good trade-off for the granularity of the ScaleDL instance and the identified anti-pattern candidates (a few coarse-grained components with

Fig. 7.7 PCM component repository extracted from the CloudStore code base

a lot of functionality vs. lots of fine-grained components). For each iteration, we had to either modify Extractor configuration parameters (e.g., too high clustering thresholds for merging components into a single component) or resolve issues within the modernized CloudStore's source code (e.g., removing test classes the Static Spotter wrongly used as main application). At the end, we got a good high-level understanding of the CloudStore implementation: We extracted a Façade component representing the servlet container, and inner servlet components that require data access components for handling database queries. This high-level view fits to our expectation and the actual implementation, thus suggesting that we successfully extracted a ScaleDL instance for the modernized CloudStore version.

After we found a good configuration and fixed all source code issues, executing the Static Spotter could be considered successful. Internally, it extracted 16 components, of which 4 are composite components (see the repository model illustrated within a tree editor in Fig. 7.7; the marked components are composite components). Overall, this amount of components provides us a good overview of the system and reflects how the implementation code is actually structured, e.g., in terms of database accesses.

However, the Static Spotter's resulting anti-pattern candidate list was empty for the modernized CloudStore version. Obviously, this version did not contain any statically detectable anti-patterns—as it was expected by this version's developers. They paid close attention not to introduce scalability bottlenecks. As there are no anti-pattern candidates for running the Dynamic Spotter, no hints for more specialized instrumentation can be derived. As a consequence, we configure it to detect anti-patterns at the system's entry point, as outlined in the next section.

7.6.2 Dynamic Spotter

As described before, we had to execute the Dynamic Spotter step with the default configuration, which detects scalability issues of the overall application. We took the CloudStore source code, compiled and deployed it on appropriate instances running dedicated virtual servers. Afterward, we instrumented the system using the AIM monitoring framework. Overall, it took several days to complete all these steps, as we had no pre-existing lab version of the CloudStore running and we faced several technical issues, which we had to overcome. For example, the Dynamic Spotter needed to be adapted to our used platform and networking infrastructure, which prevented direct communication of the Dynamic Spotter with the Java virtual machines (JVMs) hosting the application in order to collect its measurements. In addition, the Dynamic Spotter's class loader needed adaptations, as it interfered with the class loader used in the Tomcat application server. At first, we configured the Spotter to examine both the application server (Tomcat with the modernized CloudStore) and the database server (MySQL with data of all books).

Once the system was deployed, instrumented, and running, we configured the Dynamic Spotter. The most time-consuming task here was to encode CloudStore's critical use case and its key scenario into a load driver script. We used Apache JMeter in our case for this, which already provides assistance for this task. We configured the maximum number of users. This value describes the number of users the system under test should be able to handle. We set it to a maximum of 100 to be tested in four steps; so, the Dynamic Spotter tests the system with 1 user, 34, 67, and 100 users. For each symptom in the anti-pattern hierarchy, we measured the instrumented system for 5 min. We changed the default requirement threshold to 100, which describes the response time SLO in milliseconds. Overall, this added another couple of days until all was configured correctly. Finally, we were able to execute the Dynamic Spotter tool using CloudScale's catalog of HowNotTos.

As a result, we got a list of tested HowNotTos, each with an annotation telling us whether it could be diagnosed in the CloudStore or not. The result is depicted in Fig. 7.8. As good news, CloudStore does not show application hiccups when executing its critical use case. Then, the tool detected a Continuously Violated Requirement. This means that CloudStore violates its SLOs continuously under high load.

Therefore, the Dynamic Spotter tried to further investigate this anti-pattern to identify the root cause. From the measurements taken from the running CloudStore, a Traffic Jam anti-pattern was diagnosed next. It describes a scalability problem, either due to software bottlenecks or hardware limitations in general.

When further refining this Traffic Jam, the Dynamic Spotter diagnosed a One-Lane Bridge anti-pattern. Looking at the Dynamic Spotter's detail view, we could see response time, which increased under growing load, while the CPU level stayed constantly low. In detail, we get a measurement tuple showing the system's average response time for each of the configured load situations (1, 37, 66, or 100) and its corresponding CPU utilization. The average response times are increasing, while

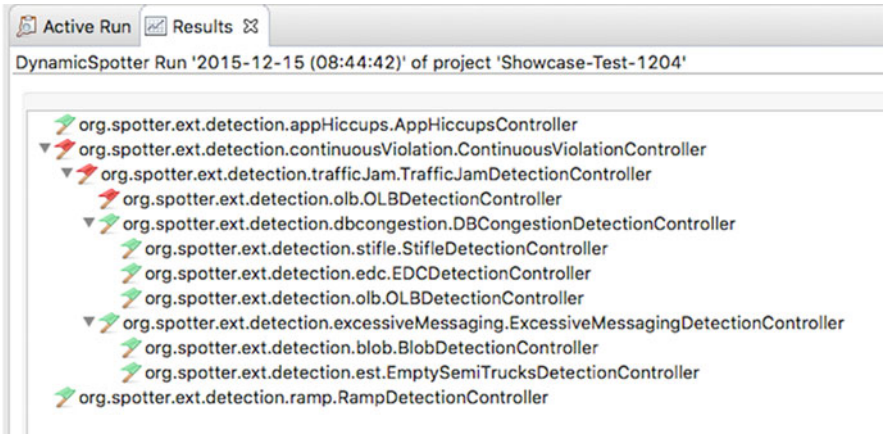

Fig. 7.8 First run of the Dynamic Spotter step

the CPU utilization is constantly around 50% usage. If there was no OLB in the modernized CloudStore version, the CPU level would, however, increase further. The OLB obviously causes the CPU utilization to stay on its level. Therefore, the system slows down under increasing load. The average response times are increasing. This scalability HowNotTo can be solved, for example, by increasing the number of concurrent processes which can access a limited resource or by reducing the time a process holds the limiting resource. After diagnosing the OLB, the Dynamic Spotter did not diagnose more anti-patterns and finished its execution.

In a second Dynamic Spotter run, we wanted to increase the statistical significance of our results by running each experiment longer (600 s instead of 300 s) and with a higher number of users (150 users in five steps instead of only 100 users in four steps). However, these increases caused an out-of-memory Exception within our Dynamic Spotter tool in the first place. We determined the root cause for this to be a bad configuration of the instrumented Tomcat application server: It had only 128 MB of heap size. This configuration was also the potential root cause for the detected OLB in the first iteration. We accordingly doubled the heap size of our Tomcat to 256 MB and were subsequently successful in running the Dynamic Spotter with increased experiment times and users. Due to these increases, the whole run through the hierarchy took 4 h (instead of only 0.5 h for the first run). However, we were rewarded in the end for this effort with a different combination of diagnosed anti-patterns.

A difference is that we also detected application hiccups (see Fig. 7.9, which shows the system's response time measurements, including the hiccups, as frequent spikes). A hiccup is an anti-pattern that often can only be diagnosed when experiments run for a longer time—which explains why we did not diagnose it in the first Dynamic Spotter run. In the second run, we discovered that CloudStore

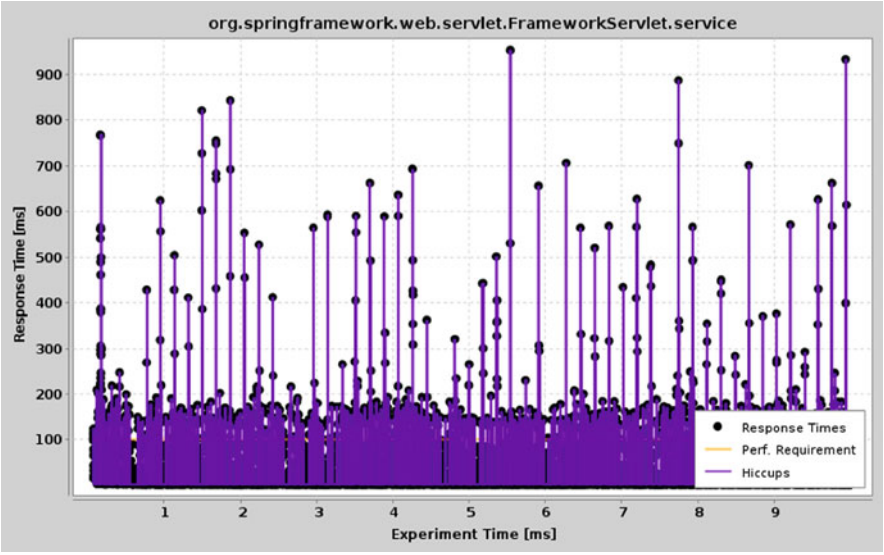

Fig. 7.9 Response time measurements from CloudStore with hiccups

suffers—from time to time—from high response-time peaks. We suspect that these peaks come from Java's garbage collection; however, we did not investigate this issue further.

7.7 Conclusion

This chapter details the steps of the CloudScale method used for analyzing and migrating implemented systems. It illustrates the Spotter tool and its substeps. The Spotter helps in identifying root causes of performance and scalability issues. Our method steps then provide hints for reengineering the system. The chapter shows the results of the method when executed on CloudStore.

From this execution, several benefits have been illustrated in the chapter. First of all, we could see that CloudScale's method supports for analyzing and migrating an implemented system works, and provides useful insights into the problems of implemented systems. We also learnt that there is no free lunch: Executing the `Use Spotters` activity takes time and requires some experience in configuring the tools, preparing the source code, and operating the instrumented system. However, given the overall effort of a development endeavor like CloudStore, the required effort is still rather low.

Even though both the Spotter tools are useful, they are not industrial, well-proven tools. In Chap. 8 we describe the required steps for aligning both tools with existing software engineering processes.

References

1. CloudScale: User Manual of the CloudScale Environment. http://www.cloudscale-project.eu/media/filer_public/2016/02/01/cloudscaleenvironment-userguide_1_1.pdf [Visited on 12/19/2016]
2. von Detten, M., Meyer, M., Travkin, D.: Reverse engineering with the reclipse tool suite. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), Cape Town, 2–8 May 2010
3. Wert, A.: Performance problem diagnostics by systematic experimentation. Karlsruhe Institut fur Technologie, Technical Report (2015)
4. Wert, A., Happe, J., Happe, L.: Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In: Proceedings of the 2013 International Conference on Software Engineering, ser. ICSE'13, pp. 552–561. IEEE Press, San Francisco (2013). [Online] Available: http://dl.acm.org/citation.cfm?id=2486788.2486861
5. Wert, A., Oehler, M., Heger, C., Farahbod, R.: Automatic detection of performance anti-patterns in inter-component communications. In: Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures, ser. QoSA'14, pp. 3–12. ACM, Marcq-en-Bareul, France (2014). [Online] Available: http://doi.acm.org/10.1145/2602576.2602579
6. Smith, C.U., Williams, L.G.: Software performance antipatterns. In: Proceedings of the 2nd International Workshop on Software and Performance, ser. WOSP'00, pp. 127–136. ACM, Ottawa (2000). [Online] Available: http://doi.acm.org/10.1145/350391.350420
7. CloudScale Wiki: HowNotTos, wiki.cloudscale-project.eu/HowNotTos:_Anti-Patterns [Visited on 12/19/2016].

Part IV
Making the CloudScale Method Happen

We introduced the CloudScale method in the previous parts, where we focused on the CloudScale method's motivation, gave a high-level overview, and finally explained all technical details of the method. In this part, we focus on a different aspect of the CloudScale method, which is equally important for its success in practice: its implementation in real-world projects.

Implementing a novel development method in a software development process is not only a matter of a well-defined, tool-supported method, but also one of motivating, educating, and encouraging developers to actually use it. Using a new method means shifting from established and familiar development steps to novel steps, unknown in the beginning and with no prior experience. Therefore, this process requires management guidance and support. In order to convince developers, but also higher-level managers, of the endeavor of implementing a new method, one needs good arguments, i.e., the benefits of the new method must be crystal clear. On the other hand, managers also need a good understanding of the costs incurred by additional method steps.

One way of judging the benefits and drawbacks of a new method is by studying reference projects. In the case of the CloudScale method, several demonstrators and reference projects have been executed. In this part, we report on them and distill the lessons learned.

In Chap. 8, we directly address both business-oriented and technical managers, and provide guidance in all steps of implementing the CloudScale method as well as in making decisions during that implementation. Our demonstrators and reference projects described in Chap. 9 serve as a starting point when you want to learn from our experience.

Chapter 8
The CloudScale Method for Managers

Steffen Becker, Gunnar Brataas, Mariano Cecowski, Darko Huljenić, Sebastian Lehrig, and Ivana Stupar

Abstract Having described the CloudScale method for engineering scalable cloud computing applications in the previous chapters, we explicitly address managers of software development processes in this chapter. It answers questions managers have in mind when considering the CloudScale method: Is it worth implementing the CloudScale method in my organization? What does it take? What are the benefits? What will be the costs? How should I get started? This chapter addresses all these questions and provides answers based on our own experience that we gained when introducing and applying the CloudScale method in practice. In the course of the chapter, we distinguish two types of managers: project managers, who are concerned with managing project teams that implement the business requirements, and technical managers, who manage the actual development efforts and take technical decisions.

The chapter is structured as follows. After a brief introduction (cf. Sect. 8.1), it first addresses project managers. We illustrate key considerations that project managers should be making when applying the CloudScale method (cf. Sect. 8.2). Afterward, we sketch how the CloudScale method interacts with other development processes (cf. Sect. 8.3) and what its pros and cons are (cf. Sect. 8.4). The remainder of the chapter addresses technical managers. First, it sketches a pilot project

S. Becker (✉)
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
e-mail: steffen.becker@informatik.uni-stuttgart.de

G. Brataas
SINTEF Digital, Strindvegen 4, 7034 Trondheim, Norway
e-mail: gunnar.brataas@sintef.no

M. Cecowski
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: mariano.cecowski@xlab.si

D. Huljenić • I. Stupar
Ericsson Nikola Tesla, Krapinska 45, 10000 Zagreb, Croatia
e-mail: darko.huljenic@ericsson.com; ivana.stupar@ericsson.com

S. Lehrig
IBM Research, Technology Campus, Damastown Industrial Estate, Dublin 15, Ireland
e-mail: sebastian.lehrig@ibm.com

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_8

in Sect. 8.5 to guide the discussion. Using this pilot, in Sect. 8.6, we briefly outline how to set up CloudScale's IDE, which can be complemented by third-party tools introduced in Sect. 8.7. We apply the CloudScale method on the pilot in Sect. 8.8.

8.1 Introduction

After introducing the CloudScale method and its tools for software architects and developers, you as project and technical managers might still wonder how the entire process of following the CloudScale method looks like in practice. Despite that the CloudScale method steps were already described in more detail in previous chapters, in particular in Chaps. 5–7, this chapter gives an overview of the entire CloudScale method targeting management aspects. First, aspects related to project management are discussed, followed by technical management aspects. For the latter, we introduce in Sect. 8.5 a simple application used as a pilot.

The goal of this chapter is to guide interested project managers with respect to the cost and effort required to perform the steps of the method. You will be informed about some of the key considerations in the CloudScale method in order to gain most benefit out of it. We will discuss the pros and cons of the CloudScale method based on our own experience. The relation of the CloudScale method to other engineering practices is discussed and possible integrations are introduced. Afterward, we guide technical managers through the process of using the CloudScale method for an analysis and optimization of a simple pilot project. This includes the setup of the CloudScale Environment, as well as the choice of the CloudScale tools and complementing tools needed to complete the CloudScale method iteration. At the end of the chapter, we briefly outline future prospects of the CloudScale method usage.

8.2 Key Considerations

When project managers consider using the CloudScale method, they should answer a set of questions for their specific organization and project context (which might be elaborated together with the technical manager). We call these questions the *key considerations* to make before implementing the CloudScale method.

Is it worth it at all? The first question that project managers should address is whether they want to invest the money and effort required in implementing the CloudScale method at all. The answer to that question depends, to a large extent, on the risk of failing to meet scalability, elasticity, or cost-efficiency requirements. The more business or mission critical these requirements are, the greater is the benefit you get out of the CloudScale method. For example, when you implement an online flower shop, it is important to scale to the customer load faced on Mother's Day as well as using only few resources during the remainder of the year.

What is my use case? Once you have decided to use the CloudScale method, the next thing to consider is the use case in which you want to use the CloudScale method. In case you have an existing development ongoing, you most likely want to verify it to identify potential issues. This often happens in the context of migrating an existing system to the cloud. In this case, you have the source code of this system. The next thing to consider is the language and maturity of the code: Is it written in Java? Does it follow a clean modularization in terms of classes or components or is the code a "big ball of mud"—grown without governance over time? In the former case, you get more support by the existing CloudScale tools; in the latter case, you should consider asking the technical manager to refactor the source code in general before addressing its scalability or before migrating it to new platforms.

How complex is my system? What analysis granularity do I need? When tool usage on the source code is possible, the next thing to consider is the system's complexity. In case it is a huge system, this has two consequences. First, the system's implementation might be too large to be consumed by the CloudScale tools. In this case, you should plan for efforts involved in adjusting the CloudScale tools to your system's complexity. Second, you should carefully consider all configuration options which have an impact on the granularity of CloudScale's analyses (cf. Sect. 5.2). In the case of huge systems, you should definitely start with a coarse-grained analysis and refine it later on.

Which analysis questions do I need to have answered? Which requirements do I have? Next, you should also consider your particular problem or question. First, you should know the quality attributes which are crucial for your mission's success. This also means, if you have not gathered scalability, elasticity, or cost-efficiency requirements and quantified them, now is a good time to do so. This will help you in your future development even when not using any of CloudScale's methods or tools. When collecting such requirements, you should also additionally collect the critical use cases and key scenarios. From our experience, problems with scalability of systems are often rooted in ill-specified or non-existing scalability requirements!

Do I already know about scalability issues? When the requirements are clarified, you should consider where in your system you may have weak spots which might prevent requirements' fulfillment. Typically, the development team of a certain software system has a good idea about the quality properties of the system. Go ahead and interview them to learn about the most critical components in your system. If you are lucky, you also learn about concrete problems with your system as it is at that time. You should use this information to focus on particular parts of your software and analyze them right in the beginning.

Which metric do I want to have analyzed? Finally, you should be clear about the type of information you would like to gather during the analysis of your system; i.e., you should have concrete questions and metrics you want to know about the system. Having a proper understanding of this helps again to focus the analyses and also ensures that the answers you get are the answers you wanted.

How do I map my system and context to the CloudScale method? Before you start analyzing your system, you should revisit the CloudScale method and check whether you fully understood all method steps needed. In particular, you have to be

able to map your system and context to the CloudScale method steps. All required inputs (documentation, code, etc.) should be available for all steps. If not, you have to gather them first.

When modeling, do I have enough skilled people? In case you want to implement a new function or large parts of your system from scratch, using a model to plan and verify your design is more useful. You should consider the maturity of your developers and of the development process before doing so. In particular, you need to ensure that you have the needed skills available in your development team. We recommend also to start modeling on a small project or subsystem before moving to large systems.

After running the method, how do I reflect on its success? All the above key considerations are pure theory as long as you have not tried to execute the CloudScale method at least once. When you have decided to do so, you should also define an evaluation plan to judge the method application in the end: Did it provide the benefits you expected from it? Were all the tools scalable for your use case? Did the tools provide enough usability? Did you provide good-enough third-party tools? What could you improve on the next iteration of the method? Did you properly identify the context of the method application or do you need to readjust this for the next iteration? In addition, you should track the time needed for each of the method steps and compare it to your expectations. In case of misalignments, identify the causes and try to improve it.

To summarize, there are several questions to consider and answer by taking your context into account. It helps to have an idea about the CloudScale method (as provided by the pilot project; cf. Sect. 8.5). It also helps to interview your technical manager or developers about the current status of the system and the most important known issues. Also, you should have clear plans where you want to go with the system, i.e., when migrating it from legacy to cloud platforms. In particular, you should know the resulting requirements based on the new or extended business models behind the newly developed system features or quality properties.

8.3 Relation to Other Engineering Methods

The CloudScale method will almost, in any case, not be the sole engineering method needed for the development of your cloud application. The CloudScale method will rather be complementing your normal development processes and methods. In that way, the CloudScale method complements classical development processes like SCRUM or the rational unified process (RUP). When looking at such processes, the CloudScale method complements several traditional steps with enhanced or additional activities and tools.

When looking at the commonalities of all engineering processes, we can identify the following *core ingredients* to engineer cloud systems:

Foundations This covers the fundamental basics, underlying concepts, guiding principles, and taxonomies of the used cloud concepts, terms, and technologies.

Implementation This covers the building blocks and practices used to realize cloud applications.

Lifecycle This aspect covers the lifecycle model of the cloud application. It typically describes end-to-end iterations of a particular cloud development and operation.

Management This aspect addresses the management questions arising when realizing a cloud application. It should cover both design time and runtime cloud management. Each management aspect should be tackled from multiple perspectives.

Our CloudScale method and its tools partly cover all these cloud engineering elements. This method contributes to the foundations of cloud concepts, terms, and technologies with scalability guidelines and a library of appropriate Architectural Templates (ATs). It furthermore provides definitions of important concepts like scalability, elasticity, and cost-efficiency. In the implementation phase, the CloudScale method provides a plethora of analyses and guiding principles that software architects can apply to systematically address scalability, elasticity, or cost-efficiency issues. CloudScale's support is more limited toward the lifecycle ingredient. However, it can cover elements related to design via models, or to quality assurance and evolution support via its Spotters. For the management ingredient in particular, we have included lots of experiences, key considerations, a pilot project, etc. in this chapter.

However, other methods and tools also cover aspects related to cloud systems' realization. For many domains, there are standard supporting engineering principles. In the cloud environment, is important to emphasize that used methods and tools have to enable the process of designing the system in a way that it leverages the power and economics of cloud resources to solve a (scalable) business problem. As cloud applications is still a growing market, it is taking a serious commercial track and there are a lot of supporting tools and ready-to-use documented HowTos, for example, specialized cloud patterns. The main focus of current offerings on the market is tools for price/cost calculation of the required IaaS resources in a particular cloud environment like Amazon EC2. Often, these offerings are much simpler than the CloudScale method, but also much more inaccurate. In addition, they are often in-transparent to the customer; i.e., the customer cannot verify whether the approach computes correct costs and whether it suggests the best IaaS provider (in contrast to suggesting IaaS providers based on hidden contractual relationships).

To summarize, there are two different relations of the CloudScale method to other engineering methods. First, it has to be integrated and mixed with classical software engineering methods like SCRUM or RUP. This integration will be organization and project specific. The CloudScale method provides method steps and tools which have been tailored to be reusable building blocks, which can be put on top of other engineering methods. Second, there are alternative or competing cloud engineering methods and tools. In particular, there are several commercial offers today on the

market. For these, you have to judge the extent to which they can complement the CloudScale method or they are in conflict with the CloudScale method. You also need to identify their real contributions in one of the above listed core ingredients. In addition, you need to judge whether they provide open and objective or biased information.

8.4 Pros and Cons of the CloudScale Method

Introducing any new step in the organizational system development process always poses a question of the benefit that the organization can gain from the newly added step (novel part of the process). There is no method that can provide gains exclusively, without any expense, because there is always effort and cost that must be invested into achieving advantages and potential profit. This is the main reason why in this chapter, we analyze the parameters that are important for making the decision on how useful the employment of the CloudScale method and its tools is. These analyses are the foundation for identifying organizational pros and cons of using CloudScale method and tools. The organization that wants to implement the CloudScale method should be prepared to actually calculate the benefits of the needed investment for the additional quality provided, in contrast to potential losses due to potential system problems and costs to maintain or reengineer bad system components in operation, or, even worse, with system failure or delays during operation.

8.4.1 Critical Success Factors for Method Adoption and Use

Successful adoption and use of the CloudScale method and tools relies on several conditions in terms of requirements.

- Having clearly instantiated requirements in order to identify particular system scalability, elasticity, and cost-efficiency needs, as well as having established system behavior scalability conditions.
- Having use cases related to system usage, which enables extracting usage patterns and correlating them with scalability requirements (primarily system work and load expected through the system lifecycle, with a focus on the behavior in the limits).
- Freedom to change currently implemented architectural patterns. Some organizations use general architectural patterns, or patterns adapted in a custom way, which in most cases need serious reengineering in order to fulfill the new system requirements. This usually requires thoughtful architectural discussions and decisions.
- An organizational development process: Many organizations use a plethora of available processes and methods, and as such, they are usually less limited and

controlled in adopting new ones. In contrast, there are big development organizations that follow standardized general-purpose or adapted/tailored development processes in a much stricter way. In such cases, it is usually required to elaborate benefits for introducing some new process or changing the steps in their own established development process environment.

- Team experience in dealing with system performance issues: This is usually also related to the selected development method and the possibility to solve overall system behavior issues or to focus on a particular system problem. Another question that needs to be considered is how much freedom does the development team have to experiment and prototype the system before they start to produce a solution.
- If the solution for the new system, or a part of it, is produced by reusing an existing code base, a good understanding of the source code that has been reused in the new system is of great importance. Similarly, a solid knowledge about the system is required when a solution for the problems with a system in operation needs to be found.

  In case of applying the CloudScale method and its tools on a system that reuses an existing code base, it is very important to have the reused source code available for analysis. If the system is built from scratch, it is crucial that your organization is willing to follow the concept of model-based development. Also, an appropriate person that can create a model of the analyzed system has to be assigned to achieve an adequate accuracy in the prediction of the potential system behavior.
- Documentation of the newly introduced method and tools: Depending on the extent of the existing documentation, users should be prepared to invest certain effort into researching and investigating new tools and their features, since they are likely not documented to the same level as mature tools.
- The tool's ability to deal with a large existing code base.
- Potential of the tools to work with so-called *grey-box* components and *grey-box* models, meaning that the capability of creating models and conducting analysis does not depend on having the entire code available; instead, it should be possible to just describe the component behavior. This can be very important for systems which access external services without available source code.
- Usage of existing frameworks as a base for developing or encapsulating new services: In this case, an architect is not interested in the analysis of existing frameworks, but only in analyzing the interaction with a custom-developed component.

If the provided method and its tools can fulfill the requirements for their implementation in the development process of an organization, it is possible to predict successful usage and user's satisfaction. In the case of the CloudScale method and its tools, fulfillment of stated requirements is the foundation for their application on the system under analysis.

8.4.2 Organizational Issues

The development of new systems and services is a team effort in any organization. The development process starts with the collection of requirements and defining the expected development scope. The product manager, together with the customer (e.g., in cloud-based system, it is most likely a service provider), formulates both functional and extra-functional requirements and determine the main focus of the cloud system's scalability requirements. Based on the collected requirements and the expected system behavior, the system architect makes early decisions on the overall system architecture. This is the foundation for the organizational decision which parts of the expected final architecture can be reused from previously developed systems, and what new services need to be developed. This is where the first challenges may arise because the combination of certain existing components/services can look promising, and the decision needs to be made quickly about the way the system will be realized. In such situations, it is crucial to determine whether the proposed realization of the system is feasible, or at all possible, and how it will impact the expected system behavior. Also, it is important to consider what can happen to the scalability requirements when composing services according to the selected architecture. If an organization skips this step, which aims at identifying the appropriate solution and making the right decisions about the system early on, and immediately jumps to service implementation, it can lead to costly issues found during the testing phase, or even worse, during the system's operational phase. Managing scalability requirements from the beginning of the system development requires full understanding of the scalability concepts and usage of all needed tools and methods that can support architects and developers to make fast selections of good solutions. In terms of organizational preconditions, achieving such a way of working can sometimes be challenging due to the complexity of the decision-making process in large companies. This is due to heterogeneous teams participating in the development process using the method and tools in different system lifecycle phases.

8.4.3 Costs

Using almost any kind of method, and especially introducing a new one in the development process, results in certain costs due to the time and effort invested in performing specific method steps. The costs involved when using the CloudScale method can be structured as follows:

New process steps The CloudScale method introduces process steps that will be new compared to the current methods employed in the organization's development process. Performing each of those steps has costs in terms of human effort.

Employee training Training of the involved personnel requires teaching them about the conceptual aspects of the CloudScale method (e.g., cloud computing

paradigm, SLOs, scalability concepts, etc.), as well as training them in the use of the CloudScale method and its tools. From our experience, the entrance barrier is somewhat lower for CloudScale's migration and evolution support, as it is more closely oriented toward analyzing the system's implementation. Training personnel in model-based analysis typically requires certain time, since most developers and software architects working in practice are unfamiliar with such approaches. Hence, they need to learn and understand modeling techniques and languages, as well as identifying the right model abstractions. In addition, CloudScale's tools require some familiarity with stochastic theory, which is a knowledge that in practice often needs to be refreshed, especially in an industrial environment. We estimate that training these skills on a basic level requires a 2- or 3-day workshop.

New organizational processes The organization itself must also be adapted to the CloudScale method. New ways of collaboration may be required due to the different roles involved in conducting the CloudScale method.

8.4.4 Covering the Cost of the CloudScale Method Adoption

As mentioned in the previous section, the CloudScale method will inevitably produce a certain cost, and this expense should be offset by savings in the overall system lifecycle costs. In that way, adoption of the CloudScale method will become profitable, and is able to benefit the organization on the long run. We identify four ways of reducing the overall lifecycle costs when using the CloudScale method:

Better quality When a service suffers from insufficient scalability, their users become dissatisfied and leave. Ultimately, in addition to losing customers (and in severe cases the organization's reputation), additional money may be lost by investing in the system's maintenance and paying for SLO violation penalties. A controlled system design from the beginning of the product lifecycle will result in better system quality, thus reducing the potential cost of complex and expensive problem-solving activities in the operational system phase.

Less redesign It takes an additional engineering effort to fix a service with poor scalability. Other than just introducing the cost of providing the problem solution, your engineering team will not be able to build new functionality due to their involvement in fixing the problem. Additionally, redesigning a service takes time, during which the original services will suffer from an inadequate scalability.

Less gold plating A development team which is eager to avoid scalability problems may use too much engineering effort when designing parts of a service. As a result, development costs will be higher, but since a scalable service may sometimes also be more complex, maintenance costs may also increase. A better overall view of the system's scalability requirements may reduce gold plating, so engineering efforts are used only where required.

Lower operating costs Services with poor efficiency will require a considerable amount of costly cloud computing resources. A system with poor elasticity will also be challenging to operate and will therefore require more manual tuning, thus requiring more engineering man-hours.

Having both costs and benefits in mind, development organizations should perform serious analysis before they decide to start using the CloudScale method and its tools so that they can understand the potential offset of their investment by savings in the overall system's lifecycle expenses.

8.4.5 Risks

We have identified the following risks related to using the CloudScale method:

Too high effort The effort involved in using the CloudScale method may simply be higher than what is possible to argue in an organization. This risk can be reduced if the CloudScale method is used on smaller projects first so that the organization gets a better grip of the actual costs involved in using the CloudScale method. As the organization gets more familiar with the CloudScale method and its tools, the size of the projects may gradually be increased.

Lack of accuracy It is very important to have an overall idea of the required level of accuracy when using the CloudScale method. To a large extent, the accuracy depends on the quality of the parameters used.

Based on our experience in applying the CloudScale method and its tools on differently sized systems and in different organizational environments (small, medium, and large companies), with careful selection of the project and a well-balanced granularity of the system, the mentioned risks can be mitigated and properly addressed.

8.4.6 Critical Factors for Successful Projects

Whenever an organization tries to apply a new or improved method in its development process, it is important to understand the potential implementation environment. The three essential aspects of the targeted environment are: the development team that will produce the first pilot, the system selected for the experimentation/piloting, and the time available for learning new concepts and making experimental developments. Based on these presumptions, we can define a list of critical success factors for the project that you should consider when applying the CloudScale method:

- Modeling experience—assign a development team which has experience with abstract concepts and system performance modeling.

- Time for learning the methods and tools—the team must have enough time to learn the concepts implemented by the tools and to be able to map the problem to the adequate tool.
- Compatibility with the current development environment—many development organizations already use standard (or customized) development frameworks that encapsulate tools and methods, and it is very important to have sufficient time to adapt to the new environment and have enough freedom to correlate and include the outputs of the new method in their existing environment.
- Availability of the source code for the system analysis—especially in the case of system composition, it is important to have the system's source code as well as any documented problems with the currently used frameworks.
- Organization of the existing code—currently used frameworks and applied development conventions should reflect the current state of the art.

8.5 A Pilot Project

After highlighting the considerations to be made by the project manager, in this section, we will introduce a very simple example system which can be used by technical managers or senior developers to get started with the CloudScale method. Its purpose is to help them to get an impression of what the CloudScale method is, what it provides, and how complex it is to get started, and to assess whether it is worth to consider using it in their own development projects. This *pilot* project is also part of the CloudScale integrated development environment (IDE), as an example ready to be used out of the box so that it can be understood and inspected in all aspects.

The pilot system is a simple client/server application implemented in Java. The server generates HTTP responses under pre-defined URLs (in a REST-like style) and the client issues corresponding HTTP requests. The client can be scaled up so that it generates a significant amount of load on the server. The two main operations provided by the server are (a) a computation of a dynamic number of Fibonacci numbers, and (b) a simple query operation, which, however, contains a synchronized block, i.e., can be executed only mutually exclusive. Figure 8.1 shows an excerpt of the code for the query operation where the `call` method of the `OLB` singleton contains the synchronized block.

```
1  @GET
2  @Path("testOLB")
3  @Produces(MediaType.APPLICATION_JSON)
4  public String testOLB() {
5          OLB.getInstance().call();
6          return "Hello␣from␣OLB␣Test␣Method!";
7  }
```

Fig. 8.1 Source code of the server-side example query method in the pilot project

```
1  public final class OLB {
2  [..]
3      /**
4       * Method leading to a One Lane Bridge.
5       */
6      public synchronized void call() {
7          try {
8              Thread.sleep(TIME_TO_SLEEP);
9          } catch (final InterruptedException e) {}
10     }
11 }
```

Fig. 8.2 Source code of the call() method in the OLB singleton

At runtime, the client can execute this method by issuing an HTTP request to the server for the URL `http://<servername>/testOLB`. The idea of the two methods of the pilot project is that one method is rather resource intensive (fibonacci), while the other represents an One-Lane Bridge (OLB) scalability HowNotTo (cf. Sect. 2.10; code shown in Fig. 8.2). Both methods are easy to understand in a few seconds and illustrate both common problems of web services and the capabilities of the CloudScale method and its tool support.

The pilot project can be used to showcase the CloudScale method. It covers both use cases of the CloudScale method. Software developers can easily create a model of the system. This can be done manually using the editors, as the system's complexity is really low, or by using the Extractor tool, as the system is provided as object-oriented Java code (i.e., it fulfills the Extractor's prerequisites). Using the model, software developers can analyze questions like:

- What is the response time of the system under varying load?
- What is the system's capacity?
- How does the system scale up or down during elastic adaptations?
- etc.

On the other hand, the pilot project can also be used in the second CloudScale method use case, the detection of HowNotTos. This can be done statically (again, the necessary prerequisite is that a model can be extracted by the Extractor) or dynamically by executing the pilot project. The project's client and server come with the needed stand-alone web server and client libraries so that they should run as soon as a Java virtual machine (JVM) can be found in the target environment. When executed, the CloudScale's Dynamic Spotter can detect the HowNotTos which have been injected into the example (i.e., the OLB or the excessive CPU consumption HowNotTos). To get things up and running here also with little effort, the Dynamic Spotter provides a custom-tailored load generator for the pilot project which can be used with minimum configuration effort.

Overall, this pilot should give project and technical managers a good idea of the benefits and considerations when wondering whether to try the CloudScale method.

We will also come back to the pilot project when we give a more detailed high-level walk-through through the CloudScale method based on the pilot in order to highlight the management decisions needed when executing the CloudScale method in Sect. 8.8 (in contrast to the technical aspects of the method discussed in the previous chapters).

8.6 Setting Up the CloudScale Environment

When the decision has been made to use or try the CloudScale method, the first thing you want to do is to learn the CloudScale method and to install its tool support in form of the CloudScale Environment. CloudScale's tools are altogether integrated in the so-called **CloudScale Environment**. This integration makes it easy to use all tools from a coherent user interface providing a unified user experience. In addition, the CloudScale Environment supports you when following the CloudScale method: It provides a build-in CloudScale method view, which you can use as a workflow engine to track your progress in the CloudScale method. The CloudScale Environment can freely be downloaded at CloudScale's web page [1].

It is available for all major platforms. It requires a JVM on the developer machine, as it is based on the well-known Eclipse IDE, which is implemented in Java. There is no need to install it; downloading and extracting the provided files should be enough; i.e., it is easy to try it out without polluting your development environment too much.

After starting the CloudScale Environment for the first time, we suggest to get used to it using the provided pilot project. It can be found under `Examples` as the `Minimum Example`. Using the pilot, you can go through all CloudScale tools. They can be accessed using the most common parameters only from the CloudScale perspective. However, in case you need to fine-tune a single tool or access advanced features, the CloudScale Environment also provides dedicated perspectives which provide access to the advanced features and settings of each tool.

Overall, the CloudScale Environment provides the following tools and guides software architects in the order in which they have to be optimally applied:

ScaleDL editors The CloudScale Environment provides several editors to create and modify Scalability Description Language (ScaleDL) models. CloudScale's catalog of ATs is integrated in these editors, allowing software architects to efficiently create even complex models.

Extractor The Extractor is a reverse engineering tool for automatic model extraction. It parses source code and generates partial ScaleDL models that can be further used by the Analyzer and the Spotter.

Analyzer The Analyzer allows to analyze ScaleDL models regarding scalability, elasticity, and cost-efficiency of cloud computing applications at design time. For these capabilities, CloudScale integrated novel metrics for such properties into the Analyzer. Analyses are based on analytical solvers and/or simulations. The

Analyzer particularly supports to analyze self-adaptive systems, e.g., systems that can dynamically scale out and in.

Spotter The Spotter allows to statically and dynamically detect scalability issues. For a static detection, the Spotter automatically detects search patterns on ScaleDL instances created by the Extractor. Found patterns are interpreted as potential scalability anti-patterns. All scalability anti-patterns are defined in a pre-defined but extensible pattern catalog.
For a dynamic detection, the Spotter provides a framework for measurement-based, automatic detection of software performance problems in Java-based enterprise software systems. The framework combines the concepts of software performance anti-patterns with systematic experimentation.

Distributed JMeter [2] Distributed JMeter is a workload generator application. When estimating resource demands, e.g., to parametrize ScaleDL models, such a workload generation is needed. Distributed JMeter can be deployed on Amazon web services (AWS) or OpenStack.

To get started, go through each tool and use it on the provided pilot project!

8.7 Complementing Tools

The engineering of cloud computing applications cannot be completely automated; however, appropriate tools can help software engineers in becoming more efficient. For the CloudScale method, CloudScale's Environment is a good choice because it is tailored for an efficient use within this method.

The CloudScale method is not tied to the official CloudScale tools: other tools can substitute or complement the official ones. This independence has the benefit that there is no vendor lock-in, and that tools that are already in use can be retained.

The following list of tools exemplifies their integration into the CloudScale method. The list is not exhaustive and should only serve for exemplification and initial pointers.

Palladio [3] Palladio is the open-source software architecture simulator that underlies CloudScale's Analyzer. Palladio can be used in stand-alone mode, without depending on the CloudScale Environment.

Apache JMeter [4] Apache JMeter is an open-source application to generate workload and measure performance metrics. It was the basis for CloudScale's Distributed JMeter, but can also be used in stand-alone mode.

Kieker [5] Kieker is an open-source performance monitoring framework. Kieker allows to instrument source code with probes which gather response time measurements at runtime. Response time measurements are especially important for estimating resource demands, e.g., to parametrize ScaleDL models. CloudScale does not provide a dedicated tool for this purpose; CloudScale's evaluations also used Kieker.

Dynatrace Application Monitoring [6] Dynatrace Application Monitoring is a commercial alternative to Kieker with advanced features that ease performance

problem detections. The tool therefore may replace dynamic detection parts of CloudScale's Spotter.

JProfiler [7] JProfiler is a commercial application to profile Java applications. Profiling helps in understanding performance characteristics of applications, which finally allows to identify critical use cases.

Java VisualVM [8] Java VisualVM is a freely available profiling tool for Java applications. The tool comes together with Oracle's JVM and therefore provides a good alternative to JProfiler.

R [9] R is a freely available programming language and environment for statistical assessment of data. With R, the data from performance monitoring tools can be appropriately prepared to be used as resource demand estimates.

8.8 Following the CloudScale Method for the Pilot Project

This section will provide a walk-through through the CloudScale method based on the provided pilot project. As such, it brings together the lessons learnt from the previous sections on the pilot and the tools, and shows key considerations and management decisions which are useful for the small example system.

When starting with the CloudScale method, the first step is to elicit SLOs and derive critical use cases and key scenarios from them (cf. Sect. 5.5). For the pilot project, we assume in the following that it has a response time SLO which says that both provided operations should react in acceptable times for Internet users. This can be considered to be 2 s. Therefore, we define SLO violations to happen if the mean response times over 10 s intervals exceeds those of 2 s. As critical use case, we consider a mix of equally calling both the OLB and the Fibonacci service. A key scenario is one where 100 concurrent users access the system hosted on a single machine for this. The latter can also serve as technical requirements for scalability in this simplified example.

In the CloudScale method, we have to consider then whether we want to model the system or whether we want to base analyses on code. As we have no real idea of the properties of the pilot project, we consider using the code as the main basis for the first method iteration. An additional aspect which suggests using code-based analysis is that the code is provided in Java and is well-structured.

After making this decision, we have to run the Spotter on the pilot project. In a first step, the Static Spotter extracts a model from the code using the Extractor and, when configured correctly (i.e., using the right granularity, which is a low one for this simple project), spots the OLB method already and reports it as a potential scalability HowNotTo. In the second step, we run the pilot project using the provided built-in server and use the Dynamic Spotter on it, together with the provided load generator. We set the number of concurrent users to 100 and the SLO violation threshold to 2 s. After running the Dynamic Spotter, we analyze the results and find that the OLB method—due to its limited concurrency—violates our technical requirements.

Hence, we can rework the pilot project. For example, we could assume that the synchronized keyword is a mistake made when implementing the method. As a simple fix, we can remove it and then rerun the analysis. Now, everything should be fine, as one server is usually strong enough to deal with 100 users.

This concludes the CloudScale method for the pilot project. However, we could for example go ahead and add another service to the pilot and use a model to analyze this evolved system. We would extend the critical use case and the key scenario to cover also the new method. Using the Extractor and manual model annotations would yield a model to be analyzed. Using this model, we could then check for example what the system's capacity is for the 2 s response time SLO.

8.9 Conclusion

In this chapter we discussed the CloudScale method from the perspective of managers considering to introduce the CloudScale method into their development processes. Introducing a new method in the current development process established within an organization will inevitably introduce expenses as well. For this reason, the decision on implementing a new method has to be based on a carefully performed analysis of potential benefits and novel costs implied by the integration of the novel process steps, which require effort and time. This chapter presents a practical point of view regarding the adoption and usage of the CloudScale method and its tools. We identify potential issues, costs, and risks of the CloudScale method adoption, as well as how those risks can be mitigated, and what are the factors determining the success of both the introduction of the new method and the project it will be applied on. One of the main challenges related to the CloudScale method is the time it takes to get familiar with the concepts it brings to its potential users (i.e., system performance modeling, etc.) and the readiness of the organization to invest in the proper integration of such methods in their currently established processes and environments. The whole process of following the CloudScale method is explained on a simple pilot so that the reader can promptly get a grip on the method steps. We consider and present different aspects of the CloudScale method adoption from the perspective of development teams in organizations that are already using various development processes and frameworks.

Despite all considerations, the CloudScale method provides major benefits to the development of scalable, elastic, and cost-efficient cloud computing applications. It offers benefits to software architects to make the right engineering decisions, but also to managers as to how to avoid risks of implementing systems which will fail in the end to meet their scalability, elasticity, or cost-efficiency requirements.

The overall objective for further work on the CloudScale method is to introduce improvements, such as simplifying the use of the tools (i.e., users could be presented with fewer choices, and the manual modeling effort could be reduced), and potentially further reducing the need for new engineering knowledge. This also requires more studies of how scalability, elasticity, and cost-efficiency are actually

handled in organizations today and how it could be handled more effectively in the future. With such additional studies, we can also provide more guidance with respect to choosing the right granularity level or planning for the amount of manual effort needed. Additional improvements such as more ATs or scalability anti-patterns can drastically reduce the modeling effort and increase tools' usability and, therefore, increase the return on investment. In this way, the CloudScale method may become a part of standard practice in contemporary software development, especially for projects where scalability, elasticity, and cost-efficiency requirements are major threats to a project's success.

References

1. CloudScale: The CloudScale Environment (2016). http://www.cloudscale-project.eu/results/tools/ [Visited on 12/19/2016]
2. CloudScale: Distributed JMeter (2016). http://github.com/CloudScale-Project/Distributed-Jmeter [Visited on 06/20/2016]
3. Palladio: The software architecture simulator (2016). http://www.palladio-simulator.com [Visited on 06/20/2016]
4. Apache JMeter: (2016). http://jmeter.apache.org [Visited on 06/20/2016]
5. Kieker: (2016) http://kieker-monitoring.net [Visited on 06/20/2016]
6. Dynatrace Application Monitoring: (2016). http://www.dynatrace.com/en/application-monitoring/ [Visited on 06/20/2016]
7. JProfiler: (2016). https://www.ej-technologies.com/products/jprofiler/overview.html [Visited on 06/20/2016]
8. Java VisualVM: Java Virtual Machine Monitoring, Troubleshooting, and Profiling Tool (2016). https://visualvm.github.io/ [Visited on 06/20/2016]
9. R: The R Project for Statistical Computing (2016). https://www.r-project.org [Visited on 06/20/2016]

Chapter 9
Case Studies

Darko Huljenić, Ivana Stupar, and Mariano Cecowski

Abstract As described in the previous chapters, the CloudScale method lets software architects manage scalability, elasticity, and cost-efficiency throughout the whole system lifecycle. To illustrate its usage, this chapter describes an application of the CloudScale method and its tools on two industrial case studies. The first case study—Ericsson's electronic health record (EHR)—is an electronic health record software, and the second one—Kantega's Flyt CMS—is a content management system (CMS). The case studies exemplify scenarios that require an engineering approach to ensure the system is scalable and performs well. The providers of the stated industrial cases are different organizations that follow their own internal development processes. The chapter closes by briefly pointing to complementing case studies that were used to evaluate the CloudScale method, both separately and in the context of the CloudScale method.

The following sections cover the essential steps and tools in the CloudScale method for the EHR (Sect. 9.1) and Flyt CMS (Sect. 9.2) case studies. Subsequently, the complementing case studies are overviewed (Sect. 9.3).

9.1 Case Study: Electronic Health Record

The first case study inspects an electronic system for managing health records that was developed by Ericsson. The system, called electronic health record (EHR), is an example of a service for which it is crucial to achieve scalability due to its functionality and a large user base. In order to ensure EHR's scalability, the migration of the EHR system to the cloud was considered as a solution. The cloud-based EHR could significantly decrease infrastructure over-provisioning due to the elastic properties of the cloud resources. However, in order to successfully migrate EHR to the cloud, it was necessary to identify potential scalability bottlenecks and

D. Huljenić • I. Stupar (✉)
Ericsson Nikola Tesla, Krapinska 45, 10000 Zagreb, Croatia
e-mail: darko.huljenic@ericsson.com; ivana.stupar@ericsson.com

M. Cecowski
XLAB d.o.o., Pot za Brdom 100, 1000 Ljubljana, Slovenia
e-mail: mariano.cecowski@xlab.si

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7_9

problems in existing code bases. CloudScale tools and method were used to assist in evaluating possible problems and solutions, as well as to predict the cloud-based system behavior. The focus of the EHR case study are actions of the CloudScale method that require software architects to apply the tools Extractor, Analyzer, and Static Spotter.

9.1.1 Electronic Health Record

EHR provides digitally stored patient health information, supporting care, education, and research. It stores information on medications, past medical history, immunizations, laboratory data, radiology reports, etc. The EHR system needs to share data with healthcare providers, insurance institutions, government agencies, and patients, making it a data-centric system with a large user base. As such, it is crucial that the EHR system is adequately provisioned, and that the EHR services delivered to the end users fulfill the service-level objectives (SLOs). The traditional non-cloud version of EHR is implemented as a web-based enterprise system.

EHR is one of the main components of the Ericsson Healthcare Exchange (EHE) [1] platform. The EHE platform is a part of Ericsson's healthcare portfolio, deployed on a national level in Croatia. Several platform services are integrated in many healthcare provider institutions, resulting in a large user base. For example, the e-Prescription service is integrated in more than 2300 general practitioner offices, over 2500 dentist offices, 192 pediatrician offices, and more than 1100 pharmacies.

The traditional on-premises version of the EHR system is centralized and consists of an EHR database and a service layer developed above the database. Database mechanisms such as database clusters, connection pools, and database replication ensure system scalability and responsiveness up to a certain level, which can be guaranteed by extensive, but limited, infrastructure resources. The infrastructure used for the non-cloud EHR deployment has to be dimensioned according to the highest demands in order to be able to handle workload peaks, which is achieved by over-provisioning. The resource demand per user is non-linear.

A high-level view of the EHR platform architecture can be seen in Fig. 9.1. The core of the EHR platform provides basic functions such as workflow management, authorization and authentication, error handling, data model, messaging, and reporting support. The access to the EHR services is enabled by service APIs. The EHR platform also provides an API so the external services can use the EHR core operations. Further details on the EHR architecture and implementation can be found in [2].

A cloud-based EHR solution is beneficial because of the scalability and elasticity offered by the cloud environment, but also because of cost minimization. A cloud solution also facilitates the provisioning of healthcare products and services to patients located in remote areas and to patients with limited access to quality medical services. However, in the process of migrating the EHR platform to the cloud, it was hard to estimate the optimal service provisioning. Hence, in the

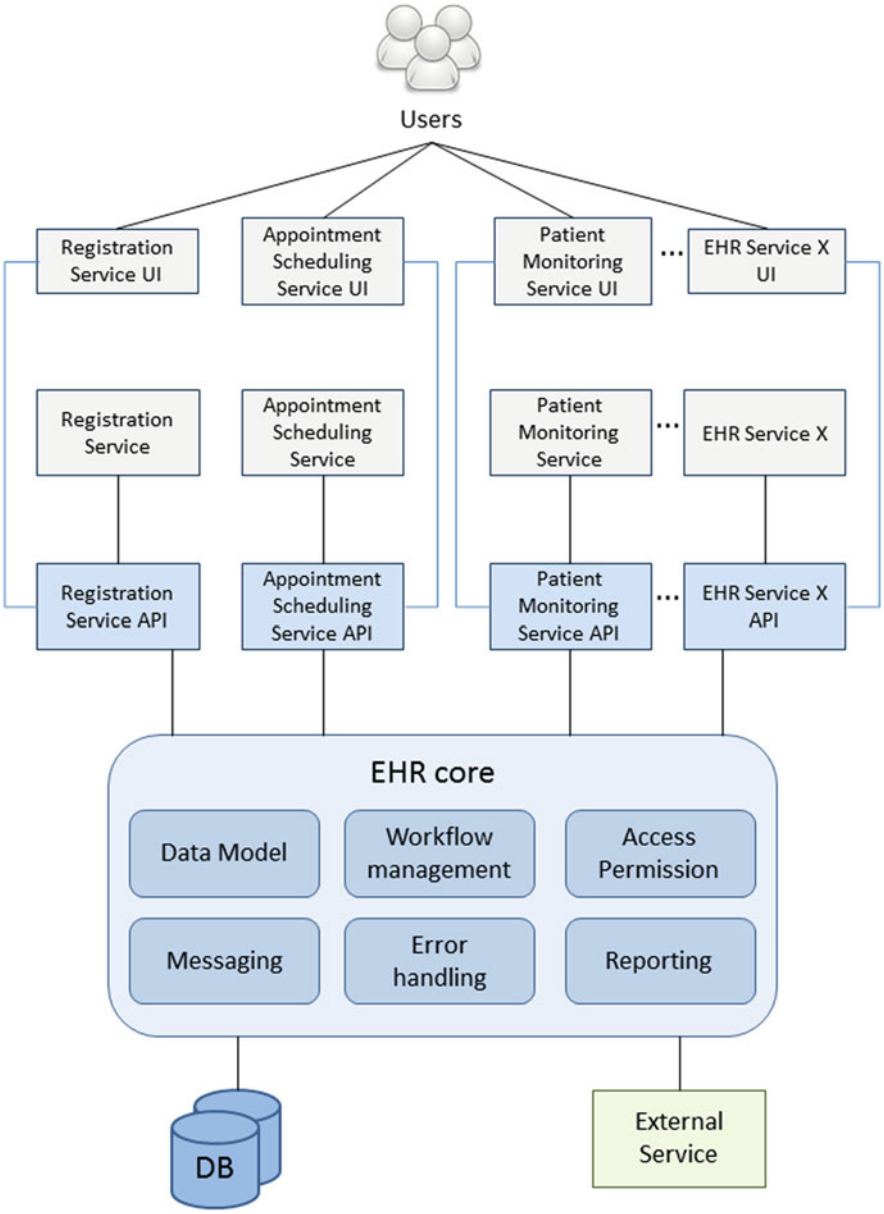

Fig. 9.1 EHR system architecture

case of EHR migration, the CloudScale method and tools were used to evaluate system scalability and potential issues, and to provision an optimal amount of cloud resources for a given workload in a cost-effective way.

There are several scenarios in the EHR service implementation that will potentially affect system scalability. One of them is the change of the EHR service load, including possible growth of the service user base. The number of EHR users might grow due to the new services, integration of additional institutions, or deployment of the EHR solution in countries with lager population. Also, for specific annual events (e.g., regular flu) and not-so-frequent events that result in increased patient number (e.g., bird flu, large earthquakes, or tsunami), there is a need for an elastic system scale-up. However, the system needs to scale down as the workload decreases to reduce operational cost. Constant system resource usage monitoring is necessary to be able to determine when and how to adjust the amount of resources allocated to the service. The work variation of the EHR service is mainly linked to the size of the health records, which can grow unpredictably. Some of the medical data records can be very large, due to treatment images, video files, sensor data for remote patient monitoring, etc. Additional consideration includes the change of resources and infrastructure price, which will affect the service cost. Changes in the primary healthcare process (e.g., regulatory changes) can produce changes in functional implementation of existing EHR services. Such functional changes can have an impact on the work or load of the EHR service, as well as on the addition of the new service offered by the EHR platform. All of the mentioned scenarios can affect system scalability and the fulfillment of system SLOs. Modeling the new services and predicting scalability impact on the existing system could in such cases be performed using CloudScale tools. The actual analysis performed on the EHR service is described in the next section, with a focus on finding optimal service provisioning and current scalability issues.

9.1.2 Applying the CloudScale Method and Tools to Electronic Health Record

The focus of the EHR case study was to apply the CloudScale method in the context of engineering the migration of the EHR solution from traditional on-premises infrastructure to cloud infrastructure. The CloudScale method was used for evaluating what is the predicted system scalability under specific workload, and if the design and architecture of the EHR system are ready to be used for a cloud-based solution. Since the CloudScale method is supported through the CloudScale Environment, which integrates all of the CloudScale tools, the method steps were easy to follow.

As one of the initial steps of the CloudScale method, software architects need to make a decision on creating the model of the EHR system manually or by using reverse engineering to extract the model out of the EHR source code. Since the EHR service was already implemented, the Extractor tool was used to extract parts of the model from the EHR system's source code in order to shorten the model preparation time. The EHR system is written in Java and consists of several modules

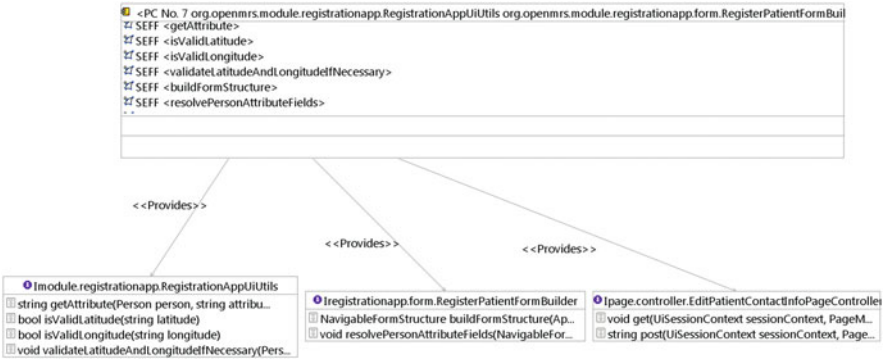

Fig. 9.2 Repository model diagram of an EHR module component generated by the Extractor tool after layout rearrangement

organized in projects containing the source code. The system and repository models were generated for each of the projects that were used as an input for the Extractor tool. An example of the extracted repository model can be seen in Fig. 9.2.

The initial layout of the diagram components had to be rearranged, and additional manual improvements have been introduced. As the main effort in improving the model obtained by reverse engineering, we identified the creation of service effect specifications (SEFFs) for each model component. Users should have a good understanding of the metrics affecting the process of clustering and merging components in the model. Obtaining the wanted level of granularity requires an iterative approach in finding the best extraction configuration. Also, since the Extractor input is a Java project, the granularity of the extracted model was determined by the organization and structure of the source code. To adjust the level of granularity to one that should be convenient for further analysis and proceeding with the CloudScale method, the whole model of the EHR system was created by combining the manually created model components (mostly related to the EHR platform core) with the parts of the model generated using the Extractor tool (EHR services). Figure 9.3 conveys an impression of the static structure of the resulting model.

Following the CloudScale method, the source code was analyzed using the CloudScale Static Spotter tool with the purpose of finding anti-patterns in current implementation of the EHR system, and to see if the design has to be adjusted for the cloud-based EHR. The tool was used on a set of 22 projects containing the source code of the EHR system core and services. During the Static Spotter analysis, we focused on finding scalability anti-patterns. After running the static code analysis, the Static Spotter detected two EHR modules that used synchronized methods in the source code, possibly resulting in scalability issues due to the One-Lane Bridge (OLB) anti-pattern. Source code inspection confirmed that these were the only projects from the input set using the synchronized methods. It is, however, important to note that the detection of the anti-pattern was also affected by the

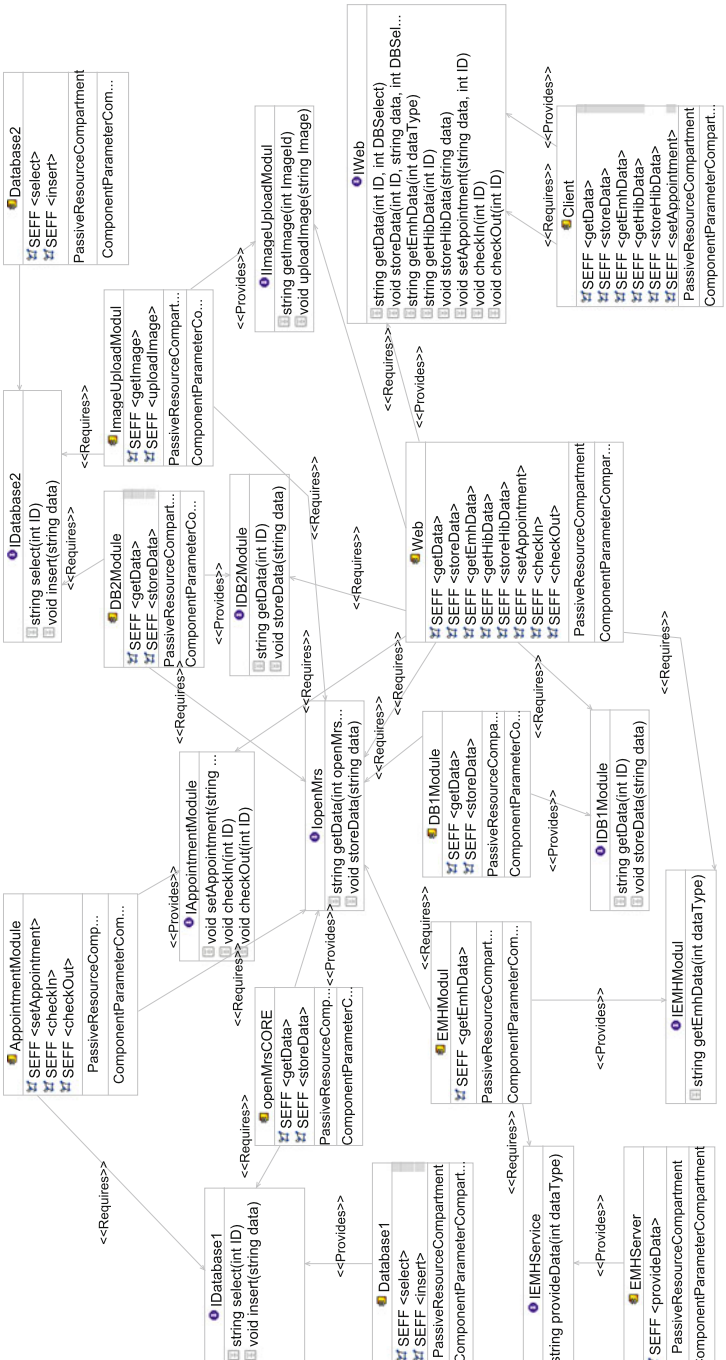

Fig. 9.3 Model of the EHR system used in the Analyzer tool

configuration parameters provided before the start of the analysis. Since a certain amount of details from the source code is lost during the process of merging and clustering, it is to be expected that the information about the possible anti-pattern implementation might be lost as well. If the parameters used to generate the model from the source code are set to create merged components, parts of information from the source code might not be preserved in the model and the Static Spotter might not be able to detect anti-patterns, although they might exists in the source code.

As we proceeded with the analysis, we evaluated the forward engineering scenario using the previously created model of the EHR system. The aim was to find potential scalability bottlenecks with CloudScale's Analyzer tool. In order to validate results provided by the CloudScale Analyzer tool against the measurements conducted in the cloud environment, a test-bed was prepared for deploying the EHR system. The private IaaS cloud used for EHR service deployment was deployed using the OpenStack software platform. The test environment consisted of three hosts, as shown in Fig. 9.4, employed for the roles of controller, compute, and storage nodes.

Both EHR application and EHR database were deployed as virtual machine instances. User requests were simulated using a load generator. In order to evaluate

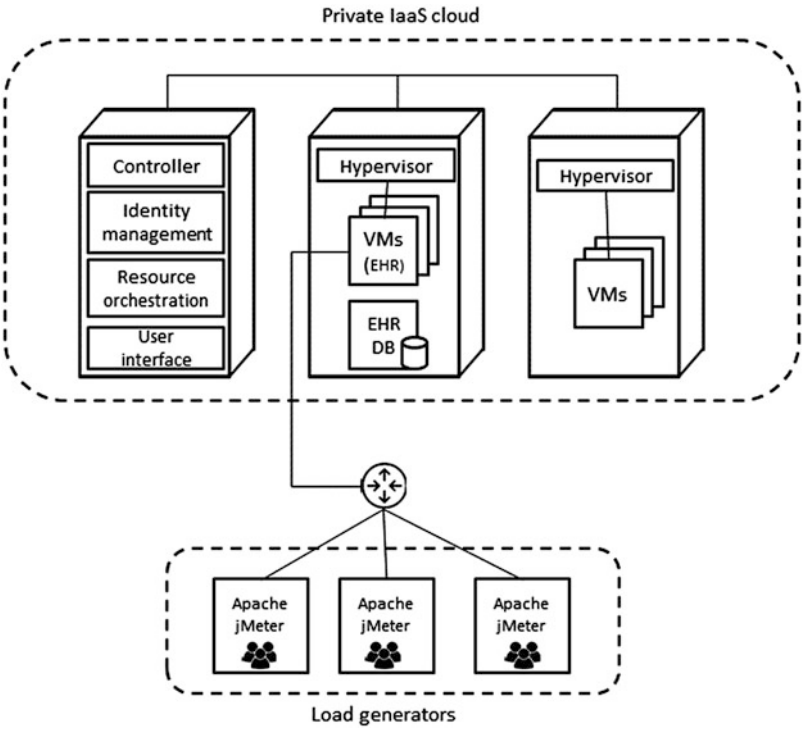

Fig. 9.4 EHR cloud deployment validation environment

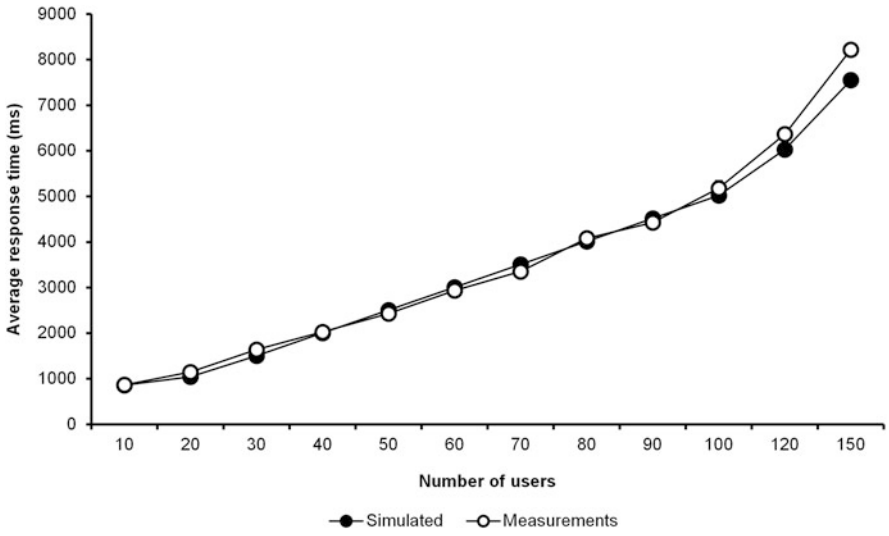

Fig. 9.5 EHR Simulated and measured average response times

Table 9.1 Instance types used for EHR cloud deployment

Instance name	Resource properties
Small	RAM: 2048 MB, Disk: 20 GB, VCPU 1
Medium	RAM: 4096 MB, Disk: 40 GB, VCPU 2
Large	RAM: 8192 MB, Disk: 80 GB, VCPU 4

model accuracy, the response times of the simulated requests were compared to the response times obtained from measurements (Fig. 9.5). The simulated results slightly underestimated the response times when the number of parallel users was greater than 100. The mean absolute percentage error of the model for the observed load range was 4.11%, which demonstrated good model accuracy.

The focus of the analysis were metrics most relevant for an industrial business case—capacity and the number of SLO violations. By using capacity metrics, we wanted to identify what part of the current EHR service user base can be served by different cloud instance types without violating the defined SLOs. This is why we also needed to track the number of SLO violations. We observed the scenario of retrieving information about the patient from the EHR database. The analysis was performed using an SLO specifying that the response time of the observed user request should not exceed 2 s.

The capacity metric was estimated for several different deployment configurations. We deployed the EHR platform on three different instance types (small, medium, and large instance), whose specifications can be seen in Table 9.1.

The measurements were performed to determine the capacity of a certain service deployment. For each deployment configuration, the number of response time SLO violations was measured while varying the number of parallel user requests. The

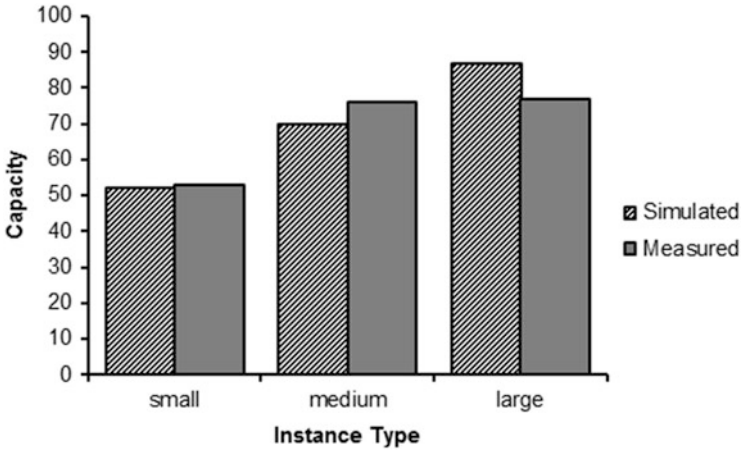

Fig. 9.6 EHR: simulated and measured capacity for different instance sizes

capacity of each configuration was noted as the maximum number of parallel user requests that resulted with no SLO violations. The measured instance capacities were compared with the ones obtained by the Analyzer tool (Fig. 9.6). The mean average percentage error of the model was 7.59%. The capacity determined by the Analyzer tool using the small instance was 52, very similar to the measured value of 53. The simulated capacity of the medium instance was 70, which was a lower value compared to the measured capacity of 76. However, the large instance was predicted to have a capacity of 87 parallel users, while repeated measurements on the validation environment demonstrated that the capacity of the large instance was 77, most likely due to the database processing rate, which was not entirely reflected in the simulations.

The second metric used during the validation of the EHR case was the number of SLO violations. In order to validate the implementation of this metric in the Analyzer tool, a series of measurements were performed where the number of parallel user requests was increased from 1 to 150. The system was deployed on the small instance (Table 9.1) in the previously described test environment. The number of SLO violations was observed with the response time thresholds of 2 s (Fig. 9.7). The root mean square error (RMSE) was calculated for each set of measurements, indicating the difference between actual measurements and values predicted by the model using the number of SLO violations metric in the Analyzer. The following figure demonstrates the percentage of requests resulting in the violation of the response time SLO, for both measured and simulated data (RMSE = 6.46%).

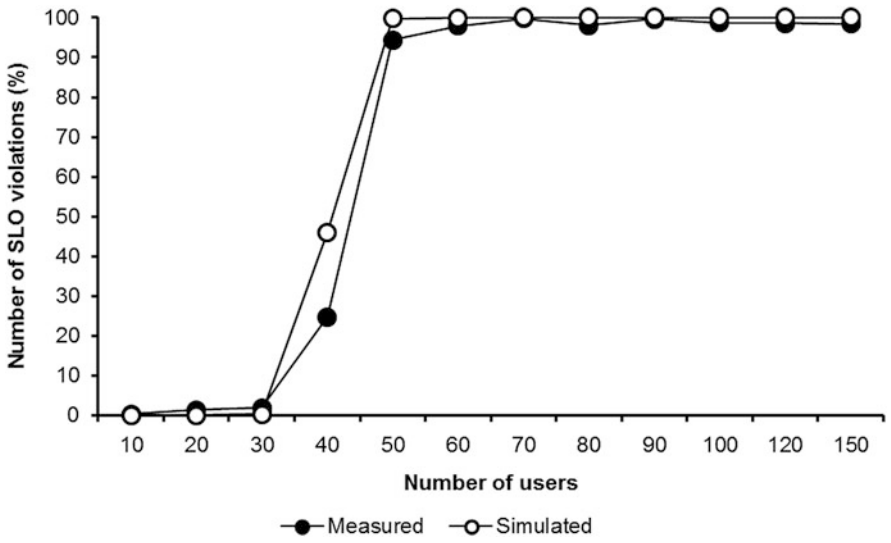

Fig. 9.7 EHR: simulated and measured number of SLO violations

9.1.3 Discussion of the Electronic Health Record Case

The EHR case used several activities from the CloudScale method and followed the CloudScale scenario of system migration to cloud environments. We first used the Extractor tool to successfully generate a model of each EHR module. However, the granularity of the extracted model was determined by the organization and structure of the source code. This means that the highest possible level of model abstraction was the level of system modules. From an industrial point of view, reverse engineering support in the creation of the model is beneficial because it makes the modeling process less time-consuming in the case of large code bases. In the EHR case, this turned out to be limiting in terms of adjusting the model granularity level according to the user's needs so the extracted model is useful in terms of enhancing it with monitoring data. However, we found that parts of the model generated by the Extractor tool can be used during the manual creation of the model; thus, all model components do not have to be specified from scratch.

The conclusion of the Static Spotter tool validation is that the tool generally returned expected output. However, the results for generating a model from source code varied depending on the Static Spotter's component-merging parameter. Generating a model for one of the projects failed, possibly due to the lack of enough main memory needed to process a large quantity of source code (>190k lines of code). Using the tool in other projects resulted in successful model creation and anti-patterns were detected when most of the information from the source code was preserved in the model. The experiment with the model component merging led to the conclusion that pattern detection is highly dependable on the parameters

used for model generation, since the detection process is based on the model of the system. Because of this fact, users of the tool will have to be well informed about the parameters used in model creation. Also, in the case of larger volumes of source code, it is expected that the Static Spotter tool faces issues in allocating enough main memory for source code processing. The benefit of the Static Spotter is the simplicity of use, resulting in a short learning curve, and detection of design-related issues, which is a non-trivial task, especially in systems with large code bases—a common case of products in Ericsson's portfolio, e.g., the EHR platform. The benefits offered by the tool will increase further with the number of implemented scalability anti-patterns.

We used the Analyzer to conduct simulations of several EHR usage scenarios, including different deployment strategies and workloads. The simulation-based analysis confirmed our assumption that the bottleneck of the EHR system deployed as a conventional non-cloud distributed system was the database. We proceeded with the analysis of the EHR system deployed to the cloud environment, with the focus on determining capacity, scalability, and elasticity system capabilities while monitoring the number of SLO violations. Model-based analysis has demonstrated promising results, which might lead to significant cost savings, since it is possible to determine optimal system provisioning without having to deploy an operational service. However, it is crucial to achieve a high level of model accuracy, which can be time-consuming. From the perspective of an industrial tool user, the Analyzer is a powerful but complex tool, and it takes time to understand it for producing accurate models.

9.2 Case Study: Kantega's Flyt CMS

Kantega's Flyt CMS is an open-source content management system (CMS) targeting anything from small personal web pages to large enterprise solutions, and is being used by many of Kantega's customers, both in private and in public sector. Kantega is a Norwegian SME providing IT consultancy services. The municipality of Trondheim uses Flyt CMS for providing information to its inhabitants, and in peak situations, e.g., in the case of strikes, the public wants to retrieve urgent information at the same time. It is thus critical that Flyt CMS satisfies its SLOs. In this case study, the Dynamic Spotter tool was applied to detect potential scalability issues by monitoring the operational Flyt CMS system.

9.2.1 Flyt CMS

Flyt CMS [3] is a CMS solution built with Java. It enables publishing and administrating content from any popular web browser and offers a flexible template system, as well as a platform for building custom software. The most common user

operations are read, update, and search for content. In addition, there are multiple administration features that ease the administration of the CMS.

Kantega has used Flyt CMS for many different customers, with customization suiting their different needs and requirements. It is currently a non-cloud solution. However, as it gains an increasing user base, moving to the cloud could be a viable alternative, since the likelihood of facing performance issues will increase with an increasing number of users. Flyt CMS is typically hosted in virtualized environments that, in a sense, emulate a cloud environment.

Flyt CMS is a three-layer architecture, data-centric application that contains logic for integrating data from different local sources and indexing content for fast searches. Figure 9.8 shows Flyt CMS' three-layer architecture. The Flyt CMS application handles data transmissions to and from the database, including dataflows from different sources. It enables administration tasks, access control, content management, and delivery of the content to the users. Flyt CMS is flexible with regard to integrating with different external data sources like business systems, data feeds, etc. Additional information regarding Flyt CMS implementation can be found in [2].

The case study on Flyt CMS provides an insight into its scalability, which is valuable for customers with a large user base and more concurrent activities on

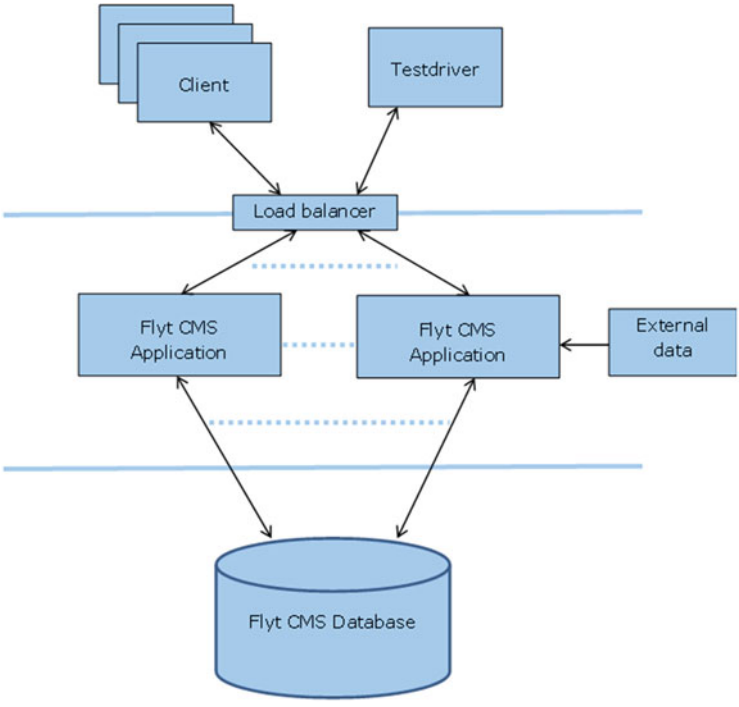

Fig. 9.8 Overview of the Flyt CMS architecture

the CMS. Flyt CMS used the CloudScale method in the context of monitoring an already deployed and running application. CloudScale's Dynamic Spotter provides a means for measurement-based testing, with an automatic analysis based on the chosen symptom or group of symptoms. Kantega has previously used Apache JMeter and HP LoadRunner as performance testing tools, which are easily integrated with the Dynamic Spotter through its extension plug-ins. Therefore, it was very convenient to include the Dynamic Spotter as a standard tool in the Kantega testing toolset.

9.2.2 Applying the CloudScale Method and Tools to Flyt CMS

While the EHR case deals primarily with the migration scenario, Flyt CMS serves as a case that utilizes CloudScale's Dynamic Spotter for monitoring and analyzing services in the operational phase. The Dynamic Spotter analysis was done with the OLB pattern, which looks for single points in the application which are used by many paths and thus hinders the application from scaling.

To gain results from the Dynamic Spotter, several prerequisites should be met. First, the running application must be started with an instrumentation agent that can instrument the byte code and report about measurements during runtime. Second, a load generation tool, which is not a part of the Spotter itself, must place a load on the running application. The Flyt CMS is defined as a Maven project written in Java, and it could be started with the instrumentation agent provided by the original Dynamic Spotter project. In addition, after setting up a load generation script through JMeter, a load generation tool that already exists in the collection of workload adaptors in the Dynamic Spotter, all prerequisites were fulfilled and the Dynamic Spotter was consequently applied to the Flyt CMS case.

There are two scopes available for the Dynamic Spotter analysis that specify where to look for scalability issues—the entry-point scope and the database scope. The entry-point scope instruments the starting servlet classes and reports about a scalability issue as the response time is increasing while having a constant (and low) CPU level. Knowing that there is an issue at entry level usually does not give enough information for pinpointing the problem in the Flyt CMS code base. The next option used is the "database scope", which gives detailed results per SQL query. However, as there are many classes that use the same queries, more information was needed for the analysis.

The central performance element in the Flyt CMS use case is how the CMS handles the communication with the database. Therefore, a major scalability weakness is related to this communication. To enable a major weakness in the Flyt CMS use case, a number of concurrent database connections from the application to the database were removed by setting the database pool the maximum. This configuration is done in the application settings, and cannot be discovered by a static tool. To test whether the Dynamic Spotter is able to pinpoint the limitation

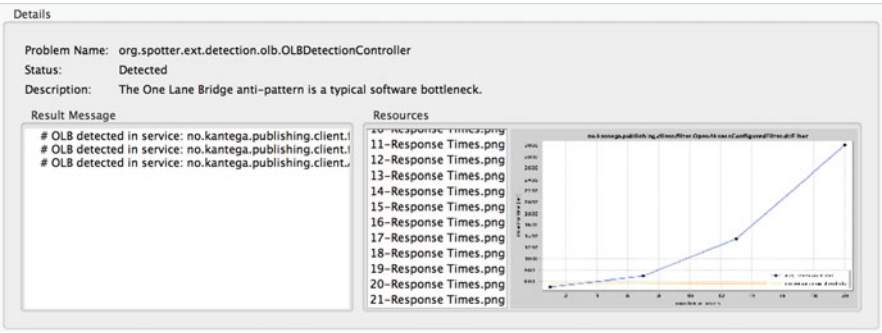

Fig. 9.9 Dynamic Spotter executed on the Flyt CMS use case, indicating increasing response times and the detection of the OLB anti-pattern

in database pool sizes, a test was run with the OLB pattern at both the entry scope and the database scope. A pre-defined response time threshold that should be used during the Dynamic Spotter's analysis was set. For both scopes, the Dynamic Spotter correctly discovered a scalability issue and reported on this with a description (Fig. 9.9).

9.2.3 Discussion of the Flyt CMS Case

The Dynamic Spotter was used for the discovery of potential scalability issues on Kanetga's Flyt CMS. The focus was on detecting the OLB anti-pattern in a running service. Finding the root cause of an OLB has involved detection runs in both the entry and database scopes. Both scopes provide valuable information, but in the case of a detected OLB, it is a necessity to identify the actual root cause that typically is between the entry and database scopes. The entry scope generally tells that an OLB is present in the application as a whole, but it does not answer the question on where the problem really is. Since the Dynamic Spotter is available as an open-source project, Kantega managed to get around this shortcoming by customizing the entry scope by adding a trace functionality that instruments the methods used by the classes originally specified for instrumentation, hence providing more in-depth information at the entry level. Although the addition of the user-defined scope was not reported to require a lot of effort, having more options on what level to instrument out of the box was noted as being very beneficial. The addition of more anti-patterns supported by the Dynamic Spotter and the ability to automate the process of detecting them have the potential of significantly decreasing the manual work in Kantega's traditional performance testing.

9.3 Additional Case Studies for the CloudScale Method

The CloudScale project was applied to existing industry solutions to test and evaluate CloudScale tools in a real-life environment. In total, there were five systems used for the CloudScale tools' evaluation. These systems were used in various phases of the tools' implementation, and have hence been employed for the testing and validation of the different tools' functionalities. Some of them were used more extensively during the project, while other covered a smaller fragment of the CloudScale project results. Table 9.2 describes the usage of the CloudScale tools by each of the use cases.

Two industrial use cases, **EHR** and **Flyt CMS**, were the focus of the project validation activities. These case studies are described in detail in the previous sections of this chapter (Sects. 9.1 and 9.2). An additional open-source showcase was planned to allow for the evaluation and reproducibility of the results related to the CloudScale method and each tool performed by any party.

Early on during the project, the **CloudStore** e-commerce sample application was designed and implemented as an additional result of the project to be used as a showcase. The CloudStore showcase has been already described in detail in Sect. 3.7, including its storyline of expected load growth throughout time. CloudStore was found to have scalability problems related to its database storage. First, some configuration problems were detected (related to the connection pool); later on, the inherent problem of a relational database cluster was found to become a bottleneck during high loads.

Additional to these three use cases, two industry solutions were included in the validation process. SAP HANA Cloud, used in the earlier phases of the project, and GeoServer, a very popular geographical information service implementation. Both use cases made use of the CloudScale tools in order to analyze potential scalability problems in their implementation and typical deployments.

GeoServer is a part of XLAB's GAEA+ geographical information services. Being one of the cornerstones of the solution, it was important to detect any potential scalability issues it could present, in particular for the given features being used. Having several features based on GeoServer as a composite service, the ability of scaling GeoServer efficiently would allow for providing services to additional clients with well-understood marginal costs. CloudScale was able to detect a few

Table 9.2 CloudScale tools usage by use-cases

Tool	ENT	SAP	Flyt	GeoServer	CloudStore
Extractor	Yes	Yes	Yes	Yes	Yes
Analyzer	Yes	–	–	–	Yes
Static spotter	Yes	–	Yes	Yes	Yes
Dynamic spotter	–	Yes	Yes	Yes	Yes
CloudScale method	Yes	–	Yes	Yes	Yes
CloudScale Environment	Yes	–	Yes	Yes	Yes

potential issues in GeoServer, though under closer inspection, these issues did not represent any major scalability problem, since they were located on sections of the system that are executed only during the start-up of the service.

SAP's HANA offered two of its components, SuccessFactor Service and Document Service, as additional applications tested by CloudScale.

While all of the stated cases contributed to the development and improvement of the CloudScale tools, the focus of the CloudScale method validation was on the EHR and Flyt CMS cases, which serve as an example of industrial products with a large user base and offer a perspective of employing the CloudScale method in a commercial environment for gaining relevant insights related to service engineering.

9.4 Conclusion

This chapter details two case studies used for inspecting the CloudScale method and tools in an industrial context. The first industrial case, EHR, follows the CloudScale method in a migration scenario, using the source code of EHR implemented for a traditional on-premises deployment and predicting the service behavior in a cloud environment. The second case study, Flyt CMS, is used for a scalability issue detection on a running service. Basic functionality, architecture, and some of the major scalability concerns are presented, together with the possible situations affecting service scalability. Several additional systems used for the testing and validation of the CloudScale tools used in different stages of their development are also presented.

The CloudScale method offered guidelines in approaching the migration of the EHR system to the cloud, and the resulting analysis provided insights into the behavior of the EHR system based on its model. The analysis provided by the CloudScale tools was found to be very useful for capacity planning of the infrastructure where the EHR system was planned to be deployed in the cloud environment. Also, the results of the simulations in terms of response times and the number of SLO violations can be used as guidelines when it comes to quality-of-service (QoS) assurance. The use of CloudScale's Dynamic Spotter on the Flyt CMS case demonstrated the possibility of significantly reducing manual effort in the process of troubleshooting the scalability issues on operational services.

Generally, the observed usability of tools was estimated as good. All of the CloudScale tools are available through the CloudScale Environment, offering a systematic and convenient view of the projects and tool results. However, since tools such as the Analyzer are fairly complex, software architects will likely have to invest some time in training and studying CloudScale's documentation and in creating accurate service component models in order to gain benefit from using CloudScale tools. Nevertheless, the toolset offers functionality that is especially interesting from the industrial point of view (e.g., the extraction of the model from the service source code, model-based scalability analysis, automated scalability anti-pattern detection on an operative service), which can be utilized to significantly reduce the cost of

experimental scalability analysis and the time to find the problem solution. Further details on case studies and detailed lessons learned can be found in a dedicated deliverable of the CloudScale project [2].

References

1. Ericsson Healthcare Exchange - EHE: http://www.ericsson.hr/ericsson-healthcare-exchange (2016) [Visited on 12/02/2016]
2. CloudScale: Project Deliverable D4.3: Requirements and validation, final version http://www.cloudscale-project.eu/media/filer_public/2016/02/02/d43_requirements_and_validation_third_version.pdf (2016), Visited: 1 December
3. Flyt CMS GitHub Repository: https://github.com/kantega/Flyt-cms (2016) [Visited on 12/02/2016]

Glossary

Analyzer The Analyzer allows to analyze ScaleDL models regarding scalability, elasticity, and cost-efficiency of cloud computing applications at design time.

Capacity Capacity is the maximum workload a service can handle as bound by its SLOs.

Cost-Efficiency Cost-efficiency is a measure relating demanded capacity to consumed services over time.

Elasticity Elasticity is the degree to which a service autonomously adapts capacity to workload over time.

Extractor The Extractor is a reverse engineering tool for automatic model extraction by parsing and analyzing source code.

Operation An operation specifies the name, type, parameters, and constraints for invoking an associated behavior.

Scalability Scalability is the ability of a service to increase its capacity by expanding its quantity of consumed lower-layer services.

ScaleDL editors Editors provided by the CloudScale IDE to create or update all aspects of the ScaleDL modeling language.

Service-Level Objective, SLO The quality-of-service target that must be achieved for each of a service's operations.

Spotter The Spotter allows to statically and dynamically detect scalability issues in implemented and running systems.

Workload Workload is the combined characterization of work and load, where work is the characterization of the data that is processed by a service's operations, and load is the characterization of the quantity of consumer requests to a service's operations at a given time.

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7

Index

3-layer, 36, 72

Abstraction, 58
Activity diagram, 55
Agile Unified Process, 41
Analyzer. *See* CloudScale Analyzer, 161, 173
  case study, 173
  example, 119
Apache Hadoop, 37
Apache JMeter, 179
Application Hiccups, 38
Architectural Templates. *See* HowTos, 70
  3-layer, 72
  catalog, 72
  concepts, 70
  example, 70
  horizontal scaling, 72
  loadbalancing, 72
  SPOSAD, 72
  tool support, 73
  vertical scaling, 72
Artifact, 88
ATs. *See* Architectural Templates

Big data, 37
Box-and-line diagram, 55
Broad network access, 51, 53
Business information systems, 36

Capacity, 9
Case studies, 167
Cloud application, 48
Cloud computing, 6, 48
  applications, 52, 54
  characteristics, 51
    broad network access, 51, 53
    measured service, 52, 54
    on-demand self-service, 51, 53
    rapid elasticity, 52, 53
    resource pooling, 52, 53
  modeling, 54
  requirements, 53
CloudScale
  Analyzer, 32
  method, 24
  Spotter, 34
    dynamic, 34
    static, 34
CloudScale Environment, 161
  workflow engine, 161
CloudScale method, 17
  accuracy, 87
  analyzing a modeled system, 96
  analyzing an implemented system, 97
  costs, 18, 156
  critical use cases, 92
  factors for successful projects, 158
  granularity, 86
  idenfy risks, 92
  key scenario, 93
  notation, 88
  objectives, 86
  operation, 99
  organizational issues, 156
  precision, 87
  prerequisites, 19
  risks, 158

S. Becker et al. (eds.), *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications*, DOI 10.1007/978-3-319-54286-7

scope, 87
steps, 90
success factors, 154
technical requirements, 94
CloudStore, 26, 56, 181
Complementing tools, 162
Component diagram, 55
Cost-efficiency, 11
analysis, 34
requirements, 31
Critical use cases, 28, 92

Data & control flow, 88
Data flow, 88
Data Transfer Objects, 141
Deployment diagram, 55
DevOps, 14
Distributed computing, 48
Docker, 50
Document Service, 182
DropBox, 52
Dynamic Spotter. *See* CloudScale Spotter
dynamic, 138, 179
case study, 179
Example, 143

EHE. *See* Ericsson Healthcare Exchange
EHR. *See* Electronic Health Record
Elasticity, 10
analysis, 33
requirements, 30
Electronic Health Record, 167, 168, 170, 181
Empty Semi-Truck, 38, 141
Ericsson, 167
Ericsson Healthcare Exchange, 168
Eventually-consistent, 51
Excessive Dynamic Allocation, 38, 141
Expensive Database Calls, 40
Extended Palladio Component Model, 73
specify, 111
External artifact, 88
External manual task, 88
Extractor, 79, 161, 170
case study, 170

Flyt CMS, 167, 177, 181

GAEA+, 181
GeoServer, 181
GoogleDrive, 52

Granularity, 151
during modeling, 104
resource granularity, 105
service granularity, 105
usage granularity, 105
Grid computing, 48

Hadoop MapReduce, 37
High performance computing, 48
Horizontal scaling, 37, 72, 140
HowNotTos, 37
Application Hiccups, 38
Empty Semi-Truck, 38
Excessive Dynamic Allocation, 38
Expensive Database Calls, 40
One-Lane Bridge, 38, 97, 136, 140, 171
resolving, 140
spot dynamically, 138
spot statically, 136
The Blob, 38
The Ramp, 38
The Stifle, 40
HowTos, 35
3-layer, 36
catalog, 35
Hadoop MapReduce, 37
horizontal scaling, 37, 140
loadbalancing, 36, 140
MapReduce, 37, 140
sharding, 36
SPOSAD, 37
static content, 36
vertical scaling, 37, 140
HP LoadRunner, 179
HPC. *See* high performance computing
HTML5, 49

JMeter, 179

Kantega, 167, 177, 181
Key considerations, 150
Key scenario, 93
Key scenarios, 28

LIMBO, 56
Loadbalancing, 36, 72, 140

Management
Cost-efficiency, 13

Elasticity, 13
Proactive, 16
Reactive, 14
Scalability, 13
Manual tasks, 88
MapReduce, 37, 140
Measured service, 52, 54
Micro-service, 50
Microservice, 16
Microsoft OneDrive, 52
Model-view-controller, 49
Modeling hints, 58
Monitoring, 35
Monitors, 76
MVC. *See* Model-view-controller

On-demand self-service, 51, 53
One-Lane Bridge, 38, 97, 133, 136, 140, 171, 179
OpenStack, 173
OpenUP, 41
Operations, 7
Overview Model
specify, 109
Overview model, 63
concepts, 63
definition, 63
example, 64
tool support, 65

Palladio Component Model, 74
allocation model, 74
examples, 76
resource environment model, 74
system model, 74
tool support, 78
usage model, 75
Parallel tasks, 88
PCM. *See* Palladio Component Model
Pilot project, 159
Planning horizon, 28

Rapid elasticity, 52, 53
Rational Unified Process, 41
Resource Pooling, 141
Resource pooling, 52, 53
RESTFull API, 50
Role, 88
developer, 89
product manager, 89
service consumer, 89
service provider, 90
system architect, 89
system engineer, 89
Roles
CloudScale, 89
NIST, 90
RUP. *See* Rational Unified Process, 153

SAP HANA Cloud, 181
Scalability, 10
analysis, 32
requirements, 30
ScaleDL, 24
Architectural Templates, 70
Extended Palladio Component Model, 73
model specification, 32
Overview model, 63
Usage Evolution, 66
ScaleDL editors, 161
SCRUM, 153
SEFFs. *See* Service Effect Specifications
Self-adaptation, 56
Self-adaptation rules, 75
Service Effect Specifications, 74
Service level objectives, 8, 27
Service-oriented Architecture, 6
Service-oriented front end applications, 49
Sharding, 36
Single page web applications, 49
SLOs. *See* Service level objectives
SPOSAD, 37, 72
Spotter. *See* CloudScale Spotter, 162
Spotters
Dynamic Spotter, 138
Static Spotter, 136
Spring, 57
State chart, 55
Static content, 36
Static Spotter. *See* CloudScale Spotter static, 136, 171
case study, 171
Example, 141
StoryDiagrams, 56
SuccessFactor Service, 182
System
deployment, 35
operation, 35
realization, 35

Technical requirement, 94
The Blob, 38, 141
The Ramp, 38

The Stifle, 40
Tool-driven process, 88
Tools
  Dynatrace, 162
  JMeter, 162
  JProfiler, 163
  Kieker, 162
  Palladio, 162
  R, 163
  VisualVM, 163
TPC-W, 56

UML MARTE, 24
UML2, 54
Unified Process, 41
Usage Evolution, 66
  concepts, 66
  definition, 66
  example, 67, 106
  specify, 106
  tool support, 69
Usage evolution, 95
Use-cases, 54

Vertical scaling, 37, 72, 140

Web application, 49
Web Scale IT, 4
Workload, 8

XLAB, 181