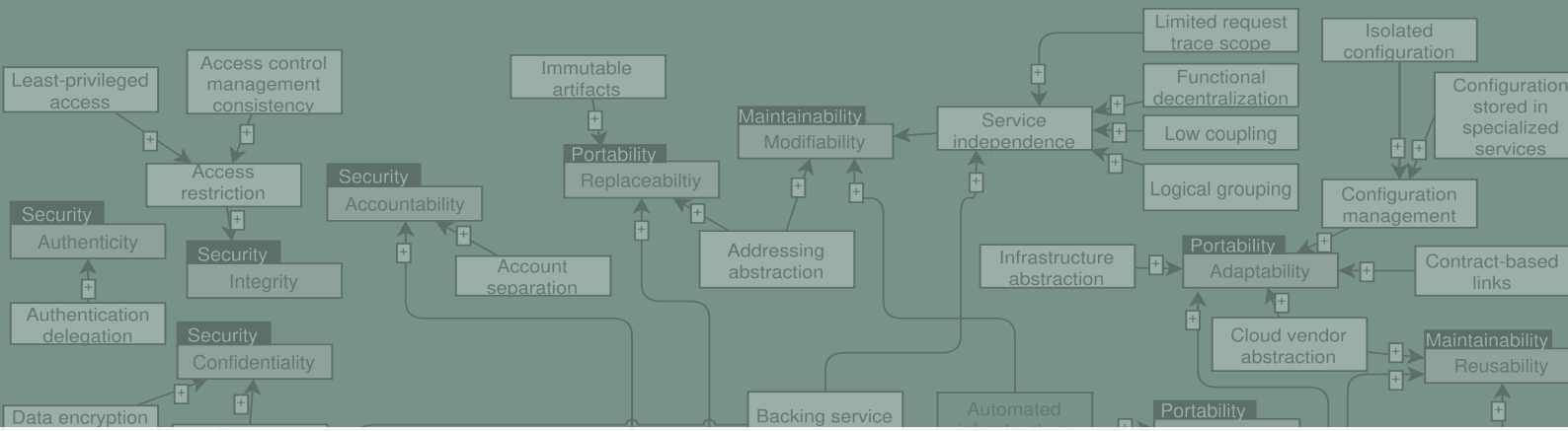
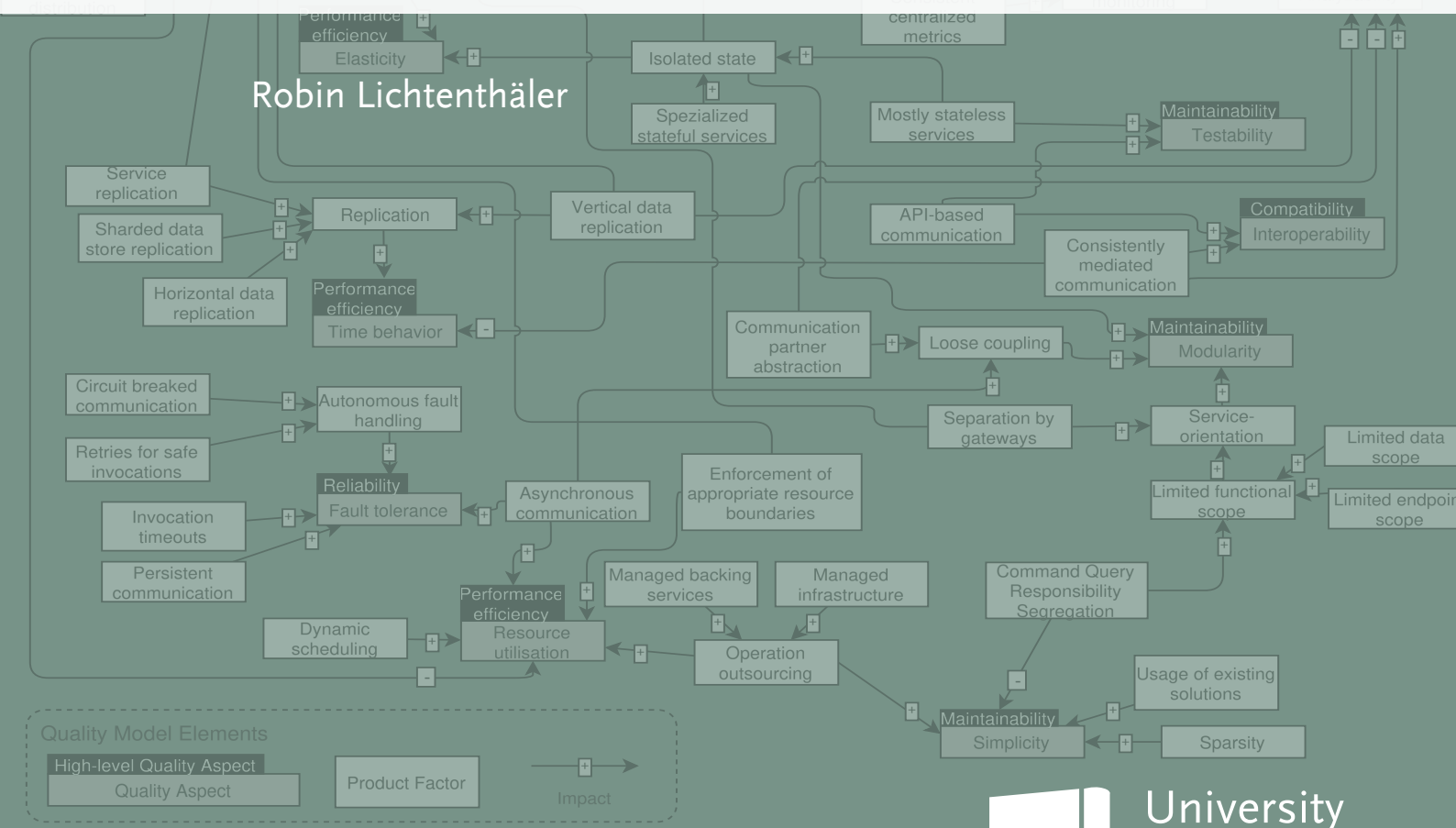
49 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg



Cloud-native Software Architecture Evaluation

A Quality Model and its Practical Applicability

Robin Lichtenthäler



University of Bamberg Press

49 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Band 49

University of Bamberg Press
2026

Cloud-native Software Architecture Evaluation

A Quality Model and its Practical Applicability

Robin Lichtenthäler

University
of Bamberg
Press
2026

Bibliografische Informationen der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg als Dissertation vorgelegen.
1. Gutachter: Prof. Dr. Guido Wirtz
2. Gutachter: Prof. Dr. Daniel Beimborn
Tag der mündlichen Prüfung: 11.03.2026

Herstellung und Druck: docupoint, Magdeburg
Umschlaggestaltung: University of Bamberg Press
Umschlaggraphik: Robin Lichtenthäler

University of Bamberg Press, ubp@uni-bamberg.de, Bamberg 2026
ROR https://ror.org/004fa0x02

ISSN: 1867-6197 (Print)
ISBN: 978-3-98989-116-6 (Print)
eISSN: 2750-8560 (Online)
eISBN: 978-3-98989-117-3 (Online)

URN: urn:nbn:de:bvb:473-irb-114696x
DOI: https://doi.org/10.20378/irb-114696

Für M und L

Acknowledgments

My journey at the University of Bamberg started in 2012 with the beginning of my bachelor studies. With this thesis, it comes to an end. Over the years I have met many people who accompanied me on the way and to whom I thus want to say *Thank you!* on this occasion although I cannot mention everybody by name.

In particular, I want to thank my supervisor Prof. Dr. Guido Wirtz for his trust and support. I actually met him in my very first lecture at the university *Einführung in die Informatik* with no idea that I would once work at his chair. Due to his pragmatism and kindness, I really enjoyed doing research, teaching and overall being a part of the Distributed Systems Group. A special thanks also goes to the other members of my dissertation committee, Prof. Dr. Daniela Nicklas as the committee chair and Prof. Dr. Daniel Beimborn who is also the second assessor. Both already inspired and motivated me during my bachelor and master studies.

Working at the Distributed Systems Group was a privilege and it would not have been the same without such great colleagues. I am grateful to Johannes Manner, Sebastian Böhm, and Stefan Winzinger for being together in the *last generation*. For shaping the teaching at the chair, collaborating in research, and always being supportive, I want to thank Stefan Kolb, Andreas Schönberger, Simon Harrer, Linus Dietz, Jörg Lenhard, and Matthias Geiger. And without Cornelia Schecher, I would probably be filling out and printing administrative forms right now instead of writing these lines. Thank you for your support in all respects.

I will also miss the daily, and sometimes extensive, lunch breaks together with the former and current members of the SWT and SYSNAP groups, as well as the new DSG group. Supporting bachelor and master theses was also a big part of my work at the university. I am more than happy, that some also lead to papers that I could publish together with Mike Prechtl, Isabell Sailer, Niels Knoll, Franka Knoch, Anton Frisch, and especially Karolin Dürr who also implemented the first prototype for the modeling feature of the Clounaq tool.

Last but not least, I want to thank my family, especially my parents and my brother, for their support and love throughout all this time.

Kurzfassung

Cloud-native wird als Begriff in der Praxis und im akademischen Umfeld genutzt, um zu beschreiben, wie Software, die optimal in Cloud Umgebungen betrieben wird, entwickelt werden sollte. Die Popularität des Begriffs führt zu einem breiten Bestreben, web-basierte geschäftliche Anwendungen auf eine cloud-native Art und Weise zu entwickeln, um von den Vorteilen moderner Cloud Umgebungen zu profitieren. Aufgrund der Komplexität moderner Cloud Umgebungen umfasst der Begriff cloud-native allerdings ein breites Spektrum an Aspekten, die es zu berücksichtigen gilt, vom Design einer Anwendung hinsichtlich Struktur und Kommunikation bis zur Technologie- und Infrastrukturauswahl. Ein tiefgreifendes Verständnis von cloud-nativem Anwendungsdesign und Entscheidungen, welche cloud-nativen Charakteristiken für eine Anwendung von Vorteil sein können, stellen Softwarearchitekten und -entwickler daher vor Herausforderungen.

Diese Arbeit formuliert und validiert ein hierarchisches Qualitätsmodell für cloud-native Softwarearchitekturen mit einem Fokus auf Designzeit-Aspekte. Cloud-native Charakteristiken sind als Faktoren im Qualitätsmodell dargestellt, welche durch Einfluss-Beziehungen mit Qualitätsaspekten verknüpft sind. Eine Einfluss-Beziehung beschreibt, wie ein Qualitätsaspekt beeinflusst wird, wenn der Faktor in einer Anwendung vorhanden ist. Zusammen mit dem Qualitätsmodell wird ein Modellierungsansatz für cloud-native Architekturen erarbeitet und das Clounaq Tool entwickelt. Das Clounaq Tool ermöglicht die Modellierung und Evaluierung von Softwarearchitekturen basierend auf dem Qualitätsmodell. Softwarearchitekten und Entwickler können damit die Auswirkungen cloud-nativer Charakteristiken besser verstehen und für neue oder bestehende Anwendungen evaluieren, wie deren Design von cloud-nativen Charakteristiken profitieren kann.

Das entstandene Qualitätsmodell erklärt den Begriff cloud-native aus einem Qualitäts-Blickwinkel heraus. Es erklärt, wie Anwendungen auf eine cloud-native Art und Weise entwickelt werden können, um bestimmte Vorteile zu erhalten. Durch die Strukturierung in Form eines Qualitätsmodells können Softwarearchitekten entscheiden und priorisieren, welche Charakteristiken cloud-nativer Anwendungen für eine bestimmte Anwendung relevant sind. Der vorgestellte Modellierungsansatz zeigt auf, wie Kernaspekte cloud-nativer Softwarearchitekturen formal in textueller und graphischer Form dargestellt werden können. Der innerhalb von Clounaq implementierte Katalog von Architekturmetriken ist mit dem Modellierungsansatz abgestimmt und kann von Softwarearchitekten in der Praxis, aber auch im wissenschaftlichen Umfeld für die Arbeit an cloud-nativen Softwarearchitekturen verwendet werden.

Abstract

Cloud-native is used as a term in industry and academia for describing how software should be built in order to run in cloud environments properly. The usage and popularity of the term leads to the desire to build web-based enterprise applications in a cloud-native way to benefit from the advantages of modern cloud environments. However, due to the complexity of modern cloud environments, cloud-native spans a broad range of aspects from the design of an application's structure and communication to the selected deployment technologies and infrastructure. Thus, understanding thoroughly what a cloud-native application design means and deciding how a specific application can benefit from cloud-native patterns and characteristics, is challenging for software architects and developers.

This work presents the formulation and validation of a hierarchical quality model for cloud-native software architectures focusing on design time aspects. Cloud-native characteristics are formulated as factors for the quality model and linked to quality aspects through impact relationships. An impact describes how a quality aspect is influenced, if a factor is present in a system. Together with the quality model, a modeling approach for cloud-native software architectures and the Clounaq tool was developed. The Clounaq tool offers software architects a model-based method for designing and evaluating software architectures based on the quality model. For the evaluation approach, architectural measures quantify the factors of the quality model. Software architects and developers can thus better understand consequences of cloud-native characteristics and evaluate for new or existing applications, how the application architecture design can benefit from cloud-native characteristics.

The resulting quality model captures the topic of cloud-native software architectures from a quality-focused point of view. It explains how applications can be built in a cloud-native way to achieve certain benefits. Through the characterization in the form of a quality model, software architects can evaluate and prioritize which cloud-native characteristics are relevant and applicable to a certain application at hand. The presented modeling approach demonstrates how core aspects of cloud-native software architectures can be represented in a formal textual and a graphical notation. The Clounaq tool includes implemented calculations for the catalog of architectural measures aligned with the modeling approach and can thus be used by practitioners and researchers for future work on cloud-native software architectures.

Contents

List of Figures

List of Tables

List of Listings

List of Definitions

List of Abbreviations

AWS Amazon Web Services

API Application Programming Interface

ADL Architecture Description Language

BPMN Business Process Model and Notation

CaaS Container as a Service

CI/CD Continuous Integration / Continuous Delivery

Clounaq Cloud-native architectural quality

CNA Cloud-native Application

CNCF Cloud-native Computing Foundation

CPU Central Processing Unit

CQRS Command and Query Responsibility Segregation

CSV Comma-separated values

DDD Domain-Driven Design

DNS Domain Name Service

EC2 Elastic Compute Cloud

EKS Elastic Kubernetes Service

ESB Enterprise Service Bus

FaaS Function as a Service

GQM Goal Question Metric

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IaaS Infrastructure as a Service

IaC Infrastructure as Code

IAM Identity and Access Management

IP Internet Protocol

JAR Java Archive

JRE Java Runtime Environment

JSON JavaScript Object Notation

MDE Model-Driven Engineering

NIST National Institute of Standards and Technology

OCI Open Container Initiative

OPA Open Policy Agent

PaaS Platform as a Service

RBAC Role-Based Access Control

RDS Relational Database Service

REST REpresentational State Transfer

RPC Remote Procedure Call

SaaS Software as a Service

SOA Service-oriented Architecture

SOAP Simple Object Access Protocol

SOC Service-oriented Computing

SPA Single-Page Application

SVG Scalable Vector Graphic

TLS Transport Layer Security

TOSCA Topology and Orchestration Specification for Cloud Applications

UI User Interface

UML Unified Modeling Language

URL Uniform Resource Locator

VM Virtual Machine

VPC Virtual Private Cloud

XaaS Anything as a Service

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Part I.

Background and Problem Identification

Cloud is about *where* we're computing.
Cloud-native is about *how*.

Cornelia Davis [60]

1. Introduction

Cloud computing can nowadays be considered a mature concept. It has been integrated as a common part of companies' IT portfolios. According to a survey by O'Reilly done in 2021, 90% of respondents were using cloud computing [178]. Although this seems impressive, it has to be noted that O'Reilly is a technology-focused company and respondents were recruited from subscribers of their newsletter [178]. A more comprehensive insight is provided by Eurostat, although only for Europe. According to their report on the use of cloud computing by enterprises, "*42.5% of EU enterprises bought cloud computing services in 2023*" [81] showing an increase of 4.2% from 2021 [81]. It can thus be stated that adoption of cloud computing is significant and increasing. Nevertheless, there are many ways how cloud computing technologies can be used. And due to continuous evolution and innovation, the range of cloud computing offerings gets broader and in parts also more complex. Deciding which cloud offerings to use, how to configure and integrate them, and how to build suitable applications for them can become challenging.

Cloud-native has therefore recently emerged as a concept. According to the O'Reilly Technology Trends for 2024 [179], it has become the most widely used topic on their platform in the overall context of cloud computing. Cloud-native as a term describes applications or technologies that are built specifically *for* the cloud [179]. That means these systems are on the one hand able to take advantage of the possibilities modern cloud technologies offer and can on the other hand also deal with inherent issues of cloud computing. The aim of developing systems specifically for the cloud thus means improving the quality of a system in one way or another.

To support the design and development of applications in a cloud-native way, a quality model is presented in this work. This quality model for cloud-native software architectures describes how cloud-native characteristics impact quality aspects. An approach is furthermore presented that enables evaluations of software architectures according to this quality model. Evaluations are based on modeled application architectures. The contribution of this work, therefore, is twofold: First, the quality model is a conceptual contribution providing a more detailed and structured understanding of what cloud-native means for the design and deployment of applications. Second, the evaluation approach with corresponding tooling is a practical contribution providing a means to apply the quality model to application architectures.

1.1. Context

Since the aim of this work is to support developers in designing and implementing *cloud-native applications*, this perspective needs to be emphasized and set in contrast to other perspectives. More specifically, this work considers the topic of cloud-native neither from a *cloud infrastructure* provider perspective nor from the perspective of cloud software vendors who provide software that enables building cloud-native applications. Such enabling software can be called *cloud-native middleware* and is exemplified by the projects listed in the Cloud-native Computing Foundation (CNCF) Landscape[1]. While these projects are also labeled as cloud-native, these are not in the focus of the quality model presented here. Instead, by using such cloud-native middleware, certain characteristics described in the quality model can be implemented.

The perspective of this work is that of a cloud consumer. The type of applications which the presented quality model targets can be best described as *Enterprise Applications* as outlined in more detail by Cerny et al. [50] who refer to Fowler [89] as a basis. In line with the conception of these authors, enterprise applications have embedded knowledge of the specific domain in which they are used, satisfy either external or internal needs (i.e., customer or employee needs), offer user-focused or machine-focused interfaces, and persist data according to the domain in which they operate. Furthermore, enterprise applications are typically implemented as web-based applications. That means, they are typically used through a frontend component accessible via a browser. In the case of desktop- or mobile-based frontend components, at least the backend components are deployed on and accessed via the web. Being web-based is a necessary requirement for an application to be deployable to a cloud environment which is by its definition "*available over the network*" [190]. How such enterprise applications can be designed as cloud-native applications is exactly the focus of this work. This focus, however, only extends up to the point from which on domain-specific knowledge would be required. This is because the approach is intended to be applicable independent of a specific business domain. Applications are therefore considered on a level of components that interact with each other and are deployed on a cloud infrastructure, but not on a detailed *business logic* level that is domain-specific.

This described focus is summarized in Figure 1.1 a), and in addition to the type of applications, the term quality also needs to be set in context. According to the ISO 25000 standard: "*The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value.*" [122] This means, ultimately, quality has to be considered relatively in comparison to specified needs, not absolutely. A quality evaluation is thus often considered as a comparison between a measured value describing a current state and a specified value capturing a target state. Such a specification of required targets, however, is

[1] `https://landscape.cncf.io/`, visited 2025-10-20

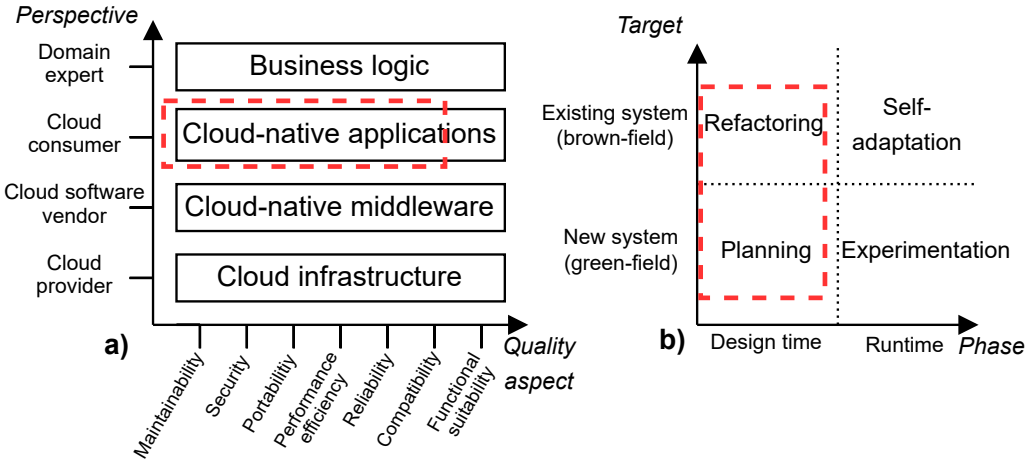


Figure 1.1.: Context of the approach:
a) Type of software and quality b) Lifecycle phase

specific to a certain application or type of applications and thus can only be done individually for different use cases.

In contrast to this general understanding of quality evaluations, the evaluations proposed and supported by this work focus on examining software architectures according to the presence of characteristics associated with cloud-native applications. The underlying quality model states how the presence of such characteristics impacts different quality aspects Based on the quality model, these evaluations provide insights on how applications already benefit from or could be changed to benefit from cloud-native characteristics. An interpretation of the evaluations and a derivation of specific actions on how to improve an application based on individual requirements and constraints, however, is up to the developers of an application. Furthermore, the scope of quality is considered broadly, that means multiple sub-aspects of quality are included. This is because the aim is to cover the breadth of the topic of cloud-native by being able to express different impacts of cloud-native characteristics. Otherwise, certain consequences of using a cloud-native approach might be neglected. One sub-aspect, however, is not included in the quality model, namely that of *Functional suitability* [122]. The functional suitability is specific to a certain application and the type of domain-specific functionality it provides. An evaluation of it is therefore not possible in a general way focusing on the topic of cloud-native. Thus, it is excluded from the quality model presented in this work.

In addition to the type of applications and quality aspects considered in this work, it should be made clear in which phases of the software development process the presented quality model and evaluation approach are intended to be used. Because the evaluation approach is relying on a modeled software architecture, it can be used independently of a specific application implementation. This is

intentional and enables the approach to be applied for evaluating potential architectures for a newly designed application during *planning*. But also existing applications can be evaluated and thus checked for potential improvements that can be introduced by *refactoring*. Thus, it is applicable throughout the whole software development process, although only at design time, as shown in Figure 1.1 b).

1.2. Problem Statement

In the following, the problem this work aims to solve is derived based on a set of hypotheses. First it is described why there is a need in supporting developers who aim to benefit from the advantages of cloud-native applications. And second, it is described how such support could be provided and which problems need to be tackled while doing so.

Cloud-native is a topic of great interest for cloud application development, as shown by the recent O'Reilly Technology Trends for 2024 [179]. Definitions of what the term means have been provided by various entities. From an industry perspective, the CNCF [56] has come up with the following definition for the term itself:

> **Definition 1.1 (Cloud Native)**
> *"Cloud native practices empower organizations to develop, build, and deploy workloads in computing environments (public, private, hybrid cloud) to meet their organizational needs at scale in a programmatic and repeatable manner. It* ***is characterized by loosely coupled systems that interoperate in a manner that is secure, resilient, manageable, sustainable, and observable.***
> *Cloud native technologies and architectures typically consist of some combination of containers, service meshes, multi-tenancy, microservices, immutable infrastructure, serverless, and declarative APIs — this list is non-exhaustive."* [56, v1.1]

This definition describes more of an approach than applications in specific. In contrast, from the perspective of academia, and considering explicitly applications, Kratzke & Quint [151] have formulated the following definition:

> **Definition 1.2 (Cloud-native Application)**
> *"A cloud-native application (CNA) is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform."* [151]

As it can be seen from these two definitions cloud-native covers a range of topics. Furthermore, both definitions leave room for further additions ("[...] *this list is non-exhaustive.*" [56]) or own interpretations ("[...] *cloud-focused design patterns* [...]"

[151]). One topic of cloud-native considers how applications are designed, especially considering how they communicate. Cloud-native systems should be loosely coupled [56] and use declarative Application Programming Interfaces (APIs) [56]. This aspect of application design and their communication characteristics is further substantiated by the reference to the microservices architectural style [56, 151]. Microservices are a research topic for themselves with a main focus on how business logic should be distributed over independent services to reach agility in system evolution and operation. Because of their distributed nature, communication between such independent services is a major concern.

Another topic of cloud-native is which technologies are chosen for implementing and packaging applications depending on the characteristics they provide. For application deployment, self-contained deployment units [151], typically containers [56], should be used. Using containers simplifies the deployment on various infrastructures and enables a fast replacement of component instances to support resilience, scalability, and updates. For data storage or middleware services, technologies should be used that support cloud-native aspects, such as horizontal scalability [151] or resiliency [56].

The infrastructure used for cloud-native applications also has a major impact on application characteristics. So-called self-service elastic platforms [151], with Kubernetes[2] as the prime example, support DevOps [161] principles and provide built-in functionalities such as automated scaling, automated restarts, or rolling upgrades. By relying on immutable infrastructure [56], the provisioning of underlying infrastructure, especially virtual machines, is repeatable and can be automated. This enables for example the provisioning of a fallback infrastructure in case of datacenter failures for a quick system recovery.

As a cross-cutting concern, the monitoring of applications [99] on different layers plays an important role. Monitoring supports the recognition of and reaction to failures, the detection of potential security incidents, or an optimization of resource consumption. This aspect is mostly associated with the operation of an application, but needs to be implemented at design time already. Systems with a comprehensive monitoring are also called observable [56] systems. And apart from the already mentioned automated scaling, cloud-native systems can also dynamically adjust the resource usage or component placement [99] during operation.

From this description of relevant topics, it can be seen that a range of aspects are important for implementing applications in a cloud-native way. Furthermore, these aspects are not independent, but need to be considered from an integrated point of view. For example, to benefit from a deployment infrastructure that provides automated scaling, the application also needs to be designed accordingly. Components which are scaled horizontally by the platform on which they are deployed should be designed stateless [151]. And required state needs to be stored with data storage technologies that support horizontal scaling themselves to avoid

[2] `https://kubernetes.io/`, visited 2025-10-20

bottlenecks through unbalanced scaling behavior. In summary, these observations lead to the formulation of the first hypothesis:

> ***Hypothesis 1*** *Cloud-native spans a broad range of aspects such as application design, technology selection, deployment infrastructure, and operations. Decisions in each aspect need to be considered in combination for assessing an application architecture.*

Assuming that hypothesis 1 is true, the problem comes up of how this complexity can be dealt with when aiming to develop applications in a cloud-native way. How could a comprehensive overview of cloud-native concepts be provided? Which application characteristics should be focused on and why?

Independent of which aspect from Hypothesis 1 is considered, building applications in a cloud-native way is typically associated with certain benefits. These benefits can be characterized by associating them with quality aspects that describe in which specific dimension an application is improved, if a cloud-native characteristic is fulfilled by an application. For example, if services are replicated over different data centers, the aim is to minimize latencies [99] and thus improve the quality aspect *Time-behavior* [122]. When components of a cloud-native application are upgraded, this should happen seamlessly without disrupting production [99] which targets the quality aspect *Availability* [122]. And also *Security* [122] as a quality aspect is emphasized as having to be included in an application architecture from the beginning [99], for example by utilizing a consistent access control approach over the different components of an application.

The breadth of the topic of cloud-native thus also extends to the different dimensions of quality. But these quality aspects can serve as an overarching ordering concept, since quality aspects can be structured in so-called quality models, "*that categorize product quality into characteristics, which in some cases are further subdivided into sub-characteristics. This hierarchical decomposition provides a convenient breakdown of product quality*" [122]. While early quality models associated quality aspects directly with measures (see Ferenc et al.'s work [85] for an overview), more recently hierarchical quality models have been introduced. They conceptualize characteristics of an application as factors between quality aspects and measures [295]. Thus, a hierarchical quality model should be applicable for structuring and explaining how different characteristics of cloud-native applications impact different quality aspects. Such a quality model could also provide a more structured understanding to the topic of cloud-native. It is therefore the subject of the second hypothesis:

> ***Hypothesis 2*** *A hierarchical quality model is able to express consequences of architectural design decisions in the context of cloud-native applications based on application characteristics impacting quality aspects.*

Assuming the existence of a quality model as stated in Hypothesis 2, the question arises how the knowledge contained in the quality model can be used during

application development? Ideally, support should be provided both, for developing new applications in a cloud-native way and for improving existing applications or even transforming applications into cloud-native applications.

Applying a quality model means evaluating an application based on the quality model to gain insights and an evaluation result describing the state of an application. Based on the evaluation results, actions can be derived to further evolve an architecture and strengthen certain quality aspects, if desired. The fundamental question is how the application or system to evaluate should be represented. Different options are available, from using the source code of an application directly to informal descriptions in prose-form. To decide which representation option is suitable, several factors need to be considered. First and foremost, the representation needs to include all relevant information required to perform the quality evaluation. For applications, as considered in this work, a challenge is that relevant information is not only included in source code, but also in additional artifacts, such as deployment descriptions [50], or even documentation. For reproducibility and to avoid ambiguities, a formal and structured representation is preferable. A structured representation is also machine processable. Thus, corresponding tooling can build on it to automate the evaluation procedure. A challenge for implementing corresponding tooling, however, is the technological heterogeneity within the context of cloud-native applications. Components of an application can be implemented in different programming languages, deployed using various tools and platforms, and depending on where an application is deployed, cloud providers have their own specific peculiarities.

An alternative and common choice, instead of using source code directly, are models [8]. Models are often used in so-called architecture optimization approaches [7] for which quality evaluations can be seen as a part of an overall approach. By using a model as an architectural representation for cloud-native applications, information specifically required for an evaluation can be included in the model. And details such as those specific to the business domain of an application can be left out. Therefore, Hypothesis 3 proposes to use an architectural model to represent cloud-native application architectures and perform quality evaluations based on modeled architectures.

> ***Hypothesis 3*** *An architectural model of a software system can be used to represent an application in a domain-independent way that allows for including necessary information to formally express application characteristics specific to cloud-native applications.*

The quality model proposed in Hypothesis 2 already enables qualitative evaluations of applications. However, for meaningful evaluations also a quantitative approach is desirable. Thus, the question arises how this can be achieved.

The data basis necessary for a quantitative evaluation is provided by the architectural model proposed in Hypothesis 3. The connecting element between the quality model and this architectural model are architectural measures that quantify

the extent to which certain factors are present in a system. Architectural measures have been proposed and discussed in literature in the past already [320]. But because such measures are taken at design time and the actual properties of a system can only be determined at runtime, finding meaningful and ideally generalizable measures is a challenge. To be meaningful, measures need to accurately quantify the factor of a system that they are associated with. Thus, an interpretation of the measure's value needs to be found that accurately describes the impact a measure has on an application's runtime characteristics. This can for example be achieved by using runtime measures as a validation mechanism.

Suitable architectural measures can be either adapted from existing literature or newly defined based on the factors of the quality model. If their meaningfulness based on the quality aspects in question can also be validated, they can be used to quantify evaluations of software architectures. This is summarized in the last hypothesis:

> ***Hypothesis 4*** *By applying architectural measures to a modeled software architecture of a cloud-native application, architectural characteristics can be quantified and, in combination with a quality model, impacts on quality aspects can be evaluated.*

In summary, the overall problem that this work aims to solve is how the development of cloud-native applications can be supported in a structured way, given the fact that cloud-native as topic covers a broad range of aspects. Based on the stated hypotheses, an approach is presented that on the one hand, supports these hypotheses and on the other hand represents a solution to the given problem.

1.3. Contributions

To provide the mentioned support for developing cloud-native applications, this work makes two main contributions. The first contribution (C1), the quality model, can be seen as a more theoretical contribution. The second contribution (C2), the Cloud-native architectural quality (Clounaq) approach with corresponding tooling, instead, is a more practical contribution building on the first. The contributions have been developed and presented through a range of publications, as shown in Table 1.1.

(C1) Quality Model for Cloud-native software architectures
The quality model is based on the Quamoco [294] metamodel. It is a hierarchical quality model containing product factors that describe cloud-native characteristics structured according to multiple quality aspects adopted from the ISO25000 standard [122]. It provides a quality-based view on the topic of cloud-native and can be used to evaluate software architectures by assessing to what extent the factors described in the quality model are present in an architecture.

(C1.1) Factors and Impacts
The factors of the quality model are either product factors or quality aspects. Product factors describe a certain characteristic of an architecture and have been formulated based on literature on the topic of cloud-native. Quality aspects describe more abstract quality attributes and are adopted from the ISO25000 standard [122]. Relationships between factors are represented by Impacts. An impact describes how one factor is impacted by the presence of another factor. Impacts have been formulated based on literature and were validated by a survey and partly by experiments.

(C1.2) Entities, Measures, and Evaluations
Entities represent specific parts of the software in focus. They have been specified based on the factors of the quality model with the aim of representing a system so that the factors can be evaluated. Measures have been derived from literature where possible or newly defined. The calculation of measures is defined based on the specified entities. Evaluations interpret measure values to evaluate factors and impacts resulting from evaluation results.

(C2) Clounaq approach
The **Clou**d-**na**tive **q**uality evaluation approach is implemented in the Clounaq tool[3]. It has been implemented as a web-based tool that provides features to view the quality model, create software architecture models, and evaluate created models with the quality model. Modeling is a manual approach, but the evaluation is then done automatically.

(C2.1) Modeling approach
Based on a review of available ADLs, TOSCA [208] has been selected as a formal modeling basis. A TOSCA extension is provided as a profile to model software architectures using the entities specified in C1.2. Architectural models can be created and modified with a graphical editor in the Clounaq tool.

(C2.2) Evaluation approach
The implemented evaluation approach calculates the measures specified in C1.2 for a modeled software architecture and evaluates each factor based on the measure values. The evaluation results are returned in a graphical way with details on the calculations. Deriving actual decisions from that on how an application should be built or changed, however, is up to the developer(s) building an individual application.

Table 1.1.: Publications presenting core contributions of this work

Ref.	Title	Year	Cont.	Method(s)
[169]	Towards a Quality Model for Cloud-native Applications	2022	C1.1, C1.2	Literature-based
[71]	An Evaluation of Modeling Options for Cloud-native Application Architectures to Enable Quality Investigations	2022	C2.1	Literature-based, Prototyping
[174]	Cloud-Native Architectural Characteristics and their Impacts on Software Quality: A Validation Survey	2023	C1.1	Questionnaire-based Survey
[171]	Formulating a quality model for cloud-native software architectures: conceptual and methodological considerations	2024	C1.1, C1.2	Literature-based
[168]	Clounaq - Cloud-native architectural quality (Tool demonstration)	2024	C2.1	Prototyping
[145]	Evaluating Cloud-Native Deployment Options with a Focus on Reliability Aspects	2024	C2.1	Use Case-based
[172]	An Experimental Validation of Architectural Measures for Cloud-Native Quality Evaluations	2025	C1.2, C2.2	Experiment
[173]	Evaluating Cloud-native Software Architectures with Clounaq	2025	C2.2	Use Case-based

[3] `https://clounaq.de`, visited 2025-10-20

1.4. Outline

The structure of this work is aligned with the hypotheses presented in Section 1.2 and the contributions listed in Section 1.3. As it can be seen in Figure 1.2, relevant foundations are presented in sections 2.1, 2.2, and Section 2.3. These are the basis for the core results presented in chapters 4, 5, and 6. How these results have been achieved from a methodological perspective is described in Chapter 3. Chapter 4 presents the quality model in detail, that means the specific elements, how they have been derived from literature, and how they can be made operational and evaluated for a given software architecture. Chapter 5 presents the modeling approach by justifying the chosen level of abstraction, the significance of the chosen entities, and their relationships. Chapter 6 describes the implemented approach and tooling for the quality evaluation of cloud-native software architectures. It thus combines the results from Chapter 4 and Chapter 5 and integrates them to enable the application of the quality model to a specific modeled software architecture. Finally, in Chapter 7 implications for theory, practice, and future work are derived from the gained results, followed by a discussion of related work in Chapter 8. The conclusion in Chapter 9 completes this work.

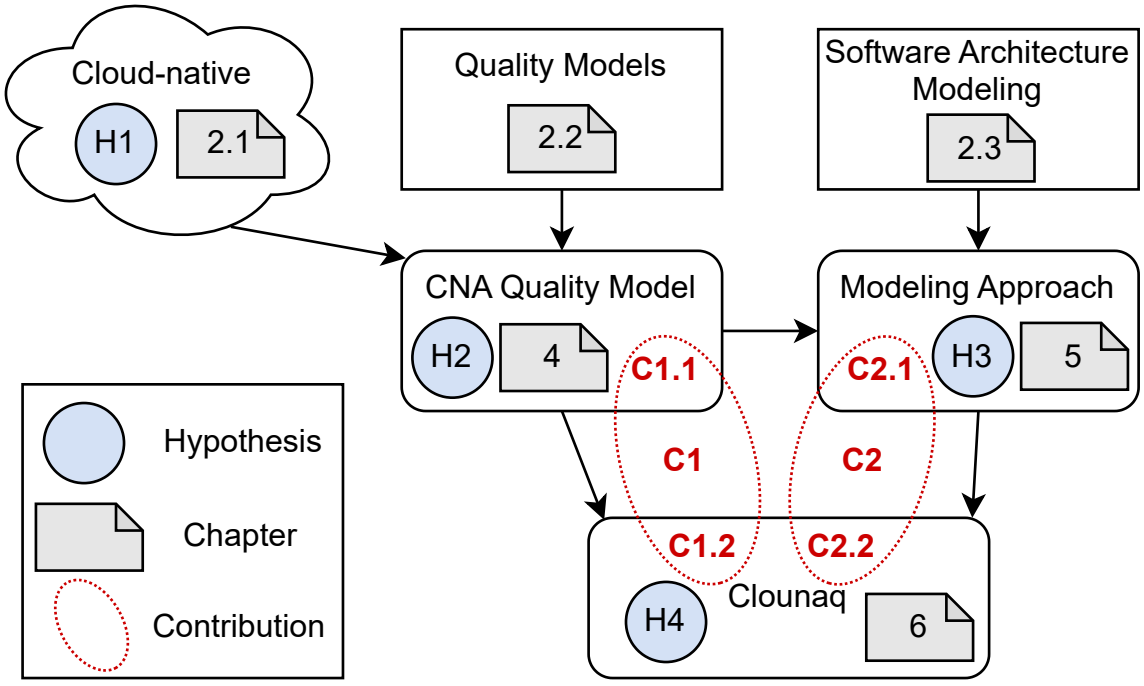


Figure 1.2.: Conceptual outline with hypotheses and contributions linked to chapters

Dad, what are clouds made of?
Linux servers, mostly.

Internet meme

2. Conceptual Foundations

Because the aim of this work is to formulate a quality model for cloud-native software architectures, the term cloud-native is explained in Section 2.1. Fundamentals on quality models are introduced in Section 2.2, and how software architectures can be represented is explained in Section 2.3.

2.1. Cloud-native

In this section, hypothesis 1 *("Cloud-native spans a broad range of aspects")* is supported.

The topic of cloud-native needs to be understood in the context of the topics from which it emerged. The two main foundations for cloud-native are cloud computing on the one hand and service-oriented computing on the other hand. These are described in the following sections 2.1.1 and 2.1.2 to then show how cloud-native applications emerged from them in Section 2.1.3.

2.1.1. Cloud Computing

The emergence of cloud computing as a possibility for executing software is often associated with the year 2006 when Amazon launched its Elastic Compute Cloud (EC2) offering [151, 301]. From then on, cloud computing became more and more important and has also grown to an important field of academic research. Cloud Computing has been defined by the National Institute of Standards and Technology (NIST) in 2011 as follows:

> **Definition 2.1 (Cloud Computing)**
> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [190]

A core aspect here is that computing resources are made available from a *cloud provider* to a *cloud consumer* [79, Chapter 4]. Different deployment models for cloud infrastructure are listed by Mell and Grance [190], including the possibility to operate an on premises *private cloud* where provider and consumer are the same organization. However, the most common model is that of the *public cloud*. In the public cloud model, the cloud provider aggregates work in a shared pool of resources and

can thus make more efficient use of resources, saving overall cost [302]. For the cloud consumer, a major advantage is the direct availability of a large amount of resources on-demand without an upfront investment [79, Chapter 3]. Based on the definition and these aspects, a set of essential characteristics for cloud computing systems can thus be identified which are listed by Mell and Grance [190]:

- *On-demand self-service*: A cloud consumer can request resources anytime in an automated way.
- *Broad network access*: Cloud services are available over the network and can be accessed from heterogeneous devices.
- *Resource pooling*: Resources are provided to consumers in a multi-tenant setup with no control over the actual physical resources for the consumers.
- *Rapid elasticity*: The amount of provided resources can be rapidly increased or decreased in an automated way.
- *Measured service*: Resource usage is measured and controlled in a traceable way for both providers and consumers.

These core characteristics have remained accurate and relevant even while cloud platforms have evolved and developed further. What has evolved, however, are what Mell and Grance call *service models* [190]. Service models describe the types of resources that are made available to consumers. In the beginning, basic computing resources, assignable to the *Infrastructure as a Service (IaaS)* service model, for example Virtual Machines (VMs), were offered by cloud providers [99]. Resources from this service model are provided with extensive administrative access to cloud consumers. Thus, a high customization is possible, but it also means that cloud consumers take over administrative responsibilities to a large extent. By 2010, platform services emerged [99], which are assigned to the *Platform as a Service (PaaS)* service model. For *PaaS* resources, the cloud provider takes over more operational responsibility, but also limits the extent to which resources can be customized and controlled by consumers [79]. That has the advantage for consumers that, by relying on pre-configured, ready-to-use resources, deployment is accelerated. Also, additional features such as automated scaling can be offered by the provider [79]. In addition, *Software as a Service (SaaS)* is seen as the third major service model [79, 190]. In SaaS, the consumer books a complete application that is intended for end-user usage and has very limited possibilities for configuration and customization.

The focus in this work is on resources from the IaaS and PaaS levels and in the context of these, cloud platforms have evolved significantly. One major aspect is the emergence of a range of additional service models. They are the result of new technological advancements or the expressed needs of cloud consumers. One example is the advent of containers [214] as a technology. On the one hand, containers have transformed the way PaaS offerings are operated [214]. And on the

other hand, they led to the emergence of a new service model: Container as a Service (CaaS) [290]. Another example are cloud functions and the topic of serverless computing [289]. Specifically, Function as a Service (FaaS) has been introduced as an additional service model [289, 290]. Some authors see FaaS as a new service model separate from PaaS [182]. Erl et al. [79] see IaaS, PaaS, and SaaS still as the main service models and other, new, service models as being sub models. This evolution in service models has also led to the proposal of new categorizations of service models. One approach simply refers to service models as Anything as a Service (XaaS) in order to be flexible when introducing new service offerings [135]. In any case, many different service models have emerged and can be booked by cloud customers, specific to their individual requirements. They differ in the type of resource that is offered, but also in the level of operational and administrative effort that is either taken over by the provider or left to the consumer. Reviews of cloud computing service models have summarized the state-of-the-art continually [117, 275], but the field continuously evolves.

Another major aspect that has evolved over the years is how resources can be provisioned and managed. While originally the typical way to request cloud resources was via a web interface, APIs were quickly made available by cloud providers. Resource provisioning and management can therefore also be automated based on such APIs. That lead to the development of additional tooling which makes use of the APIs provided by cloud providers. One major aspect covered by such tools is provisioning and deployment which is nowadays supported by Infrastructure as Code (IaC) tools [235, 253]. Other aspects related to the management of cloud resources after they are provisioned are monitoring [193], scaling [53], cloud-edge orchestration [44], or performing upgrades [310].

The number of available service models, functionalities of additional cloud tools, and different cloud providers with their own specific focuses has lead to a complex space for cloud environments and applications running in them. Structured approaches for developing software for the cloud are therefore important. It is not only about whether to use cloud computing or not, but about how specifically it should be used. One step in this direction has been the formulation of patterns, such as the cloud computing patterns presented by Fehling et al. [84].

To summarize, cloud computing is an established technological concept that is, however, still evolving. Major areas of evolution are the types of resources offered by cloud providers in terms of service models and the approaches and tools for provisioning and managing cloud infrastructures together with the applications running on them. The essential characteristics for cloud computing formulated by Mell and Grance [190], nevertheless, still capture the core aspects of cloud computing. Thus, they remain relevant for understanding also more recent developments in cloud computing.

2.1.2. Service-oriented Computing

Service-oriented computing, as the second topic of major importance for cloud-native, has emerged in the early 2000s [218]. The fundamental question to which service-orientation is seen as the answer is how business functionalities can be implemented with software in a way that is effective, cost-efficient and flexible in the long term. In the context of this work, the description of Papazoglou et al. is adopted as a definition:

> **Definition 2.2 (Service-oriented Computing (SOC))**
> *"Service-oriented Computing (SOC) is the computing paradigm that utilizes services as fundamental elements for developing applications. [...] Services are self-describing, open components that support rapid, low-cost composition of distributed applications."* [218]

Individual services are autonomous and provide specific functionalities which can be accessed in a platform-independent way [219]. Thus, services are *interoperable* and their functionalities can be *reused* for different business requirements [78]. *Interoperability* and *reusability* are the basis for *service composition* [218, 258]. It describes the concept of implementing business processes as a composition of relevant service functionalities that, if used together, form a larger, more complex functionality or complete application. Service composition can be dynamic in the sense that specific services are not known a priori, but discovered during execution based on their published service descriptions [258]. By the services being autonomous and platform-independent themselves [219], they can be operated in flexible environments and they can evolve independently of other services. Thus, conceptually, service-oriented computing as a paradigm is based on the idea of composing applications from autonomous, distributed services.

Thomas Erl has conceptualized service-orientation (as a basis for SOA, see 2.1.2.1) in more detail in a series of books. First, with a technology focus in 2005 [75]. Then, with a focus on core concepts in 2007 [76], also as a companion to his SOA Design Patterns books published in 2008 [77]. And finally, in an updated version considering newer technologies and microservices (see 2.1.2.2) in 2016 [78].

Throughout these books, Erl has characterized service-orientation through eight core design principles [75, Ch.8][76, Ch.4][77, Ch.4][78, Ch.3]:

- *Standardized Service Contract* - Published technical interfaces of services should be standardized, consistent and appropriately scoped.
- *Service Loose Coupling* - Services should be loosely coupled to reduce dependencies and allow for independent service evolution.
- *Service Abstraction* - Implementation details of services should be hidden by providing more abstract interfaces.
- *Service Reusability* - Services and their functionalities should be reusable in different contexts.

- *Service Autonomy* - Services should be autonomous regarding their environment and resources.
- *Service Statelessness* - State should only be introduced when necessary to make services scalable.
- *Service Discoverability* - Services should be easy to find and understand allowing for service reuse.
- *Service Composability* - Services should be designed with composition in mind.

By following these design principles, service-oriented solutions aim at reaching the strategic goals of *Increased Intrinsic Interoperability*, *Increased Federation*, *Increased Vendor Diversification Options*, and *Increased Business and Technology Domain Alignment* [76, Ch.3][77, Ch.4]. Fulfilling these goals leads to the following strategic benefits: *Increased ROI*, *Increased Organizational Agility*, and *Reduced IT Burden* [76, Ch.3][77, Ch.4]. The benefits of service-orientation are clear, but putting the listed principles in practice with actual implementations to reach these benefits is challenging and requires corresponding technologies, methods and significant domain knowledge. Regarding such specific technologies and methods, a substantial amount of research and development has been done over time. This has been documented in a structured way by Papazoglou et al. in 2008 with their *research roadmap* for service-oriented computing [219]. Papazoglou et al. differentiate between *Service foundations*, *Service composition*, *service management*, and *Service engineering* as key areas for which they reviewed the state-of-the-art and formulated research challenges [219]. Several challenges have been addressed by now, but also the state-of-the-art has shifted. Seemingly promising technologies have not reached the expected relevance. For example, a major topic was web services technology and the so-called WS-* stack [298] which provided implementations and standards for developing services. It is by now, however, not widely used anymore, as Schermann et al. state that in practice REpresentational State Transfer (REST) is preferred over XML-RPC and SOAP is used rarely [254]. This has also been investigated by Pautasso et al. [220] with a comparison between RESTful web services and "big" web services. Another technology tightly connected with service-oriented computing were Enterprise Service Buses (ESBs) which were seen as state-of-the-art by Papazoglou et al. [219]. They are, however, not of large interest anymore, as stated by Schermann et al. [254].

Bouguettaya et al. have reviewed service-oriented computing from a more recent perspective with their *service computing manifesto* [39]. They criticize that services computing has not yet reached its full potential, because aspects going beyond technology have not been properly addressed yet. They do however build a bridge to the topic of cloud computing by highlighting the fact that by using cloud technologies for deploying services, published services have seen a significant increase [39]. The potential synergies between SOC and cloud computing

were already seen earlier by Wei and Blake [301] who highlight that cloud computing is well suited to facilitate service deployment and management. As the most recent review, Plebani et al. [228] revisit the history of SOC and state that fundamental SOC functionalities have become a commodity in many areas. Research now focuses on applying SOC in new areas of interest or more specific technologies. Overall, SOC is a more abstract, but fundamental, paradigm, which has been studied comprehensively in academia and which has significantly impacted the way applications are built in practice.

2.1.2.1. Service-oriented Architectures

Closely connected to SOC is the topic of SOA. In comparison to SOC, SOA, as an architectural model, describes the technical application and implementation of SOC in an enterprise context. Thus, SOA is more concrete than SOC. The term SOA, however, has also been used extensively in research and practice, sometimes with different meanings, sometimes interchangeably with SOC. In the context of this work, the viewpoint of Erl is adopted who sees SOA as a service-oriented technology architecture that is implemented through a specific combination of technologies, products, APIs, and infrastructure which is unique within each enterprise [76, Ch.3]. In his earliest book, Erl also provides a concise definition for SOA:

> **Definition 2.3 (Service-oriented Architecture (SOA))**
> *"SOA is a form of technology architecture that adheres to the principles of service-orientation. When realized through the Web services technology platform, SOA establishes the potential to support and promote these principles throughout the business process and automation domains of an enterprise."* [75]

However, due to the mentioned technology-focus, he included *Web services technology* [298] explicitly which should be seen only as an example rather than a necessity from today's perspective. To support the previously stated goals of SOC, the specifically chosen combination, and thus architecture, within an enterprise should establish the following four characteristics according to Erl [77, Ch.4][78, Ch.4]:

- *Business-Driven* - The technology architecture is aligned with the business architecture and evolves over time driven by the business architecture.
- *Vendor-Neutral* - The architecture is not based on a single proprietary vendor platform, but allows for combination and replacement of technologies from different vendors.
- *Enterprise-Centric* - A meaningful part of the enterprise is represented to achieve reuse across previously separated areas.
- *Composition-Centric* - Services can be composed repeatedly and in different ways to enable agility regarding business change.

The overall SOA within an enterprise can be differentiated further into the following types of architectures for which different technologies might be relevant and decisions can be made separately [77, Ch.4][78, Ch.4]:

- *Service Architecture* - The architecture of a single service, covering its implementation and interface.
- *Service Composition Architecture* - The architecture of a set of services composed together, covering how the composition is implemented and managed.
- *Service Inventory Architecture* - The architecture supporting a set of related services, acting as a boundary for these services and potentially requiring standardization for aspects of services within the inventory.
- *Service-Oriented Enterprise Architecture* - The architecture of the whole enterprise, covering and integrating all services, compositions, and inventories.

Further advise on how a SOA can be implemented in an enterprise is often provided in the form of *patterns* [77, 247]. These can be assigned to one of the mentioned types of architecture and provide more concrete guidance on how to design and integrate services according to the principles of SOC. The impact of SOA on quality has also been investigated in research in different ways, including quality models [90, 107] and with a focus on the impact of patterns [96]. To summarize, SOA is the topic of applying SOC to an enterprise context, specifically for building enterprise applications with a service-oriented architectural model and corresponding technologies. The core service-oriented principles have proven successful, but the applied technologies and methods have evolved over time.

2.1.2.2. Microservices Architectural Style

More recently, in comparison to SOA, the topic of *microservices* has emerged. First introduced in 2011 [67], the topic has been popularized by an article from Lewis and Fowler [166]. It has gained interest and broad resonance especially in practice [67, 296], but over time also in research [9, 43, 92, 129]. The microservices architectural style is often considered as an evolution or successor of SOA [9, 24, 67, 221, 222]. While SOA, specifically the variant based on the Web services technology stack and ESBs, was popular with a lot of companies striving to implement it, many were not able to get the desired benefits out of it [24]. One reason was the perceived complexity of SOA and its corresponding technologies [67]. In contrast to the focus of SOA on *service reuse* and *composition* at that time, the focus of the microservices architectural style is on *service independence* [49, 126, 228]. Service independence increases isolation between services for better replaceability and maintainability [24]. In this context, the complexity of communication through

ESBs and Web services protocol stacks [49] was abandoned in favor of more lightweight communication styles, especially REST based on Hypertext Transfer Protocol (HTTP) or asynchronous messaging [24]. Cerny et al. identify a shift from smart routing capabilities of ESBs in SOA to smart services in the microservices architectural style [49], aligned with the *Smart endpoints and dumb pipes* principle listed by Lewis and Fowler [166]. By limiting the scope of a service to a small cohesive set of functionalities, the focus is shifted from dynamic usage of services to dynamic replacement, since smaller services are easier to replace, if required.

Increased service independence is enabled by upcoming Continuous Integration / Continuous Delivery (CI/CD) technologies [161] which facilitate automation. Thus, responsibility for developing and operating services can be given to DevOps [161] teams that independently evolve services. In turn, CI/CD-based deployments are also facilitated by independent services. Furthermore, service independence is supported by the trend towards virtualization via containerization [214] and especially cloud computing environments for deploying and operating containerized services [126, 214]. Through cloud computing and automation, deploying and operating a larger set of distributed services has become feasible. Thus cloud computing can be seen as a core enabler of the microservices architectural style [43, 92, 126, 321].

Apart from the microservices characteristics that differentiate it from other SOA implementations, there are actually a lot of commonalities between SOA and microservices. Especially, the rapid and flexible development of enterprise applications as a core SOC goal [218] is associated with microservices [126]. The microservices architectural style embraces core SOC principles like *loose coupling* or *service contracts*. This supports the opinion that the microservices architectural style is another particular implementation of SOA following the SOC paradigm. Zimmermann has investigated this aspect in detail and by distilling knowledge from the microservices descriptions by Lewis and Fowler [166], as well as by Newman [202], formulated the following seven *microservices tenets* [321]:

1. Single-responsibility, independent services which encapsulate data and logic while exposing fine-grained interfaces, typically based on RESTful HTTP or asynchronous messaging.

2. Business-driven development practices and pattern languages such as Domain-Driven Design [82] are employed.

3. Cloud-native application design principles are followed (e.g. IDEAL [84]: isolated state, distribution, elasticity, automated management, loose coupling).

4. Multiple computing paradigms (e.g., functional and imperative) and storage paradigms (e.g., relational and NoSQL) are used: polyglot programming and persistence.

5. Lightweight containers are used to deploy services.
6. Decentralized continuous delivery is practiced during service development.
7. DevOps-based, lean, but holistic and largely automated approaches to configuration, performance and fault management are employed.

As it can be seen from these tenets, the microservices architectural style conforms to core SOC design principles (see Section 2.1.2): *Standardized Service Contract* through tenet 1, *Service Loose Coupling* through tenets 1 and 3, *Service Abstraction* through tenet 1, *Service Reusability* through tenet 2, *Service Autonomy* through tenets 1,2,3,4, and 5, *Service Statelessness* through tenet 2.

Also, the core SOA characteristics stated by Erl [78, Ch.4] (see Section 2.1.2.1) are supported by the microservices architectural style: *Business-Driven* through tenet 2, *Vendor-Neutral* through tenets 4 and 5, *Enterprise-Centric* through tenet 2 and *Composition-Centric* through tenets 1 and 3. Furthermore, the importance of cloud computing as an enabling technology for microservices is evident.

Despite the numerous benefits associated with the microservices architectural style, there are also challenges, especially based on the inherent distribution of microservices-based architectures [24]. Another core challenge is service granularity [126, 296], meaning how to scope services or how *micro* services should be. It is mainly influenced by the specific business requirements a system is supposed to fulfill. Thus, it is hard to find an answer from a technical point of view. Nevertheless, this question has been researched comprehensively, especially from the viewpoint of migrating to a microservices architecture [1, 20, 42, 94]. Since many companies already had existing applications when microservices became popular, the question of how to migrate and split these applications into microservices-based architectures is relevant, too.

To summarize, the microservices architectural style can be seen as the contemporary approach and technological solution for implementing service-oriented systems. It has evolved from the previous technological approach to SOA and was enabled by other driving forces such as CI/CD and cloud computing in particular.

2.1.3. Cloud-native Applications (CNAs)

The term *cloud-native* and its history have been investigated based on a systematic mapping study by Kratzke & Quint [151]. According to them, it occurred at the very beginning of cloud computing around 2006, then disappeared, and was used increasingly again starting around 2015. This can also be seen in Figure 2.1 where the number of found publications per year when searching for specific terms is plotted. The search terms cover the previously presented topics: *cloud computing* (Section 2.1.1), *service-oriented* (Section 2.1.2 and Section 2.1.2.1), *microservices* (Section 2.1.2.2), and *cloud-native*. It shows how interest in service-oriented computing started around 2000 with a peak in 2009 and the impact of cloud computing

starting in 2006. The more recent topics of microservices and cloud-native started to gain interest around 2015. This year is of particular importance, because in June 2015 the first commit of Kubernetes was pushed to GitHub[4] and in July 2015 the CNCF was announced, backed by numerous high-profile companies[5]. The major impact from practice is also noted by Kratzke & Quint [151]. Cloud-native can be seen as the latest trend in the area of enterprise applications. It builds on the topics of cloud computing as such and service-oriented computing with the microservices architectural style as its most recent practical realization.

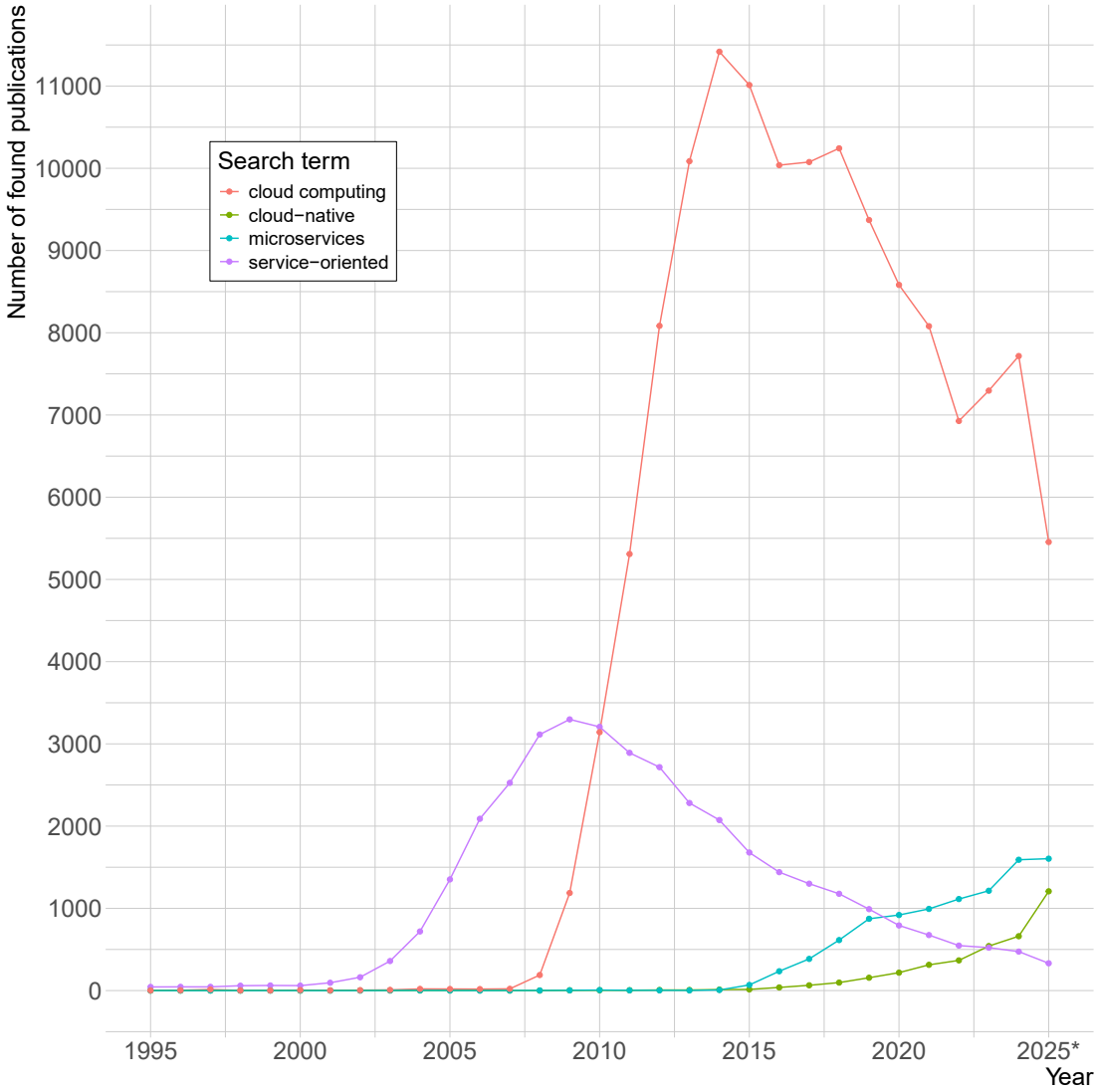


Figure 2.1.: Number of found publications over time per search term[6]

[4] https://kubernetes.io/blog/2024/06/06/10-years-of-kubernetes/, visited 2025-10-20

[5] https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-technologies/, visited 2025-10-20

[6] Source: https://www.semanticscholar.org. Search invoked with filter on "Computer Science" as a field of study. *Data for 2025 is only until 2025-10-20 and thus preliminary.

In fact, cloud-native can be seen as a product of microservices and the continuous evolution in cloud computing. Driven by the goal of implementing applications in a fast, agile, and cost-efficient way, the cloud-native style of building applications relies on the concepts developed in these areas. This has already been described by Leymann et al. who state that "*the term "native cloud application" suggests an application to fully benefit from all the advantages of cloud computing*" [167]. To do so, applications need to be build in a specific way. Leymann et al. [167] refer to the IDEAL properties stated by Fehling et al. [84]: **I**solation of state, **D**istribution, **E**lasticity, **A**utomated Management, and **L**oose coupling. Fehling et al. [84] describe in detail, how these properties embrace the essential cloud characteristics (see 2.1.1): Because cloud environments are inherently distributed and heterogeneous to enable *resource pooling*, applications should consist of separated components that can be distributed among cloud resources. Elasticity takes advantage of the *rapid elasticity* and *measured service* characteristics to optimize resource usage depending on the current workload in a cost-efficient way. Isolating state is closely related to elasticity and also embraces the resource pooling characteristic. By differentiating strictly between stateless and stateful components, they can be assigned separately to corresponding cloud resources which are pooled depending on their specific characteristics. Because cloud consumers have no control over the actual physical resources, automated management is required to react to changing environments. It is enabled through the *on-demand self-service* and *measured service* characteristics which enable management components to observe a cloud environment and react on-demand in an automated way. Because resources and components constantly change, loose coupling facilitates the management of individual components and mitigates impacts of failing components.

Also, Toffetti et al. highlight that "*there are several advantages in embracing the cloud*" [281] which differentiates cloud-native applications as "*adapted and transformed to leverage the dominant cloud computing models*" [281], from other applications just deployed in the cloud. However, Toffetti et al. also note that experience in cloud computing has shown that "*one should plan for failure, striving to produce resilient applications on unreliable infrastructure*" [281], because cloud environments are built from commodity hardware for cost effectiveness. This adds another perspective to cloud-native, also included by Pahl et al. [216] in their list of architectural principles for cloud software. Besides *service-orientation* and *virtualization*, Pahl et al. highlight *uncertainty* and *adaptivity* as core influencing factors for cloud-native software architectures. Adaptivity is required to deal with "*variable resources, system errors, and changing user characteristics*" [216] and adaptation processes need to handle uncertainty in provisioning times, monitoring data and underlying resources. The core point is that cloud computing has inherent challenges and uncertainties that applications need to deal with and adapt to. Cloud-native applications are built for this, as also stated by Davis: "*Cloud-native software is designed to anticipate failure and remain stable even when the infrastructure it's running on is experiencing outages or is otherwise changing*" [60, Ch.1].

In addition to the already presented definitions 1.1 and 1.2, which have focused rather on technical characteristics, it is thus possible to define cloud-native applications also in a non-technical way. This definition entails the possibility of being a valid definition in the long-term, even with the continuous evolution in cloud computing technologies:

> **Definition 2.4 (Cloud-native application (non-technical))**
> *A cloud-native application takes full advantage of the benefits offered by cloud computing while at the same time tolerating inherent difficulties of cloud environments.*

To achieve this, the current approach is to apply the technologies and methods described by the definitions of the CNCF [56] and by Kratzke & Quint [151]. These technologies and methods embrace the core cloud computing characteristics. Using a set of microservices that are developed and managed independently is supported by the *on-demand self-service* characteristic through which DevOps teams can deploy and operate a service independently of a dedicated operations team. This also covers other aspects of automation, which is essential for managing microservices, and relies on the *on-demand self-service* characteristic. The distributed communication over the network between microservices is facilitated through *broad network access*. Elasticity, especially horizontal scalability associated with microservices, is supported by the *rapid elasticity* characteristic. Self-contained deployment units, especially containers, fit well to the *resource pooling* characteristic which includes no control over the actual physical resources. Extensive control over physical resources is also not needed, because containers have all required dependencies included. The execution characteristics of containers also fit well with *resource pooling*, because they are designed to be lightweight and to be run in a clustered way independent of the underlying hardware. The practice of using immutable infrastructure is supported through *on-demand self-service* and *rapid elasticity*, because instead of manipulating existing infrastructure, new infrastructure can be rapidly provided to replace previous infrastructure. The *measured service* characteristic is exploited by more recent service models such as FaaS. FaaS is also used increasingly in microservices-based architectures [24]. Likewise, for monitoring and observability, *measured service* is a core characteristic.

While cloud-native applications embrace the core cloud computing characteristics, their emergence would not be possible without the ongoing evolution in cloud computing and associated technologies, especially virtualization. Gannon et al. highlight this by describing the need for what they call a *service fabric* [99]. It is the platform on which the actual services are operated and which includes additional functionalities, like monitoring and automated restarts [99, 138]. This is supported by Kratzke and Peinl [150] who formulated a cloud-native application reference model in which their cluster layer represents this fabric. It is typically realized through a container orchestrator with Kubernetes currently being the most commonly used one. Additionally, cloud computing providers now of-

fer managed supporting services, like databases, caches, or message brokers as stable and trusted resources [99]. Consumers pay for them by usage which further facilitates application development and deployment. A currently emerging and researched topic is multi-cloud [8] where the idea is to rely on multiple cloud providers for application development and operation. Multi-cloud native applications further exploit the *resource pooling* characteristic in the sense that not only the physical resources are pooled by a cloud provider, but also the virtualized resources from different cloud providers are pooled.

In the evolving area of cloud-native applications, keeping an overview is challenging. This is also recognized by Wei et al. [300], who assembled a comprehensive mapping of cloud-native practices to quality aspects and available tools and technologies. The number of tools and technologies gathered by them in the context of cloud-native, together with the discussion of cloud-native originating from the topics of microservices and the continuous evolution in cloud computing, demonstrate the broad scope of the topic of cloud-native. This scope covers the aspects **application design** (service scope and communication), **technology selection** (deployment technologies and supporting services), **deployment infrastructure** (cloud service model and infrastructure management), and **operation** (monitoring, scaling, handling of failures), as discussed in the problem statement (1.2).

2.2. Quality Models

Ensuring the quality of software has always been a core part of software engineering [35, 85]. In earlier days, starting in the 1970s, individual metrics have been used to measure for example the complexity of source code [85]. But it has become apparent that relations between characteristics of software and its quality are more complex. Quality can be considered in various dimensions and may be difficult to evaluate, because "*software does not directly manifest quality attributes. Instead, it exhibits product characteristics that imply or contribute to quality aspects*" [68]. Furthermore, quality depends on the specific context of a considered application [141]. This dependence on the context of an application also means quality is, in the end, always relative to the requirements stated for an application. Thus, a structured approach is needed for evaluating and ensuring the quality of software. For this reason, quality models are used in software engineering to enable a structured evaluation of the quality of a system according to quality attributes deemed important for it [85]. This is shown in an abstract way in Figure 2.2.

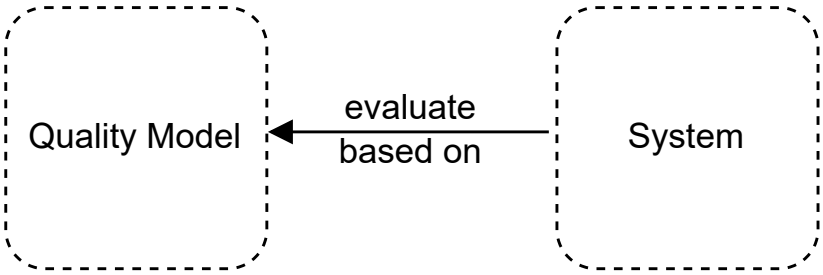


Figure 2.2.: A quality model is used to evaluate a system

A quality model, once available, can be used to evaluate many systems (if applicable to them), enabling also a comparison of systems. There are different variants of quality models, but in general all are based on the following definition:

> **Definition 2.5 (Quality Model)**
> *"defined set of characteristics, and of relationships between them, which provides a framework for specifying quality requirements and evaluating quality"* [122]

The set of characteristics, relationships and all other parts of a quality model can be called *elements*, as shown in Figure 2.4. The term *element* is adopted from Dromey [69] who used it to describe the parts of his generic quality model. Because the term element is more abstract, it covers different types of quality models that can be defined and structured in various ways. Lochmann & Goeb [176] have developed a unifying model for software quality which aims at covering all different types of quality models. From their point of view, the elements of a quality model can be conceptualized according to the meta-model in Figure 2.3

A property is a tuple of an attribute and an entity where the attribute characterizes the entity [176]. An attribute can be a quality aspect, for example Availability or a specific characteristic, for example the size. An entity can be a system as

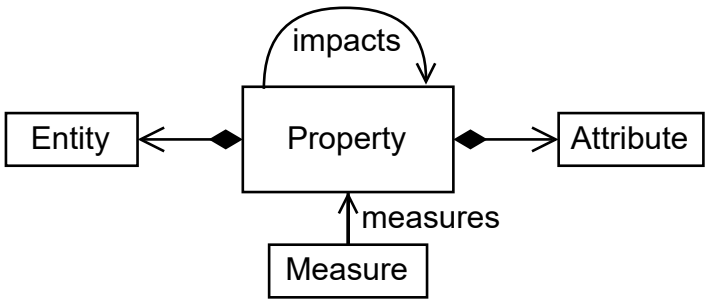


Figure 2.3.: Elements according to the unifying quality meta-model by Lochmann & Goeb [176]

a whole or, for example, a class within object-oriented software. Through impacts relationships can be formulated so that specific characteristics can impact more abstract quality aspects. Although it narrows the types of elements, the unifying meta-model still provides flexibility regarding the kind of elements used to specify a quality model and aligns with definition 2.5. It also aligns with the generic quality model of Dromey [69], because product properties and quality attributes are both represented through properties and the "*means of linking them*" [69] are given through the impacts.

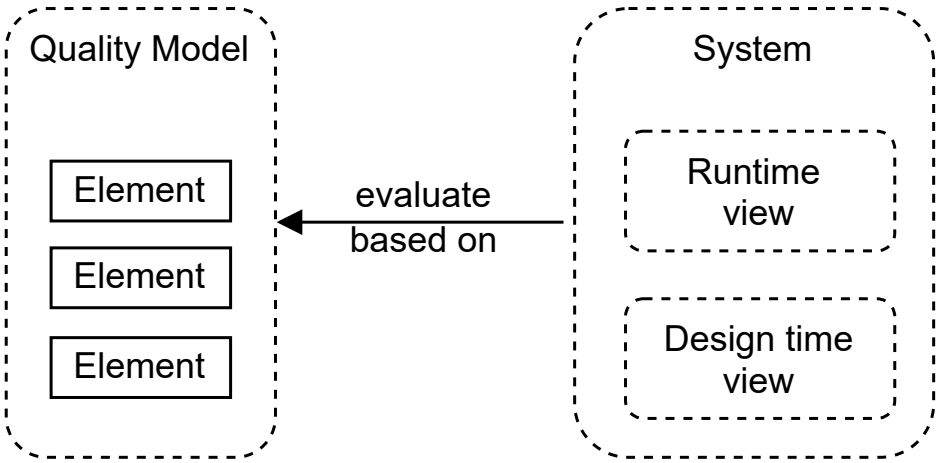


Figure 2.4.: A quality model contains a set of elements and systems can be evaluated at design time or runtime

Furthermore, it is important to make a distinction between the views that can be applied for evaluating a system (see Figure 2.4). An evaluation can focus on the internal characteristics of a software which are evaluated by analyzing the internal implementation of a system (*Design time view*). Or an evaluation can focus on external characteristics of a software which can only be evaluated at runtime when observing its behavior (*Runtime view*). This distinction is in line with the ISO 25010 [122] standard which considers internal and external quality. Also, Kitchenham et al. [142] differentiate between "externally visible properties" and "internally visible properties". For both types of characteristics impacts on quality aspects can be stated. Typically, impacts of external characteristics are more intuitive, for

example, the externally visible uptime has a positive impact on availability, the higher it is.

Based on these core aspects of quality models, the different types of quality models that exist and how they can be differentiated is discussed in the following.

2.2.1. Types of Quality Models

Quality models can be differentiated based on their *view*, their *structure*, and the *domain* they focus on. A basic distinction for types of quality models is the *view* on quality that is applied. Kitchenham & Pfleeger [141] proposed a detailed differentiation between the following views:

- *transcendental view*: Quality can be recognized but not defined.
- *user view*: Quality is rated from the user perspective.
- *manufacturing view*: Quality is a result of the development process.
- *product view*: Quality is based on specific software product characteristics.
- *value-based view*: Quality depends on how much a customer is willing to pay.

Dromey [69] differentiates mainly between process-focused models (*manufacturing* view) and product-focused models (*product* view). He states that too much focus had been put on process-focused models and argues for a focus on product-focused quality models. The ISO 25010 standard [122], in turn, puts a focus on the *product* view, by providing quality aspects for *product* quality models, and the *user* view, by providing quality aspects for *quality in use* models. Currently, most quality models that have been developed are product quality models (e.g., [165]), as stated by Ferenc [85] who reviewed the state-of-the-art of quality models. This is supported by later reviews [95, 203, 309] with Nistala et al. [203] stating that only 8% of quality models found by them consider the *process* view (e.g., [63]). The predominant view on quality thus focuses on the software product itself, as does the quality model presented in this work.

Nevertheless, there are still different ways for *structuring* product quality models. The options range from simpler approaches, such as checklists, over rule-based approaches used in static analysis tools, to more complex quality models aiming at explaining and analyzing quality in an integrated way [176]. The most common option regarding more complex quality models are so-called *hierarchical quality models*. They structure the elements of a quality model in an interconnected hierarchy [85] as shown in Figure 2.5.

The specific hierarchy depends on the quality model but can, for example, cover elements from lower-level measures to higher-level quality aspects. Hierarchical quality models can integrate and interrelate multiple quality aspects and enable a detailed evaluation of software quality. To build such a hierarchy, however, a well-founded theory is required which can explain the stated relationships within

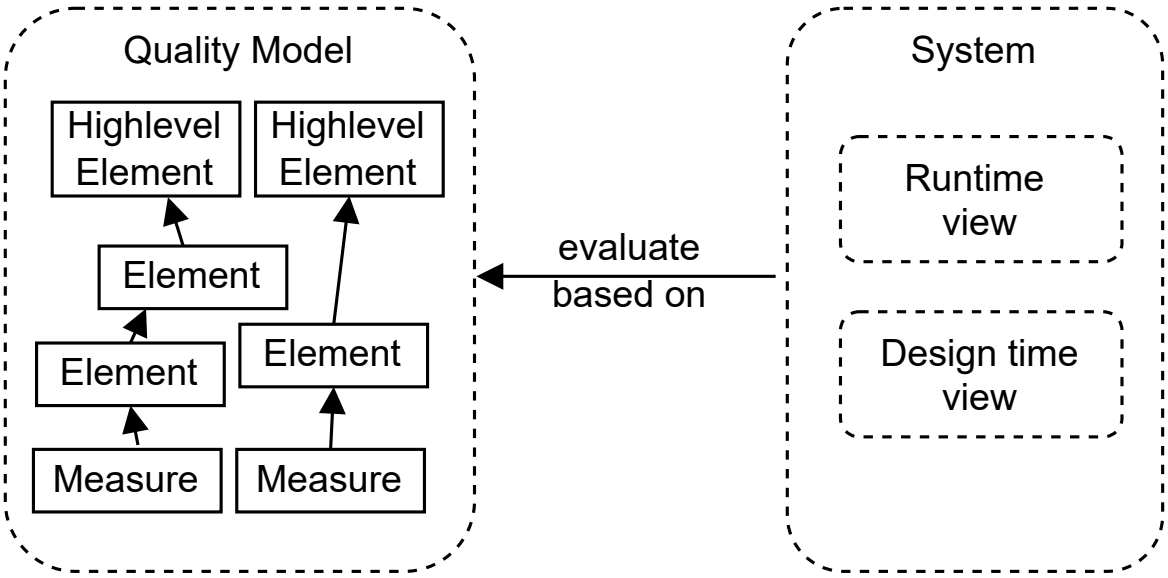


Figure 2.5.: In hierarchical quality models elements are structured in a hierarchy

a hierarchy. From a historical perspective it can be seen that early quality models for example from Boehm [35], McCall [186], or Dromey [68] have been formulated and described in comprehensive research reports using experience and logical implication. As another example, Bansiya et al. [22] state for their hierarchical quality model that in an object-oriented system the characteristic of coupling impacts extendability. According to them, high coupling has a negative impact on extendability. Coupling can be quantitatively measured by counting for each class to how many other classes it is directly related [22] and thus extendability can be evaluated. Hierarchical quality models typically contain the elements proposed by Lochmann & Goeb [176] in their unifying meta-model (see Figure 2.3). Properties can be used to analyze systems in a qualitative way, while measures add quantifiability. Regarding the structure of quality models, hierarchical quality models are the most common approach. This is because they enable an analysis of more complex relationships between characteristics of a system and quality aspects. These relationships, however, must be comprehensible and relevant.

Finally, quality models can be differentiated based on the covered *domain*. Because models inherently abstract from actual systems and have a specific focus based on which this abstraction is applied, quality models are formulated and used specific to certain domains while abstracting from other aspects. Over time, quality models for different domains have been developed. These include quality models for object-oriented systems [22], embedded systems [184], Web services [213], SOA architectures [107], microservices [231], or more recently machine learning systems [260].

The quality model developed and presented in this work is a hierarchical product quality model focusing on the domain of cloud-native applications. As such it can be understood from the presented theory on quality models and be differentiated from other quality models according to the mentioned aspects.

2.2.2. Relation to Patterns, Best Practices, and Smells

Using quality models is one approach within software engineering for evaluating and ensuring quality in a system. It has to be seen in the context of other approaches which are used extensively as well and which can partly be integrated.

One important concept is *Design Patterns*. Patterns "*solve specific design problems*" [98] and "*help designers reuse successful designs by basing new designs on prior experience*" [98]. Design Patterns have become popular in software engineering with the book by Gamma et al. [98] on Design Patterns for object-oriented software being a major influencing factor. According to them, a Design Pattern covers four essential elements [98]: A *pattern name*, for reference and easier communication; a *problem*, describing when to apply a pattern; a *solution* describing the required elements for implementing a pattern; and *consequences* which result from the application of a pattern. Design Patterns have been captured and cataloged for various areas, such as enterprise integration via messaging [116], workflows [287], or more recently service APIs [323]. An important aspect is that patterns are not invented but identified and collected from practical experience. Also in the areas of cloud computing [84, 125], microservices [242, 277], and cloud-native applications [60, 120] patterns have been collected.

Design Patterns are closely related to quality models in the sense that they shape the internal characteristics of a system and thus influence external characteristics of a system. Quality aspects that are impacted by the usage of a pattern are part of the consequences stated for a pattern, but are not the only thing to consider. Some works directly focus on the relationships between patterns and quality aspects, for example regarding microservices [285]. The impacted quality aspects are, however, not the primary focus of design patterns. Their main focus is providing a proven solution to a certain kind of problem. From the point of view of this work, patterns can be reflected in quality models through certain formations of entities with specific attributes. When an architecture is evaluated, the fact that a certain pattern was used, can be recognizable in a quality model through corresponding architectural measures which measure a property. The quality aspects that a property impacts should then be the same as those also targeted by the pattern. Therefore, patterns are also adopted in this work, especially from practitioner books [60, 118, 120, 242], but also from academic literature [58, 204, 205, 313, 314]. Quality models, in contrast to patterns, aim at a more integrated point of view including relationships and dependencies between characteristics of a system.

Similar to, and sometimes identical with patterns, are *best practices*. Best practices can be understood as rules for how certain things should be done. In comparison to patterns, best practices cover a broader scope. That means best practices consider not only technical solutions, but also aspects considering the development process or deployment practices. Best practices are typically applied globally in a system while a pattern is applied only to a certain part of a system. Examples for best practices cover API design [244], cloud application design [105] or cloud security aspects [239]. While best practices also aim at improving the quality of a

system in a certain dimension, they are typically used as a list of practices to apply. The individual practices are not placed in a more integrated context, as hierarchical quality models do. Checking and ensuring a list of best practices could, however, also be seen as a basic form of performing a quality evaluation.

Finally, another concept related to the quality of systems are *(architectural) smells*. Smells describe certain characteristics of a system that have a negative connotation and may lead to problems. Thus, the existence of smells may decrease the quality of a system. While similar to best practices in the sense that smells are based on a set of rules, an architectural evaluation regarding smells does not focus on positive aspects but the existence of negative aspects. The goal is to find and eliminate them. Smells are typically eliminated by *refactoring* [322]. The identification and refactoring of architectural smells has been investigated for component-based architectures [100], microservices-based architectures [230, 276], cloud architectures [322], and more recently in a more general, comprehensive review by Mumtaz et al. [199]. In comparison to quality models, the focus on architectural smells generally takes a different perspective. However, the insights provided by work on architectural smells can also be adopted in quality models.

In summary, different approaches for improving the design or quality of software exist. Hierarchical quality models can be seen as a more advanced option in comparison to the other presented concepts, because a focus is set on an integrated perspective. This might, however, also make quality models more complex to use. The knowledge embedded in design patterns, best practices, and smells can also be integrated in quality models which is why literature considering these concepts are also a main foundation for this work. In practice, all approaches are applied, because the common goal is to improve software in a structured way.

2.2.3. Formulating Quality Models

Parts of this section have been taken from [171].

Formulating a quality model refers to the process of creating a new quality model. In essence, this means that the elements of a quality model need to be specified and set into relation. Different approaches exist to do this and some constraints need to be considered.

Before starting with the actual formulation of a quality model, the aspects described in Section 2.2.1 should be decided upon. That means a desired *view* for the quality model should be chosen, the types of elements which make up the *structure* of the quality model must be defined, and the targeted *domain* needs to be stated. Guided by these aspects, the actual quality model can then be formulated. For this, Dromey has proposed a 5-step process [69] aligned with his generic quality model elements (see Section 2.2):

1. Identify high-level quality attributes
2. Identify product components

3. Identify and classify quality-carrying properties of components
4. Propose axioms for linking product properties to quality attributes
5. Iterative refinement based on evaluation

Dromey argues that at the beginning a top-down approach can be used to decide on the high-level quality attributes and decompose them on one further level. Further decomposition of quality attributes is discouraged by Dromey, because it does not provide practical value [68]. Instead, it is better to then continue from the specific product components and proceed from the "*tangible to the intangible*" [68]. This can be seen as a bottom-up approach of analyzing product characteristics and linking them to quality attributes. According to Dromey, it is important to formulate adequate, direct links between product characteristics and quality attributes [68, 69]. Furthermore, different product characteristics can impact the same quality aspect and are not mutually exclusive in their effect [69].

The formulation process proposed by Dromey is not an absolute rule. Instead, as the variety of quality models and used formulation approaches [95, 203] show, there is room for maneuver. Especially, as the last step of Dromey's process shows, iterative improvement is important and thus formulated elements from a previous step might also be revised or removed in a following step. Wagner et al. have documented their formulation process for a quality model for Java and C#-based projects based on the Quamoco meta-model [295]. They combined a top-down with a bottom-up approach and used iterations for formulating and validating their quality model. These iterations included consideration of existing tools and literature, expert workshops and continuous reviews. Overall, Wagner et al. started with collecting existing measures and then formulated product factors based on them as well as the entity hierarchy as required by these factors [295]. Afterwards, Wagner et al. mapped the product factors to quality aspects by specifying impact relationships and finally formulated evaluations for completing the quality model. As a more recent example, Siebert et al. [260] have reported their construction process in detail which is overall comparable to the one proposed by Dromey, but varies in certain more detailed aspects.

Independent of the concrete process steps used, the resulting quality model should satisfy certain characteristics. AL-Badareen et al. [5] have investigated rules and constraints to consider when formulating hierarchical quality models. One exemplary rule is that there should be only one path from an element to a certain top-level quality aspect [5] in order to keep evaluations regarding this quality aspect unambiguous. The most important characteristic, however, for a quality model to be useful and of good quality itself, is that the conditions and relations it describes reflect reality. An exact reflection of reality is impossible, because a model, per its definition, has to abstract from the real world. But it should be ensured that software quality relations are modeled as realistic as possible. This is also stated by Wagner et al. who emphasize the importance of a *rationale* [294] for each stated

impact relation between elements of a quality model. The respective rationale included in a quality model is a result of the approach used for formulating a quality model. Different approaches have been listed by Moody [194], including for example: theory-based (deductive), experience-based (codification), observation-based (inductive), or consensus-based (social). Based on the available possibilities, a suitable choice needs to be made on the approach used for formulating a quality model. In any case, the chosen approach should be documented and explained so that it can be reproduced and evaluated by others. In the end, the elements of a quality model need to be formulated in a clear and understandable way and the relationships between elements need to be accurate and proven.

Part of an understandable and accurate quality model is also the structure that is used. This refers specifically to the types of elements used in a quality model. Over time, different formalism and corresponding meta-models have been proposed for formulating the elements of a quality model, as documented in the review by Galli et al. [95]. Meta-models define types of quality model elements to prescribe a syntax for specifying quality models with clear semantics for the corresponding elements. In particular, the Quamoco meta-model [294, 295] has been the result of an extensive research effort on hierarchical quality models. It can be used to formulate new quality models based on it. Quamoco is presented in more detail in Section 2.2.3.1, because it is also used as a basis for the quality model presented in this work. If quality models are formulated based on a common meta-model they are also comparable and corresponding methods and tools might be reusable for different quality models. In a similar way, standards have been developed for specifying quality as such and quality models in particular. Notable standards are the ISO 9126 [121] and its successor ISO 25000 [122]. They define high-level quality aspects and sub-aspects into which the aspects can be decomposed. By relying on the same standards, quality models are also comparable.

Applying a structured and thorough approach for the formulation of a quality model is important for it to be meaningful. To be useful, also in practice, another important aspect is tooling. Effective tooling support is essential for the adoption of quality models as stated by Yan et al. [309]. Furthermore, a quality model should support different phases of the software development process. Nistala et al. [203] differentiate between different phases in which quality models can be used (planning, realization, documentation, and assessment) and then map the type of elements of quality models to the phases in which they are used. Through this, they identify significant differences between quality models regarding the completeness of their included elements. A completely formulated model supporting all phases is more likely to be successful.

In summary, formulating a meaningful and useful quality model requires effort. By relying on existing approaches and formalism, the formulation process can be facilitated and substantiated. The formulation process should be structured and documented.

2.2.3.1. Quamoco

Quamoco is a meta-model for hierarchical quality models that has been developed in a comprehensive research effort [294, 295]. The meta-model defines a set of element types which can be used to create hierarchical quality models. These element types and their relations are depicted in Figure 2.6 and explained in the following.

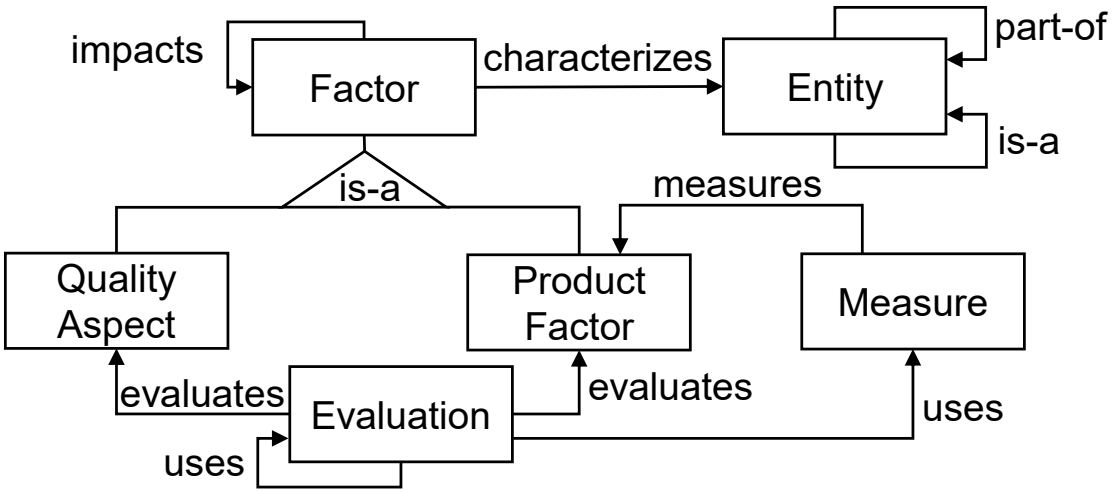


Figure 2.6.: Essential elements of Quamoco [294]

The core element type of the Quamoco meta-model is a *factor* which can be differentiated into either a *quality aspect* or a *product factor*. Quality aspects represent the rather abstract quality characteristics for example from the ISO 25000 standard [122]. Product factors capture tangible characteristics of a software product which should be measurable for a specific software based on *measures*. Through the *impact* relationship, it is possible to define a hierarchy of factors where quality aspects define the top-level elements and product factors are structured below. An impact relationship can be defined coarsely, for example, as being positive or negative, describing how a factor impacts another factor, if that factor is present in a system.

More detailed statements about how product factors impact quality aspects, also using quantitative approaches, can be made using *evaluations*. Evaluations are defined for quality aspects and product factors and use the available measures as well as evaluation results of other factors. Evaluations can be considered as rules that configure the exact way in which the quality of a software system can be evaluated per factor and through the defined hierarchy, in the end for each high-level quality aspect. Evaluations can also be set in relation to each other, for example, to enable aggregations of evaluations. To connect the quality model with the actual software under consideration, *entities* are used to represent different parts or components of a software system. Entities are defined using `part-of` and `is-a` relationships, effectively enabling the representation of a software architec-

ture tailored to the quality model. The extent to which product factors are present in a certain software system is characterized through the specific set of entities which describes this software system. Based on how these entities are structured and which properties they have, the existence of product factors can be measured This enables, in the end, a quality evaluation of a software system. When applying the Quamoco meta-model to the abstract view on hierarchical quality models from Figure 2.5 to create a quality model, the depiction in Figure 2.7 results. It shows how the elements of the Quamoco meta-model are used to formulate a quality model and how these are connected to an actual software system to enable its evaluation.

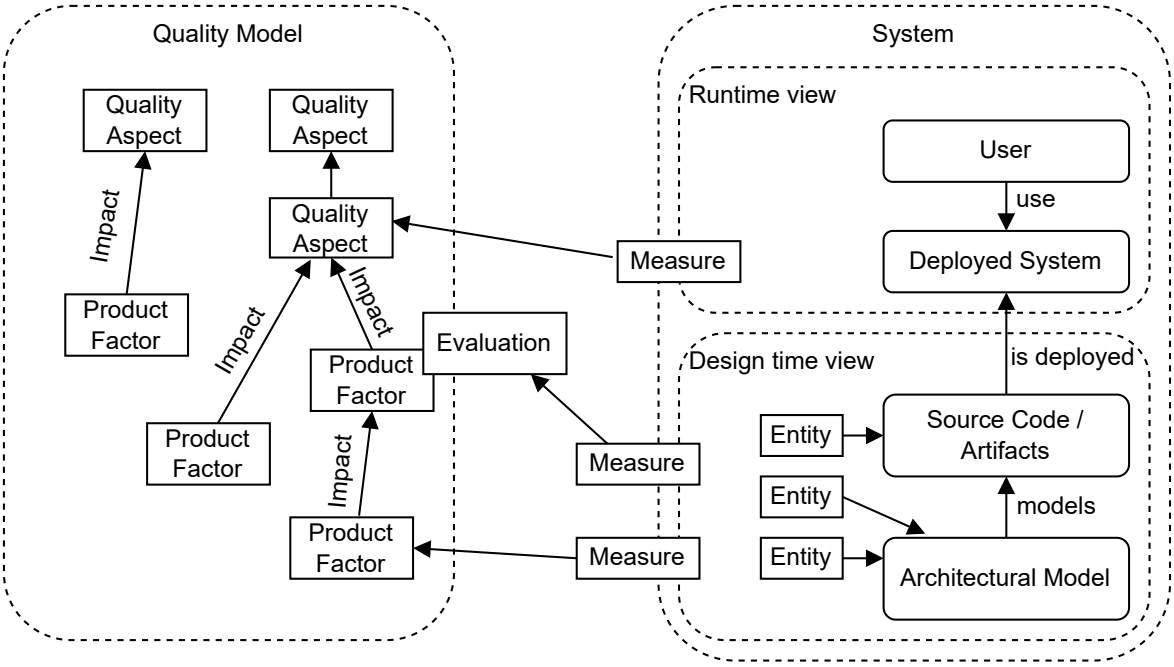


Figure 2.7.: A Quamoco-based quality model to evaluate a system

The main connection to the actual system is through the representation of the system through entities. Depending on the domain and scope which should be used to evaluate a system, entities can be used for representing parts of the source code or other artifacts directly. Or they can be used to represent parts of a more abstract architectural model which in turn models the actual implementation. Based on the available entities and their properties, measures are defined which need to be suitable for evaluating corresponding product factors. Quamoco itself is not focused only on the design time, but can also be used at runtime. The Quamoco elements can be used for specifying measures that are calculated at runtime and which can in turn be used to evaluate product factors and quality aspects from a runtime perspective as well. For the rest of this work, the terms introduced by Quamoco as shown in Figure 2.6 are used for describing the approach for formulating and validating the quality model presented in this work.

Wagner et al. [294] state that factor evaluation results should be reported on a scale between 0 and 1. 0 indicates that a factor is not present in a system while 1

indicates that a factor is fully present in a system. Such a quantitative evaluation of factors therefore requires measures which can be used to evaluate the presence of factors in a system in a meaningful way. The topic of measure formulation is non-trivial and therefore explored in the following section in more detail.

2.2.3.2. Measure Formulation

Measures, and in particular architectural measures, are essential for quantitative quality evaluations of software architectures. The term *measure* is used throughout this work for consistency, for what is called both measure and metric in literature. Herbst et al. note that: "*Whereas in mathematics, the term metric is explicitly distinguished from the term measure (the former referring to a distance function), in computer science and engineering, the terms metric and measure overlap in meaning and are often used interchangeably.*" [113]. The decision to use measure consistently is aligned with the Quamoco meta-model (Figure 2.6) and other literature on measures [196].

Measurement in software engineering is challenging and discussed in detail by Morasca [196]. Because software engineering is a human-intensive discipline, repeatability of experiments (e.g., developing a software product) is limited. Thus, several aspects of software measurement are more similar to measurements in social sciences rather than so-called "hard" sciences [196].

An important distinction for measures in software engineering is whether they measure an internal or an external property of an entity [196]. Internal properties (e.g., size or complexity) are "easy" to measure because they only rely on knowledge of the artifact of interest itself. External properties (e.g., qualities such as maintainability or reliability) cannot be measured based solely on the artifact itself. Instead, they require the consideration of the environment and interactions between the artifact and its environment. External properties are thus more difficult to measure. However, external properties are more interesting from a practical point of view which is why typically the "*measurement of an internal attribute of a software artifact (e.g., the size of a software design) is interesting only because it is believed or it is shown that the internal attribute is linked to some external attribute of the same artifact (e.g., the maintainability of the software design)*" [196].

Furthermore, a measure always measures a property of a specific entity [196] which is in line with the unifying meta-model by Lochmann & Goeb [176] (see Figure 2.3). Thus, a measure also always needs to be formulated on the basis of a specific entity.

The challenging aspect for formulating measures is that they need to be appropriate for measuring the property of interest for the corresponding entity and that they are meaningful regarding the eventual goal, typically predicting an external property. Morasca discusses two basic approaches for formulating measures [196]: using measurement theory or using an axiomatic approach.

Measurement theory is the preferable approach according to Morasca. According to measurement theory, a measure is a mapping from the so-called *Empirical*

Relational System to the so-called *Numerical Relational System* [196]. The Empirical Relational System covers intuitive knowledge about properties of entities while the Numerical Relational System quantifies this knowledge. Because in general an arbitrary assignment of measures is possible, an important aspect from measurement theory is that a measure needs to satisfy the *representation condition* [196]. This means only fully sensible measures must be chosen which clearly represent the property that should be quantified while all measures that contradict intuition must be discarded. Different kinds of scales (nominal, ordinal, interval, ratio) can be used for measures [196]. Independent of what kind of scale is used, a statement made based on a measure must be true even if a different scale is used for the same measure.

However, Morasca also states that using measurement theory can often be too restrictive in its constraints, impeding the formulation of measures. While the mentioned concepts of measurement theory should be striven for, a more relaxed approach, used more often in software engineering, is the axiomatic approach [196]. The axiomatic approach relies on the formulation of axioms that codify knowledge accepted as common sense. Using axioms as a basis, measures can be formulated that comply with these axioms and are thus grounded on a set of axioms which can become commonly accepted over time, if the formulated measures turn out to be meaningful. Furthermore, measures defined using the axiomatic approach do not need to be discarded because of the more strict measurement theory. Instead, they can be taken up at first and then be refined over time [196].

In addition, Morasca presents an approach for measure formulation named GQM/MEDEA (Goal Question Metric / MEasure DEfinition Approach) that builds on the Goal Question Metric (GQM) approach [25]. It includes helpful guidelines for formulating measures, such as the orientation towards a specific goal or the consideration of knowledge about the environment. However, it is focused more on the specification of measures that can be used directly for linking them to external properties, rather than applying them in the context of a an architectural quality model.

Also building on the GQM approach, Becker et al. [27] discuss the formulation of measures specifically for cloud services. Referring to Erl [80], they state the following properties as requirements for formulating measures: *quantifiability*: a clear and appropriate unit of measurement is set; *repeatability*: repeated measurements provide identical results; *comparability*: units of measurement need to be standardized; *easy obtainability*: measurement is based on a non-proprietary, common form. While these properties focus on cloud services and rather on external properties of cloud services, they can also serve as guiding characteristics for formulating measures for internal properties, i.e. architectural measures.

Especially for internal properties, various measures for the same property can be formulated, but their usefulness, also in comparison to each other, might not be obvious at first. To tackle this problem, different approaches for measure selection have been proposed in literature. Lochmann et al. [177] experimented

with machine learning based predictors to reduce the number of measures used in a predictor without decreasing the performance too much. They show that, while the best performance can be achieved with more measures, it's also possible to use more focused quality models with fewer measures that still provide an acceptable performance. Their approach, however, is based solely on a statistical evaluation of performance without considering individual measures in detail. Van der Bent et al. [288], in contrast, while building a quality model for Puppet[7] code, decreased the number of measures used in their quality model through the calculation of correlations. Based on a dataset of repositories, they calculated all formulated measures and pair-wise correlation values between measures. When two measures were highly correlated, one could be dropped in favor of the other. Both approaches can be used in the process of formulating measures to select meaningful measures from a set of potential measures.

As mentioned, measures are always defined for a certain entity. If a quality model covers a range of entities structured in an entity hierarchy and it should be possible to provide evaluations for different entities, the question arises of how measures may be aggregated or disaggregated. This has also been considered in literature by Mordal et al. [197] who discuss different methods of aggregation from simple averages, over weighted averages to more complex approaches. An important aspect is that improvements made in individual parts of a system might not be recognizable in aggregated evaluations for the whole system. The aggregation method should thus be selected and implemented carefully.

Apart from measure aggregation along an entity hierarchy, also aggregation along a quality model hierarchy needs to be considered, as discussed by Ulan et al. [283]. There, the same problem exists: information might be hidden by using simple or even weighted average approaches. Instead, Ulan et al. introduce their aggregation approach based on a joint distribution of a set of observed software artifacts. Software products are then ranked according to this distribution. However, the approach requires an existing large-enough data set for creating such joint distributions.

Overall, when formulating and selecting measures for quality models, existing literature and knowledge should be considered. For example, Arvanitou et al. [14] have linked architectural measures to quality attributes in an extensive mapping study. By relying on their results, the meaningfulness of measures can be evaluated. In a more subjective approach, Zimmermann [320] has also discussed the availability and usefulness of architectural measures. While he emphasizes the overall usefulness of presented measures from his experience, he states that the measures always need to be evaluated with care and in the context of the specific application at hand. He furthermore states, that it is an open research question of how the usefulness of architectural measures could be evaluated in a general way [320].

[7] `https://www.puppet.com/`, visited 2025-10-20

In summary, the formulation of measures should be done in a structured approach, focused on the specific goal of measurement and considering the corresponding entities in focus. While an initial formulation can be guided by an approach and aim to satisfy certain requirements, the meaningfulness of measures needs to be continuously evaluated and validated when they are used for quality evaluations. This is also considered in more detail in Section 2.2.5.

2.2.4. Interpreting Measures for Formulating Evaluations

The mentioned work by Arvanitou et al. [14] links architectural measures directly to quality aspects. In hierarchical quality models, especially those based on Quamoco, architectural measures are used to evaluate product factors first. In Quamoco, specific evaluation elements are therefore specified which use measure values to evaluate the presence of a product factor within a system or based on another entity. Thus, evaluation elements need to interpret measures in a meaningful way to evaluate factors accordingly.

The interpretation of measures depends on various aspects such as the value range, the existence of empirical data to which specific values can be compared to, or the level of abstraction on which a measure might be aggregated. A challenge is that measure values calculated for entities can have different kinds of value ranges. Mordal-Manet et al. calls values of such measures “raw metrics” [198]. These raw values need to be interpreted by mapping them into a format consistent to the chosen evaluation approach that provides evaluation results for product factors. A decision has to be made which values to use for such a mapping. Discrete values or ordinal ranks provide an intuitive understanding and reduce the complexity of quality evaluations. However, discrete mappings may hide the effects of smaller changes to a system. Such changes may not be reflected in changes of discrete values, although on the level of the raw measures they would be recognizable [198]. An alternative are mappings to continuous values, but then a meaningful mapping function is required. Independent of which kind of mapping is chosen, any mapping is an interpretation. Interpretations are often based on common sense and experience [198, 246] rather than empirically validated methods. Ideally, however, measure value interpretations should be grounded in empirical insights about collections of comparable systems in order to know about “typical” values of measures [86, 308]. Based on such known distributions of measure values, untypical measure values could be identified. It would then also be possible to derive thresholds for measure values relative to other systems, as proposed by Alves et al. [10], Oliveira et al. [212] or van der Bent et al. [288]. If no empirical insights about measure values are available, interpretations should at least be aligned with intuitive reasoning, expert knowledge, and common sense [284].

Similar to the calculation of measures, also their interpretation is based on specific entities. If calculation and interpretation are too fine-grained regarding the entity hierarchy, the results may be overwhelming, impeding a meaningful eval-

uation. If calculation and interpretation are too coarse-grained, important details may be hidden. This is also described by Mordal et al. [197] in the context of metric aggregation: If measure values of entities are aggregated using a simple average, this may smooth out measure values that would otherwise characterize a problematic condition.

Formulating evaluations for a quality model, thus, requires a consideration of these challenges in order to make quality models meaningful. Overall, in practice quality models should not only provide understanding, but also guidelines for improvement [197]. The aspects of understanding and deriving insights from evaluation results are mainly supported by the chosen form and presentation of evaluation results.

2.2.5. Validating Quality Models

Parts of this section have been taken from [170, 171].

As already mentioned, the formulation of a quality model should be done iteratively. That means, after elements have been formulated, they are not final, but may be revised in a later iteration. The key approach for deciding on whether to keep or revise an element is validation [309].

In the context of this work, **validation** is understood as the methodological approach for ensuring the validity of a theory, in this case the theory underlying a quality model. It is distinguished explicitly from **evaluation**. Evaluation is understood as the approach of using a quality model for evaluating the quality of a software. While the terms validation and evaluation may be used differently or even interchangeably in literature, this is the way they are used in this work.

As stated in Section 2.2.3, a quality model necessarily abstracts from reality, but the abstraction should nevertheless reflect reality as closely as possible for meaningful evaluations of software. Through validation, this reflection of quality impacts actually observable in reality is ensured. In other words, the theory underlying the formulation of quality model elements is validated using suitable approaches.

Validation is closely related to formulation and a core aspect is how the theory used for the formulation of a quality model is substantiated. Different types of rationale can be used for substantiating a theory based on the applied methodology. Types of rationale are: (1) **logical/axiomatic** where theory is based on logical reasoning or common knowledge, (2) **literature-based** where theory has been developed and stated elsewhere in scientific literature, (3) **empirical** where theory is substantiated via empirical evidence, for example from interviews with experts or structured surveys among a larger set of participants. By validating a theory, or a quality model in this case, it is ensured that its theoretical concepts can be confirmed and explained also by a different type of rationale. For example, if a quality model has been formulated based on literature, different types of rationale could be an empirical perspective from domain experts or quantitative data from

benchmarking experiments. Existing literature on performing validations of quality models has been reviewed by Yan et al. [309] who focus on commonly used model elements. They found *expert opinion* to be a commonly used approach (39%), but in the same amount of works, no explicit validation was done (39%) [309]. Regarding earlier quality models, AL-Badareen et al. [5] state that quality models are often formulated in a subjective manner. They refer to a finding from Kitchenham and Pfleeger [141] that "*software quality models suffer from a lack of rationale for the relationships between quality characteristics and how the lowest levels properties are composed into an overall assessment of higher level quality characteristics*" [5]. More recently proposed quality models, especially domain-specific quality models, increasingly use empirical validation approaches. These include expert opinions [184, 185, 260] or questionnaire-based surveys [104, 139, 189]. This is supported by Moody [194] who states that empirical methods are important for the acceptance of conceptual models (which is a super-set of models used in software engineering and therefore also includes quality models) in practice.

There are several aspects of a quality model that can be validated and validations can in turn be done in several ways based on the mentioned types of rationale. In an investigation of validation approaches and scopes for quality models [170], an important aspect that emerged is that not all types of validation are possible during all phases of the creation process of a quality model. Especially in an early phase, when only quality aspects, product factors, and their respective impact relations are formulated, a complete validation of the quality model based on an assessment of software is difficult because of the missing measures and evaluations. Nevertheless, it is possible to validate these formulated elements with a suitable approach that includes an additional type of rationale. For example, if a quality model has been formulated based on experience by experts, it could be validated using an experimental approach relying on measures of an actual software system. Or, if a quality model has been formulated based on theory, e.g., relying on literature, it could be validated through an empirical survey. Empirical approaches, such as expert interviews or surveys may not require the complete formulation of all elements of a quality model. These can, for example, be used to validate factors by asking for the clearness of their descriptions or their applicability to software systems. Or they can be used to validate the formulated impact relationships between factors by asking for the type and strength of impacts certain factors have on other factors.

The phase of the creation process of a quality model also influences the scope for which a validation can be done. Different scopes of validation are depicted in Figure 2.8 and labeled V1 to V7.

It can be seen that the validation of factors themselves (V1), the impacts between factors (V2) and relative weights of impacts (V3) can be done solely based on factors proposed for a quality model. Therefore, these can also be done early in the creation process of a quality model (early in the sense that only these elements are defined). For example, using interviews with experts, Gerpheide et al. [104] have

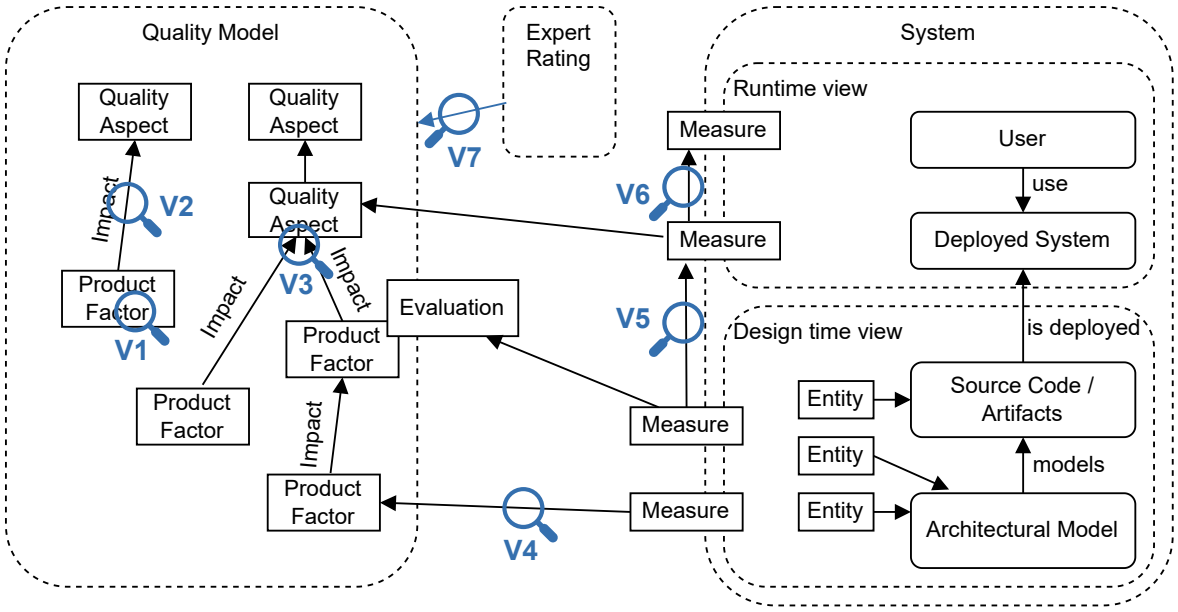


Figure 2.8.: Validation scopes for quality model validations [170]

defined and validated factors (V1). Mayr et al. [184] early validated their model (V1, V2, V3) regarding comprehensibility, appropriate level of abstraction, and consistent classifications through conducting multiple workshops. Lampasona et al. [158] present an approach to rate the minimality and completeness of factors (V1) using interviews with experts. Surveys among practitioners have been used by Mehmood et al. [189] and Gerpheide et al. [104] to validate impacts (V2) and their weights (V3) from product factors on quality aspects by calculating the agreement of respondents regarding the existence and type of impacts. Similarly, Khomh et al. [139] validated impacts and their weights (V2, V3), although they investigated design patterns instead of product factors.

Other validation scopes require the formulation of all elements of a quality model. The validation of the meaningfulness of measures for product factors (V4) and the validation of relationships between architectural measures and runtime measures considering the system itself (V5) or the users using a system (V6) can only been done if the formulated quality model includes corresponding measures. A validation of the complete quality evaluation process enabled by a quality model based on an expert rating (V7) also requires the quality model to be completely formulated. For example, Braeuer et al. [40] validated measures by comparing them with previously gained measurements (V4). Not explicitly focusing on quality models, but focusing on the correlations between measures, Calero et al. [46] have validated architectural measures for relational database systems by relating them to measures for quality aspects (V5). Similarly, Singh et al. [264] have done this with architectural measures for object-oriented systems (V5), Kvam et al. [155] for legacy systems, Yu et al. [311] for web services, and Knoll et al. [146] for microservices-based systems. An investigation of correlations between external measures and user measures (V6) has been done by Jung et al. [131] considering

the ISO 9126 standard [121]. Lastly, Kläs et al. [143] validated factors based on diversification (V1) of measures and overall validity of a quality model through a comparison with expert ratings (V7). In addition to the validation of measures in the specific context of quality models, Misra et al. [192] have proposed a framework for validating new measures which considers formal validation as well as empirical approaches.

Based on the concepts for formulating quality models presented in Section 2.2.3 and the options for validating quality models presented in this Section 2.2.5, an own structured approach can be defined for creating a quality model. An iterative approach of formulation, validation, and refinement through new formulations can be taken and validation can be applied repeatedly and from different perspectives. As a general rule it can be stated that the more validations from different perspectives are performed, the more confidence exists that the quality model is appropriate and usable. Validations during the creation process, also called *formative evaluations* [111], should complement validations of a completely formulated quality model to ensure the validity of a quality model early on. Overall, the validation of quality model elements can be seen in the context of the already mentioned "*representation condition*" from measurement theory as also done by Galli et al. [95] who state that "*properties of real-world entities measured are mapped in numeric representations in such a way that the numeric representations are equivalent to the reality*" [95].

2.3. Software Architecture Modeling

As the aim of this work is to evaluate software architectures of cloud-native applications, the topic of software architectures and how they can be represented, specifically in the context of cloud computing, is introduced in this section. Software architecture is a core part of the software engineering discipline [268] concerned with approaches and methods for developing complex software systems in a structured way. Kruchten et al. [152] in 2006 reviewed the history of software architecture as a research topic and state that it started to emerge in 1990 when software systems became more and more complex and the need arose to design and analyze systems at a higher level of abstraction. How systems could be described at such a higher level of abstraction was formalized by Perry & Wolf [226] in 1992. They propose a model that depicts software architecture as a tuple of *element*, *form*, and *rationale* [226]. Elements can be: *processing elements* which are components that process and transform data, *data elements* which contain the information that is used and transformed, and *connecting elements* which connect the other elements, for example messages being sent from one component to another. The architectural *form* encompasses properties which constrain elements and *relationships* which constrain how elements may be connected. *Rationale* provides a motivation and explanation for the choice of specific elements. According to Perry & Wolf [226], it is an integral part of software architecture which shows how design

decisions satisfy constraints and requirements of a system. By describing the software architecture of a system using this model, the software can be evaluated and evolved in a structured way. This is also picked up by Wasserman [297] who positions software architecture within the software engineering discipline and argues that software architecture is a key aspect for the quality of a system, especially its maintainability. And Sommerville states that "*The non-functional requirements depend on the system architecture — the way in which these components are organized and communicate.*" [268]. According to him, designing the software architecture of a system should be the first step in a software design process as it involves fundamental decisions about how to structure and organize a system. Fundamentally changing the software architecture later in the software development process is expensive and should be avoided [268].

Meanwhile, the term software architecture has also been defined in a standard:

> **Definition 2.6 (Software Architecture)**
> *"fundamental concepts or properties of an entity in its environment and governing principles for the realization and evolution of this entity and its related life cycle processes"* [123]

This definition includes the already mentioned aspects together with a consideration of the environment in which a system is operated and its underlying life cycle process. Because design decisions for a software architecture are influenced to a large extent by the specific domain and environment of a system, software architectures have also been investigated and researched specific to certain areas. One of these areas is cloud computing where the decisions of which cloud service models to use and how to build an application based on them are integral. Therefore, Pahl et al. [216] have refined the software architecture of a cloud application as a *cloud architecture* and define it as follows:

> **Definition 2.7 (Cloud Architecture)**
> *"an abstract model of a distributed cloud system with the appropriate elements to represent not only application components and their interrelationships, but also the resources these components are deployed on and the respective management elements"* [216]

Core elements of a cloud architecture are thus the components of an application and the cloud resources which are used for deploying these components. Kratzke and Peinl [150] have investigated cloud architectures, specifically considering cloud-native applications. They formulate a layered cloud-native stack as a reference architecture for designing cloud applications. These layers can be used as architectural viewpoints and encompass from the lowest layer to the highest: the *physical host* layer, the *virtual host* layer, the *node* layer, the *cluster* layer, the *(micro-)services* layer, and the *application* layer. On each layer, the elements making up the architecture on this layer can be designed and analyzed.

As shown by the model of software architecture described by Perry & Wolf [226] and by the definition for cloud architecture, software architectures are typically described as models that abstract from the actual system. They contain only the information relevant for the architectural design and analysis. Representing software through models has a long history in software engineering already [268]. Models can take different forms, but for a structured usage, typically explicit modeling languages are used. Modeling languages, in comparison to programming languages, are typically defined based on a meta-model and their abstract syntax is not context-free, but already includes certain semantics [73]. Modeling languages specifically for representing software architectures are called ADLs. Many ADLs have been introduced over the years [188, 250], but only a few became popular and are in widespread use [181], such as Unified Modeling Language (UML) [38]. One reason is that models abstract from the software and the suitable type of abstraction differs depending on the specific purpose of which there can be many. Another reason is discussed by Lago et al. [157]: Often informal, ad-hoc types of architectural descriptions, such as simple box and line diagrams, are used for discussions among software architects and developers. Such informal descriptions are easy to use and fine for building an understanding of an architecture or high-level discussions. But because of their missing structure, they are difficult to integrate more thoroughly in the software development process, for example, for continuous architectural evaluations. More structured ADLs, however, often do not come with adequate tooling or other features improving their usability which impedes their usage, because it takes effort to use them [157]. The additional effort and complexity of using structured ADLs needs to be worth it. This has been envisioned for some time already under the term Model-Driven Engineering (MDE). MDE, however, has not reached the adoption imagined by academia. It is used only in certain areas or companies [157] and overall adoption in practice is challenging, also due to the large diversity of ideas regarding its usage [106]. More recently, Soares et al. [266] have reviewed the usage of ADLs for continuous evaluations of software architectures. They also come to the conclusion that tooling and automation is a key aspect for wider adoption. For creating or improving ADLs, not only the already mentioned tooling and usability aspects need to be considered, but also the elements of the language itself must be carefully designed as discussed by Sehestedt et al. [256]. They argue that the quality of an ADL can be evaluated according to its completeness, consistency, correctness, and clarity [256].

New ADLs for specific areas are regularly proposed, for example, for microservices [163] or multi-cloud architectures [8]. The findings by Soares et al. [266], nevertheless hold: To gain significant adoption in practice, the proposed benefits of using a specific ADL must outweigh its complexity and effort of usage. For this, automation and usability are key aspects to consider.

In the following, a more detailed look is provided on the purposes that modeling (and thus architectural modeling) can be used for in Section 2.3.1. Section 2.3.2 discusses the different views or layers of abstraction that a software architecture

modeling language may consider and Section 2.3.3 considers typically used notations and syntax for modeling languages. Finally, modeling languages specifically for cloud computing are presented in Section 2.3.4. Topology and Orchestration Specification for Cloud Applications (TOSCA) is presented in more detail, because it is used as a basis for the modeling approach presented in this work.

2.3.1. Modeling Purpose

The different purposes that modeling can be used for can be differentiated based on the software development lifecycle and whether a new or an existing system is considered. According to Sommerville [268, Ch.5], models can be used to facilitate discussions about a new or existing system, they can document an existing system, or be used to generate an initial implementation for a new system. In the context of this work, these purposes are considered either as **documentation** or **implementation & deployment** which can both be attributed to the design time phase. In addition, more recent approaches also propose the usage of models at runtime (specifically called *models@runtime* [34]) which is assigned to the purpose of **operation** in the context of this work.

Subsumed under the purpose of **documentation** are those usages of models where the model unidirectionally depends on the (to be created) system. That means, while modeling artifacts are used for the design and development of a system, they are merely an abstract representation of the system and not used for transforming them into an implementation or to generate other implementation or deployment artifacts out of them. Models with the purpose of documentation represent aspects of a new or existing system for discussions among developers. Regarding new systems, models can be used to reach a common understanding and reduce uncertainty about the following implementation, leading to higher quality software [19]. For existing systems, models can be used to perform architectural evaluations. Architectural evaluations have the goal of improving quality aspects of a system by analyzing a system and finding possibilities for improvement. A first overview on architectural evaluation approaches has been provided by Dobrica & Niemela [66] where model-based approaches, however, only play secondary role, in comparison to scenario-based approaches. In the more recent work by Aleti et al. [7] architecture evaluation and optimization approaches are reviewed that use a variety of models for representing software architectures or to model certain behavior of a system. In specific, more recent examples of using models for the evaluation of software architectures by representing them in an abstract way are the works by Ntentos et al. [206] or Cerny et al. [51] focusing on the analysis of microservices-based architectures. Whether the goal is to design a new system or to improve an existing system, a challenge of using models for documenting systems is always how changes in a system are reflected in its model. That means how an implementation and its model can be kept in sync. While automation would

be a solution here, it requires effort to be built, and it needs to be built specifically for certain modeling approaches and implementation technologies.

When modeling and implementation artifacts are connected more closely, the purpose typically is to support **implementation & deployment** directly with models. This purpose has gained increased interest through Model-Driven Engineering (MDE) [91]. The idea is to model a system's architecture, or parts of it, and then generate executable code or deployment scripts out of these models [233]. Thereby, time can be saved, because boilerplate code does not need to be written, and important characteristics can be included consistently throughout the components of a system. If models should be used for implementation & deployment, however, these models need to be more formalized and contain all necessary details for an adequate generation of code or deployment artifacts [268, Ch.5]. It has been shown that MDE is most effective if used for specific parts of a software system, instead of systems as a whole [304]. That also means the models used for such purposes are specific to a certain aspect and domain which they support and where they can be beneficial. Examples of using models for implementation & deployment aspects include the modeling of business processes using Business Process Model and Notation (BPMN) and their execution on a BPMN engine [102], or more recently the modeling of cloud infrastructures and the transformation of such models into executable deployment scripts [253].

Finally, extending the usage of the models along the software development process, they can also be used at runtime for the **operation** of a system. This concept has been called *models@runtime* by Blair et al. [34] and has since then been used in several approaches. In an extensive literature review, Szvetits & Zdun [274] provide a comprehensive overview on aims and techniques for this approach. A runtime model captures the current state of a system, characteristics of this model are monitored, and, if required, changes are applied to the model that are directly reflected in the running system. This enables a more dynamic software behavior and can support different aims such as policy enforcement or error handling. More recently, such concepts have also been referred to as self-adaptiveness [191, 303]. Mendonça et al. discuss self-adaptiveness in the context of microservices [191]. From their investigation, it becomes clear that architectural models which can be used at runtime are again different to models used for documentation, implementation, or deployment. A main challenge is the integration of such approaches with different technologies as well as their applicability in broader contexts [191].

2.3.2. Modeling Point of View

Because models represent an abstraction of a system, the specific abstraction that a model focuses on is an important differentiation. This can also be referred to as the point of view that a model takes. The potential points of view a model can take are numerous, can vary in scope, and may rely on information not available

from a system itself, but, for example, from the organization developing a system. Typically, the point of view focuses either on aspects of the structure or the behavior of a system, as also exemplified by the types of models enabled by UML [38]. Considering software architectures, the 4+1 view by Kruchten [153] has become famous. It differentiates between five points of view on a system architecture: the *logical* view, the *development* view, the *process* view, the *physical* view, and the *scenario* view. For the different views, Kruchten [153] also proposes different modeling approaches and highlights their respective interdependencies. Another example of how modeling can take different points of view regarding software architectures has also been presented in Section 2.1.2.1. Erl [77, Ch.4][78, Ch.4] differentiates between the different types of architecture relevant in SOAs from focusing on a single service to considering an enterprise as a whole.

Staying in the context of this work, however, points of view relevant for the evaluation of software architectures of cloud-native applications are presented in more detail in the following. A general differentiation can be made between a business-focused and technology-focused perspective, as also done in the review of service description languages by Nistala et al. [201].

From a technology-focused perspective, several levels for modeling cloud applications and infrastructure are of interest [31]. Starting from the lowest levels, models can be used to represent the **hardware infrastructure** on which a system is running. Kuroda and Gokhale [154], for example, model the hardware infrastructure for a private cloud data center to support its provisioning. On top of that is **software infrastructure**. Elements of interest on this level are for example virtual machines, container platforms, but also managed cloud services. Models with this point of view can be used to describe the deployment topology of a system. The purpose can on the one hand be to document the required infrastructure on which the components should run, but it can also be used for automating the provisioning and deployment [306]. Considering an **individual component**, models can be used to describe its internal architecture, that means the classes, methods, libraries, or frameworks used in it. As an example, Lenhard et al. [164] have evaluated source code metrics for identifying architectural problems using an architectural model of a system. Abstracting from internal details of components, and considering how several components interact with each other to form a larger system is an additional point of view. The focus then is on **component interactions**. An example is the approach by Ntentos et al. [206], who evaluate architectures of microservices-based systems. By modeling components, such as services, databases or messaging systems and how they are set up to interact with each other, they evaluate architectures according to their conformance to microservices patterns. While this considers all potential interactions, specific sequences of interactions executed to provide business functionalities can be modeled as **workflows**. A workflow is the technical perspective which specifies the sequence of component calls and potential constraints. Modeling languages for workflows have been

reviewed by Börger et al. [45] and more recently Farshidi et al. [83], with the most prominent example being BPMN.

The **business perspective**, in contrast to the technology perspective, focuses on the business functionalities provided by a system. The alignment and integration of the business perspective with the technical perspective has been the goal of Enterprise Architecture (EA) research where models and visualizations play a major role, as shown by Zhou et al. [319]. Therefore, also modeling languages for the business perspective exist. Mostly, such languages, however, also include technical aspects, as in the case of ArchiMate [159]. Or such languages are created for supporting a specific aspect of the implementation, as in the case of the Entity-relationship model (ERM) [52]. Within the business perspective, a coarse differentiation between **processes** and **data** can be made, although a more fine-grained differentiation is also possible [319]. A business process focus view would consider the sequence of different activities required to fulfill a certain task A data view, instead, considers the different data objects used within a system either based on how they are used in processes or how they are related to each other.

As stated, the mentioned points of view are exemplary and many more specific viewpoints can be formulated based on the purpose and scope of a modeling approach. The points discussed here in more detail, however, are relevant for the consideration of cloud application architectures and are used in Section 2.3.4 for a comparison of modeling approaches.

2.3.3. Notations and Syntax

Modeling artifacts need to have a certain form that is adequate for the specific purpose a model should be used for and suitable for representing the required point of view that a model should represent. The form that is used is generally called the notation and the rules governing how the notation may be used for modeling is called the syntax [195]. Typical notations are either **visual**, **textual**, or **mathematical**. The rigor of the underlying syntax of a notation can vary and depends on the intended usage for the modeling approach.

Most commonly associated with modeling approaches are **visual** notations where shapes are prescribed that can be drawn to represent components and their connections. A combination of different shapes with corresponding connections forms a diagram. As Shaw & Clements state: "*For as long as complex software systems have been developed, designers have described their structures with box-and-line diagrams and informal explanations*" [257] The rigor regarding the syntax can be less formal if a diagram is used to reach an understanding about a system and is discarded afterwards [268]. In other cases, specific semantics are associated with certain shapes or connectors. This is for example the case with UML [38]. It relies on visual notations with a specific syntax. The variety in visual notations is large and also their usefulness varies. Therefore, it is important to think about the meaning given to shapes and connectors and whether the chosen syntax is adequate for the purpose

at hand. Often understandability is essential. Moody [195] provides guidelines and examples regarding useful and appropriate visual notations.

Furthermore, models can use **textual** notations, comparable to code, where the syntax is defined in a specification This is done, for example, by Rademacher et al. [233]. For textual notations a specific syntax is easier to enforce. Thus, textual notations may be better suited for modeling approaches aiming to support implementation & deployment. But tool support should be provided, for example, in the form of a linter, and textual models can become long and complex which may hinder understandability.

And as a more special case of textual notations, there are **mathematical** representations which explicitly use formal mathematical notations to define possible architectural models based on set or graph theory. An example is the approach by Ntentos et al. [206] who also formally express constraints to check the conformance of an architecture to certain patterns. The benefits of using a mathematical notation, such as the possibility to rely on existing mathematical operations, however, need to be weighted up against the additional effort required to create formally correct models.

While a modeling language might support different representations based on different notations, there typically is one representation in focus which is used in day-to-day work and which fits the intended purpose of the modeling best. Transformations between different notations are nevertheless possible and should be automated by corresponding tools. Choosing a suitable notation for a modeling approach requires the consideration of the purpose of the model, the point of view of the model, and whether suitable tooling is already available or can be created.

2.3.4. Cloud Application Modeling Languages

Of special relevance for this work are modeling approaches and ADLs for cloud applications. Therefore, an overview on existing approaches in this area is provided in the following with an additional, more detailed description of TOSCA in Section 2.3.5 TOSCA is the language used as a basis for the modeling approach presented in Chapter 5.

An extensive, and still the most recent, review on cloud modeling languages has been published by Bergmayr et al. [29]. They reviewed 19 cloud modeling languages, that means languages that can model applications and their usage of cloud resources, mainly stemming from an academic context. Based on a comparison framework, the characteristics and capabilities of existing modeling languages were analyzed. Bergmayr et al. found diverse origins and purposes of modeling languages. The aim of harmonizing the diversity in cloud offerings and environments using an abstract model, however, stands out. The core elements represented by cloud modeling languages are the application components and the cloud infrastructure resources used for application deployment. In addition, languages may include details specific to their respective aim, for example elasticity or

pricing aspects. The main purpose of languages is implementation & deployment, with goals being deployment automation, selecting an efficient cloud deployment approach or enabling portability through an abstract description of required resources. Overall, the reviewed modeling languages are diverse themselves, that means Bergmayr et al. found a lack of interoperability between modeling languages and no common, standardized basis. Regarding usability, they note that "*it appears clear that accompanied tools are beneficial for CML's usefulness*" [29].

Another review by Nawaz et al. [201] considers a broader scope of languages with service description languages for cloud services as a starting point. By analyzing the description languages provided by cloud providers to describe their cloud offerings and description languages proposed in academic literature, Nawaz et al. also report a high diversity and no consensus on common abstractions. They also find a focus on the purpose of deployment and provisioning where such description languages can be used to automatically find, select, and provision suitable cloud resources. The lack of consensus and interoperability between different service description languages can be explained by the fact that cloud providers have no interest in supporting consumers to develop portable applications, deployable on various offerings by different providers.

Moving the focus from cloud environments to application architectures, more recently ADLs have also been investigated in the context of microservices-based architectures. Microservices are closely related to cloud-native applications, especially when it comes to their deployment. Thus, modeling microservices-based architectures is also of interest. Alshuqayran et al. [9] did a systematic mapping study on the topic of architecting microservices. Regarding architectural representations, they found that typically informal diagrams are used and in some cases UML, but no comprehensive, microservices-specific modeling language. This finding is supported by Francesco et al. [92] who also did a systematic mapping study with the same focus. They found a majority using informal box-and-line notations, but for certain specific aspects of microservices-based architectures, also formal notations are used, such as BPMN for functionalities supported by several microservices. Francesco et al. [92] furthermore highlight the need for a commonly used language to reason about architectures and especially quality attributes of these. They consider TOSCA as a potentially suitable language, but found only marginal use in practice. Lastly, Lelovic et al. [162] performed a systematic mapping study specifically on ADLs for microservices-based systems. They compare available languages based on a mix of characteristics covering the aspects that can be modeled, but also how models are used and which notation is used. Aspects of interest include business processes, workflows, and service communication within a system. Lelovic et al. [162] identify several languages suitable to represent microservices aspects, but state that no language is broadly adopted and able to represent microservices-based architectures holistically.

Overall, modeling approaches in the context of cloud applications are mostly applied for cloud resource provisioning and application deployment. Also, even

more recent approaches highlight this. For example, Sandobalin et al. [252] present ARGO, a model-driven approach to graphically model and then provision cloud infrastructure resources. Wurster et al. [305] developed the Essential Deployment Meta Model (EDMM) which functions as a meta-model for describing cloud deployment topologies and addresses the mentioned issue of a lack of interoperability between cloud modeling languages.

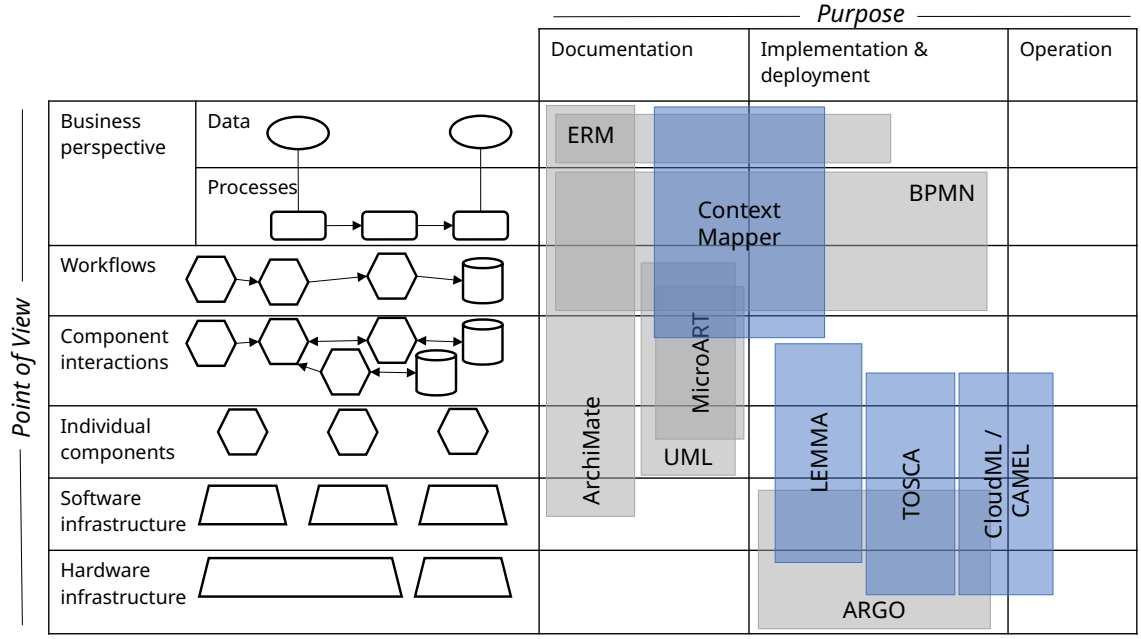


Figure 2.9.: A Categorization of Architectural Modeling Languages

The purposes and points of view presented in the previous sections can be used to compare modeling approaches and ADLs with each other. This has been done in previous work to identify a suitable modeling approach for the quality evaluation approach presented in this work [71]. The comparison is shown in Figure 2.9 and has been slightly extended. In the comparison, it can be seen how cloud-focused modeling languages can be positioned in comparison to other well known modeling languages and approaches, including UML [38], BPMN [209], Entity-Relationship Model (ERM) [52], and ArchiMate [159].

Modeling languages that were part of the detailed comparison [71] are highlighted in blue. CloudML [87], CAMEL [2], and TOSCA [208] are similar in their aim of enabling the modeling of deployment topologies for cloud applications in order to automate the deployment and management of applications across different cloud environments. CloudML and CAMEL are closely related, because CAMEL can be seen as an advancement of CloudML that adds additional capabilities. One added capability is a runtime support which is why CAMEL also partly fulfills the purpose of operation.

LEMMA [234] and Context Mapper [132] on the other hand stem from the microservices context with LEMMA focusing on the model-driven specification and generation of microservices-based applications. Context Mapper focuses on the

modeling and evaluation of communication and integration strategies between services, based on Domain-Driven Design (DDD). Context Mapper thus supports a more detailed representation of communication relationships while in LEMMA the deployment of an application is to some extent integrated as a sub-aspect.

As additional, more recent examples, MicroART [109] and ARGO [252] are also depicted. MicroART [109] is a modeling and tooling approach for recovering microservices-based architectures from source code to document and evaluate the architecture of a system. ARGO [252] is a modeling approach for specifying cloud infrastructure resources that can then be automatically provisioned.

In summary, it can be said that cloud-focused modeling languages have a focus on the selection and deployment of cloud resources suitable to an application at hand while microservices-focused modeling languages lay a focus on the inter-service communication. Because cloud-native encompasses both focuses, a cloud-native modeling approach might need to take an integrated point of view.

2.3.5. TOSCA

TOSCA is a standard published by OASIS Open[8] that provides a specification format for describing deployment and orchestration topologies for cloud applications.
TOSCA 1.0 was published in 2013 and used XML as a base text format. It received wide attention from practice and especially academia [28]. Because TOSCA is "just" a specification, it cannot be used right away for application development. For that, corresponding tooling is required which was mainly implemented by academia [33], namely with the OpenTOSCA environment[9]. The main goal of TOSCA is to enable an automated deployment and management for portable cloud applications [33]. Thus, the main aspects are that (1) applications, that means components and their deployment topologies, can be modeled in a vendor-neutral, and thus portable, way together with required deployment and maintenance operations. (2) a TOSCA-compliant engine can parse the model and automatically deploy it to a suitable cloud environment. Although the standard itself only describes a textual notation, also a graphical notation has been implemented within the Winery tool [147]. With TOSCA 1.3, Extensible Markup Language (XML) was complemented by YAML Ain't Markup Language (YAML) as a base text format, aiming for more readable models. The standard has been continuously revised and TOSCA 2.0 was published in 2025, now only based on YAML. With TOSCA 2.0, breaking changes were introduced, but the majority of existing tooling has been implemented for TOSCA 1.3. It thus remains to be seen, to what extent tooling is revised to support TOSCA 2.0 and to what extent new tooling will be developed. The standard also contains only abstract types now, so all additional ef-

[8] `https://www.oasis-open.org/`, visited 2025-10-20
[9] `https://www.opentosca.org/`, visited 2025-10-20

fort to make TOSCA practically applicable is community-driven[10]. When types are specified for describing application components or cloud resources, this is done within so-called *profiles*. While TOSCA 1.3 contained a default *simple profile* with predefined types for virtual machines or database components, this has been removed in 2.0. It is now completely up to the implementers of TOSCA to provide types and tooling with which applications can actually be deployed. There is no reference implementation or similar, for example, provided by OASIS, which can be used to test TOSCA practically. An excerpt of core TOSCA elements from the standard is shown in Figure 2.10. It is used in the following to illustrate core concepts of TOSCA.

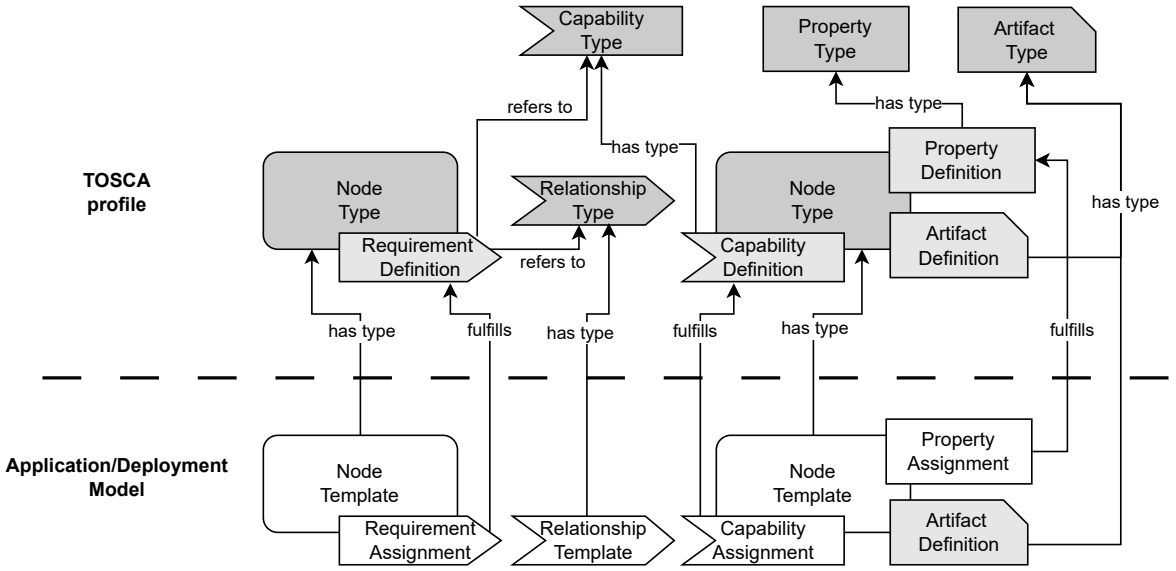


Figure 2.10.: Excerpt of core TOSCA [208] elements

When specifying a TOSCA profile, it is necessary to provide types for the different elements of the TOSCA standard. The core elements are *nodes* and *relationships*. *Nodes* represent the components of an application which includes both application components, for example a web service, and cloud resources, for example a virtual machine. *Nodes* can be connected via *relationships* that essentially connect a *requirement* of a *node* to a *capability* of another *node*. For example, if a web service requires a host on which it can run this can be fulfilled with a hosting capability of a virtual machine. By connecting the web service with a hosted-on relationship to the virtual machine, the requirement of the web service is satisfied. When a TOSCA engine parses this model, it knows that a virtual machine needs to be provided and that the web service should be executed on this virtual machine.

To model, for example, virtual machines, it is necessary to specify a *node type* for virtual machines in a profile. A *node type* can define several details, specifically *capabilities* (of certain *capability types* that can also be specified) and *requirements*. Furthermore, a *node type* can define *properties* that enable the storage of detailed information, for example a virtual machine image name. *Properties* can be of basic

[10] `https://github.com/oasis-open/tosca-community-contributions/`, visited 2025-10-20

types such as *string*, *number*, or *boolean*. But also custom types can be created by specifying *property types* Furthermore, *artifacts* can be used to represent code or deployment artifacts that are needed for components, for example, the actual virtual machine image. Before an *artifact* can be defined for a node, a corresponding *artifact type* needs to be specified. *Relationship types* are required to specify *relationships* between *nodes*. By defining *requirements* and *capabilities* on the *node type* level based on available *relationship types*, it can be ensured that only valid *relationships* are formed later in actual models. As shown in Figure 2.10, the types of a profile can be used to create actual deployment models for applications. Actual application topology instances are called *templates* in TOSCA. Thus, *node templates* and *relationship templates* describe the actual instances of components of an application. They are called *templates*, because the actually running instances are created from these *templates* by a TOSCA engine. By building in support for specific profiles in a TOSCA engine, it can be ensured that any application model based on a supported profile can be deployed and managed with that TOSCA engine.

Because TOSCA is used in this work mainly for a documenting purpose in order to model and evaluate software architectures of cloud-native applications, the presented concepts from the TOSCA standard are those relevant for this aim. Other concepts, such as the specification of *interfaces*, *operations* and *workflows* should nevertheless be mentioned. These are necessary for the purposes of deploying and managing an application.

> The computer scientist should be a “universalist”,
> having the inquiring mind of the empirical scientist,
> the modelling and abstraction ability of the mathematician,
> and the tool building and implementation ability of the engineer.
>
> *Peter Wegner [299]*

3. Methodology

Parts of this chapter have been taken from [171].

This chapter describes the overarching methodology that has been applied for developing the Clounaq approach. It therefore combines the methodologies used within the individual steps that lead to the resulting quality model and evaluation approach. Section 3.1 describes the methodology applied to formulate and validate the quality model. Section 3.2 describes how the tooling support was implemented that enables a practical application of the Clounaq approach.

3.1. Formulating and Validating the Quality Model

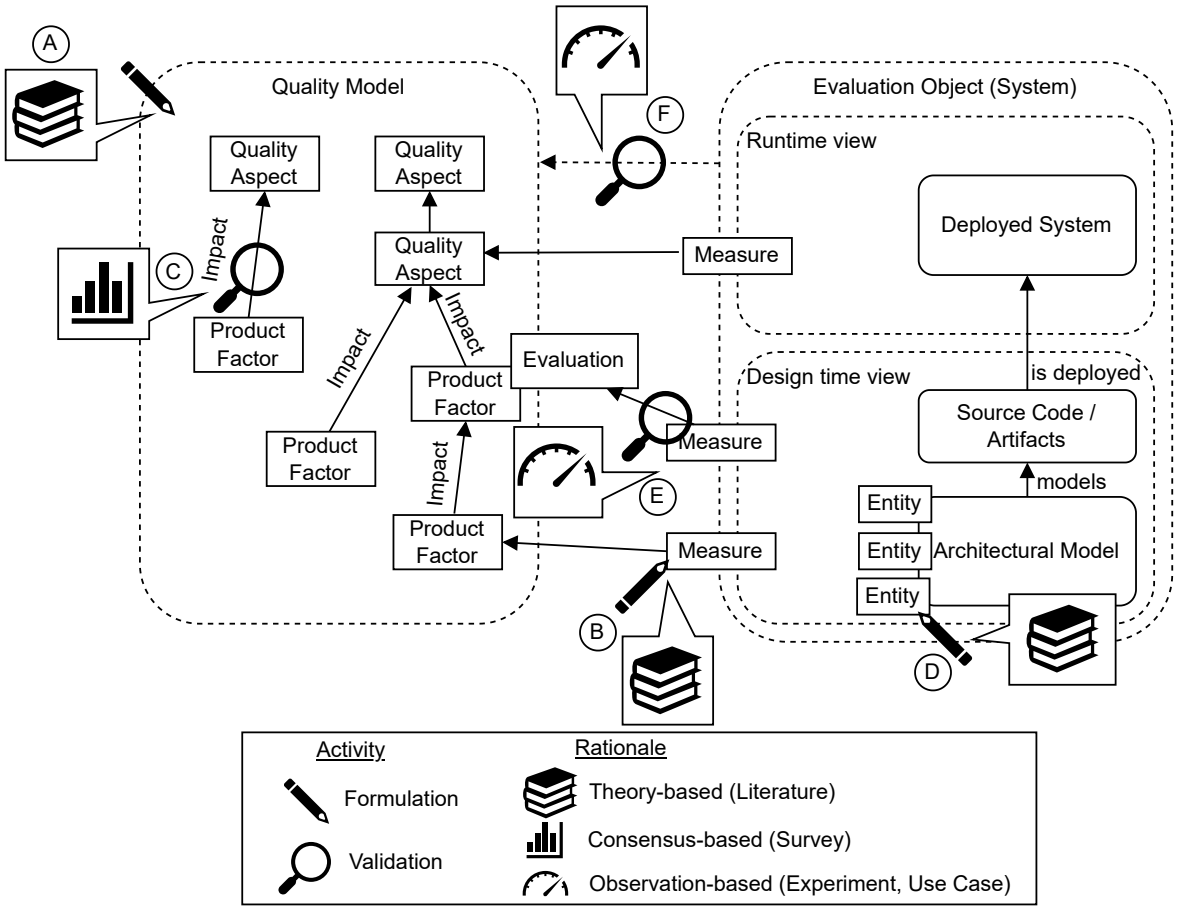


Figure 3.1.: The overall methodology for developing the quality model

Figure 3.1 shows the methodology in a condensed form, applied to the conceptual view of a quality model and how it is connected to a system that should be evaluated. As the overall aim of this work is to develop a quality model that can evaluate systems according to cloud-native characteristics, this methodology chapter describes the steps taken to do so. In Figure 3.1, the steps **A** to **F** are depicted to show which part of the quality model they cover and which specific method was applied. Each step either formulated (depicted with ✎; also compare to Section 2.2.3) or validated (depicted with 🔍; also compare to Section 2.2.5) a part of the quality model. For this, different kinds of rationale were used and therefore different kinds of methods were applied. Steps **A**, **B**, and **D** were theory-based (📚), that means literature searches were conducted to formulate parts of the quality model. Step **C** was consensus-based (📊), because a survey among experts based on a questionnaire was used to validate impacts. And steps **E** and **F** were observation-based (🏎), meaning that either an experiment or use case study was done to validate parts of the quality models using the gained observations.

3.1.1. Step A: Initial Quality Model Formulation

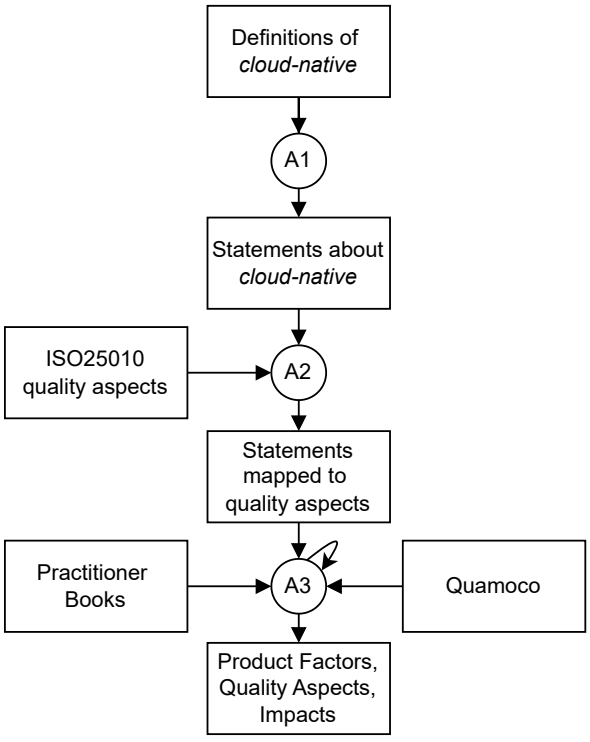


Figure 3.2.: Details for Step A

For the initial formulation of the quality model [169], a literature-based approach was used. This is shown in detail in Figure 3.2. Using a top-down approach, firstly, definitions of *cloud-native* from the academic literature [84, 99, 151, 216, 281, 307] and from practice [55, 240, 292] were used as basis. From these definitions, distinct statements about characteristics of cloud-native were extracted, combined,

and collected (A1). To build on a proved foundation, the ISO25010 [122] standard "Systems and software Quality Requirements and Evaluation (SQuaRE)" was then added. More specifically, the quality aspects from the *Product quality model* were used as top-level quality aspects and the statements were mapped to these quality aspects (A2). Finally, the initial quality model based on the Quamoco meta-model [294] was formulated (see Section 2.2.3.1). This formulation was done in an iterative approach (A3). As the topic of *cloud-native* is mainly driven by practice, practitioner books on the topic were scanned as a further literature basis. The used books are listed in Table 3.1. Statements from these books were added to the previously collected statements to formulate product factors backed by such statements.

Table 3.1.: Practitioner books used in A3

Title	Author(s)	Year	Ref.
Cloud Native Infrastructure	Justin Garrison and Kris Nova	2017	[101]
Cloud Native Java	Kenny Bastani and Josh Long	2017	[26]
Cloud Native Patterns	Cornelia Davis	2019	[60]
Cloud Native	Boris Scholl et al.	2019	[255]
Microservices Patterns	Chris Richardson	2019	[242]
Cloud Native Transformation	Pini Reznik et al.	2019	[241]
Cloud Native DevOps with Kubernetes	John Arundel and Justin Domingus	2019	[13]
Kubernetes Patterns	Bilgin Ibryam and Roland Huß	2020	[118]
Building Secure and Reliable Systems	Heather Adkins et al.	2020	[3]
Design Patterns for Cloud Native Applications	Kasun Indrasiri and Sriskandarajah Suhothayan	2021	[120]
Practical Process Automation	Bernd Ruecker	2021	[249]
Cloud Native Architecture and Design	Shivakumar Goniwada	2021	[108]

The mappings between product factors and quality aspects were transformed into impact relationships. Through the iterative approach, new knowledge from literature was gradually included, and product factors were continuously refined. The result from this step, therefore, was an initial quality model with product factors, quality aspects, and connecting impact relationships between them.

3.1.2. Step B: Literature Search for Measures

To identify suitable measures that can be used to evaluate the factors formulated in Step A (3.1.1), an additional literature search was performed. Guidelines for planning, structuring and reporting the search were followed [140, 236] and a detailed description can be found online[11]. The goal of this search was to extract existing architectural measures from literature which have previously been used to evaluate the characteristics of cloud-native software architectures as described by the product factors of the quality model. This literature search has, in a first iteration, been done together with the initial formulation [169]. It has been repeated and revised at a later point in time [171] to include newly published literature and find additional measures (see Figure 3.3).

Two search strings were used for the literature, aiming to cover the topic of cloud-native and to find literature explicitly describing architectural measures that can be calculated based on a software architecture.

Therefore, the first search string sets a focus on the topic of microservices architecture (see 2.1.2.2), because it is closely related and has been in focus of research. To incorporate the fact that the terms "measure" and "metric" are often used for the same thing in literature, both are used in the search string, also in their plural forms:

> (Abstract:(architecture)) AND (Abstract:(measure) OR Abstract:(measures) OR Abstract:(metric) OR Abstract:(metrics)) AND (Abstract:(service-oriented) OR Abstract:(microservices) OR Abstract:(microservice))

With the second search string a focus is set on cloud applications and their quality. The constraint on "quality" is due to the fact, that "cloud computing" is also included as a keyword, because not all literature may explicitly use the term "cloud-native". But "cloud computing" is a broad topic itself (see Section 2.1.1) and may lead to many irrelevant results if it stays for itself. For this search string, also "measure" and "metric" are both included:

> (Abstract:(cloud-native) OR Abstract:("cloud computing")) AND (Abstract:(measure) OR Abstract:(measures) OR Abstract:(metric) OR Abstract:(metrics)) AND (Abstract:(quality))

It has to be noted, that these two search strings represent their latest iteration as used in the 3rd revised search. The 1st and 2nd search used exactly the same search strings, but upon a closer review of them, it was realized that literature might have been missed due to the usage of their singular or plural forms of keywords. For example, in the initial search terms, the term "microservices" was included, but not "microservice". Thus, the paper by Zdun et al. [315] was

[11] `https://r0light.github.io/cna-quality-model/search-process`, visited 2025-10-20

not found during the 2nd search, although relevant measures are presented in it. Thus, the search string were revised to account for this.

The search strings were used to search the ACM Digital Library[12], IEEE Xplore[13], and Springer Link[14].

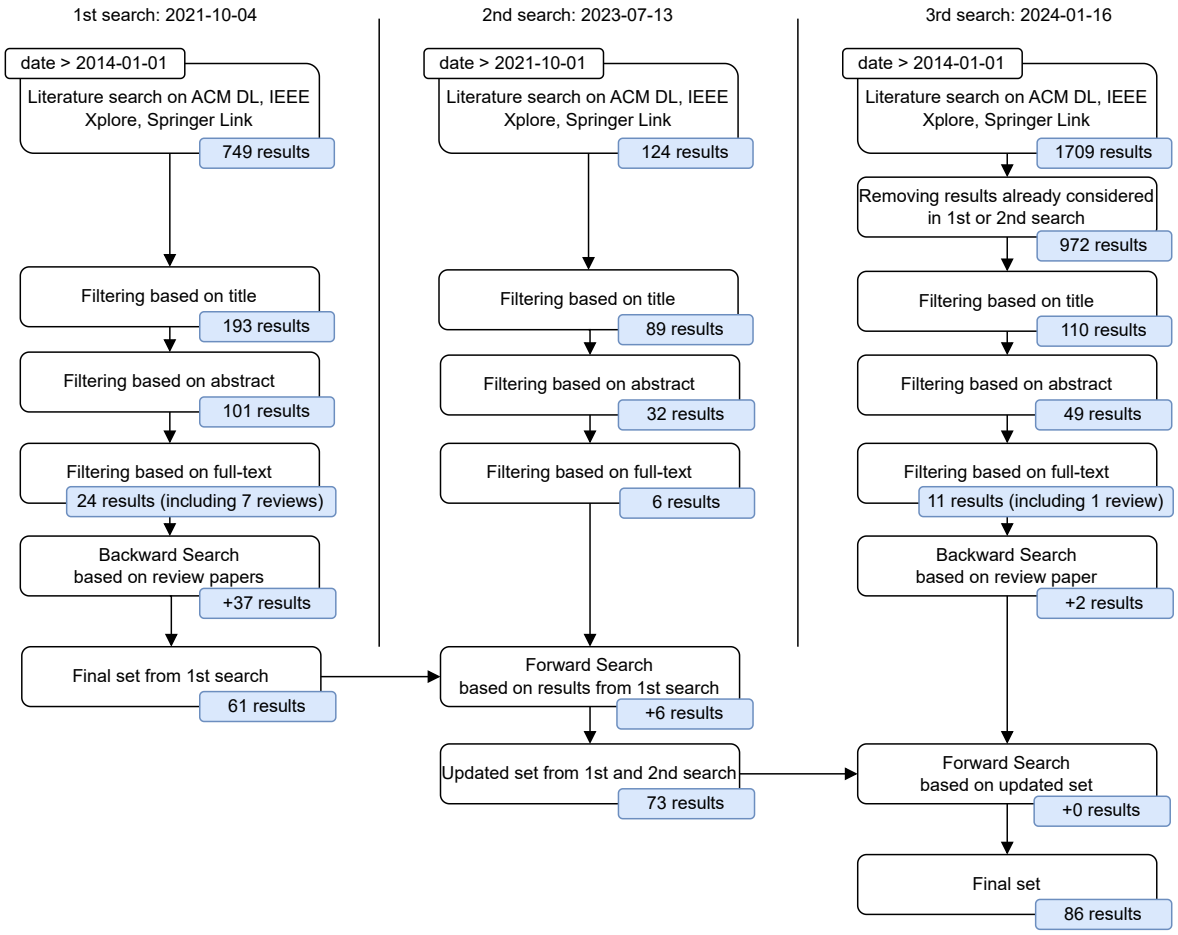


Figure 3.3.: Merged search processes for step B (Literature Search for Measures)

An overview of all three searches and their corresponding analysis steps is provided in Figure 3.3. In the first search a publishing date filter was set to include only work published from 2014 on, as this marks the year when cloud computing really emerged in the form required for cloud-native applications (2.1.1). The third search then used that same date filter to include also previously missed literature. However, an additional step was then necessary to filter out results which had already been analyzed in the first or second search.

To identify results, containing descriptions of architectural measures relevant to the quality model, and filter out irrelevant results, the following criteria were checked for each result. To be relevant, literature has to:

[12] `https://dl.acm.org/`, visited 2025-10-20
[13] `https://ieeexplore.ieee.org/Xplore/home.jsp`, visited 2025-10-20
[14] `https://link.springer.com`, visited 2025-10-20

- take a perspective corresponding to that of application developers (in contrast to the perspective of cloud providers),
- consider software architectures on a level suitable to the quality model (in contrast to lower levels such as internal component design or higher levels such as the end-user perspective),
- present measures that can be evaluated at design time (in contrast to runtime specific measures),
- clearly specify measure calculations so that they can be adopted

These criteria were applied together with a general check of topic-fit, first by analyzing only the title, then by analyzing the abstract, and finally by analyzing the full-text, as shown in Figure 3.3.

Where applicable, also backward and forward searches were done. For example, backward searches were done based on found review papers that aggregate a set of potentially relevant literature. And a forward search was done based on the previously identified literature in the second and third search iteration. Overall, the searches lead to final set of 86 publications containing relevant measures applicable to the developed quality model.

The measures from these publications were then adapted to the specific entities of the quality model and a formula for their calculation was derived, based on the entities of the quality model. Furthermore, each measure was assigned to a product factor for which it provides a meaningful measurement and could thus support its evaluation. It was also stated which entities were needed to calculate the measure, and it was derived on which entities a measure could be applied. For example, a measure that can be calculated for an individual service can often also be calculated for a whole system by averaging the values across all services of a system. If it was possible to calculate a measure on a coarser level by aggregating it or to calculate it on a finer level by using only a subset of entities, this information was added to a measure.

3.1.3. Step C: Impact Validation

Before continuing with the formulation of the more detailed elements of the quality model, the so far formulated elements should be validated. To do so, an empirical survey-based approach was used [171, 174]. This is because from a methodological point of view it can be applied early in a formulation process, even when not all elements of a quality model are formulated yet (see Section 2.2.5). The focus of the survey was set on the impact relationships between product factors and quality aspects. Specifically, participants were asked to state impacts based on their own experience without consideration of the impacts already formulated for the quality model. If participants stated the same impacts as in the quality model, this validates them and involved product factors.

For survey research, replicability is of particular importance [48]. In this section, therefore, the process of survey design and execution is described. For conducting the survey, the research process as presented by Kasunic [134] was followed. Kasunic lists seven stages for conducting a survey:

1. Identify research objectives - see 3.1.3.1
2. Identify & characterize target audience - see 3.1.3.1
3. Design sampling plan - see 3.1.3.2
4. Design & write questionnaire - see 3.1.3.2
5. Pilot test questionnaire - see 3.1.3.2
6. Distribute the questionnaire - see 3.1.3.3
7. Analyze results and write report - see 3.1.3.3

3.1.3.1. Identification of Factors and Impacts to Validate

The research objective of the survey was to validate and revise the elements of the quality model as it existed after Step A (3.1.1). Due to the breadth of the topic of cloud-native, the quality model had 76 factors. This amount posed a challenge to the survey design, because the time and effort potential participants are willing to invest in such surveys is usually limited. Therefore, the objective was adjusted by reducing the number of factors for the survey. To do this systematically, the number was reduced by:

1. considering only the direct impacts from factors on quality aspects (e.g., *Distributed tracing of invocations* → *Analyzability*), excluding impacts from factors on mediating factors. (e.g., *Distributed tracing of invocations* → *Automated monitoring*
2. excluding intermediate factors merely serving as 'placeholders' (e.g. *Automated monitoring*, but not *Automated restarts*)
3. excluding factors which are less specific to cloud-native applications (e.g. *Data encryption in transit*)

This lead to a total of 45 factors for the survey, in addition to the 24 quality aspects.

As the target audience, IT professionals (e.g., developers, software engineers, software architects, IT managers) were identified who have experience with implementing and operating web-based applications that run on cloud infrastructures. Their professional experience is regarded as the key enabler to reliably rate impacts of factors on quality aspects. Still, it can be considered a rather inclusive criterion, in order to avoid being too restrictive and unnecessarily reducing the number of potential participants.

3.1.3.2. Survey Setup and Pilot Study

The achievable sample size of the target audience was difficult to estimate in advance. Therefore, no explicit sampling plan was formulated, but instead the plan was to distribute the survey via appropriate channels to achieve a wide reach and collect as many answers as possible. To ensure that participants belong to the target audience, an explicit description of the target audience was presented on the welcome page of the survey site.

For the questionnaire design, a symmetric bipolar scale with a neutral alternative [62] was used for the impact ratings. This allowed for measuring also the impact strength. A 5-point scale was chosen, as too fine-grained scales have not been found to provide much benefit [175]. The scale had the following options for rating how a factor can impact a quality aspect:

- this factor has a **positive impact (++)** on that quality aspect
- this factor has a **slightly positive impact (+)** on that quality aspect
- this factor has a **no impact (0)** on that quality aspect
- this factor has a **slightly negative impact (-)** on that quality aspect
- this factor has a **negative impact (--)** on that quality aspect

Using such a scale would have allowed for using an existing survey tool such as limesurvey[15] which was initially tried. However, because a possibility should be included to also detect previously not considered impacts, all potential impacts (i.e., 45 factors * 24 quality aspects = 1080) would have needed to be rateable. With the usual question layout from existing survey tools, this would have made the survey overwhelming for participants. Therefore, a custom survey frontend[16] was built. This frontend aimed for a simple and intuitive rating by participants. A participant focuses at a single factor on one page. A factor is presented in a uniform layout with a brief description. The potentially impacted quality aspects are presented below, grouped by their high-level quality aspects. By clicking on a quality aspect, an impact can be selected and rated. A screenshot from the survey frontend visualizing this setup is shown in Figure 3.4

An additional difficulty was to estimate in advance the number of factors that participants would be willing to answer. Ideally, each participant would provide an answer to each factor (45). But the time that participants are willing to invest is usually very limited. The approach was thus tested in a pilot study where participants could rate as many factors as they wanted. In this pilot, participants were also asked to provide explicit feedback on the usability of the survey tool, the content, and the size of the survey. The pilot study was conducted for two weeks in July 2022 with six participants who represented a small sample of the target audience.

[15] `https://www.limesurvey.org`, visited 2025-10-20
[16] `https://github.com/r0light/qmsurvey-frontend`, visited 2025-10-20

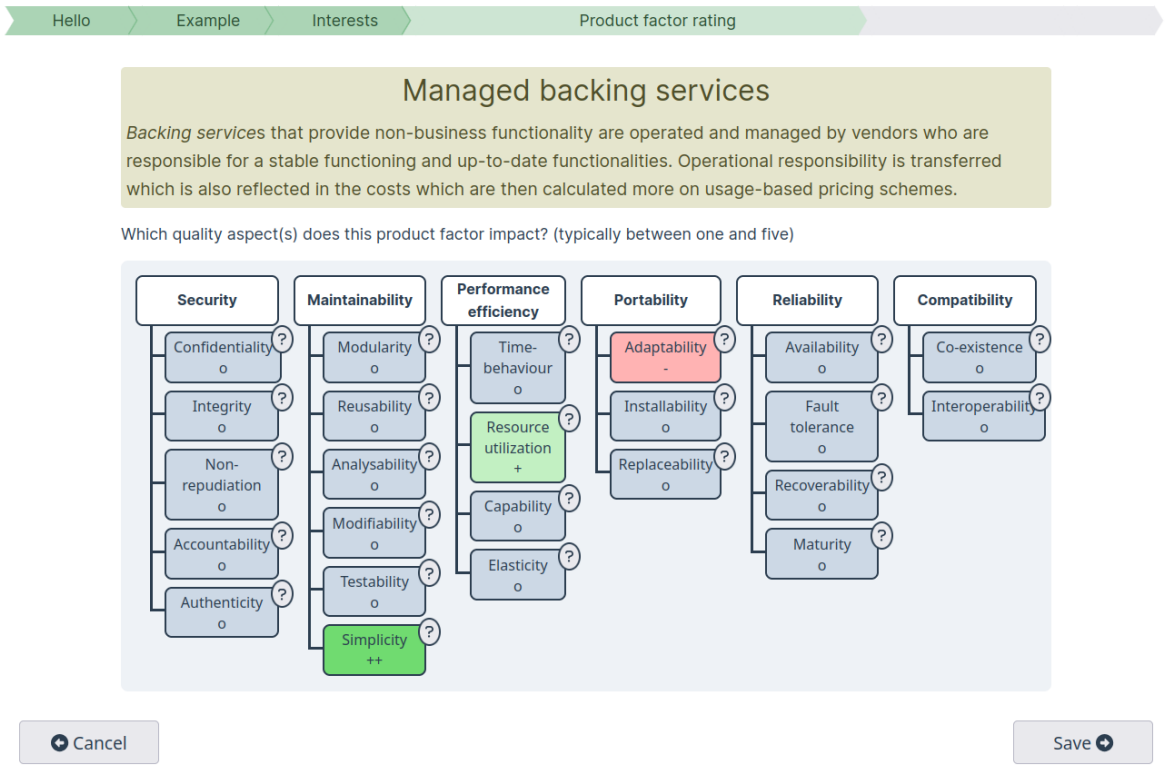


Figure 3.4.: Screenshot of the survey frontend with the page for the factor Managed backing services

It showed that interested participants might be willing to provide more answers. Others abort the survey soon if the number of questions is too high. Hence, for the actual survey, participants were able to rate as many factors as they wanted to. Furthermore, factors were grouped thematically to make it easier for participants to choose factors according to their topics of interest. Finally, demographic questions were included. These questions asked about the job area, job title, industry sector, and years of experience with software development in general and experience with cloud computing in specific.

3.1.3.3. Survey Distribution and Results Preparation

To distribute the survey, requests for participation were sent to:

- professionals who had published articles addressing a related topic on popular blogging sites, such as Medium[17]
- professionals who held a talk on a related topic at industry conferences, such as the CloudNativeCon[18]

[17] `https://medium.com/`, visited 2025-10-20
[18] `https://www.cncf.io/kubecon-cloudnativecon-events/`, visited 2025-10-20

- professionals who had published on a related topic at scientific conferences, such as the IEEE CLOUD[19]
- professionals personally known to one of the authors
- community groups specifically considering cloud computing on social media sites, such as Reddit[20]

Email was preferred as the method of communication, but also LinkedIn[21] was used, when contact details were not publicly accessible. The survey was available online from October 2022 to December 2022. During that time-frame, 42 complete submissions were received.

The gathered results were prepared and analyzed in the following ways: Firstly, by using descriptive statistics, it was analyzed how and for which factors participants provided answers. Secondly, as the main analysis, the ratings for each factor-quality aspect combination were analyzed to compare them with the impacts stated in the initial quality model. Thirdly, additional impacts, not covered by the original quality model, but derived from the ratings, were analyzed.

The results contained data in the form of a set of ratings for each factor-quality aspect combination. A rating is a numerical value of **+2**,**+1**,**0**,**-1**, or **-2** corresponding to the rating options **positive impact (++)**, **slightly positive impact (+)**, **no impact (0)**, **slightly negative impact (-)**, or **negative impact (--)**. Thus, a simple *mean* value was calculated for each set. Furthermore, to also have an indicator for the significance of results, the *Exact multinomial test of goodness-of-fit*[187] was used. It is suitable when there are multiple values of one nominal variable (the different types of impact) and the sample size is small. In essence, the test compares the observed distribution of a variable with an assumed distribution representing the null hypothesis. The smaller the probability for an observed distribution under the null hypothesis is, the more significant the result is. For the case of the null hypothesis, it is assumed that an impact can not be clearly stated, meaning that all ratings have the same probability. An exception to this is the *no impact* rating. It was selected as a default in the survey tool, if no explicit rating was stated. Therefore, its probability is assumed to be twice as high. That means, for the ratings **+2**/**+1**/**0**/**-1**/**-2** a ratio of **1**:**1**:**2**:**1**:**1** is assumed under the null hypothesis. The result of the comparison of the actual distribution with this distribution is added as the *pValue* to each set of ratings. The *mean* value together with the *pValue* form the basis for the interpretation of the results. The *mean* value expresses the type and strength of an impact, while the *pValue* is an indicator for the significance of an impact stated by participants. As the significance level $\alpha = 0.1$ was set as a commonly chosen value. Furthermore, as an additional significance indicator, a threshold of 5 for the total number of answers for a factor was chosen. The validation result is put in parentheses to mark it as uncertain, if either the p-Value is

[19] `https://conferences.computer.org/cloud`, visited 2025-10-20
[20] `https://www.reddit.com/`, visited 2025-10-20
[21] `https://www.linkedin.com/`, visited 2025-10-20

above the significance level or the number of answers for this factor is below the threshold.

An additional detailed description and all results are summarized in a report which is available online[22]. It also includes the anonymized raw survey data and the demographic data on the background of participants which shows a mixed distribution of industry and academic backgrounds.

3.1.3.4. Incorporation of the Validation Results into the Quality Model

With the survey report as a basis, the results showing significance (specifically based on the *pValues*) were used to reiterate on the elements of the quality model. Impacts that could be successfully validated were kept in the quality model. Impacts which could not be validated through the survey results were reconsidered. That means, the initially used literature was consulted again, and a decision was made of whether to remove the impact relationship or to restructure the hierarchy of factors where suitable. In addition, the newly stated impacts were analyzed, and it was evaluated whether to include them in the quality model, again by reconsidering also the initially used literature. For factors with many impacts on several quality aspects, it was reviewed whether they cover several distinctive aspects and could be split up into separate factors. For factors with unclear impacts, it was analyzed whether they can be defined and formulated more clearly.

In fact, some factors showed to be ambiguous, because either significant impacts on several quality aspects were found or because conflicting types of impact (that means positive and negative) were found for the same quality aspect. In these cases, the factors were revised as such to properly include the found impacts as new impact relations in the quality model. Revision could also mean splitting a factor into new separate factors which cover distinguishable aspects of the factor that showed ambiguous results. All refinements were based on the interpretations by the survey executors, but the initially used literature was also reconsidered to substantiate decisions. To summarize, the following refinement actions were applied to the quality model based on the validation survey results:

- Removal of an initially stated impact relation (17 times)
- Addition of a newly found impact relation (16 times)
- Reformulation of a product factor and selection of suitable impact relations (2 times)
- Splitting a product factor in two or more new product factors which cover distinguishable sub-aspects of the original product factor (1 time)

Which refinements regarding specific impacts and factors were applied is also listed online[23].

[22] `https://github.com/r0light/qmsurvey-results`, visited 2025-10-20
[23] `https://r0light.github.io/cna-quality-model/survey`, visited 2025-10-20

Overall, the validation step resulted in an updated and improved version of the quality model [171]. Finally, the results of the survey also highlighted aspects to consider for further validation, for example where an impact could not be clearly derived.

3.1.4. Step D: Modeling Entity Formulation

In parallel to the validation of existing quality model elements (Step C), the actual evaluation of software architectures was prepared by formulating suitable architectural entities to represent software architectures. This required on the one hand a conceptual definition of entities as part of the quality model to which factors can be related. And on the other hand, a description language which enables the structured representation of software architectures and which can be stored, shared and processed. Especially, for the practical applicability of the approach, a description language based on existing approaches and tooling would be preferable to enable an integration with other modeling and evaluation approaches.

An initial set of conceptual entities, fitting to the initially formulated product factors of the quality model, was derived together its initial version [169]. These entities were formulated based on the formulated product factors so that in principle the information required to measure and evaluate the product factors was available within these entities. However, only the types of these entities were specified and how they are related, but no detailed properties were specified.

The more detailed definition of entities was done based on the chosen description language and during the implementation of measure calculations and evaluations. The approach thereby was to choose an existing description language, in this case a modeling language, to build on first and then extend the modeling approach iteratively while implementing the measures needed to quantitatively evaluate product factors. The modeling approach was thereby extended based on which information was needed for the measure calculations. The overall goal thus was to keep the entity formulation as expressive as needed, but also as constrained as possible.

3.1.4.1. Choosing a Description Language as a Basis

Using architectural models to represent, analyze, and manage cloud-based applications has been done in previous work already (see Section 2.3.4). To benefit from previous work and additionally enable a potential integration with other approaches, a decision was made to review existing cloud software architecture description languages and select one existing language as the basis for the representation of software architectures to evaluate.

The selection is based on the master thesis by Karolin Dürr [70] who performed a literature search for existing architectural description languages for cloud ap-

plications[24]. From the search results, she compared found languages based on their suitability to represent entities as required by the quality model and additional factors influencing the practical applicability within the quality evaluation approach [71]. Based on this comparison, a decision was made for TOSCA [208] (see Section 2.3.5), because it is a standard supported by a larger community and its extensibility allowed for the best fit regarding the quality model requirements. An initial TOSCA profile as the modeling language basis was developed [71] based on TOSCA v1.3 and the TOSCA Simple Profile in YAML[25]. During the further work on the quality evaluation approach, however, the modeling approach was migrated to TOSCA 2.0 [208] which also led to the dismissal of the Simple Profile, because it was deprecated with the updated standard. The current modeling and description basis for the entities, therefore, is TOSCA 2.0.

3.1.4.2. Extending the Modeling Approach

In line with the initial formulation of the modeling approach being driven by the requirements of the quality model elements, an extension of the modeling approach was done driven by a use case-based investigation of implementation options for product factors of the quality model [145].

The initial investigation of implementation options was done as a part of the master thesis by Franka Knoch [144]. As a result, different specific implementation options for a range of product factors from the quality model were created[26]. The modeling approach was then extended based on the results of the work [145]. To do so, common and differing characteristics of the implementation options were analyzed, and the modeling approach was then extended in order to be able to model and differentiate these options within the approach. Effectively, this meant extending entities of the modeling approach with properties that allow for including additional information within models.

3.1.4.3. Iterative Refinement of Entity Types and Entity Properties

The subsequent refinement of entity types and their properties up to the state of the quality model as described in this work, was also driven by the requirements of the quality model. For this refinement, however, the goal was to advance the quality model and the quality evaluation approach into a consistent state so that it can be practically applied to software architectures. This means that for all product factors, a quantitative evaluation should be possible by relying on suitable measures and that evaluations are available which interpret resulting measure values and quantify impact relationships. Therefore, in an iterative approach, the following actions were performed for each product factor:

[24] `https://karolinduerr.github.io/MA-CNA-ModelingSupport/`, visited 2025-10-20

[25] `https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html`, visited 2025-10-20

[26] `https://github.com/frankakn/cloud-native-deployment`, visited 2025-10-20

- if measures are assigned to it:
  - check if these measures can be calculated based on the current entities
  - extend the entity types or entity properties to support these measures
- if no measures are available (because none were found in Step B):
  - specify at least one new measure to enable a quantification of the factor
  - extend the entity types or entity properties to support new measures
- refine the entity types and entity properties by checking if properties can be merged or expressed in a more meaningful way

Through this a few more specific entity types (e.g. Proxy Backing Service or Broker Backing Service) were added, and corresponding properties were added to entities to enable the calculation of measures. This refinement was done both conceptually for the entities in the context of the quality model and their representations within the TOSCA profile developed for the approach.

The overall outcome of this step, therefore, is a fully formulated quality model with all required elements specified. Furthermore, together with the quality model, a modeling approach based on TOSCA is available. It is integrated with the quality model in the sense that software architectures modeled based on this description language can be evaluated based on the quality model.

3.1.5. Step E: Measure Validation

Although, as an outcome of Step D, the quality model and evaluation approach is fully available, several elements are formulated solely based on literature and therefore require validation to ensure their meaningfulness. Especially, newly formulated measures for product factors for which no measures could be found in literature before, require a validation. This has been done, at least for a selection of measures, within this Step E. In specific, an experimental approach was used that measures architectural measures for a range of architectural variations of an exemplary system. The resulting values are compared with runtime measures from a load test on these architectural variations [172]. This way, the meaningfulness of the measures in terms of a quantification of corresponding product factors and their impact on quality aspects can be validated. The experiment setup and results interpretation is described in the following.

3.1.5.1. Experimental Setup

The core aspect of the used experimental setup is that different architectural variations of a system could be executed and compared. Architectural measures could then be calculated for the different variations and be combined with runtime measurements for these variations. Thus, a system was needed for which such variations could be created. The TeaStore application [293] was chosen as a use case

for the experiments. It has been created and used by researchers explicitly for such purposes. Additionally, tools and utilities for the deployment and load testing of the application are available. The original version of the TeaStore[27] was used as one architecture variation. Additional variations were created with the goal of varying the architectural measures in focus of the validation.

Following the recommendations by Bermbach et al. [30] and Papadopoulos et al. [217] for designing benchmarks in cloud environments, a focus was put on the repeatability of the experiments. The complete experimental setup is described in the following and artifacts for repeating the experiments are provided online[28]. For all implemented architectural variations, Docker images were created and are publicly available online[29]. These images can be reused to repeat experiments. Different image tags represent different variations, as described in the following:

- **original**: An unchanged version of the TeaStore
- **nocaching**: A version with caching disabled in the *persistence* service
- **withcaching**: A version with caching in the *webui* and *auth* services for applicable data aggregates
- **withfailures**: A version in which failures are simulated within services to enable availability evaluations
- **withfailures-nocaching**: A combination of *withfailures* and *nocaching* for the *persistence* service
- **withfailures-withcaching**: A combination of *withfailures* and *withcaching* for the *webui* and *auth* services

For images having **withfailures** in their tag name, a simulation of failures was included by implementing a return of a server fault during certain time periods for each endpoint of applicable services. Specifically, within a time frame of 10 seconds, a service replies only with server errors for 2 seconds. The erroneous time frame is implemented to be different for each service instance by including an offset based on the hostname. Since the hostname is set to the container id, the offset differs for each instance that is started as a new container. This approach simulates outages of individual service instances to enable an evaluation of availability within the comparatively short time of a test run.

The different images were then used within Kubernetes Deployment descriptions to create different architectural variations. For example, the number of replicas for each service was varied to vary the Service replication level measure.

Apart from these variations, through using different images and changing the number of replicas, a consistent experiment setup was used to minimize the impact of additional deployment aspects. This setup is shown in Figure 3.5.

[27] `https://github.com/DescartesResearch/TeaStore`, visited 2025-10-20
[28] `https://github.com/r0light/experimental-measure-validation/`, visited 2025-10-20
[29] `https://hub.docker.com/u/rlight?page=1&search=teastore`, visited 2025-10-20

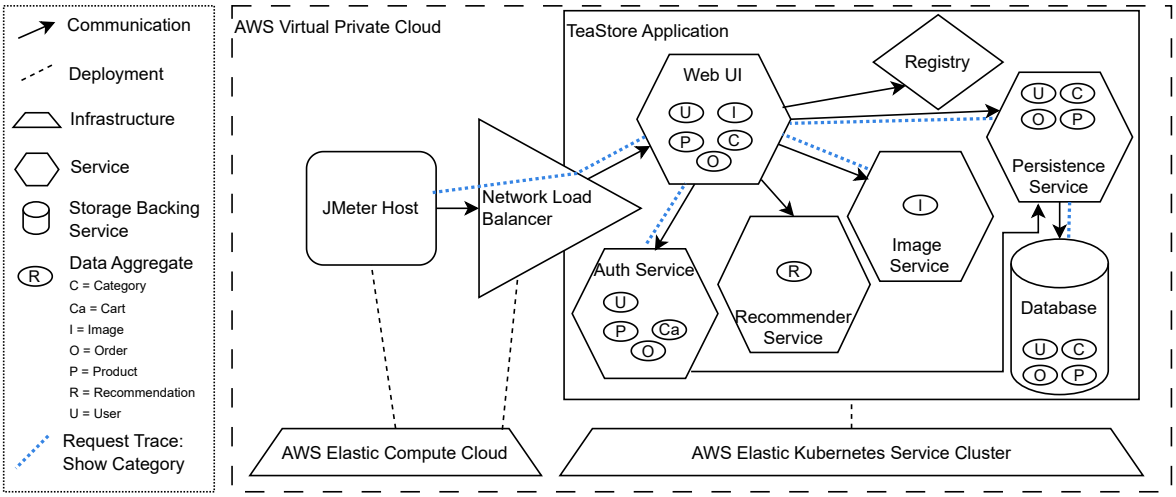


Figure 3.5.: Experimental Setup of the TeaStore application for Measure Validation Experiments [172]

To achieve a more realistic setting, the Amazon Web Services (AWS) cloud was chosen, specifically with the AWS Elastic Kubernetes Service (EKS)[30] based on managed EC2 Node Groups. All provisioning tasks were automated via Terraform[31] for a stable cluster configuration throughout experiments. To access the application within the cluster, a Network Load Balancer was provisioned via the AWS Load Balancer Controller for the Web UI Kubernetes Service. This load balancer was only reachable within a private network. An additional EC2 Node (called JMeter Host) was added to this private network to start load tests.

For the factor Horizontal data replication, instead of the default TeaStore database running within the cluster, a managed cloud database (AWS Relational Database Service (RDS))) was used to facilitate database replication. Replicas of the database were created via the AWS RDS interface. It has to be noted, however, that for the experiments, only read replicas were used for the database.

In total, 20 architectural variations were prepared, which enable investigations of changes to architectural measures. Not all measures differed for all variations, but they differed within a set of variations that were then considered in combination. For example, the variations **noreplication**, **lowreplication**, **mixedreplication**, and **highreplication** were used to investigate the measure Service replication level.

Load tests were performed with JMeter, because a JMeter test plan was already available from the TeaStore application. The used test plan is called the `browse` profile. It represents a realistic usage of the application[72, 293]. It includes a range of requests performed in a certain order to mimic typical user behavior. The different requests of the profile can be represented by Request Trace entities and are named **Index Page**, **Show Category**, **User Login**, **Product Page**, **Add Product To Cart**, and **User Logout** in the data and the results descriptions. Furthermore, **teaStore** refers to the System entity itself.

[30] `https://aws.amazon.com/eks/`, visited 2025-10-20
[31] `https://www.terraform.io/`, visited 2025-10-20

Scripts are provided to execute the JMeter profile and copy the result files created by JMeter from the JMeter host to a local machine. For the load profile, the approach presented by Eismann et al. [72] was adapted. While also using three different load levels, the respective rates were lowered to load levels of 100, 150, and 200 requests per second. This is because in the smallest setup, one instance per service was used, while Eismann et al. used multiple instances per service even in their smallest setup.

For each architectural variation, the application was deployed to the Kubernetes cluster. Then, a warm-up time of 10 minutes was used during which a load of 200 requests per second was sent to the application. After the warm-up, the actual measurements were done by 5-minute runs repeated 5 times for each load level. This process was repeated for each architectural variation once more with a fresh application deployment. Thus, a total of 30 runs were done for each architectural variation. Repetitions are important for experiments in the cloud, because cloud environments can be unstable [72] and the impact of temporary problems is thus statistically reduced.

The load tests with all architectural variations were done one after another in a time span from 2025-02-05 to 2025-02-25. All results from these experiment runs are available online as a report[32]. The report and all following evaluations have been done with R[33]. The raw data is also available online[34].

3.1.5.2. Results Interpretation

For an interpretation of the results, the raw data from each experiment run was aggregated based on required runtime measures and per entity. The runtime measures in focus were the response time (to investigate Time-behavior) and success rate (to investigate Availability). Values for these measures were calculated per experiment run for the system as a whole and per request trace within the profile. For the response time, an average value as well as the 90th percentile value was calculated. This aggregated data from the experiment runs was then combined with the corresponding architectural measures calculated with the Clounaq tool that implements the measures formulated within the quality model. Architectural measures were calculated, where applicable, for the System entity and the Request Trace entities. Out of this data set as a whole, subsets of data were built, specific to the corresponding hypotheses Thus, a variation of the architectural measure in focus of each hypothesis was available together with the corresponding runtime measure values in focus of each hypothesis. For each of the subsets, a linear regression model was calculated per entity, considering the architectural measure as the independent variable and the runtime measure as the dependent variable. For the interpretation of the results, these linear regression models are included in a plot for each entity within a subset. The aim of these regression models was

[32] `https://r0light.github.io/experimental-measure-validation/`, visited 2025-10-20
[33] `https://www.r-project.org/`, visited 2025-10-20
[34] `https://doi.org/10.5281/zenodo.15488911`, visited 2025-10-20

to investigate whether there is a relationship between the measures that is significant (based on the calculated p-values). Although non-linear regression models might have provided an overall better fit (considering R^2), the more intuitive linear regression models were preferred. This is because the intention was not to enable exact predictions, but just investigate the general relationships.

3.1.6. Step F: Validation by Use Cases

To assess the developed quality model and its corresponding quality evaluation approach also as a whole, an additional validation based on use cases was performed [173]. This validation focused less on quantitative validations of individual measures, but rather on the measure interpretations applied in the corresponding product factor evaluations. The goal was to validate whether the approach as a whole is able to highlight specific application characteristics based on the product factors and their evaluations and whether the modeling and evaluation approach can be practically applied to use case applications in general. The used approach is summarized in Figure 3.6.

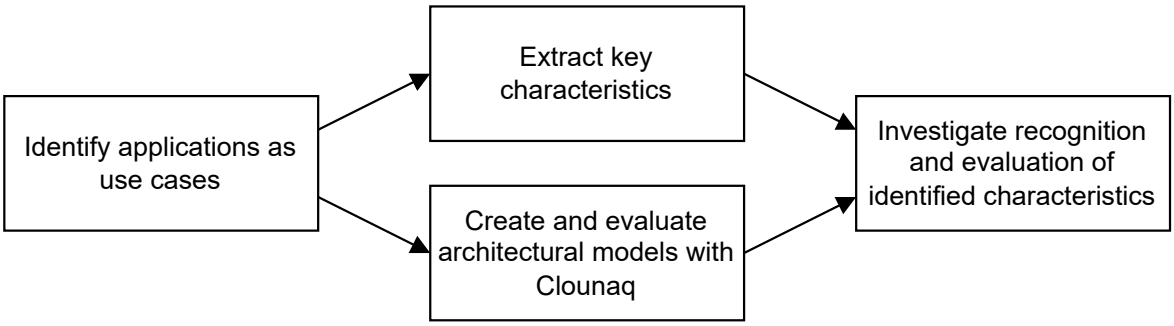


Figure 3.6.: Validation approach based on use case applications [173]

First, exemplary applications were identified and selected as use cases. Three applications were selected based on their relevance to the topic of cloud-native, their prior consideration in literature, and available documentation. These applications are:

TeaStore[35], built as a reference architecture for microservices-based systems to enable benchmarking experiments [293].

LakeSideMutual[36], implemented in the context of the Microservices API Patterns project[37], showcasing how microservices APIs can be designed.

Spring PetClinic Cloud[38], originally built to showcase features of the Spring Framework. This fork represents a distributed version that includes features from

[35] `https://github.com/DescartesResearch/TeaStore`, visited 2025-10-20
[36] `https://github.com/Microservice-API-Patterns/LakesideMutual`, visited 2025-10-20
[37] `https://microservice-api-patterns.org/`, visited 2025-10-20
[38] `https://github.com/spring-petclinic/spring-petclinic-cloud`, visited 2025-10-20

the Spring Cloud project[39]. This project supports the development of microservices-based cloud applications and has also been used in literature[282].

Next, for each of the applications, key characteristics were extracted. This was done by analyzing each application based on aspects such as the following:

- Which architectural style or which pattern is used?
- Which functionalities are implemented, and how are they provided?
- Which technologies are used for implementation, communication, storage, monitoring and configuration?
- How should the application be deployed?

The extracted key characteristics were related to product factors of the quality model to prepare for the following evaluation.

Next, the applications were manually modeled with the modeling approach corresponding to the quality model inside the Clounaq tool[40]. The models were created based on available source code, artifacts, and documentation. After finishing the modeling, the automated quality evaluation of each application architecture was triggered.

Finally, the extracted key characteristics were compared with the quality evaluation results to investigate how the characteristics are reflected in the evaluation results, whether the values of architectural measures capture these characteristics, and which additional insights are provided.

3.2. Implementation of Tooling Support

The major contribution of this work is the quality model and the quality model evaluation approach, formulated and validated as described in Section 3.1. However, to enable a practical application of the approach, tooling support is essential. As mentioned by Malavolta et al. [181] in their industry survey on requirements for architectural languages, graphical representation and usability were among the most missed features for architectural languages with usability being critical for the adoption of a language. Therefore, as an additional contribution of this work, all steps of the quality model evaluation approach are supported by a corresponding tool, named Clounaq. The supported steps are the modeling of software architectures, the automated calculation of architectural measures for a modeled architecture, and the evaluation of an architecture based on the quality model. It was decided to preliminarily provide only a manual modeling functionality, because a focus should be set on the approach together with the quality model as a whole. Implementing automated modeling support would require significant

[39] `https://spring.io/projects/spring-cloud`, visited 2025-10-20
[40] `https:/clounaq.de`, visited 2025-10-20

additional effort, because of the heterogeneity of technologies available in the context of cloud-native applications to which tooling in turn would need to be adapted. The implemented tool was also a key aspect for the validation steps E (3.1.5) and F (3.1.6).

To implement the tooling support in a structured way, it is important to specify beforehand what the tool should provide and which aspects are of importance. Thus, for the implementation, several guiding requirements have been formulated, according to which the tool was iteratively developed:

- **R1** The effort to install or set up the tool should be minimal so that it is easy to use.
- **R2** A graphical modeling feature should be included that facilitates the creation of architectural models of systems to evaluate.
- **R3** A textual representation of models should be supported as well so that models can be exported, shared, and imported.
- **R4** All evaluations should be explained and presented in a traceable way, so that results can be comprehended.
- **R5** Features for handling complexity should be included (for example shrink and expand actions, filtering possibilities, or aggregation features)
- **R6** The tool should be customizable and extendable, so that changes to the quality model can be easily introduced.

To support **R1**, it was decided to implement a web application that runs completely within the browser. As a result the tool could be hosted as a static website and no installation would be required from the user side. As a programming language, TypeScript[41] was chosen. Different graphical modeling (**R2**) libraries for JavaScript are available which were reviewed by Karolin Dürr in her master thesis as well [70]. A decision was made for JointJS[42], because at the time of the review, it showed to be the most stable and promising. Based on it, the modeling functionality (**R2**) was implemented in a first prototype [71] providing a graphical modeling option for the entities of the quality model. For the development of the graphical entity representations, Moody's design theory [195] for creating effective visual notations was considered. According to this theory, meaning should be indicated with the visual representations and representations have to be clearly distinguishable. In addition, a possibility to import and export architectural models based on the developed TOSCA profile was included (**R3**). Building on the first prototype, the Clounaq tool was further developed by including an interactive view of the quality model with explanations for all product factors and implemented measures associated with them (**R4**). Also, for the quality evaluation functionality, corresponding information on how the evaluation results are derived are provided (**R4**). To handle the potential complexity of both the quality model itself and created architectural models (**R5**), features to focus on certain entities, filter product factors

[41] `https://www.typescriptlang.org/`, visited 2025-10-20
[42] `https://www.jointjs.com/`, visited 2025-10-20

by categories and by associated quality aspects were included. Finally, to enable customization and extensibility (**R6**), a separation in the source code between specifications of quality model elements in plain JavaScript Object Notation (JSON) and the actual implementation of the tooling functionalities was enforced. Therefore, changes to the quality model, for example, adding a new impact relationship, only requires a change in one file where a JSON structure needs to be modified, without writing TypeScript code or similar.

The Clounaq tool is available as open-source software[43] and has been developed iteratively alongside the formulation and validation of the quality model. In contrast to the initial prototype [70], it now not only supports the modeling of application architectures, but also their automated evaluation. Furthermore, the quality model is included as an interactive diagram, it is possible to work on multiple architectural models in parallel, export and import functionalities have been added and from a technical perspective, the application has been migrated to Vue.js[44] as a foundation.

[43] `https://github.com/r0light/cna-quality-tool`, visited 2025-10-20
[44] `https://vuejs.org/`, visited 2025-10-20

Part II.

Results

> The need for measuring the quality of software products is almost as old as software engineering itself.
>
> *Rudolf Ferenc et al. [85]*

4. A Quality Model for Cloud-native Software Architectures

Parts of this chapter have been taken from [169, 171, 174].

In this chapter, hypothesis 2 *("A hierarchical quality model is able to express cloud-native architectural design decisions.")* is supported.

The elements of the quality model for cloud-native software architectures are described in the following. The state of the quality model described here is the result of all applied steps as described in Chapter 3. For previous versions of the quality model, please refer to the previous work [169, 171, 174].

4.1. Quality Model Overview

The quality model is formulated based on the Quamoco approach (Section 2.2.3.1) and the presentation of the quality model is thus structured based on the different elements as defined by Quamoco. An integrating overview is given in Figure 4.1. The quality model is a hierarchical quality model, and the top-level elements are the *quality aspects* placed circularly at the edges. *Product factors* are structured around these *quality aspects* with the advantage that *product factors* which impact several *quality aspects* can be drawn in the middle. The connections between factors represent *impact* relationships. An *impact* relationship signifies that the targeted factor is positively or negatively impacted, if the factor at which the *impact* originates, is present in an architecture. An interactive version of the quality model can also be found online[45]. In the following, first the *quality aspects* are defined in Section 4.2 structured by their corresponding high-level quality aspects. Since factors are based on *entities*, i.e., parts of a software architecture, these are presented in Section 4.3. At the core are then the *product factors*, listed in Section 4.4. The corresponding *impacts* are already added to the *product factors*, but considered in detail in Section 4.5. For the quantification of quality evaluations, *measures* are defined in Section 4.6. And the *evaluations* in Section 4.7 finally connect certain values of *measures* with results for the presence of *product factors* and their resulting impacts on *quality aspects*.

[45] `https://clounaq.de/quality-model`, visited 2025-10-20

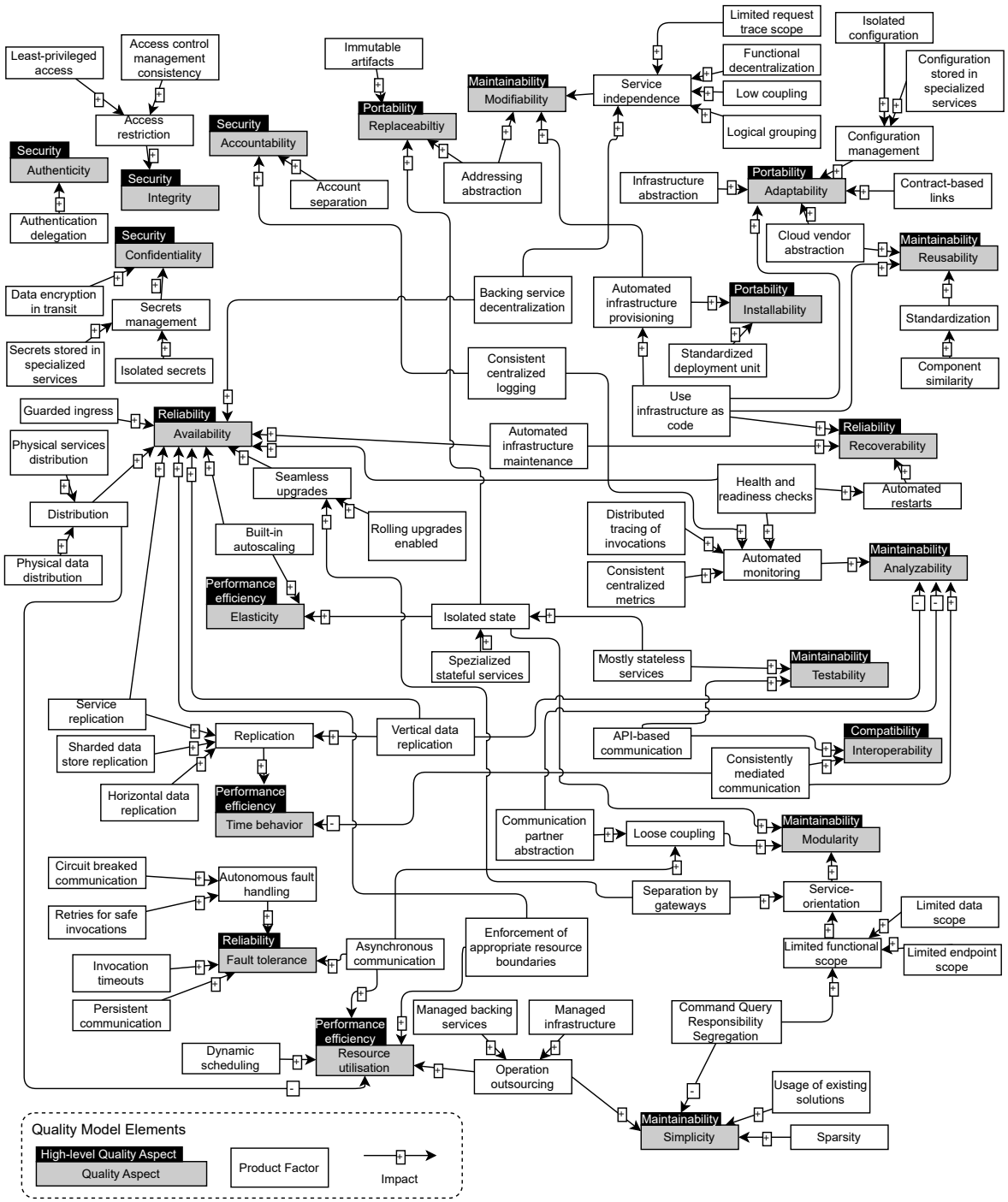


Figure 4.1.: The current state of the quality model

4.2. Quality Aspects

The ISO25000 standard [122] defines a hierarchy of quality aspects that allows for a structured and differentiating analysis of the quality of a system. The highest level quality aspects are *Security*, *Reliability*, *Maintainability*, *Performance efficiency*, *Portability*, and *Compatibility*. For each of these highest level quality aspects, which are rather abstract, sub-aspects are defined that further differentiate them. The quality model uses these quality aspects as the top-level quality aspects. The reason

for this is that it became apparent during the formulation of the quality model that for certain factors impacts to different sub-aspects of the same high-level aspect need to be modeled. While in general the quality aspects from ISO25000 could be adopted as-is, four aspects (*Capability*, *Maturity*, *Co-existence*, and *Non-repudiation*) are not included in the quality model. This is simply because no suitable factors were formulated that would be assignable to these quality aspects. In contrast, two new quality aspects were added: *Elasticity* (4.13) and *Simplicity* (4.10). *Elasticity* can be grouped under the high-level aspect of *Performance efficiency*, but it did not fit with any of the existing sub-aspects. Therefore, it was added as an independent quality aspect, similar to how Zheng et al. considered it [318]. *Simplicity* can be viewed as the positive counterpart to Complexity. It was added to the quality model under the high-level aspect of *Maintainability* in the same way as done by Plösch et al. [229] or Venkitachalam et al. [291]. It covers the general notion that complexity makes systems less maintainable and thus avoiding complexity by focusing on *Simplicity* has a positive impact on *Maintainability*.

In the following, all quality aspects are defined (mostly the definition from the ISO25000 standard [122] is adopted) and shortly described in the context of cloud computing.

4.2.1. Security

Although security is important for all kinds of software, it has a special importance for software running on cloud infrastructure. Cloud infrastructure is typically hosted and managed by an external cloud provider and thus shared with other clients of that provider. It is generally accessible via the public internet and communication also happens via potentially insecure networks. As stated by Gannon et al. [99]: "*Security is not an afterthought*" for cloud-native applications.

> **Definition 4.1 (Confidentiality)**
> *"Confidentiality describes to what extent data processed in a system is only accessible to those who actually need it and is otherwise protected from illegitimate access."* [122]

Confidentiality considers mostly data that is used in a cloud application. That typically means data that is stored in cloud databases or cloud file systems. But also data that is provided via API endpoints of components of the application needs to be protected from illegitimate access [230].

> **Definition 4.2 (Integrity)**
> *"Integrity describes how well a system is able to prevent unauthorized access or manipulation of functions and data."* [122]

Integrity goes a bit further, considering not only the protection of data from read access, but also from write access by unauthorized entities. It applies not only to

the data that is processed, but also to the components of an application themselves. If an attacker targets integrity and gains access to the components of an application, e.g. through container vulnerabilities [272], and executes malicious code, the consequences are far-reaching.

> **Definition 4.3 (Accountability)**
> *"Accountability describes to what extent it is possible in a system to trace back actions that have taken place back to the subject that performed them."* [122]

Cloud platforms nowadays have advanced Identity and Access Management (IAM) systems which also enables the assignment of accounts to different components of an application.

Thus, *Accountability* can be ensured if used appropriately to make sure actions happening within a cloud platform can be traced back to components and users [273].

> **Definition 4.4 (Authenticity)**
> *"Authenticity describes how well a system is able to identify a subject and validate its identity as well as claims made by a subject."* [122]

For *Authenticity*, various technologies and approaches are available nowadays that can be used to clearly identify users and manage their access rights, for example via different roles. Apart from how users are authenticated, it is also important for authenticity how the authentication approach within a cloud application is implemented [112] so that it can be extended to other components and used in a consistent way.

4.2.2. Maintainability

Maintainability is the quality aspect that is often the first association when talking about software quality. Although a core aspect of cloud computing is that tasks and responsibilities are handed over to the cloud platform to facilitate the operation and maintenance of an application, other complexities can arise that need to be addressed.

> **Definition 4.5 (Modularity)**
> *"Modularity describes how well a system is composed of different components, so that a change in one component has a minimal impact on other components."* [122]

Components in the context of cloud-native applications are typically services implemented for example with a microservices architectural style or based on serverless functions. What a well-modularized application looks like depends on the business domain and which functionalities are implemented in which services, but also largely on how services communicate with each other. Principles of

service-oriented architectures [64] are therefore important for a well-modularized cloud-native application.

Definition 4.6 (Reusability)
"Reusability describes to what extent parts of a system can be used in more than one system." [122]

Reusability in cloud-native applications is especially relevant for backing services that provide functionalities needed in various applications. By reusing complete components or parts of components, applications can be developed faster and maintained in a similar way.

Definition 4.7 (Analyzability)
"Analyzability describes to what extent it is possible to accurately assess the impact of an intended change as well as the extent to which failures can be diagnosed to find their cause or parts that need to be changed can be identified." [122]

To be able to analyze a system, it must provide corresponding information about its state and behavior, for example in the form of metrics or logs [149]. Components or infrastructure should therefore be used that can provide this information, ideally in a holistic and consistent way.

Definition 4.8 (Modifiability)
"Modifiability describes how well a system can be modified without introducing defects or degrading other qualities of the system." [122]

Modifiability of cloud-native applications considers changes and updates to components or infrastructure and how well, that means how fast and error-free, these can be applied [253]. Important concepts for *modifiability* are keeping components independent, automation, and abstraction.

Definition 4.9 (Testability)
"Testability describes how effective test criteria can be defined and used for a system are to check the intended behavior of a system as well as how facile it is to perform the tests to determine whether the test criteria are met." [122]

Testing cloud-native applications can take more effort because the usage of functionalities only available in a cloud environment requires tests to be executed in the cloud environment [32]. However, also techniques and approaches are available that enable a local (partial) testing of applications. Building applications in a way that still enable their *testability* is thus important.

Definition 4.10 (Simplicity)
Simplicity describes how well a system is composed of as few components as possible and includes simple instead of complex interrelations to enable a good overview and understanding of the system.

Finally, a core point of cloud computing is that complexity is taken care of by the cloud provider to simplify application development and operation [4].

Ensuring *simplicity* as the counterpart to complexity is thus important for the maintainability of an application.

4.2.3. Performance Efficiency

Performance efficiency is an important quality aspect for cloud-native applications because of the core promise of cloud computing that resources are available on-demand and basically unlimited. That means resources should be acquirable up to the desired performance level, and it is more a question of whether the acquired resources are used efficiently and how much they cost, rather than whether they are available.

Definition 4.11 (Time-behavior)
"Time-behavior describes how well a system performs in terms of processing and response times as well as the throughput rate when performing its functions." [122]

The performance of an application with respect to time depends on the one hand on its design (for example how much data needs to be transferred between how many components), but on the other hand also on how many resources are available while processing a task [193]. Since cloud applications are typically distributed, this communication overhead needs to be taken into consideration. It needs to be evaluated how the provisioning of more resources impacts *time-behavior*.

Definition 4.12 (Resource utilization)
"Resource utilization describes to what extent resources are available and used as required by a system when performing its functions, in terms of storage space needed, CPU utilization, memory usage, or network usage." [122]

As already stated, cloud platforms offer virtually unlimited resources. However, the acquired resources must be fitted to the actual demand for a good *resource utilization*.

Definition 4.13 (Elasticity)
Elasticity describes the rapidness and accurateness with which a system is able to adjust its allocated resources to the currently required amount without over- or under-allocation.

Elasticity is a quality aspect specific to cloud applications. It complements time-behavior and resource utilization with the aspect of how well and how fast an

application is able to adapt its current resources to changing demands [113] in order to provide a good performance and be resource efficient.

4.2.4. Portability

Portability in the context of cloud applications is focused mostly on the connection point between cloud consumer, i.e. application developer, and cloud provider. Depending on which cloud service model and which specific cloud services are chosen, porting an application from one cloud environment to another can be more or less difficult. The more responsibility is taken over by the provider, the less portable an application typically is. The phenomenon of being bound to a certain provider is commonly referred to as *vendor lock-in* in the literature.

> **Definition 4.14 (Adaptability)**
> *"Adaptability describes how well and how easy a system can be adapted to be executed on different or evolving software, platforms, environments, or hardware."* [122]

One aspect of portability is *adaptability* focusing on the specific changes that are necessary to port an application from one environment to another [312]. The complexity and amount of necessary changes depend on how well an application can abstract from specific technologies, services, or underlying infrastructure.

> **Definition 4.15 (Installability)**
> *"Installability describes how well a system can be installed or uninstalled completely and correctly in a specific environment."* [122]

Apart from necessary changes, also the installation process itself needs to be considered. In general, with automation and descriptive style, *installability* can be improved [253].

> **Definition 4.16 (Replaceability)**
> *"Replaceability describes how well a component or system can replace another component or system for the same purpose in the same environment."* [122]

Finally, portability can also be supported by ensuring *replaceability* of components. If components can be easily replaced, it is easier to react to common issues in cloud computing. For example, breaking changes in service offerings, deprecation of technologies, license changes or changes in pricing. A core aspect for *replaceability* is the reliance on standards [215] and specifications that are supported by a range of providers.

4.2.5. Reliability

The specific characteristics of cloud environments have significant consequences on the *reliability* of applications. On the one hand, due to the on-demand access to

resources, applications can quickly react to problems and avoid longer downtimes for example. But on the other hand, cloud providers can only provide guarantees on their services up to a certain level. Developers of cloud applications need to prepare for the fact that parts of a cloud infrastructure can be unreliable.

> **Definition 4.17 (Availability)**
> *"Availability describes to what extent a system is operational and accessible at any point in time when it is needed."* [122]

A key approach to *availability* is redundancy and replication. Thus, cloud environments, which are typically based on globally distributed datacenters, are well-positioned to enable a distributed replication of application components [200]. Developers of cloud-native applications nevertheless need to build on these possibilities in a suitable way in order to ensure *availability* of their applications.

> **Definition 4.18 (Fault tolerance)**
> *"Fault tolerance describes how well a system is able to operate even when facing software or hardware faults."* [122]

Cloud-native applications need to expect certain faults, such as network partitionings, connection issues, or virtual machine crashes [124]. Therefore, mechanisms need to be built into cloud-native applications that can deal with these issues and avoid negative impacts, essentially making them *fault-tolerant*.

> **Definition 4.19 (Recoverability)**
> *"Recoverability describes how well a system is able to recover and return to the intended state after an interruption or failure."* [122]

In contrast to *fault tolerance* which is concerned with how a system behaves while a failure is acute, *recoverability* is concerned with how a system can return from the failure state [59]. Therefore, additional measures need to be in place that for example restart components or provision new resources when necessary.

4.2.6. Compatibility

As the last quality aspect considered in the scope of this work, *compatibility* covers the aspect of how well components or systems can be integrated. In the context of cloud computing, a common approach of cloud providers is that their own service offerings are well integrated to make it more attractive for cloud consumers to stay within one ecosystem. An interesting aspect therefore is how compatibility can be ensured independently of sticking with a single cloud provider.

Definition 4.20 (Interoperability)
"Interoperability describes how well two parts of a system or two systems are able to exchange information and to process such exchanged information." [122]

Interoperability is the most relevant quality aspect in the context of compatibility, and it is concerned with how two components or systems can exchange information. This aspect is less specific to cloud computing, because it is impacted more by how components communicate, that means which kind of APIs are used and which data formats are applied. For example, APIs should be based on commonly used technologies and documented accordingly [215]. And also the data formats should be commonly used ones so that already available libraries and tooling can be used.

4.3. Entities

Before the product factors of the quality model are presented, the entities of the quality model are defined. These entities were initially formulated based on the product factors [169] with the goal of representing a software architecture in a way that the information required by the product factors can be included in the model. In an iterative process, the product factors and entities were then refined, using the defined entities to formulate the product factors more precisely. Thus, for a better understanding, the entities are now presented before the product factors. Because of the technological heterogeneity of cloud-native applications, the entities are formulated in a more abstract way, independent of specific implementations (with the *Artifacts* as an exception). An overview of all entities is provided in Table 4.1 with short descriptions for each entity. Furthermore, the formal definition of entities is summarized at the end of the section in Listing 4.1 and Listing 4.2. This is needed later for a precise definition of the measures in Section 4.6.

A single cloud-native application is represented by the *System* entity:
$SYS := (C, L, I, DM, RT, DA, BD, N)$

It acts as a wrapper for all other entities. Before discussing the main entities, *Component* (C) and *Infrastructure* (I), the more foundational and independent entities *Data Aggregate* (DA), *Backing Data* (BD), and *Network* (N) are explained. All entities can be uniquely identified by an *id* and have a *name*. Furthermore, all entities are characterized by properties (*props*) that are specific to each entity.

An important aspect for cloud-native applications is how data is processed by components. To represent data that is relevant from a domain perspective, the *Data Aggregate* entity ($DA := (id, name, props)$) is intended to be used. The name is adopted from DDD and the intention of the entity is to represent data on a more coarse-grained level. This hides, for example, details of specific fields or data types, but allows for a differentiation between data that is used in different parts of an application to provide different functionalities. An example of a *Data Aggregate* in the context of a web shop could be an `Order` *Data Aggregate*. In

Table 4.1.: Entities of the quality model

Name	Description	Relation
System (SYS)	The cloud-native application as a whole	is-a system
Component (C)	An abstract entity for representing distinguishable executable parts of the system that provide certain functionalities. It can for example be a service or a certain cloud resource. Regarding its granularity, it should, generally speaking, correspond to something that can be run as an OS process.	part-of system
Service (S)	A component that implements a business functionality.	is-a component
Backing Service (BS)	A component providing general functionalities needed by services, for example, messaging, logging.	is-a component
Storage Backing Service (SBS)	An explicitly stateful component used to store business data, e.g., a database.	is-a component
Proxy Backing Service (PBS)	A component which can act as a proxy for all kinds of communication (links) between other components.	is-a component
Broker Backing Service (BBS)	A component which acts as a communication broker, for example a message broker or an event store.	is-a component
Endpoint (E)	A communication endpoint, for example a REST endpoint, message producer/listener.	part-of component
External Endpoint (EE)	An endpoint which is explicitly publicly available.	is-a endpoint
Link (L)	A directed potential connection between a specific component and a specific endpoint of a different component. Potential in this case refers to the design time perspective, meaning that a component is implemented so that it can invoke the respective endpoint.	part-of system
Infrastructure (I)	The technical foundation where components are deployed, e.g., a container orchestration system.	part-of system
Deployment Mapping (DM)	A connection between a component or infrastructure and its underlying infrastructure on which that component or infrastructure is deployed.	part-of system
Request Trace (RT)	The whole resulting trace of a service invocation from the outside that means when an external endpoint is invoked. A request trace includes a collection of components and links.	part-of system
Data Aggregate (DA)	An aggregate which needs to be persisted and is used by services, e.g., business objects.	part-of system
Backing Data (BD)	Non-business data, e.g., config values, secrets, logs, metrics	part-of system
Network (N)	A network or subnet which covers a range of (ip) addresses and to which components and infrastructure entities can be assigned to.	part-of system
Artifact (A)	An artifact associated with a component or an infrastructure entity that enables the actual deployment or execution of that entity. For example, a Jar-File or a Container Image	part-of component

contrast, data that is used and processed by an application, but not directly relevant to the specific domain of an application, can be represented by the *Backing Data* entity: $BD := (id, name, props)$. Examples for *Backing Data* are configura-

tion data, secrets, logs, or metrics. With the properties of the *Backing Data* entity $props_{BD} := \{kind, included\_data\}$ it can be specified what *kind* of data a *Backing Data* entity represents and the data that is included.

Network entities $N := (id, name, props)$ represent networks of which *Components* or *Infrastructure* entities can be a part of. Examples are Virtual Private Clouds (VPCs) or subnets within VPCs that are commonly used in cloud environments to group together or isolate resources. With the corresponding properties further details of a *Network* can be specified: $props_N := \{ip\_version, cidr, start\_ip, end\_ip, gateway\_ip, network\_type\}$.

The first core element to represent application architectures are *Components*: $C := (id, name, props, RDA_C, RBD_C, RN_C, artifacts, providedEndpoints, externalIngressProxiedBy, ingressProxiedBy, egressProxiedBy, addressResolutionBy, authenticationBy)$.
A *Component* is an abstract entity used to represent various parts of an architecture which implement necessary functionalities of an application. A *Component* is software that offers functionalities that are either specific to the application domain or general functionalities, like messaging, logging, storage, or configuration. A Component can be described in more detail with the following properties:
$props_C := \{managed, software\_type, stateless, load\_shedding, identities, namespace\}$
In addition to *id, name,* and *props, Components* can have several relations to other entities: One or more *Data Aggregates* can be assigned to a *Component* if that *Component* uses them via relations: $RDA_C \subseteq C \times DA$. In the same way, also one or more *Backing Data* entities can be assigned to a *Component* if it uses them via relations $RBD_C \subseteq C \times BD$. The relations between *Components* and *Data Aggregates* or *Backing Data* entities can also be described in more detail via properties: $props_{RDA_C} := \{usage\_relation, sharding\_level\}$ for relations to *Data Aggregates* and $props_{RBD_C} := \{usage\_relation\}$ for relations to *Backing Data* entities. Especially the `usage_relation` property is important, because it describes whether a *Component* only uses a *Data Aggregate* or whether it persists it. A *Component* can be assigned to one or more *Networks* in which it is included via relations $RN_C \subseteq C \times N$. The *Artifacts* associated with a *Component* are specified with $artifacts \subseteq A$. *Artifacts* ($A := (id, type, props)$) can be added to *Components* or *Infrastructure* entities and represent the actual artifacts used to implement, deploy or document an entity. Examples for *Artifacts* are Java Archives (JARs) or container images.

To enable communication, each *Component* can have a number of *Endpoints* that it provides, specified with $providedEndpoints \subseteq E$. They are intended to represent an interface of a component in a more fine-grained way. An Endpoint $E := (id, name, props, RDA_E, documentedBy)$ is modeled in detail with the following properties:
$props_E := \{kind, method\_name, protocol, port, supported\_authentication\_methods, url\_path, rate\_limiting,$

$idempotent, readiness\_check, health\_check\}$. In the exemplary case of a REST API, one endpoint entity could represent one combination of a URL(specified with *url_path*) and an HTTP verb (specified with *method_name*), but may also be used even more granular (e.g., considering parameters) if needed. For an *Endpoint* it can also be specified which *Data Aggregates* are used by it with $RDA_E \subseteq E \times DA$. Furthermore, if they are documented explicitly with an *Artifact*, this is set with $documentedBy \subseteq A$. *External Endpoints* are more specific than *Endpoints*: $EE \subseteq E$. They represent *Endpoints* that are made publicly available and thus represent entry points to the application. Lastly, the following additional relations can be specified for a *Component*:

- The external ingress of a *Component* can be proxied by a *Proxy Backing Service* set with $externalIngressProxiedBy \in PBS$.
- The internal ingress of a *Component* can be proxied by a *Proxy Backing Service* set with $ingressProxiedBy \in PBS$.
- The egress of a *Component* can be proxied by a *Proxy Backing Service* set with $egressProxiedBy \in PBS$
- The addresses a *Component* uses to call endpoints of other components can be resolved by a *Backing Service*, an *Infrastructure* entity or a *Network*. This can be set with $addressResolutionBy \in BS \cup PBS \cup I \cup N$

While the *Component* entity itself can be used as a general entity, a range of more specific entities that are in an inheritance relationship with *Component* are available and should be used preferably. These are *Service*, *Backing Service*, *Storage Backing Service*, *Broker Backing Service*, and *Proxy Backing Service*. Since they extend the *Component* entity, they share the same properties, but can extend them with more specific properties. And also the mentioned relations to other entities apply to these more specific entities.

A *Service* ($S \subseteq C$) is a component that provides core functionalities specific to the domain of the application in focus. That means, an application should have at least one *Service* that implements the features relevant to the application in focus. In the case of a microservices-based application, each *Service* represents a separate part of the application.

In contrast, *Backing Service* entities ($SBS \subseteq C$) are intended to model parts of an application which do not provide key functionalities, but are nevertheless necessary for the operation of an application. Examples are configuration services, logging services, or authentication services. The specific functionality provided by a *Backing Service* can be specified with the *providedFunctionality* property listed in $props_{BS} := props_C \cup \{providedFunctionality, replication\_strategy, address\_resolution\_kind\}$. For several other functionalities, more specific entities are available:

Storage Backing Services ($SBS \subseteq C$) are all parts of an application that provide storage and persist data of an application. The typical example would be a database. Since these are explicitly stateful, also a *replication_strategy* can be specified as part of its properties: $props_{SBS} := props_C \cup \{name, shards, replication\_strategy\}$.

Broker Backing Services ($BBS \subseteq C$) are concerned with communication capabilities and should be used for example to model messaging brokers. Their properties $props_{BBS} := props_C \cup \{kind, replication\_strategy\}$ also include the *replication_strategy* property, because they can be stateful.

Finally, *Proxy Backing Services* ($PBS \subseteq C$) represent components that mediate communication either externally, for example as a load balancer, or internally, for example as a sidecar proxy within a service mesh. The specific *kind* of proxy, can be set in its properties $props_{PBS} := props_C \cup \{kind, load\_balancing\}$. If one of the more specific entities is not suitable for a model, the more abstract *Component* entity should be used.

To connect *Components*, communication relationships can be represented with *Links*. A *Link* ($L := (id, props, sourceComponent, targetEndpoint)$) is a directed connection between a $sourceComponent \in C$ and a $targetEndpoint \in E$ of another *Component*: $targetEndpoint \notin sourceComponent.E$. The properties of the *Link* entity $props_L := \{relation\_type, timeout, retries, circuit\_breaker\}$ enable the modeling of further characteristics of communication.

While *Links* represent individual potential invocations of *Endpoints* from certain *Components*, functionalities of applications are typically provided by a cooperation of several components. Such cooperations can be characterized by how a number of *Links* are causally related, i.e. invoked sequentially or in parallel to process a specific request. Therefore, *Request Trace* entities ($RT := (id, name, props, involvedLinks, referencedEndpoint)$) can be used to represent the resulting traces from such cooperations. They group a number of $involvedLinks \subseteq L$ in a structured way and reference a single *External Endpoint* ($referencedEndpoint \in EE$) which is the trigger for the execution of a *Request Trace*. The components that are part of a request trace as a result, are additionally listed in the properties: $props_{RT} := \{nodes\}$.

The second core element of an architectural model is represented by the *Infrastructure* entity ($I := (id, name, props, artifacts, RBD_I, RN_I)$). It covers all parts of an architecture that are needed to host and execute the *Components* of an application. The defining characteristic of an *Infrastructure* entity is that it shares (i.e., provides) its resources (processor, memory, disk) to another entity that can be either an additional layer of *Infrastructure* or a *Component*. For an *infrastructure* entity, also $artifacts \subseteq A$ can be specified. And an *infrastructure* entity can reference a *Backing Data* entity if it uses it, for example, if a certain configuration is stored in an *Infrastructure* entity. This is specified with $RBD_I \subseteq I \times BD$. And, similar to *Components*, *Infrastructure* entities can be assigned to one or more *Networks* in which they are included: $RN_I \subseteq I \times N$. *Infrastructure* entities also include a range of properties to further characterize them: $props_I :=$

$\{kind, environment\_access, maintenance, provisioning,$
$supported\_artifacts, availability\_zone, region, supported\_update\_strategies,$
$deployed\_entities\_scaling, self\_scaling,$
$enforced\_resource\_bounds, identities, address\_resolution\_kind\}$.

Finally, the connecting element for *Infrastructure* entities are *Deployment Mappings*. A *Deployment Mapping* ($DM := (id, props, deployed, host)$) describes a hosting relationship between an *Infrastructure* entity $host \in I$ and a hosted entity, that can be either another *Infrastructure* entity or a *Component*: $deployed \in C \cup I$. A *Component* or *Infrastructure* entity can have multiple *Deployment Mappings* to different *Infrastructure* entities and an *Infrastructure* entity can also host multiple *Component* or *Infrastructure* entities. But an *Infrastructure* entity cannot host itself: $deployed \neq host$. *Deployment Mappings* are characterized with a range of properties: $props_{DM} := \{deployment, deployment\_unit, replicas, update\_strategy,$ $automated\_restart\_policy, assigned\_account, resource\_requirements\}$.

As already described, each entity has a number of properties that further characterize an entity and which are important for the specification of measures later. These properties are listed in detail in Table 4.2, Table 4.3, Table 4.4, Table 4.5, and Table 4.6. Each property is identified by a specific **property key** that is unique per property and entity. The **value** column describes which value a property may hold. If it includes an enumeration of specific values, the value has to be one out of these enumerated options.

The specification of properties for the different entities has been done in an iterative process based on the requirements of the factors of the quality models. That means entities were initially formulated without any properties. While developing the quality model, properties were gradually added as required. The underlying trade-off is that, on the one hand, a modeled software architecture needs to contain the information required to evaluate the factors of the quality model. And on the other hand, the software architecture model should be as minimal as possible in order to reduce complexity and keeping it maintainable, especially when changes or additions need to be made over time.

Taking it all together, the presented entities with their respective properties enable a focused description of cloud-native software architectures. The information included in an architecture modeled with these entities, especially how entities are related and which properties they have, is used by the measures presented in Section 4.6 to enable quantitative quality evaluations.

Table 4.2.: Entity properties of components, relations to data aggregates, and relations to backing data.

Entity	Property key	Description	Value
Component	managed	A component is managed if it is exclusively operated by someone else, e.g. a cloud provider and the source code of the component instance is inaccessible. If the source code of a component can be changed by yourself, the component is not managed.	boolean
	software_type	The type of the software in the sense of who developed it. If it is a self-written component use "custom", if it is an existing open-source solution which is not customized (apart from configuration) use "open-source". If it is licensed proprietary software, use "proprietary".	custom, open-source, proprietary
	stateless	True if this component is stateless, that means it requires no disk storage space where data is persisted between executions. That means it can store data to disk, but should not rely on this data to be available for following executions. Instead, it should be able to restore required data after a restart in a different environment.	boolean
	load_shedding	Whether or not this component applies load shedding. That means whether the component rejects incoming load based on certain thresholds (resource usage, concurrent requests).	boolean
	identities	The identities of this component, such as account names, users or roles.	map
	namespace	If the underlying infrastructure enables the usage of namespaces, this can be specified here to logically group components. If not, everything is in the same default namespace	text
$RDA_{C/E}$	usage_relation	Describes how a component uses attached data, that means whether it just uses (reads) it for its functionality or if it also updates and persists (writes) it; possible values are usage, cached usage, and persistence	usage, cached-usage, persistence
	sharding_level	Only applicable if data is persisted by a component; If a component persists data, the sharding level describes the number of shards used; 0 acts as a placeholder if data is not persisted; 1 is the default meaning that no sharding is used; >1 is the number of shards	integer
$RBD_{C/I}$	usage_relation	Describes how a component uses attached data, that means whether it just uses (reads) it for its functionality or if it also updates and persists (writes) it; possible values are usage, cached usage, and persistence	usage, cached-usage, persistence

Table 4.3.: Entity properties of artifacts and backing services.

Entity	Property key	Description	Value
Artifact	provider_specific	Whether this artifact is (cloud) provider-specific or not.	boolean
	based_on_standard	If the artifact is based on a standard, specify it here.	none, OCI, OpenAPI, other
	self_contained	Whether this artifact is self-contained or not, that means whether it needs additional resources explicitly added to it to be used.	boolean
Backing Service	providedFunctionality	The functionality provided by this backing service	naming/addressing, configuration, authentication/authorization, logging, metrics, tracing, vault, other
	replication_strategy	The strategy used to replicate data, if multiple replicas of this backing service are deployed.	none, read-only-replication, active-active-replication
	address_resolution_kind	If address resolution is provided, state the kind here, otherwise leave as 'none'	none, discovery, DNS, other
Storage Backing Service	name	the logical name of the database	text
	shards	The number of shards this storage service is configured with, the default of 1 means no sharding is used.	number
	replication_strategy	The strategy used to replicate data, if multiple replicas of this storage backing service are deployed.	none, read-only-replication, active-active-replication
Proxy Backing Service	kind	The kind of proxy this is.	API Gateway, Load Balancer, Service Mesh, other
	load_balancing	Flag whether the proxy functionality includes load balancing	boolean
Broker Backing Service	kind	The kind of broker based on how messages are persisted and delivered. Queue is the classical message broker where messages are stored in a queue, delivered to one consumer and deleted afterwards. Topic enables a delivery of one message to several consumers, but messages are also deleted. Log enables the Event sourcing pattern, since messages/events are persistently stored and can be retrieved at any time.	queue, topic, log
	replication_strategy	The strategy used to replicate data, if multiple replicas of this broker backing service are deployed.	none, read-only-replication, active-active-replication

Table 4.4.: Entity properties of endpoints and links.

Entity	Property key	Description	Value
Endpoint	kind	The kind of endpoint which can be either "query", "command", or "event". A "query" is a synchronous request for data which the client needs for further processing. A "command" is a synchronous send of data for which the client needs a corresponding answer for further processing. An "event" is an asynchronous send of data for which the client does not expect data to be returned for further processing.	query, command, send event, subscribe
	method_name	An optional name of the method/action used. For REST APIs it can for example be specified whether it is a GET or POST method. For message brokers it can be specified whether it is a publish or subscribe action.	text
	protocol	The name of the (Layer 4 through 7) protocol that the endpoint accepts.	text
	port	The optional port of the endpoint.	text
	supported_authentication_methods	A list of authentication methods, that this endpoint supports. If the list is empty, the endpoint does not require authentication.	list
	url_path	The optional URL path of the endpoint's address if applicable for the protocol.	text
	rate_limiting	If for this endpoint rate limiting is enforced, the limit can be stated here, otherwise it is "none".	text
	idempotent	Flag to specify whether this endpoint is idempotent, meaning that the effect of a successful invocation is independent of the number of times it is invoked.	boolean
	readiness_check	Flag to specify whether this endpoint is used as a readiness check	boolean
	health_check	Flag to specify whether this endpoint is used as a health check	boolean
Link	relation_type	Type of relation, e.g. subscribes to or calls	text
	timeout	If a timeout is applied for this link, specify its length here. If no timeout is applied, use 0.	number
	retries	Number of times this invocation is retried, if it fails.	number
	circuit_breaker	Whether or not this invocation is protected by a circuit breaker	text

Table 4.5.: Entity properties of infrastructure entities.

Entity	Property key	Description	Value
Infrastructure	kind	The kind of infrastructure. Possible kinds are ”physical hardware”, ”virtual hardware”, ”software platform”, or ”cloud service”.	physical-hardware, virtual-hardware, software-platform, cloud-service
	environment_access	Describes the extent of available access to the environment in which the infrastructure is operated. With full access, one can control all aspects of the infrastructure. Limited access means that infrastructure is under control of a provider and only certain things are allowed, such as configuration. With no access infrastructure is completely managed by a cloud provider.	full, limited, none
	maintenance	How infrastructure is maintained, that means for example how updates are installed or how certificates are regenerated.	manual, automated, transparent
	provisioning	How infrastructure is initially provisioned. This can be done manually (for example through the web interface of a cloud provider), automatically coded (for example through an IaC tool), automatically inferred, if it is inferred based on deployed components, or transparent, if it is not explicitly provisioned by a consumer, but done on-demand by a provider.	manual, automated-coded, automated-inferred, transparent
	supported_artifacts	Which kind of artifacts can be deployed on this infrastructure, e.g. VM images, container images, jar archives, native executables, ...	list
	availability_zone	The name of the availability zone in which this infrastructure is provided. If it is running in multiple availability zones, provide their names as a comma-separated list.	text
	region	The name of the region in which this infrastructure is provided.	text
	supported_update_strategies	Which update strategies are supported for entities deployed on this infrastructure	list
	deployed_entities_scaling	Whether and how entities deployed on this infrastructure can be scaled horizontally	none, manual, automated-built-in, automated-separate
	self_scaling	Whether and how this entity scales itself, either horizontally or vertically.	none, manual, automated-built-in, automated-separate
	enforced_resource_bounds	Set to true if the infrastructure enforces resource bounds on deployed components, for example regarding cpu shares or memory size. Deployed entities can then only use resources up to a certain bound. Otherwise, entities can use resources as available.	boolean
	identities	The identities of this infrastructure, such as account names, users or roles.	map
	address_resolution_kind	If address resolution is provided, state the kind here, otherwise leave as ’none’	none, discovery, DNS, other

Table 4.6.: Entity properties of deployment mappings, request traces, backing data and networks.

Entity	Property key	Description	Value
Deployment Mapping	deployment	How this deployment mapping is described or ensured. Valid values can be manual, automated imperative, or automated declarative	manual, automated-imperative, automated-declarative, transparent
	deployment_unit	When this deployment mapping can be described as a specific unit of deployment (e.g. a Kubernetes Pod), state it here. Otherwise, it is custom.	custom, CNA.Artifact, Kubernetes.Resource, Kubernetes.Resource.Service, Kubernetes.Resource.Deployment, Implementation, Implementation.Bash, Implementation.Python, Implementation.Java, Implementation.JavaScript, Image.Container, Image.Container.OCI, Image.VM, Terraform.Script, Pulumi.Script, Ansible.Script, Chef.Script, Puppet.Script, CloudFormation.Script, AWS.Resource, AWS.EKS.Cluster, AWS.EC2.Instance, AWS.EC2.LoadBalancer, AWS.EC2.NodeGroup, AWS.Beanstalk.Application, AWS.RDS.Instance, Azure.Resource, Azure.ResourceManagerTemplate, GCP.Resource, OpenAPI, Spring.CloudContract, Pact.Contract
	replicas	The minimum number of replicated instances for the deployed component when it is running	number
	update_strategy	How the deployed entity is updated in case a new version is deployed. It can simply be replaced by an instance of the new version, or specific strategies like "rolling upgrade" or "blue-green" can be used.	in-place, replace, rolling, blue-green
	automated_restart_policy	If the deployed entity is automatically restarted in case of failure.	never, onReboot, onProcessFailure, onHealthFailure
	assigned_account	The name of the account under which this component is deployed (e.g. a service account from a cloud provider)	text
	resource_requirements	Explicitly stated resource requirements for this deployment. If stated the infrastructure can schedule an entity accordingly.	text
Request Trace	nodes		list
Backing Data	kind	The kind of backing data, meaning what kind of data this refers to	config, secret, logs, metrics
	included_data		map
Network	ip_version	The IP version of the requested Valid values are 4 for ipv4 or 6 for ipv6.	number
	cidr	The cidr block of the requested	text
	start_ip	The IP address to be used as the start of a pool of addresses within the full IP range derived from the cidr block.	text
	end_ip	The IP address to be used as the end of a pool of addresses within the full IP range derived from the cidr block.	text
	gateway_ip	The gateway IP address.	text
	network_type	Specifies the nature of the network in the underlying cloud infrastructure.	text

Listing 4.1: Formal specifiation of entities - I

$$
\begin{aligned}
&SYS := (C, L, I, DM, RT, DA, BD, N)\\
&DA := (id, name)\\
&BD := (id, name, props)\\
&\quad props_{BD} := \{kind, included\_data\}\\
&N := (id, name, props)\\
&\quad props_{N} := \{ip\_version, cidr, start\_ip, end\_ip, gateway\_ip, network\_type\}\\
&C := (id, name, props, providedEndpoints, artifacts, RDA_C, RBD_C, RN_C,\\
&\quad externalIngressProxiedBy, ingressProxiedBy, egressProxiedBy,\\
&\quad addressResolutionBy, authenticationBy)\\
&\quad props_{C} := \{managed, software\_type, stateless, load\_shedding,\\
&\quad identities, namespace\}\\
&\quad providedEndpoints \subseteq E\\
&\quad artifacts \subseteq A\\
&\quad RDA_C \subseteq C \times DA\\
&\quad props_{RDA_C} := \{usage\_relation, sharding\_level\}\\
&\quad RBD_C \subseteq C \times BD\\
&\quad props_{RBD_C} := \{usage\_relation\}\\
&\quad RN_C \subseteq C \times N\\
&\quad externalIngressProxiedBy \in PBS\\
&\quad ingressProxiedBy \in PBS\\
&\quad egressProxiedBy \in PBS\\
&\quad addressResolutionBy \in BS \cup PBS \cup I \cup N\\
&\quad authenticationBy \in BS\\
&S \subseteq C\\
&BS \subseteq C\\
&\quad props_{BS} := props_C \cup \{providedFunctionality,\\
&\quad replication\_strategy, address\_resolution\_kind\}\\
&SBS \subseteq C\\
&\quad props_{SBS} := props_C \cup \{name, shards, replication\_strategy\}\\
&PBS \subseteq C\\
&\quad props_{PBS} := props_C \cup \{kind, load\_balancing\}\\
&BBS \subseteq C\\
&\quad props_{BBS} := props_C \cup \{kind, replication\_strategy\}\\
&E := (id, name, props, RDA_E, documentedBy)\\
&\quad props_{E} := \{kind, method\_name, protocol, port,\\
&\quad supported\_authentication\_methods, url\_path, rate\_limiting,\\
&\quad idempotent, readiness\_check, health\_check\}\\
&\quad RDA_E \subseteq E \times DA\\
&\quad props_{RDA_E} := \{usage\_relation, sharding\_level\}\\
&\quad documentedBy \subseteq A\\
&\quad allow\_acces\_to \subseteq accounts\\
&EE \subseteq E
\end{aligned}
$$

Listing 4.2: Formal specifiation of entities - II

$A := (id, type, props)$
$\quad props_A := \{provider\_specific, based\_on\_standard, self\_contained\}$
$L := (id, props, sourceComponent, targetEndpoint)$
$\quad props_L := \{relation\_type, timeout, retries, circuit\_breaker\}$
$\quad sourceComponent \in C$
$\quad targetEndpoint \in E$
$\quad targetEndpoint \notin sourceComponent.E$
$RT := (id, name, props, involvedLinks, referencedEndpoint)$
$\quad props_{RT} := \{nodes\}$
$\quad involvedLinks \subseteq STEP$
$\quad STEP := (l_1, ..., l_n) | l_i \in L$
$\quad referencedEndpoint \in EE$
$I := (id, name, props, artifacts, RBD_I, RN_I)$
$\quad props_I := \{kind, environment\_access, maintenance, provisioning,$
$\quad supported\_artifacts, availability\_zone, region,$
$\quad supported\_update\_strategies, deployed\_entities\_scaling,$
$\quad self\_scaling, enforced\_resource\_bounds,$
$\quad identities, address\_resolution\_kind\}$
$\quad artifacts \subseteq A$
$\quad RBD_I \subseteq I \times BD$
$\quad props_{RBD_I} := \{usage\_relation\}$
$\quad RN_I \subseteq I \times N$
$DM := (id, props, deployed, host)$
$\quad props_{DM} := \{deployment, deployment\_unit, replicas, update\_strategy,$
$\quad automated\_restart\_policy, assigned\_account, resource\_requirements\}$
$\quad deployed \in C \cup I$
$\quad host \in I$
$\quad deployed \neq host$

4.4. Product Factors

Product factors can be considered the core of the quality model. They capture specific characteristics of cloud-native application architectures. For each factor it has to be decidable whether or to what extent it is present in an architecture. Because of the focus on the design time of an application, this decision needs to be possible at design time. Product factors which can only be evaluated by analyzing the behavior of a system at runtime or factors considering aspects out of the core scope of an architectural point of view, such as the deployment process, are not included.

In the following, the 73 product factors of the quality model are presented. They are the result of the initial formulation based on literature [169] and the refinement based on a survey [174]. Each factor has a **Name** and a prose *description*. For the description of product factors, the terminology introduced through the entities presented in Section 4.3 is used. Each factor is assigned to a number of Categories to account for the various aspects covered by the topic of cloud-native. The connecting element between factors are Impacts that can be either positive (↗) or negative (↘) and describe how one factor impacts another. Product factors can impact other product factors or quality aspects. It can be differentiated between *intermediate factors* which impact and are impacted by other factors, and *leaf factors* which are not impacted by further factors. Leaf factors, thus, need to be measurable through measures. To avoid ambiguities when applying the model, there is always a single path from one leaf factor to a quality aspect. Under Read more, references are provided that link to the literature based on which the factors were initially formulated and thus where additional information can be found. Finally, under Measures used for evaluation, the measure(s) are listed based on which the presence of the respective product factor in a system can be quantified. If no measures are listed, the factor is evaluated only based on the sub-factors that impact it.

The categories, grouping the product factors by the various aspects of cloud-native, are described in the following. It has to be noted, that factors can also be assigned to more than one category. Figure 4.2 shows the number of product factors assigned per category and as it can be seen, the distribution of factors over the categories is balanced. An exception, however, is the category *Application Administration* as it represents a core focus for developing and operating applications in a cloud-native way.

Application Administration This category covers all aspects related to the configuration and management of cloud applications. This means how cloud resources are used and integrated, how tasks and responsibilities are divided between the cloud provider and the cloud consumer, or which tools and technologies are applied.

Cloud Infrastructure Infrastructure builds the foundation on which applications are deployed and operated. The characteristics of the chosen infrastructure thus also determine application characteristics. Certain cloud services offer more features out-of-the-box than others. Factors inside this category, therefore, are considered with the characteristics and features of the cloud resources used as the infrastructure of an application.

Network Communication Since cloud applications are distributed over networks, the way its different components communicate is important. The amount of network traffic is typically billed by cloud providers. Furthermore, network latency can have a significant impact on application performance. And components have to expect network and communication faults. Thus, factors considering these aspects are assigned to this category.

Data Management How data is stored and processed is another important aspect to consider. Especially, because of the distributed and replicated nature of cloud applications. Thus, the factors within this category consider how data, or more specifically state, is managed to enable fast and consistent access where necessary.

Business Domain Although the quality aspects considered in the quality model exclude functional suitability, certain aspects from the business perspective nevertheless need to and can be considered on an architectural level. Factors assigned to this category consider, for example, how request traces that represent functionalities of an application are designed and distributed over the components of an application. This affects not only communication behavior but also data usage.

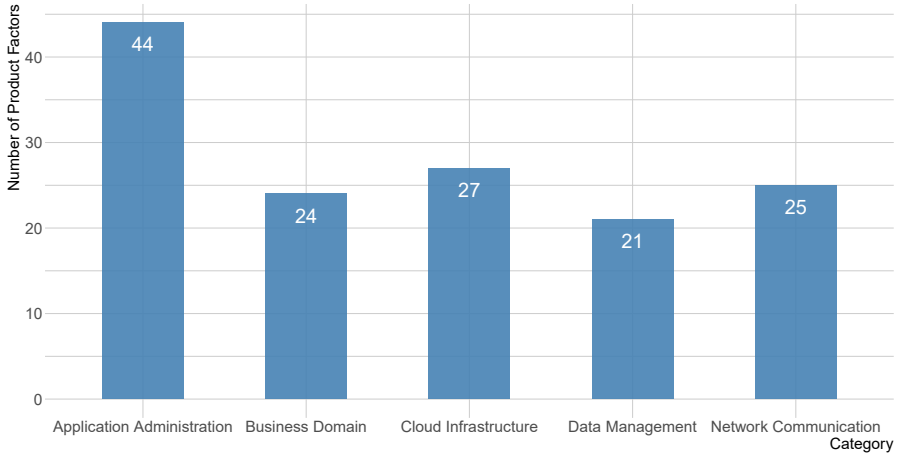


Figure 4.2.: Number of Product Factors per Category

Considering the relevance of product factors for the different quality aspects, Figure 4.3 shows an unbalanced distribution. Many product factors are important in the context of *Maintainability* while other quality aspects considering *Portability*

or *Security* do not have as many product factors relevant for them. Again, it has to be noted, that product factors can be relevant for more than one quality aspect.

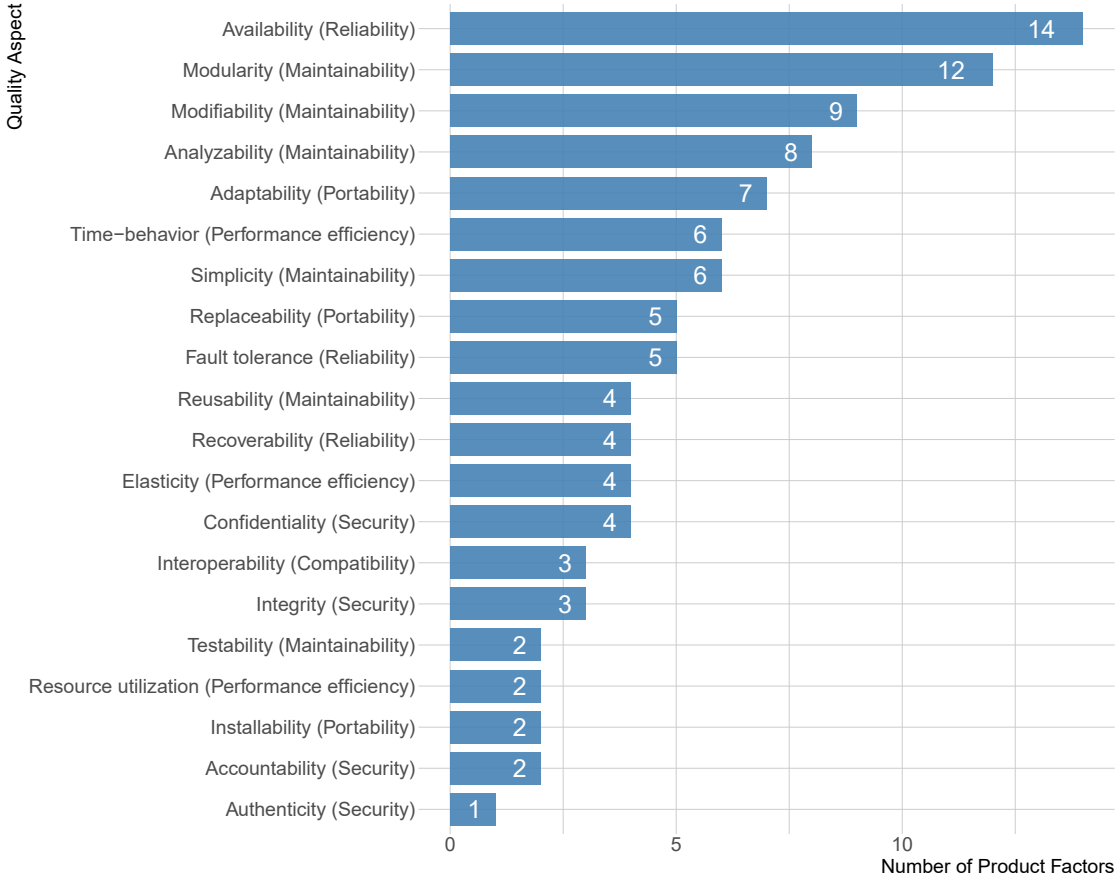


Figure 4.3.: Number of Product Factors per Quality Aspect

A specific ordering of the product factors is therefore difficult. They may be assigned to multiple categories and may impact several quality aspects. To nevertheless present product factors that are more closely related also next to each other in the text, an ordering by their impacted quality aspects is applied. Firstly, product factors impacting quality aspects related to *Security* are presented. They are followed by those impacting quality aspects related to *Maintainability*, then *Performance efficiency*, *Reliability*, *Portability*, and finally *Compatibility*.

4.4.1. Product Factors in the Context of Security

To keep data exchanged within an application confidential, the communication between components of a system can be encrypted [255]. This is covered by the factor **Data encryption in transit**. That way, even if an attacker has access to the raw packages sent through the network, the content is protected. Well-established protocols, such as Transport Layer Security (TLS), can be used for this [120]. For communication in public networks encryption should always be used. For communication via private networks, the necessity can be discussed, because encryption includes a small processing overhead. For increased Confidentiality, however, also communication via internal networks can be encrypted.

Factor **Data encryption in transit**
Data which is sent or received through a link from one component to or from an endpoint of another component is encrypted so that even when an attacker has access to the network layer, the data is protected.
Categories: Network Communication
Impacts: ↗ **Confidentiality**
Read more:
[255] 6 Encrypt Data in Transit
[120] 2 Security (Use TLS for synchronous communications)
Measure(s) used for evaluation:
Ratio of external endpoints that support TLS
Ratio of secured links

Secrets management covers aspects of how confidential data, such as credentials, are used within a system. Secrets is used as a general term for data that should not be accessible to externals. Their exposure would have severe consequences for the security of a system. **Secrets management** acts as an intermediate factor aggregating the factors impacting it.

Factor **Secrets management**
Secrets (e.g. passwords, access tokens, encryption keys) which allow access to other components or data are managed specifically to make sure they stay confidential and only authorized components or persons can access them. Managed in this case refers to where and how secrets are stored and how components which need them can access them.
Categories: Application Administration, Cloud Infrastructure, Data Management
Impacts: ↗ **Confidentiality**
Impacted by: **Secrets stored in specialized services**, **Isolated secrets**

One core aspect to managing secrets properly is to store them separately from the code which uses them. By using such **Isolated secrets**, the risk of accidentally exposing secrets, for example when sharing code or artifacts, is reduced significantly. Instead, secrets should only be stored in the environment where they are actually used [255].

Factor **Isolated secrets**
Secrets (e.g. passwords, access tokens, encryption keys) are not stored in component artifacts (e.g. binaries, images). Instead, secrets are stored for example in the deployment environment and components are given access at runtime only to those secrets which they actually need and only when they need it.
Categories: Application Administration
Impacts: ↗ **Secrets management**
Read more:
[255] 6 Never Store Secrets or Configuration Inside an Image
[3] 14 Don't Check In Secrets
Measure(s) used for evaluation:
Secrets externalization

Furthermore, secrets have to be treated differently than other backing data. For cloud-native applications, specialized backing services are available which ensure that secrets are only stored encrypted [255], never in plain-text. These backing services furthermore enable a revocation or replacement of secrets during the runtime of a system. This can be used to react to security breaches where secrets might have been compromised. As described in **Secrets stored in specialized services**, secrets are then made accessible to components only when they need them and also only to those components which are allowed to access them. Examples for such specialized backing services for secrets are Kubernetes Secrets[46] or AWS Secrets Manager[47]. ¸

Factor **Secrets stored in specialized services**
A dedicated backing service to host secrets (e.g. passwords, access tokens, encryption keys) exists. All secrets required by a system are hosted in this backing service where they can also be managed (for example they can be revoked or replaced with updated secrets). Components fetch secrets from this backing services in a controlled way when they need them.
Categories: Cloud Infrastructure, Data Management
Impacts: ↗ **Secrets management**
Read more:
[255] 6 Securely Store All Secrets
[13] 10 Kubernetes Secrets
Measure(s) used for evaluation:
Secrets stored in vault

To support **Integrity**, a general practice is **Access restriction** which means ensuring that access to data and functionalities is restricted and only granted when necessary. In the quality model this factor is an aggregating factor which combines the factors impacting it.

[46] `https://kubernetes.io/docs/concepts/configuration/secret/`, visited 2025-10-20
[47] `https://docs.aws.amazon.com/secretsmanager/`, visited 2025-10-20

Factor **Access restriction**
Access to components is restricted to those who actually need it. Also, within a system access controls are put in place to have multiple layers of defense. A dedicated component to manage access policies can be used.
Categories: Network Communication, Application Administration
Impacts: ↗ **Integrity**
Impacted by: **Least-privileged access**, **Access control management consistency**

A core factor in this context is **Least-privileged access** which describes a characteristic that actually applies to any system, but is especially relevant for cloud systems where several components may share a cloud infrastructure and are exposed to different networks. Approaches like Role-Based Access Control (RBAC) enable the implementation of this factor and are supported by Cloud providers and platforms such as Kubernetes [13].

Factor **Least-privileged access**
Access to endpoints is given as restrictive as possible so that only components who really need it can access an endpoint.
Categories: Network Communication, Application Administration
Impacts: ↗ **Access restriction**
Read more:
[255] 6 Grant Least-Privileged Access
[13] 11 Access Control and Permissions
Measure(s) used for evaluation:
Access restricted to callers

With an increasing number of components and cloud resources, the management of access control rules for a system becomes complex. As described in **Access control management consistency** using a unified, consistent approach to manage access controls helps to facilitate this task and to avoid accidental misconfiguration, because developers get familiar with this approach. Furthermore, it is possible to build additional tooling that can verify access control rules [13]. By codifying access control rules in a backing service, using for example Open Policy Agent (OPA)[48], checks can be automated and implemented consistently across a system [108].

[48] `https://www.openpolicyagent.org/`, visited 2025-10-20

Factor **Access control management consistency**
Access control for endpoints is managed in a consistent way, that means for example always the same format is used for access control lists or a single account directory in a dedicated backing service exists for all components. Access control configurations can then be made always in the same known style and only in a dedicated place. Based on such a consistent access control configuration, also verifications can be performed to ensure that access restrictions are implemented correctly.
Categories: Application Administration, Network Communication
Impacts: ↗ **Access restriction**
Read more:
[3] 6 Access Control (Access control managed by framework)
[108] 9 Policy as Code (consistently describe your security policies in form of code)
Measure(s) used for evaluation:
Consistency of supported authentication methods of endpoints
Consistency of supported authentication methods of external endpoints

Access control rules can only be evaluated based on the identity of the entity asking for access. Thus, it is necessary to identify entities which is typically done by assigning accounts to entities. Apart from accounts for end users of a system, accounts or identities can also be assigned to the different components of a System to differentiate them. While this could be done using a default account for all components, using **Account separation** leads to better **Accountability**, because components can then be uniquely identified. Restricting access or isolating components can be done more fine-grained, for example in case of an account compromising [3].

Factor **Account separation**
Components are separated by assigning them different accounts. Ideally each component has an individual account. Through this, it is possible to trace which component performed which actions and it is possible to restrict access to other components on a fine-grained level, so that for example in the case of an attack, compromised components can be isolated based on their account.
Categories: Application Administration, Business Domain
Impacts: ↗ **Accountability**
Read more:
[255] 6 Use Separate Accounts/Subscriptions/Tenants"
[3] 8 Role separation"(let different services run with different roles to restrict access)
[3] 8 "Location separation (use different roles for a service in different locations to limit attack impacts)
Measure(s) used for evaluation:
Ratio of unique account usage

To avoid implementing authentication logic several times in different components, potentially in different ways, a federalized identity management can be used [255]. This leads to **Authentication delegation** where authentication is not performed by components themselves, but delegated to a Backing Service which is specifically deployed for that task. **Authenticity** can thus be applied more consistently and is managed in a single place.

Factor **Authentication delegation**
The verification of an entity for authenticity, for example upon a request, is delegated to a dedicated backing service. This concern is therefore removed from individual components so that their focus can remain on business functionalities while for example different authentication options can be managed in one place only.
Categories: Application Administration, Business Domain
Impacts: ↗ **Authenticity**
Read more:
[255] 6 Use Federated Identity Management
[108] 9 Decentralized Identity
Measure(s) used for evaluation:
Ratio of delegated authentication

4.4.2. Product Factors in the Context of Maintainability

Cloud-native applications are often built using a microservices-based approach (see 2.1.2.2). One core concept behind this is covered by the factor **Service-orientation**. Its goal is to achieve better **Modularity** by organizing the functionality of a system in services. To fulfill **Service-orientation** several sub-factors need to be considered.

Factor **Service-orientation**
Cloud-native applications realize modularity by being service-oriented, that means the system is decomposed into services encapsulating specific functionalities and communicating with each other only through specific interfaces. Commonly, a microservices architectural style is used.
Categories: Business Domain, Data Management, Network Communication
Impacts: ↗ **Modularity**
Impacted by: **Limited functional scope**, **Separation by gateways**

The main factor for achieving service-orientation in a microservices-based architecture is **Limited functional scope** [108]. With each service having a specific responsibility, it can be managed by a small focused team that is flexible in its implementation while exposing only a service interface.

Factor **Limited functional scope**
Each service covers only a limited, but cohesive functional scope to keep services manageable.
Categories: Business Domain, Data Management
Impacts: ↗ **Service-orientation**
Impacted by: **Limited data scope**, **Limited endpoint scope**, **Command query responsibility segregation**
Read more:
[241] 9 Microservices Architecture
[3] 7 Use Microservices
[108] 3 Polylithic Architecture Principle (Build separate services for different business functionalities)

One way to limit the functional scope of a service is to limit the scope of data aggregates used by a service. Based on the ideas of DDD, the domain of an application defines how data aggregates are related. With a **Limited data scope**, a service only uses a single data aggregate or only closely related data aggregates, to keep the functionality cohesive.

Factor **Limited data scope**
The number of data aggregates that are processed in a service is limited to those which need to be administrated together, for example to fulfill data consistency requirements. The aim is to keep the functional scope of a service cohesive. Data aggregates for which consistency requirements can be relaxed might be distributed over separate services.
Categories: Business Domain, Data Management
Impacts: ↗ **Limited functional scope**
Measure(s) used for evaluation:
Cohesion between endpoints based on data aggregate usage

Another way to limit the functional scope of a service is through the endpoints offered by a service. The endpoints of a service should be cohesive based on the functionality they provide. Apart from the data aggregates used by endpoints, this **Limited endpoint scope** can also be determined based on the communication behavior within a system. If the endpoints of a service invoked by other components differ significantly, it can be an indicator that a service covers a broader scope than necessary.

Factor **Limited endpoint scope**
To keep the functional scope of services limited, the number of endpoints of a service is limited to a cohesive set of endpoints that provide related operations.
Categories: Business Domain
Impacts: ↗ **Limited functional scope**
Measure(s) used for evaluation:
Service interface usage cohesion

An additional option to further limit the functional scope of a service has become popular specifically in the context of microservices-based applications: **Command query responsibility segregation** This factor is a direct adaptation of the Command and Query Responsibility Segregation (CQRS) pattern [242]. It limits the scope of individual services, because for a data aggregate the functionalities of querying it or modifying it are separated into different services. Each service therefore has a more limited scope and its functionality can be optimized accordingly. For example, the service responsible for performing queries can use libraries and a data model specifically for that while the command service uses its own data model [60] optimized for ensuring business constraints during modifications. For some changes it is easier to modify services using CQRS, because they are concerned with either the query side or the command side, not both. Changes that impact both sides, however, might require more effort, because they need to be made in several places. A negative impact to **Simplicity** is therefore also added to the factor.

Factor **Command query responsibility segregation**
Endpoints for read (query) and write (command) operations on the same data aggregate are separated into different services. Changes to these operations can then be made independently and also different representations for data aggregates can be used. That way operations on data aggregates can be adjusted to differing usage patterns, different format requirements, or if they are changed for different reasons.
Categories: Network Communication, Business Domain
Impacts: ↗ **Limited functional scope** ↘ **Simplicity**
Read more:
[60] 4.4
[242] 7.2 Using the CQRS pattern
[26] 12 CQRS (Command Query Responsibility Segregation)
[120] 4 Command and Query Responsibility Segregation Pattern
[108] 4 Command and Query Responsibility Segregation Pattern
Measure(s) used for evaluation:
Read write separation for data aggregates

A core principle of service-orientation is that functionalities are offered via well-defined interfaces. Apart from the interfaces of individual services, it is possible to further abstract from the actual implementations of functionalities within services by using gateways. Gateways act as a kind of proxy and offer an interface that can either replicate the interfaces of services proxied by it, offering only a reduced set of these endpoints, or by offering additional endpoints building on the endpoints offered by proxied services [120]. Either way, by applying **Separation by gateways**, components of a system or groups of components are abstracted from each other. Furthermore, gateway components can take over tasks typically associated with backing services, such as authorization, load throttling, or logging [60, 120]. Although, this is covered separately by the factor **Guarded ingress**. Finally, gateways improve the possibility to perform **Seamless upgrades**, because

clients only know the gateway component that can seamlessly redirect traffic to an upgraded service once it's ready without the client noticing.

Factor **Separation by gateways**
Individual components or groups of components are separated through gateways. That means communication is proxied and controlled at specific gateway components. It also abstracts one part of a system from another so that it can be reused by different components without needing direct links to components that actually provide the needed functionality. This way, communication can also be redirected when component endpoints change without changing the gateway endpoint. Also, incoming communication from outside of a system can be directed at external endpoints of a central component (the gateway).
Categories: Network Communication, Business Domain
Impacts: ↗ **Service-orientation** ↗ **Seamless upgrades**
Read more:
[60] 10.2
[242] 8.2
[26] 8 Edge Services: Filtering and Proxying with Netflix Zuul
[120] 7 API Gateway Pattern
[120] 7 API Microgateway Pattern (Smaller API microgateways to avoid having a monolithic API gateway)
[108] 4 "Mediator" (Use a mediator pattern between clients and servers)
Measure(s) used for evaluation:
Degree of separation by gateways

Systems require state to provide meaningful functionalities, but dealing with state in cloud-based systems can be challenging. Storing state is even described as "*the hardest aspect of architecting a distributed, cloud-native architecture*" by Goniwada [108]. To keep the complexity of handling state in certain parts of the system architecture only, stateless components should be clearly separated from stateful components as described by the factor **Isolated state**. It acts as an aggregating factor for two sub-factors that cover the two main concerns of isolating state.

Factor **Isolated state**
Services are structured by clearly separating stateless from stateful services. Stateful services should be reduced to a minimum. That way, state is isolated within these specifically stateful services which can be managed accordingly. The majority of stateless services is easier to deploy and modify.
Categories: Data Management, Business Domain
Impacts: ↗ **Modularity** ↗ **Replaceability** ↗ **Elasticity**
Impacted by: **Mostly stateless services**, **Specialized stateful services**
Read more:
[108] 3 Be Smart with State Principle

In addition to separating stateless from stateful components, designing services in a stateless way should be the preferred approach [255]. Therefore, most ser-

vices should be designed stateless while the number of stateful services should be reduced to a minimum (In any meaningful application, however, state cannot be avoided). As described in the factor **Mostly stateless services**, stateless services are a better fit to core aspects of cloud computing [60]. Stateless service instances can be replaced, scaled out, and scaled down at runtime quickly and easily, enabling an exploitation of the on-demand resources characteristic (see Section 2.1.1) . Furthermore, stateless services are easier to test, because the test setup is simplified.

Factor **Mostly stateless services**
Most services in a system are kept stateless, that means not requiring durable disk space on the infrastructure that they run on. Stateless services can be replaced, updated or replicated at any time. Stateful services are reduced to a minimum.
Categories: Data Management, Business Domain
Impacts: ↗ **Isolated state** ↗ **Testability**
Read more:
[60] 5.4
[255] 6 “Design Stateless Services That Scale Out
[108] 3 Be Smart with State Principle, 5 Stateless Services
Measure(s) used for evaluation:
Ratio of stateless components
Degree to which components are linked to stateful components

Those services which do hold state need special attention [60]. All stateful services in a system, should be **Specialized stateful services** in the sense that they provide mechanism for dealing with consistency and replication. Replication is important in cloud environments to be able to react to outages or network partitioning. But replicating state requires choosing and implementing a corresponding replication strategy that ensures consistency or at least eventual consistency. Specific storage backing services, which provide such strategies, should therefore be used for the stateful parts of a system. In Kubernetes, for example, there is a specific resource for that, so-called stateful sets [118] which build a basis for that.

Factor **Specialized stateful services**
For stateful components, that means components that do require durable disk space on the infrastructure that they run on, specialized software or frameworks are used that can handle distributed state by replicating it over several components or component instances while still ensuring consistency requirements for that state.
Categories: Data Management, Business Domain
Impacts: ↗ **Isolated state**
Read more:
[60] 5.4
[118] 11 “Stateful Service”
Measure(s) used for evaluation:
Ratio of specialized stateful services
Ratio of suitably replicated stateful services

As a consequence of the distributed nature of cloud environments, components should be loosely coupled where possible. This leads to greater independence in terms of communication and thus better **Modularity** at design time and tolerance to network communication problems at runtime. Different dimensions of **Loose coupling** are relevant and captured by sub-factors of this factor.

Factor **Loose coupling**
In cloud-native applications communication between components is loosely coupled in time, location, and language to achieve greater independence.
Categories: Business Domain, Network Communication
Impacts: ↗ **Modularity**
Impacted by: **Asynchronous communication**, **Communication partner abstraction**

To decouple components in time, **Asynchronous communication** can be used. That means components are connected through asynchronous links over which messages are sent without awaiting a response [60]. This can happen either directly between components or communication can be mediated via a broker backing service acting as a message bus [255]. One communication partner then sends messages to the broker via an asynchronous links while the other communication partner receives messages asynchronously by being subscribed to the broker. The communicating components are less coupled then, because it's not necessary that both components are ready at the same time for a successful communication. The message broker nevertheless needs to be available as well.

Factor **Asynchronous communication**
Asynchronous links (e.g. based on messaging backing services) are preferred for the communication between components. That way, components are decoupled in time meaning that not all linked components need to be available at the same time for a successful communication. Additionally, callers do not await a response.
Categories: Network Communication, Business Domain
Impacts: ↗ **Loose coupling**
Read more:
[60] 4.2
[255] 6 Prefer Asynchronous Communication
[242] 3.3.2, 3.4 Using asynchronous messaging to improve availability
[120] 3 Service Choreography Pattern
[249] 9 Asynchronous Request/Response (Use asynchronous communication to make services more robust)
[108] 4 Asynchronous Nonblocking I/O
Measure(s) used for evaluation:
Degree of asynchronous communication
Asynchronous communication utilization

In addition to asynchronous messaging as such, using broker backing services also enables **Communication partner abstraction**. This means components are not aware of the identity of their actual communication partners, because communication is based on events rather than directed messages. A component that is interested in a specific type of event can subscribe to it and receive events without the event producer knowing the subscriber [242]. This also makes components less coupled in their life-cycles, because a subscribing component may be created later than a producing component. A potential drawback, however, is that communication is harder to trace [249], thus **Communication partner abstraction** can have a negative impact on **Analyzability**.

Factor **Communication partner abstraction**
Communication via links is not based on specific communication partners (specific components) but abstracted based on the content of communication. An example is event-driven communication where events are published to channels without the publisher knowing which components receive events and events can therefore also be received by components which are created later in time.
Categories: Network Communication
Impacts: ↗ **Loose coupling** ↘ **Analyzability**
Read more:
[242] 6 Event-driven communication
[249] 8: Event-driven systems "event chains emerge over time and therefore lack visibility.
Measure(s) used for evaluation:
Event sourcing utilization metric

Another, more fundamental, concept of abstraction in terms of communication is **Addressing abstraction**. It applies to both synchronous and asynchronous communication and concerns the addresses used by components to contact other components via links. Instead of using Internet Protocol (IP) addresses directly, abstract names are used which are then resolved to the specific addresses by suitable mechanisms. While Domain Name Service (DNS) exists as a mature technology for the global internet, systems in private networks historically still often implement communication via direct addressing. With the move to the cloud where infrastructure is created and destroyed on-demand, **Addressing abstraction** is necessary and implemented for example via service discovery [101] or also DNS in the case of Kubernetes. **Addressing abstraction** has a positive impact on **Modifiability** because components can be upgraded without causing problems, even if the address of the upgraded component changes. And it also impacts **Replaceability** positively because the component behind a name can be replaced more easily without impacting clients. Nevertheless, additional backing services or specific infrastructure are needed to realize **Addressing abstraction**.

Factor **Addressing abstraction**
In a link from one component to another the specific addresses for reaching the other component is not used, but instead an abstract address is used. That way, the specific addresses of components can be changed without impacting the link between components. This can be achieved for example through service discovery where components are addressed through abstract service names and specific addresses are resolved through service discovery which can be implemented in the infrastructure or a backing service.
Categories: Network Communication
Impacts: ↗ **Modifiability** ↗ **Replaceability**
Read more:
[60] 8.3
[118] 12 Service Discovery
[242] Using service discovery
[101] 7 Service Discovery
[120] 3 Service Registry and Discovery Pattern
[26] 7 Routing (Use service discovery with support for health checks and respect varying workloads)
[120] 3 Service Abstraction Pattern (Use an abstraction layer in front of services (for example Kubernetes Service))
[108] 4 Service Discovery
Measure(s) used for evaluation:
Service discovery usage

Factor **Usage of existing solutions for non-core capabilities**
For non-core capabilities readily available solutions are used. This means solutions which are based on a standard or a specification, are widely adopted and ideally open source so that their well-functioning is ensured by a broader community. Non-core capabilities include interface technologies or protocols for endpoints, infrastructure technologies (for example container orchestration engines), and software for backing services. That way capabilities don't need to self-implemented and existing integration options can be used.
Categories: Cloud Infrastructure, Application Administration
Impacts: ↗ **Simplicity**
Read more:
[241] 9 Avoid Reinventing the Wheel
[3] 12 Frameworks to Enforce Security and Reliability
Measure(s) used for evaluation:
Ratio of non-custom backing services

While the core of a system implements domain-specific functionalities, many functionalities of cloud-native applications, for example message brokering, are not specific to a certain domain and are typically provided by backing services. Because such non-core capabilities are similar for applications across domains, they

have typically been implemented elsewhere already and thus existing solutions can be used [241]. This **Usage of existing solutions for non-core capabilities** simplifies application development, because the complexity of implementing such non-core capabilities is avoided and furthermore supported by a broader community. Also, new features and bug fixes are potentially coming from a community, reducing maintenance effort down to performing library updates.

While relying on existing external solutions avoids the effort of implementing functionalities, there is a risk that maintenance of such solutions may cease and a replacement is needed. In such cases, using components that adhere to standards and commonly accepted specifications, can mitigate such risks. The reason is that software developed to use components which adhere to a standard or a specification is reusable also for other components that adhere to the same standard or specification. Furthermore, by relying on **Standardization**, implementation and maintenance effort can be reduced if a standard is commonly known and understood by developers. Developing additional functionalities based on components that support that standard is then faster and less error-prone. Various standards in cloud computing exist [261], although cloud providers may be reluctant to support open standards for their infrastructure or services to bind consumers to their environments.

Factor **Standardization**
By using standardized technologies within components, for interfaces, and especially for the infrastructure, backing services and other non-business concerns, reusability can be increased and the effort to develop additional functionality which integrates with existing components can be reduced.
Categories: Application Administration
Impacts: ↗ **Reusability**
Measure(s) used for evaluation:
Ratio of standardized artifacts
Ratio of entities providing standardized artifacts

Factor **Component similarity**
The more similar components are, the easier it is for developers to work on an unfamiliar component. Furthermore, similar components can be more easily integrated and maintained in the same way. Similarity considers mainly the libraries and technologies used for implementing service logic and service endpoints, as well as their deployment.
Categories: Application Administration
Impacts: ↗ **Reusability**
Read more:
[241] 9 Reference Architecture
Measure(s) used for evaluation:
Component artifacts similarity
Infrastructure artifacts similarity

Apart from the usage of standards and specifications, **Reusability** can already be supported by implementing components within a system in a similar way. Through **Component similarity**, that means using the same or similar programming languages and libraries, the effort for understanding and making changes to components can be reduced. For example, a new functionality that needs to be added to several components can be implemented once and then reused for other components if they are similar enough. **Component similarity** can also be enforced for an application by providing a reference architecture [241].

To operate and maintain a system, **Analyzability** is important. If a system is analyzable, problems can be detected, and it is possible to decide where and how to introduce changes to either avoid introducing problems or improving a system regarding a certain aspect. The characteristic of being able to analyze and understand a system based on the outputs it provides is also called observability [108]. A core aspect in this context is **Automated monitoring** that needs to be planned and implemented for a system from the start. The factor **Automated monitoring**, however, is an aggregating factor combining a range of aspects that are covered more in detail by several sub-factors.

Factor **Automated monitoring**
Cloud-native applications enable monitoring at various levels (business functionalities in services, backing-service functionalities, infrastructure) in an automated fashion to enable observable and autonomous reactions to changing system conditions.
Categories: Application Administration, Business Domain, Network Communication, Data Management
Impacts: ↗ **Analyzability**
Impacted by: **Consistent centralized logging**, **Consistent centralized metrics**, **Distributed tracing of invocations**, **Health and readiness checks**
Read more:
[108] 3 High Observability Principle

Logging is a core practice in software development, relevant for all types of software. In cloud environments, however, it has a special significance, because components are executed on remote infrastructure that can be created and destroyed on-demand. By sending logs to an explicit logging backing service, logs can be preserved even when the component instance that created them does not exist anymore. Furthermore, it is possible to analyze logs across different components and their corresponding infrastructure. This is captured by the factor **Consistent centralized logging** which proposes using a central logging backing service for a system to store logs [255], collecting logs in a consistent format [255] and that provides functionalities for log aggregation and analysis [60, 242]. Having a centralized logging approach in place for the components of a system, also impacts **Accountability** in a positive way, because actions that happened in a system can be traced back to component instances or users. Logs can be either pushed by components to the logging backing service or pulled by the logging backing service from

the components. Whether to use a push or a pull based approach needs to be assessed by application developers. A push-based approach places a higher burden on components while a pull-based approach adds complexity to the logging backing service and it needs to be able to discover newly added components. Especially for short-lived components, a push-based approach can be more favorable.

Factor **Consistent centralized logging**
Logging functionality, specifically the automated collection of logs, is concentrated in a centralized backing service which combines and stores logs from the components of a system. The logs are kept consistent regarding their format and level of granularity. In the backing service also log analysis functionalities are provided, for example by also enabling a correlation of logs from different components.
Categories: Application Administration, Data Management
Impacts: ↗ **Automated monitoring** ↗ **Accountability**
Read more:
[60] 11.1
[255] 6 Use a Unified Logging System
[255] 6 Common and Structured Logging Format
[242] 11.3.2 Applying the Log aggregation pattern
[241] 10 Observability
[101] 7 Monitoring and Logging
[3] 15 Design your logging to be immutable
[13] 15 Logging
[26] 13 Application Logging
[26] 13 Audit Events (capture events for audits, like failed logins etc)
[249] 11 Custom Centralized Monitoring
[108] 19 One Source of Truth
Measure(s) used for evaluation:
Ratio of components or infrastructure nodes that export logs to a central service

In a similar way to logging, also the gathering and calculation of metrics should be supported by a centralized backing service [101]. Deciding which metrics are relevant for a specific system can be challenging, but suggestions on commonly helpful metrics exist [13]. Gathering **Consistent centralized metrics** also enables metric aggregation over different entities of a system. With continuously gathered metrics, it is possible to quickly identify and resolve issues, for example by including an alerting system that is triggered when predefined thresholds for metrics are exceeded or undercut. Finally, additional backing services can build on a metrics backing service to implement automated behavior in a system, such as autoscaling.

Factor **Consistent centralized metrics**
Metrics gathering and calculation functionality for monitoring purposes is concentrated in a centralized component which combines, aggregates and stores metrics from the components of a system. The metrics are kept consistent regarding their format and support multiple levels of granularity. In the backing service also metric analysis functionalities are provided, for example by also enabling correlations of metrics.
Categories: Application Administration, Business Domain
Impacts: ↗ **Automated monitoring**
Read more:
[60] 11.2
[255] 6 Tag Your Metrics Appropriately
[242] 11.3.4 Applying the Applications metrics pattern
[101] 7 Monitoring and Logging, Metrics Aggregation
[241] 10 Observability
[13] 15 Metrics help predict problems
[26] 13 Metrics
[13] 16 The RED Pattern (common metrics you should have for services
[13] 16 The USE Pattern (common metrics for resources
[108] 19 One Source of Truth
Measure(s) used for evaluation:
Ratio of components or infrastructure nodes that export metrics

Factor **Distributed tracing of invocations**
For request traces that span multiple components in a system, distributed tracing is enabled so that traces based on correlation IDs are captured automatically and stored in a backing service where they can be analyzed and problems within request traces can be clearly attributed to single components.
Categories: Application Administration, Network Communication
Impacts: ↗ **Automated monitoring**
Read more:
[60] 11.3
[255] 6 Use Correlation IDs
[242] 11.3.3 Using the Distributed tracing pattern
[101] 7 Debugging and Tracing
[241] 10 Observability
[13] 15 Tracing
[26] 13 Distributed Tracing
[249] 11 Observability and Distributed Tracing Tools (Use Distributed Tracing)
[108] 19 One Source of Truth
Measure(s) used for evaluation:
Distributed tracing support

In addition to logging and metrics, the third main pillar of observability in cloud-native systems is distributed tracing [149]. This is covered by the factor **Distributed tracing of invocations**. If a system supports distributed tracing, the individual services are instrumented so that they track incoming traffic via their endpoints and outgoing traffic via invoked links. In addition, correlation ids are added to communication [26]. When all tracked communication is reported to a backing service for distributed tracing, the resulting traces can be analyzed regarding their complexity and performance. While the request trace entity is closely related to this concept, it allows for a more general modeling of how interaction is happening in a system. The factor **Distributed tracing of invocations** instead covers the technical aspect of whether distributed tracing is implemented within a system.

Factor **Health and readiness checks**
All components in a system offer health and readiness checks so that unhealthy components can be identified and communication can be restricted to happen only between healthy and ready components. Health and readiness checks can for example be dedicated endpoints of components which can be called regularly to check a component. That way, also an up-to-date holistic overview of the health of a system is enabled.
Categories: Application Administration
Impacts: ↗ **Automated monitoring** ↗ **Automated restarts** ↗ **Availability**
Read more:
[255] 6 Implement Health Checks and Readiness Checks
[118] 4 Health Probe
[242] 11.3.1 Using the Health check API pattern
[101] 7 State Management
[13] 5 Liveness Probes
[13] 5 Readiness Probes
[26] 13 Health Checks
[120] 1 Why container orchestration?, Health monitoring
[108] 4 Fail Fast, 16 Health Probe
Measure(s) used for evaluation:
Ratio of services that provide health endpoints
Ratio of services that provide readiness endpoints

An additional product factor that supports **Automated monitoring**, but is also important for other quality aspects, is **Health and readiness checks**. To fulfill this factor, components include special endpoints that allow checking their status [242]. A differentiation is made between health and readiness. Health refers to whether a component is operating in a normal, expected way or whether there is erroneous, unexpected behavior. Readiness refers to whether a component is ready to process requests. A component can be healthy but not ready, for example while it is starting up and initializing itself [13]. If a component is not healthy, it is also not ready. The existence of **Health and readiness checks** , apart from enabling the monitoring of a system as such, also impacts the possibility of **Automated**

restarts, because unhealthy components requiring a restart can be detected that way. Furthermore, by directing traffic only to ready components, **Availability** can be impacted positively.

The way how infrastructure, required to run a system, is managed has consequences for the maintainability of a system. In specific, Reznik et al. [241] write that "*Any manual work that is required in between the changes committed by the developer and the delivery to production will significantly reduce the speed of delivery*" [241, Pattern: Automated Infrastructure (Chapter 10)]. Thus, automation should be preferred over manual work and through **Automated infrastructure provisioning** of required infrastructure, modifications are simplified and can be done more quickly. The factor, therefore, has a positive impact on **Modifiability** if it is present in a System, especially its Infrastructure. The core point is whether Infrastructure entities can be provisioned automatically, that means with little or no manual actions. This could, for example, be the case when the infrastructure is managed by a cloud provider and can be provisioned once needed by the deployment of a Component [241]. Another case would be the usage of IaC approaches for which an additional product factor **Use infrastructure as code** exists. Because also the initial deployment of a system is simplified by automated provisioning, another positively impacted quality aspect is **Installability**.

Factor **Automated infrastructure provisioning**
Infrastructure provisioning should be automated based on component requirements which are either stated explicitly or inferred from the component which should be deployed. The infrastructure and tools used should require only minimal manual effort. Ideally it should be combined with continuous delivery processes so that no further interaction is needed for a component deployment.
Categories: Cloud Infrastructure, Application Administration
Impacts: ↗ **Modifiability** ↗ **Installability**
Read more:
[241] 10 Automated Infrastructure
[108] 5 Automation
Measure(s) used for evaluation:
Ratio of automatically provisioned infrastructure

The usage of IaC provides not only automation but also the description of infrastructure in a storable format. As described for the factor **Use infrastructure as code** with a stored infrastructure description it is possible to provision infrastructure repeatedly [108]. This positively impacts **Reusability** and **Recoverability**, because infrastructure can be brought up again quickly and consistently if needed to scale out or to recover from a failure. Furthermore, when infrastructure is implemented in a declarative way [108], changes can be made to the declaration and the corresponding IaC tool manages the required changes, improving **Modifiability**. Through a modularization and parameterization of IaC artifacts, it is possible to adapt a required infrastructure quickly to a different environment, which improves **Adaptability**.

Factor **Use infrastructure as code**
The infrastructure requirements and constraints of a system are defined (coded) independently of the actual runtime in a storable format. That way a defined infrastructure can be automatically provisioned repeatedly and ideally also on different underlying infrastructures (cloud providers) based on the stored infrastructure definition. Infrastructure provisioning and configuration operations are not performed manually via an interface of a cloud provider.
Categories: Cloud Infrastructure, Application Administration
Impacts: ↗ **Modifiability** ↗ **Adaptability** ↗ **Reusability** ↗ **Recoverability**
Read more:
[255] 6 Describe Infrastructure Using Code
[108] 16 Declarative Deployment, 17 What Is Infrastructure as Code?
Measure(s) used for evaluation:
Ratio of infrastructure with IaC artifact

Another factor, important in several dimensions and impacted by several subfactors, is **Service independence**. Independence of services is a core principle in microservices-based architectures that enables autonomy for services during their lifecycle. Independence can be ensured on several layers and is often accompanied by decentralization [108]. The quality aspect that is impacted positively if services are independent is **Modifiability**.

Factor **Service independence**
Services are as independent as possible throughout their lifecycle, that means development, operation, and evolution. Changes to one service have a minimum impact on other services.
Categories: Business Domain, Network Communication, Cloud Infrastructure, Application Administration, Data Management
Impacts: ↗ **Modifiability**
Impacted by: **Low coupling**, **Functional decentralization**, **Limited request trace scope**, **Logical grouping**, **Backing service decentralization**
Read more:
[108] 3 Decentralize Everything Principle (Decentralize deployment, governance)

Each link from one component to the endpoint of another component represents a dependency. The component which provides the endpoint has to respect this dependency to keep the other component functional. This is called coupling. By designing a system in a way that **Low coupling** is achieved, the **Service independence** can be increased. Coupling should be low, but can obviously not be avoided completely. Therefore, it is also important to design endpoints in a way that the internal implementation can be evolved without impacting the endpoints of a component too much. The factor **Loose coupling** is related to this factor, but considers how endpoints and links can be designed, while Low coupling considers more the number and density of links.

Factor **Low coupling**
The coupling in a system is low in terms of links between components. Each link represents a dependency and therefore decreases service independence.
Categories: Business Domain
Impacts: ↗ **Service independence**
Measure(s) used for evaluation:
Number of components a component is linked to relative to the total amount of components
Degree of coupling in a system

From a domain perspective, independence between services can be reached by decentralizing functionalities provided by a system. Ideally, unrelated functionalities are separated from each other by providing them by different services. However, services within a single system are typically related to each other, otherwise they would be separated in different systems. Thus, there is a trade-off between separating services from each other based on their functionalities and being able to provide functionalities for which multiple services need to collaborate. A balance must be found, but the general notion that **Functional decentralization** leads to more independent services still applies.

Factor **Functional decentralization**
Business functionality is decentralized over the system as a whole to separate unrelated functionalities from each other and make components more independent.
Categories: Business Domain
Impacts: ↗ **Service independence**
Measure(s) used for evaluation:
Data aggregate spread
Request trace similarity based on included components

Functionalities that are based on the collaboration of multiple components are represented as request traces in the context of the quality model. Another aspect of service independence therefore is how long and complex request traces are within a system. As described by the factor **Limited request trace scope**, their length and complexity should be limited, otherwise services are less independent. The factor is related to **Low coupling**, because request traces are based on the links within a system, but it adds another perspective. Only by analyzing request traces it can be determined whether only a few or many services need to collaborate to process requests.

Factor **Limited request trace scope**
A request that requires the collaboration of several services is still limited to as few services as possible. Otherwise, the more services are part of a request trace the more dependent they are on each other.
Categories: Business Domain, Network Communication
Impacts: ↗ **Service independence**
Measure(s) used for evaluation:
Average complexity of request traces
Maximum number of services within a request trace
Request trace complexity

Another possibility for increasing independence of components in cloud-native systems is **Logical grouping**. It can be implemented in different ways, therefore the factor is defined rather abstract. The core idea, however, is always to introduce groups to which components are assigned. Components which are assigned to the same group share certain resources, like hardware, networking, or backing services, while components that are not are more independent. A suitable example are Kubernetes namespaces [13]. Different namespaces can exist within a cluster and only those components running in the same namespace share resources while they are isolated from components in other namespaces.

Factor **Logical grouping**
Services are logically grouped so that services which are related (for example by having many links or processing the same data aggregates) are in the same group, but services which are more independent are separated in different groups. That way a separation can also be achieved on the network and infrastructure level by separating service groups more strictly, such as having different subnets for such logical groups or not letting different groups run on the same infrastructure. Potential impacts of a compromised or misbehaving service can therefore be reduced to the group to which it belongs but other groups are ideally unaffected.
Categories: Cloud Infrastructure, Application Administration, Business Domain
Impacts: ↗ **Service independence**
Read more:
[255] 6 Use Namespaces to Organize Services in Kubernetes
[13] 5 Using Namespaces
[120] 1 Why container orchestration?; Componentization and isolation
Measure(s) used for evaluation:
Namespace separation

This notion of whether resources or components are shared by a group of components , is also important for the factor **Backing service decentralization**. This factor is focused on backing services in specific, which also includes storage backing services [120] and broker backing services. Decentralizing their usage by the services of a system, means that for example not a single backing service provides

a functionality for the whole system, but several backing services exist that are assigned to different parts or groups of a system [249]. This approach also positively impacts **Service independence**, because changes applied to one backing service or problems that occur in one backing service impact only a part of the system and not the whole system.

Factor **Backing service decentralization**
Different backing services are assigned to different components. That way, a decentralization is achieved. For example, instead of one message broker for a whole system, several message brokers can be used, each for a group of components that are interrelated. A problem in one messaging broker has an impact on only those components using it, but not on components having separate message brokers.
Categories: Application Administration, Cloud Infrastructure, Data Management
Impacts: ↗ **Service independence**
Read more:
[120] 4 Decentralized Data Management (decentralized data leads to higher service independence while centralized data leads to higher consistency.)
[120] 4 Data Service Pattern (As having a negative impact because multiple services should not access the same data);
[249] 2 Different Workflow Engines for different services
[108] 5 Distributed State, Decentralized Data
Measure(s) used for evaluation:
Average weighted storage backend sharing
Average weighted broker backend sharing
Ratio of storage backend sharing
Ratio of broker backend sharing

Factor **Operation outsourcing**
By outsourcing the operation of infrastructure and components to a cloud provider or other vendor, the operation is simplified because responsibility is transferred. Furthermore, costs can be made more flexible because providers and vendors can provide a usage-based pricing.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Simplicity**
Impacted by: **Managed infrastructure**, **Managed backing services**
Measure(s) used for evaluation:
Ratio of provider-managed components and infrastructure

In a general way, complex systems are harder to maintain. The opposite to complexity is **Simplicity**. A core promise of cloud computing is that system development and deployment is simplified by the cloud provider taking over operational tasks from the cloud consumer. Therefore, the factor **Operation outsourcing** captures that and has a positive impact on **Simplicity**. It is an intermediate factor

and aggregates two sub-factors that differentiate between different parts of a system which can be managed by a cloud provider.

The most important type of entity which can be managed by a cloud provider instead of the developers of a system themselves is infrastructure. The factor **Managed infrastructure**, therefore, captures this fact, and it is the more fulfilled, the more infrastructure is managed by the provider, with the developers having only limited or no control at all. The differences in cloud service models, such as IaaS, PaaS, or CaaS, in which cloud providers take over different levels of responsibility, are represented by the different types of infrastructure entities modeled for a system. An explicit differentiation of these service models is thus not part of the factor itself but made through the modeling of the system.

Factor **Managed infrastructure**
Infrastructure such as basic computing, storage or network resources, but potentially also software infrastructure (for example a container orchestration engine) is managed by a cloud provider who is responsible for a stable functioning and up-to-date functionalities. The more infrastructure is managed, the more operational responsibility is transferred. This will also be reflected in the costs which are then calculated more on usage-based pricing schemes.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Operation outsourcing**
Measure(s) used for evaluation:
Ratio of fully managed infrastructure

Factor **Managed backing services**
Backing services that provide non-business functionality are operated and managed by vendors who are responsible for a stable functioning and up-to-date functionalities. Operational responsibility is transferred which is also reflected in the costs which are then calculated more on usage-based pricing schemes.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Operation outsourcing**
Read more:
[255] 6 Use Managed Databases and Analytics Services
[13] 15 Don't build your own monitoring infrastructure (Use an external monitoring service)
[26] 10 managed and automated messaging system (operating your own messaging system increases operational overhead, better use a system managed by a platform)
Measure(s) used for evaluation:
Ratio of managed backing services

In addition to relying on managed infrastructure, a common approach in cloud-native systems is to also use **Managed backing services**. Because backing services provide functionalities that are not domain-specific and thus not specific to a certain system, there may be no advantage in building and operating them your-

self. In contrast, a cloud provider has experience and may operate such backing services more efficiently, because several consumers use them. To simplify operation, for databases, messaging services or analytics services, therefore, managed solutions can be used [255].

Finally, to conclude the factors in the context of maintainability, **Sparsity** is considered. It covers the notion that by relying on a cloud provider, not only the operational effort can be reduced by letting a provider manage parts of a system, but the total number of resources that need to be considered can be reduced. The fewer resources developers need to be concerned with, the simpler is also the maintenance of a system.

Factor **Sparsity**
The more sparse a system is, the fewer components there are which need to be operated and maintained by the developers of a system. This covers all types of components, such as services, backing services, storage backing services, and also the infrastructure.
Categories: Application Administration, Business Domain
Impacts: ↗ **Simplicity**
Measure(s) used for evaluation:
Number of components

4.4.3. Product Factors in the Context of Performance Efficiency

A core characteristic of cloud computing is on-demand access to a virtually unbounded amount of resources (see Section 2.1.1). This availability of resources enables **Replication** in different dimensions with the goal of improving performance efficiency, typically **Time-behavior**. The factor **Replication** aggregates the different types of replication which are represented through individual sub-factors.

Factor **Replication**
Business logic and needed data is replicated at various points in a system so that latencies can be minimized and requests can be distributed for fast request handling.
Categories: Application Administration, Data Management, Cloud Infrastructure
Impacts: ↗ **Time-behaviour**
Impacted by: **Service replication**, **Horizontal data replication**, **Vertical data replication**, **Sharded data store replication**

The most common type of replication is **Service replication**, meaning that multiple replicas of services are deployed in parallel. This is also called horizontal scaling or scaling out [255]. While this factor is related to **Built-in autoscaling**, it does not cover the aspect of how scaling is done, but simply the fact whether multiple replicated instances of services are deployed. The reason for this type of replication improving **Time-behavior** is twofold: Firstly, by having multiple rep-

licas, load can be distributed among these to achieve consistent response times for requests. Secondly, by having replicas available in different locations, traffic can be directed to replicas that are closer to the client, therefore, reducing latency. In addition, **Service replication** also impacts **Availability** positively, because even if a single instance of a service becomes unavailable for some reason, other instances may still be operational to serve requests [60].

Factor **Service replication**
Services and therefore their provided functionalities are replicated across different locations so that the latency for accesses from different locations is minimized and the incoming load can be distributed among replicas.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Replication** ↗ **Availability**
Read more:
[60] 5.1. Cloud-native apps have many instances deployed
[255] 6 Design Stateless Services That Scale Out
Measure(s) used for evaluation:
Service replication level
Amount of redundancy

Factor **Horizontal data replication**
Data is replicated horizontally, that means duplicated across several instances of a storage backing service so that a higher load can be handled and replicas closer to the service where data is needed can be used to reduce latency.
Categories: Application Administration, Data Management
Impacts: ↗ **Replication** ↗ **Availability**
Read more:
[255] 6 Use Data Partitioning and Replication for Scale
[108] 4 Data Replication
Measure(s) used for evaluation:
Storage replication level

Another type of replication that has to be considered separately is **Horizontal data replication**. Similar to **Service replication**, it considers whether multiple replicas of a component are deployed, but instead of typically stateless services, the focus is on stateful storage backing services. The goal is to replicate the data required by a system and thus, those components which store the data need to be replicated. However, replicating a stateful service requires more effort because consistency requirements need to be handled. Different strategies for data replication exist [108], for example read-only replication where consistency is ensured by allowing writes only to a single storage backing service instance and other instances are read-only. Setups in which writes should be allowed to several instances require algorithms for distributed coordination to ensure data consistency among instances. Therefore, the storage backing services that are used need to support and implement such replication strategies which most modern database systems do, but some-

times with limitations. The overall goals of data replication are similar to that of **Service replication**: Improve **Time-behavior** by distributing the requests for data and having data available closer to those clients or components which need it. And improve **Availability** by being able to serve data from another instance, if one instance is unavailable.

Also considering how data is used, the factor **Vertical data replication** describes not how data is replicated within replicas of the same component, but across different components. The idea is to replicate data aggregates along a request trace, so that when components interact with each other to fulfill a request, links through which data aggregates are queried from another component are not invoked every time. When network communication via links can be avoided, the overall latency decreases, having a positive impact on **Time-behavior**. The core technique to implement that is caching [26, 255]. Caching cannot always be used, depending on how frequently data changes and how recent the status of data in a component has to be. But for data aggregates that change rarely, caching can have a significant impact on **Time-behavior**. Also, **Availability** can be impacted positively, if a data aggregate from a cache can be used while the actual component storing a data aggregate is unavailable. However, **Analyzability** can be impacted negatively, if problems in a request trace need to be analyzed or reproduced, that may or may not occur depending on whether cached data is used.

Factor **Vertical data replication**
Data is replicated vertically, that means across a request trace so that it is available closer to where a request initially comes in. Typically, caching is used for vertical data replication.
Categories: Application Administration, Data Management
Impacts: ↗ **Replication** ↘ **Analyzability** ↗ **Availability**
Read more:
[255] 6 Use Caching
[26] 9 Caching (Use an In-Memory cache for queries to relieve datastore from traffic; replication into faster data storage)
[120] 4 Caching Pattern
Measure(s) used for evaluation:
Ratio of cached data aggregates
Data replication along request trace

The last option for replication considered within the quality model is **Sharded data store replication**. It can be applied within a storage backing service and for one or several data aggregates. The idea behind sharding is to split large collections of data into a number of shards based on a certain attribute within the data that makes sense from a domain perspective, such as location [120]. Thus, there are multiple parts, in this case called shards, for a single data aggregate, but the specific values within the shards do not overlap. When data is queried, ideally only a single shard needs to be used, reducing the amount of data that needs to be processed and thus

improving Time-behavior. Furthermore, also the overall load can be distributed among the different shards.

Factor **Sharded data store replication**
Data storage is sharded, that means data is split into several storage backing service instances by a certain strategy so that requests can be distributed across shards to increase performance. One example strategy could be to shard data geographically, that means user data from one location is stored in one shard while user data from another location is stored in a different shard. One storage backing service instance is then less likely to be overloaded with requests, because the number of potential requests is limited by the amount of data in that instance.
Categories: Application Administration, Data Management
Impacts: ↗ **Replication**
Read more:
[120] 4 Data Sharding Pattern
[108] 4 Data Partitioning Pattern
Measure(s) used for evaluation:
Level of sharding across storage backing services

Factor **Enforcement of appropriate resource boundaries**
The resources required by a component are predictable as precisely as possible and specified accordingly for each component in terms of lower and upper boundaries. Resources include CPU, memory, GPU, or Network requirements. This information is used by the infrastructure to enforce these resource boundaries. Thereby it is ensured that a component has the resources available that it needs to function properly, that the infrastructure can optimize the amount of allocated resource, and that components are not negatively impacted by defective components which excessively consume resources.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Resource utilization** ↗ **Availability**
Read more:
[255] 6 Define CPU and Memory Limits for Your Containers
[13] 5 Resource Limits
[118] 2 Defined Resource requirements
[13] 5 Resource Quotas (limit maximum resources for a namespace)
[108] 3 Runtime Confinement Principle, 6 Predictable Demands
Measure(s) used for evaluation:
Ratio of infrastructure entities enforcing resource boundaries
Ratio of deployment mappings with stated resource requirements

Resources, such as CPU, memory, or network bandwidth, can be acquired in cloud environments flexibly and on demand. Their usage is metered and billed in a granular way. For an optimal **Resource utilization**, all components should therefore have enough resources available to function properly, but resources should also not be over-provisioned to avoid excessive cost. There are two key things to

consider in this regard as described by the factor **Enforcement of appropriate resource boundaries**: Firstly, the resource requirements of components should be known as precisely as possible and they should be made explicit [13, 108]. These requirements are typically specified as a request and a limit value as in the case of Kubernetes [118], effectively describing a boundary within which the provided resources should be. Appropriate resources can then be made available to components . Secondly, the specified resource boundaries should be enforced by the infrastructure on which components are deployed [13, 255]. This means, the requested resources should be guaranteed to components and components trying to use more than their maximum resources should be constrained to avoid negative impacts on other components. This also improves **Availability**, because a component excessively consuming resources cannot take away resources from other components.

Factor **Dynamic scheduling**
Resource provisioning to deployed components is dynamic and automated so that every component is ensured to have the resources it needs and only that many resources are provisioned which are really needed at the same time. This requires dynamic adjustments to resources to adapt to changing environments. This capability should be part of the used infrastructure.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Resource utilization**
Read more:
[241] 10 Dynamic Scheduling
[101] 7 Resource Allocation and Scheduling
[118] 6 Automated Placement
[120] 1 Why container orchestration?; Resource Management
[120] 1 Why container orchestration?; Automatic provisioning
[108] 16 Automated Placement
Measure(s) used for evaluation:
Ratio of components deployed dynamically

Because infrastructure of cloud-native systems is created and destroyed on-demand over time, there should be no fixed scheduling of components on specific hardware. Instead, **Dynamic scheduling** should be used which means that a component is scheduled to be run, but the specific hardware on which it is run is selected dynamically at runtime by an orchestration platform [241]. The selection of suitable infrastructure to run components can include several factors. Most importantly, the resource requirements of a component should be met, but also factors like current utilization of infrastructure or current cost can be incorporated. A dynamic infrastructure enables an efficient usage of resource and thus improves **Resource utilization**. While current orchestration systems, such as Kubernetes, have automated the placement of component instance to optimize **Resource utilization** [118], an instance is typically not rescheduled, after it has been started. Further optimization would thus be possible by also rescheduling component in-

stances while they are running. In any case, scheduling decisions should be made automatically by the orchestration platform.

As the last factor in the context of performance efficiency, **Built-in autoscaling** should be provided by the infrastructure of a cloud-native system. That means, the capability of automatically horizontally scaling the components of a system, should be an inherent part of the infrastructure [255]. A default strategy is typically applied, for example, based on CPU utilization, but a configuration of scaling policies should be possible. Autoscaling includes both up- and down-scaling and thus improves **Elasticity**, because the amount of resources is adjusted to suit current demand in a timely way. Furthermore, **Availability** is impacted positively, because autoscaling can react to sudden traffic increases and avoid a system being unavailable due to instances being overloaded.

Factor **Built-in autoscaling**
Horizontal up- and down-scaling of components is automated and built into the infrastructure on which components run. Horizontal scaling means that component instances are replicated when the load increases and components instances are removed when load decreases. This autoscaling is based on rules which can be configured according to system needs.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Availability** ↗ **Elasticity**
Read more:
[255] 6 Use Platform Autoscaling Features
[118] 24 Elastic Scale
[26] 13 Autoscaling
[120] 1 Why container orchestration?; Scaling
[108] 5 Elasticity in Microservices
Measure(s) used for evaluation:
Deployed entities autoscaling
Infrastructure autoscaling

4.4.4. Product Factors in the Context of Portability

Regarding portability, an important aspect is the type of infrastructure on which the components of a system are deployed. The type of deployment unit supported by this infrastructure and used for the deployment of components has a major impact on how well the system could be adapted to a different infrastructure. For a better **Adaptability**, software platforms, like Kubernetes, are used which can be run on different kinds of infrastructure or in a managed way. The specific underlying infrastructure is abstracted by this platform and for the deployment of components, deployment unit types suitable to that platform are used, for example Pods. With this **Infrastructure abstraction**, a system can be adapted to a different kind of infrastructure, as long as it supports these deployment units. The underlying infrastructure of the software platform is irrelevant. Therefore, components do not depend on specific hardware requirements, but instead are described as more abstract deployment units which can be run on a specific software platform which in turn can run on various types of underlying infrastructure about which the components do not need to worry.

Factor **Infrastructure abstraction**
The used infrastructure such as physical hardware, virtual hardware, or software platform is abstracted by clear boundaries to enable a clear differentiation of responsibilities for operating and managing infrastructure. For example, when a managed container orchestration system is used, the system is operable on that level of abstraction meaning that the API of the orchestration system is the boundary. Problems with underlying hardware or VMs are handled transparently by the provider.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Adaptability**
Read more:
[26] 14 Service Brokers (make use of service brokers as an additional level of abstraction to automatically add or remove backing services)
[108] 3 Location-Independent Principle
Measure(s) used for evaluation:
Ratio of abstracted hardware

If the abstraction away from specific hardware is seen as relevant in a vertical direction, the corresponding horizontal direction would be an abstraction away from specific cloud providers. **Cloud vendor abstraction** is what is typically in focus when considering the portability of cloud applications. A system operating in a certain cloud environment can be adapted to an environment of another vendor, if the deployment units used for the components and the interfaces of managed services are supported or provided by both cloud vendors. The choice of deployment units and interfaces therefore impacts the **Adaptability** and if this choice abstracts from provider-specific technologies **Adaptability** is impacted positively. Another topic in this regard is multi-cloud. It means the ability of a system to be operated across different cloud environments concurrently. The abstraction from

specific cloud providers is key for a multi-cloud approach [120] and also positively impacts **Reusability**, because components can be reused across different cloud environments.

Factor **Cloud vendor abstraction**
The managed infrastructure and backing services used by a system and provided by a cloud vendor are based on unified or standardized interfaces so that vendor specifics are abstracted and a system could potentially be transferred to another cloud vendor offering the same unified or standardized interfaces.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Adaptability** ↗ **Reusability**
Read more:
[120] 1 Dynamic Management; Multicloud support
Measure(s) used for evaluation:
Non-provider-specific infrastructure artifacts
Non-provider-specific component artifacts

Configuration of components, especially where or how to reach other components, is a main aspect to consider for adapting a system to run in a different environment. Proper **Configuration management** therefore improves **Adaptability**. As a factor, however, it is an intermediate factor aggregating its sub-factors.

Factor **Configuration management**
Configuration values which are specific to an environment are managed separately in a consistent way. Through this, components are more portable across environments and configuration can change independently of components.
Categories: Application Administration, Data Management
Impacts: ↗ **Adaptability**
Impacted by: **Isolated configuration**, **Configuration stored in specialized services**

The basis for proper **Configuration management** is **Isolated configuration**. That means configuration data should not be stored within components, especially not hard-coded in their implementation. Instead, it is stored within the environment in which a system is deployed, for example using environment variables [60]. To adapt a component to a different environment, only the configuration data of a component needs to be changed, while the artifacts themselves remain untouched. Environment variables are the recommended approach, because it is simple to use them and they work in all operating systems and environments [118]. Also, technologies such as Docker and Kubernetes rely on environment variables for configuration [118]. While the actually used configuration should always be stored in the environment, it does not mean that no configuration can be included in a component. For development purposes it makes sense to include default values within a component, but an external configuration should then take precedence. There can even be a range of options where configuration data is stored which is then used by a component with differing priority. Potential options include prop-

erty sources inside a component, configuration data stored in a server context, command-line arguments, or environment variables.

Factor **Isolated configuration**
Following DevOps principles, environment-specific configurations are separated from component artifacts (e.g. deployment units) and provided by the environment in which a cloud-native application runs. This enables adaptability across environments (also across testing and production environments)
Categories: Application Administration, Data Management
Impacts: ↗ **Configuration management**
Read more:
[60] 6.2 The app's configuration layer
[118] 18 EnvVar Configuration
[255] 6 Never Store Secrets or Configuration Inside an Image
[3] 14 Treat Configuration as Code
[120] 1 Decoupled Configurations
Measure(s) used for evaluation:
Configuration externalization

Factor **Configuration stored in specialized services**
Configuration values are stored in specialized backing services and not only environment variables for example. That way, changing configurations at runtime is facilitated and can be enabled by connecting components to such specialized backing services and checking for updated configurations at runtime. Additionally, configurations can be stored once, but accessed by different components.
Categories: Application Administration, Data Management
Impacts: ↗ **Configuration management**
Read more:
[118] 19 Configuration Resource
[242] 11.2 "Designing configurable services
[13] 10 ConfigMaps
[26] 3 Centralized, Journaled Configuration
[26] 3 Refreshable Configuration
Measure(s) used for evaluation:
Configuration stored in config service

While configuration data is typically picked up by component instances once during initialization and then remains unchanged, in dynamic cloud environments a more advanced option is to enable changes to configuration at runtime. This approach of refreshable configuration [26], however, requires specialized backing services managing configuration data and components need to be able to detect changes to configuration data. This is covered by the factor **Configuration stored in specialized services** and different approaches exist including configuration services [242] or configuration managed by the environment as in the case of Kubernetes ConfigMaps [13] where the default approach, however, is to re-

start pods upon a configuration change. Another advantage of specialized backing services for configuration management is that configuration can be managed in a central place [26]. That way, configuration that is used by multiple components and which should be applied consistently is changed in a single place only. Furthermore, configuration should be stored and versioned, for example with git, to also enable rollbacks in case of failures caused by faulty configurations [26].

A major decision in cloud applications is how components are deployed, especially which technologies and formats are used to package, install, and execute them. In the context of **Installability** relevant aspects are which requirements the components have regarding the infrastructure they are deployed on and whether the technologies and formats used are supported by the environments of different cloud providers. If deployment artifacts of components are self-contained [108], that means they contain all software needed to run them, they have fewer requirements regarding the infrastructure and installation is easier. Furthermore, by relying on standardized formats with agreed on interfaces and widespread adoption, a support for deployment on various cloud infrastructures can be achieved. Containers are the prime example for a **Standardized self-contained deployment unit** because they include all required software in their image making them self-contained. Furthermore, with the specifications of the Open Container Initiative (OCI), standards are set which are supported by a range of technological platforms, especially container runtimes. Containers can nevertheless be kept small and their start-up times are faster than more traditional VM-based approaches.

Factor **Standardized self-contained deployment unit**
The components are deployed as standardized self-contained units so that the same artifact can reliably be installed and run in different environments and on different infrastructure.
Categories: Cloud Infrastructure, Application Administration
Impacts: ↗ **Installability**
Read more:
[241] 10 Containerized Apps
[3] 7 Use Containers (smaller deployments, separated operating system, portable);
[120] 1 Use Containerization and Container Orchestration
[101] 7 Application Runtime and Isolation
[108] 3 Deploy Independently Principle (deploy services in independent containers), Self-Containment Principle, 5 Containerization
Measure(s) used for evaluation:
Standardized deployments
Self-contained deployments

Another important factor that is typically enabled by containerization [120] is to use **Immutable artifacts**. The core idea is to implement, build and test artifacts at design time so that they are immutable at runtime. By disallowing changes to artifacts at runtime, the runtime state can always be set up fully from the previously

built artifacts. For upgrades to components, the instances are not modified, but replaced by a newer version which has also been implemented, built and tested completely at design time. Therefore, the upgrade process is predictable, and it is also possible to roll back to a previous version in case of problems by deploying the previous artifact again. Overall, **Replaceability** is impacted positively because a deployed component instance can always be replaced with a new instance in case of a failure or replaced with a new version in case of an upgrade [108].

Factor **Immutable artifacts**
Infrastructure and components of a system are defined and described in its entirety at development time so that artifacts are immutable at runtime. This means upgrades are introduced at runtime through replacement of components instead of modification. Furthermore components do not differ across environments and in case of replication all replicas are identical to avoid unexpected behavior.
Categories: Application Administration
Impacts: ↗ **Replaceability**
Read more:
[255] 6 Don't Modify Deployed Infrastructure
[120] 1 Containerization
[108] 3 Process Disposability Principle, Image Immutability Principle
Measure(s) used for evaluation:
Replacing deployments

4.4.5. Product Factors in the Context of Reliability

A cloud-based system exposed to the public internet and used by clients is expected to be available anytime, but it is also exposed to incoming traffic of potentially large scale and with potentially harmful content. Therefore, using **Guarded ingress** is important, meaning that all incoming traffic (ingress) is controlled to avoid harm from unusual ingress and ensuring the **Availability** of the system. Unusual ingress can occur without malicious intent, simply because the load is very high due to a special event. In any case, the ingress should be monitored and measures should be in place to take action if the load is unusually high or if it takes long to process requests. Such measures can be rate limiting [255], throttling [13], or load shedding [13]. Especially rate limiting and load shedding take into account information about clients and can thus also be applied per client to keep a service available for others. The overall goal is always graceful degradation [13], meaning that if a system is not able to handle all load or traffic, functionality should not degrade completely, but partially, so that at least some clients can be served. And even if a system is able to handle unusually high traffic by scaling out, **Guarded ingress**, for example, with rate limiting, is important to detect malicious traffic that might lead to unexpected costs because a system scales out, but the traffic does not generate actual revenue. Mainly, **Guarded ingress** should be in place for external endpoints, but it can also be applied internally. It can be implemented within each

component or within a proxy component, such as a gateway, where policies can be applied consistently, so that services can focus on their core domain logic.

Factor **Guarded ingress**
Ingress communication, that means communication coming from outside of a system, needs to be guarded. It should be ensured that access to external endpoints is controlled by components offering these external endpoints. Control means for example that only authorized access is possible, maliciously large load is blocked, or secure communication protocols are ensured.
Categories: Network Communication, Application Administration
Impacts: ↗ **Availability**
Read more:
[255] 6 Implement Rate Limiting and Throttling
[3] 8 Throttling (Delaying processing or responding to remain functional and decrease traffic from individual clients) (should be automated, part of graceful degradation)
[3] 8 Load shedding (In case of traffic spike, deny low priority requests to remain functional) (should be automated, part of graceful degradation)
[108] 5 Throttling
Measure(s) used for evaluation:
Ratio of components whose external ingress is proxied
Ratio of rate limiting endpoints

While cloud computing relies on abstracting from actual servers, physical hardware nevertheless is used in the end and hardware can fail or become unreachable over the network. This fact is built into cloud environments where a differentiation is made between availability zones that map to actual data centers and thus enables a cloud consumer to see where infrastructure is running. For different cloud services, there often is a choice between provisioning in a single availability zone or in multiple zones. The more availability zones are used, the more costly a deployment becomes. But such a **Distribution** over more than one availability zone has a positive impact on **Availability**, because a failure in one availability zone does not mean that the whole system is unavailable. The factor **Distribution** is an intermediate factor, aggregating two sub-factors.

Factor **Distribution**
Components are distributed across locations and data centers for better availability, reliability, and performance.
Categories: Data Management, Cloud Infrastructure
Impacts: ↗ **Availability**
Impacted by: **Physical data distribution**, **Physical service distribution**
Measure(s) used for evaluation:
Number of availability zones used by infrastructure

One sub-factor for distribution is **Physical data distribution** explicitly focusing on stateful components that persist the data aggregates needed within a system.

Distributing data over more than one physical location is crucial, but also complex. It is crucial to enable the availability of data, even if one location is down or not reachable [255]. But a decision needs to be made on how to replicate the data. Backups can be a basic option, but may not always reflect the latest state. With read replication, a retrieval of data is ensured even if there is a failure, but updates may then not be possible for a certain time. Active-active replication requires a consensus protocol between the different storage backing service instances to ensure data consistency, but it could enable a continuous availability even if one instance is unavailable. The factor is related to **Horizontal data replication** and **Sharded data store replication** but it puts the focus on the physical locations over which data is replicated.

Factor **Physical data distribution**
Storage Backing Service instances where Data aggregates are persisted are distributed across physical locations (e.g. availability zones of a cloud vendor) so that even in the case of a failure of one physical location, another physical location is still useable.
Categories: Data Management, Cloud Infrastructure
Impacts: ↗ **Distribution**
Read more:
[255] 6 Keep Data in Multiple Regions or Zones
[120] 4 Data Sharding Pattern: Geographically distribute data
Measure(s) used for evaluation:
Number of availability zones used by storage services

The distribution of stateless services over different physical locations as described by the factor **Physical service distribution** is less complex than the distribution of stateful services. New service instances can be deployed in different availability zones without the need to coordinate with other service instances. For high availability services instances are kept running in different availability zones so that a direct takeover is possible, if service instances become unavailable. Otherwise, it would also be possible to deploy new service instances in another availability zone on-demand, but then there might be a certain downtime depending on the initialization time of an instance.

Factor **Physical service distribution**
Components are distributed through replication across physical locations (e.g. availability zones of a cloud vendor) so that even in the case of a failure of one physical location, another physical location is still useable.
Categories: Cloud Infrastructure
Impacts: ↗ **Distribution**
Measure(s) used for evaluation:
Number of availability zones used by services

To ensure the **Availability** of a system also during upgrades of services, **Seamless upgrades** need to be supported. These can be implemented in different ways, but

the overall goal is always to redirect traffic from an old version to a new version seamlessly so that there is no downtime experienced by clients. Therefore, there must be a capability to redirect traffic and a possibility to run differing versions in parallel. This is detailed further in the sub-factors impacting this factor.

Factor **Seamless upgrades**
Upgrades of services do not interfere with availability. There are different strategies, like rolling upgrades, to achieve this which should be provided as a capability by the infrastructure.
Categories: Application Administration, Cloud Infrastructure, Network Communication, Business Domain
Impacts: ↗ **Availability**
Impacted by: **Separation by gateways**, **Rolling upgrades enabled**

Factor **Rolling upgrades enabled**
The infrastructure on which components are deployed provides the ability for rolling upgrades. That means upgrades of components can be performed seamlessly in an automated manner. Seamlessly means that upgrades of components do not necessitate planned downtime.
Categories: Application Administration, Cloud Infrastructure
Impacts: ↗ **Seamless upgrades**
Read more:
[60] 7.2
[255] 6 Use Zero-Downtime Releases
[118] 3 Declarative Deployment
[241] 10 Risk-Reducing Deployment Strategies
[13] 13 Rolling Updates
[120] 1 Why container orchestration?; Rolling upgrades
Measure(s) used for evaluation:
Rolling update option
Rolling updates

With an infrastructure that has **Rolling upgrades enabled** it is possible to upgrade components deployed on that infrastructure in a controlled way avoiding downtime. Rolling means that an infrastructure is able to run different versions of the same component in parallel for a certain time frame. Typically, instances of the new version are started gradually, and the traffic is partly redirected to these new instances [60]. If there are no errors, all traffic is eventually routed to the new instances and the old instances are gradually stopped. Alternatively to rolling upgrades, blue/green upgrades [60] can be used where all new instances are first started while the old instances still serve requests and then at a certain point all traffic is redirected to the new instances. While such upgrade processes could be executed manually as well, in a cloud-native system such capabilities should be provided by the infrastructure in an automated way.

Similar to how components should be upgraded in a way that does not interrupt availability, also the infrastructure on which components run should be maintained without impacting the operation of the system. Automation is a key concept here [108] and maintenance tasks like operating system upgrades or certificate regeneration should be done automatically by the infrastructure as described by the factor **Automated infrastructure maintenance**. By keeping the infrastructure up-to-date automatically, **Availability** is ensured. And through the automation of maintenance tasks, a system's **Recoverability** is impacted positively as well, because if infrastructure needs to be restarted for some reason, such tasks are performed automatically as well, reducing the time it takes to recover.

Factor **Automated infrastructure maintenance**
The used infrastructure should automate regular maintenance tasks as much as possible in a way that the operation of components is not impacted by these tasks. Such tasks include updates of operating systems, standard libraries, and middleware managed by the infrastructure, but also certificate regeneration.
Categories: Cloud Infrastructure, Application Administration
Impacts: ↗ **Availability** ↗ **Recoverability**
Read more:
[241] 10 Automated Infrastructure
[108] 5 Automation
Measure(s) used for evaluation:
Ratio of automatically maintained infrastructure

Factor **Persistent communication**
Links persist messages which have been sent (e.g. based on messaging backing services). That way, components are decoupled, because components need not yet exist at the time a message is sent, but can still receive a message. Communication can also be repeated, because persisted messages can be retrieved again.
Categories: Network Communication
Impacts: ↗ **Fault tolerance**
Read more:
[120] 5 Event Sourcing Pattern: Log-based message brokers
Measure(s) used for evaluation:
Service interaction via backing service
Event sourcing utilization metric

In the context of **Fault tolerance** a factor that can have a positive impact on it is **Persistent communication**. Especially for asynchronous communication via broker backing services, it is possible to persist the messages that are sent via the broker. That way, if a component fails, also communication that has been lost during the component outage can be recovered, because the messages still exist and can be received by the component that failed. With persistent communication, a System is therefore more tolerant to communication faults that can happen within a cloud environment, especially if a System is distributed. An explicit approach

based on this concept is event sourcing [120]. Communication happens via events that capture incidents relevant to the domain of a system. These are stored in a specialized service , called the event store, and can be retrieved by other components from there.

A common approach for implementing systems in a fault-tolerant way is to resolve or at least react to faults automatically if they occur. Ideally, a fault then has no effect on the processing of requests within a system . Of course, this is not possible for all kinds of faults and in all contexts, but where applicable such **Autonomous fault handling** can have a positive impact on the **Fault tolerance** of a system. The factor **Autonomous fault handling** is an intermediate factor aggregating different sub-factors, currently focusing mostly on communication faults.

Factor **Autonomous fault handling**
Services expect faults at different levels and either handle them or minimize their impact by relying on the capabilities of cloud environments.
Categories: Network Communication, Cloud Infrastructure
Impacts: ↗ **Fault tolerance**
Impacted by: **Invocation timeouts**, **Retries for safe invocations**, **Circuit breaked communication**

A basic fault tolerance mechanism that should be used independently of the cloud context in any distributed communication over the network is **Invocation timeouts**. A timeout value is set, to specify after which time a request should be canceled, if no response is received [120]. It does handle the fault in the sense that the fault is not perceived from the outside, but at least the problem becomes apparent and the invoking component is not blocked infinitely while waiting for a response. Although **Invocation timeouts** are a basic mechanism, they are relevant in distributed cloud-based systems, and they are also the basis for more complex fault tolerance mechanisms.

Factor **Invocation timeouts**
For links between components, timeouts are defined to avoid infinite waiting on a service that is unavailable and a timely handling of problems.
Categories: Network Communication
Impacts: ↗ **Autonomous fault handling**
Read more:
[120] 3 Resilient Connectivity Pattern: Time-out
[242] 3.2.3 Handling partial failures using the Circuit Breaker pattern
[108] 5 Timeout
Measure(s) used for evaluation:
Ratio of links with timeout

Building on timeouts, another mechanism for **Autonomous fault handling** is to use **Retries for safe invocations**. If a request times out, it can be retried a finite amount of times [255]. For transient, that means temporary problems, this might be enough for resolving the issue, if one of the retries executes successfully. Retries, however, can be used straightforward only for certain types of endpoints.

Specifically, endpoints that should be retried need to be safe, meaning that they do not change state. Thus, they are idempotent [249] and can be invoked multiple times. Retrying endpoints that are not idempotent might lead to erroneous states, because it is possible that an invocation is retried, although the first request was successful, but it was considered by the invoking component as having failed.

Factor **Retries for safe invocations**
Links that are safe to invoke multiple times without leading to unintended state changes, are automatically retried in case of errors to transparently handle transient faults in communication. That way faults can be prevented from being propagated higher up in a request trace.
Categories: Network Communication
Impacts: ↗ **Autonomous fault handling**
Read more:
[60] 9.1
[255] 6 Handle Transient Failures with Retries
[255] 6 Use a Finite Number of Retries
[26] 12 Isolating Failures and Graceful Degradation: Use retries
[120] 3 Resilient Connectivity Pattern: Retry
[249] 9 Synchronous Request/Response (Use retries in synchronous communications)
[249] 9 The Importance of Idempotency (Communication which is retried needs idempotency)
[108] Idempotent Service Operation, Retry, 5 Retry
Measure(s) used for evaluation:
Ratio of links with retry logic

Factor **Circuit breaked communication**
For links a circuit breaker implementation is used which avoids unnecessary communication and therefore waiting time if a communication is known to fail. Instead, the circuit breaker immediately returns an error response of a default response, is possible, while periodically retrying communication in the background.
Categories: Network Communication
Impacts: ↗ **Autonomous fault handling**
Read more:
[60] 10.1
[255] 6 Use Circuit Breakers for Nontransient Failures
[242] 3.2.3 Handling partial failures using the Circuit Breaker pattern
[26] 12 Isolating Failures and Graceful Degradation: circuit breaker
[120] 3 Resilient Connectivity Pattern: Circuit breaker
[108] 4 Circuit Breaker
Measure(s) used for evaluation:
Ratio of links with complex failover

An even more advanced mechanism than retries is to use a so-called circuit breaker [120]. It is applied on the client side and all invocations of an endpoint are proxied by the circuit breaker. If invocations of that endpoint fail, they can be retried. But if the issue persists, the circuit breaker changes its state to directly return any further invocations of the problematic endpoint . The circuit breaker response can either be an error message or a default answer, if applicable, so that a partial request processing is possible [242]. The advantages are that clients can proceed directly without waiting for timeouts or retries and that services which might not be able to respond because they are overloaded have time to recover since the circuit breaker blocks additional requests. After a specified time, the circuit breaker will retry reaching the endpoint and if it succeeds again, all other requests are again sent to the actual service again.

Because component instances in cloud environments may fail for various reasons, a common approach is to apply **Automated restarts** which need to be implemented in the infrastructure or an external backing service that has the ability to monitor components and restart or redeploy them on the corresponding infrastructure. **Automated restarts** can enable a system to quickly recover in the case of temporal or unusual failures which do not require fixes within a component [26]. For more complex faults that require changes in the implementation, of course a fix introduced through an upgrade of the component is required. A challenge is how to detect that a component has failed and should be restarted. This can be done either via the deployment infrastructure, because it detects that a process has failed with an error exit code. Or, as a better approach, health checks can be used. These are explicitly intended for this use case and components providing them therefore also have a positive impact on the applicability of **Automated restarts**.

Factor **Automated restarts**
When a component is found to be unhealthy, that means not functioning as expected, it is directly and automatically restarted. Ideally this capability is provided by the infrastructure on which a component is running.
Categories: Cloud Infrastructure, Application Administration
Impacts: ↗ **Recoverability**
Impacted by: **Health and readiness checks**
Read more:
[26] 13 automatic remediation
[120] 1 Why container orchestration?; High availability
[108] 5 Self-Healing
Measure(s) used for evaluation:
Deployments with restart

4.4.6. Product Factors in the Context of Compatibility

The communication between services happens via APIs [241] and how well they can be used to integrate services defines how interoperable components within a system are. Good **API-based communication** depends on various aspects, like the architectural style and consistency of an API, but mainly whether the API provides the functionality required by clients within a limited number of cohesive endpoints. In any case, the API of a component should be understandable and well-documented to facilitate **Interoperability**. For that, interface specifications [3] can be used, such as the popular OpenAPI[49] specification for RESTful APIs. With a well-specified interface, that can be understood well, aligns with common interface conventions, and documents potential edge cases, implementing links to a component is facilitated in the sense that it can be done faster and safer. Thus, the endpoints of a component should be documented with an interface specification artifact to fulfill this factor. An additional positive impact exists on **Testability** because with a structured interface specification it is also possible to write tests that check the behavior of an interface based on the specification.

Factor **API-based communication**
All endpoints that are offered by a service are part of a well-defined and documented API. That means, the APIs are based on common principles, are declarative instead of imperative, and are documented in a standardized or specified format (such as the OpenAPI specification). Communication only happens via endpoints that are part of such APIs and can be both synchronous or asynchronous.
Categories: Network Communication, Business Domain
Impacts: ↗ **Interoperability** ↗ **Testability**
Read more:
[241] 9 Communicate Through APIs
[3] 6 Understandable Interface Specifications (Use Interface specifications for understandability
[26] 6 Everything is an API (Services are integrated via APIs)
[120] 2 Service Definitions in Synchronous Communication (Use a service definition for each service);
[120] 2 Service Definition in Asynchronous Communication (Use schemas to define message formats);
[108] 3 API First Principle
Measure(s) used for evaluation:
Ratio of documented endpoints

On top of specifications that document an API, it is also possible to use more binding contracts that explicitly capture how clients use an API. This is covered by the factor **Contract-based links** and the concept of consumer-driven contract testing [26] is the main example of how this factor can be fulfilled. A contract, again formulated using a structured format, specifies the expectations of a client

[49] `https://www.openapis.org/`, visited 2025-10-20

regarding an API, e.g. the endpoints of a service. By testing against this contract, it can be verified that the integration between client and service works as expected, ensuring **Interoperability**. If the client asks for changes or new functionalities, this can be guided by adapting the contract and the service can then be modified to fulfill the contract again. Furthermore, if changes are made in the service, it can be validated whether these changes break existing client invocations based on the contracts. And if a potential breaking of a contract is detected, the change can be delayed or adjusted.

Factor **Contract-based links**
Contracts are defined for the communication via links so that changes to endpoints can be evaluated by their impact on the contract and delayed when a contract would be broken. That way consumers of endpoints can adapt to changes when necessary without suddenly breaking communication via a link due to a changed endpoint.
Categories: Network Communication, Business Domain
Impacts: ↗ **Interoperability** ↗ **Adaptability**
Read more:
[26] 4 Consumer-Driven Contract Testing (Use contracts for APIs to test against)
Measure(s) used for evaluation:
Ratio of endpoints covered by contract

Many previously described factors cover mechanisms and characteristics implemented in addition to domain-specific aspects. These are often implemented similarly for several components within a system. Therefore, it can make sense to bundle such things that should be applied consistently within a system in a unifying approach. Especially characteristics in the context of network communication can be covered by mediating communication via proxies alongside components. The factor **Consistently mediated communication** covers this and describes exemplary tasks that can be implemented within mediating backing services instead of services themselves. A common approach to do so is to use a service mesh[60]. When using a service mesh, so-called sidecars are deployed alongside components which act as a proxy mediating all in- and out-going communication of a component. The sidecars are controlled by a central control plane and from there policies regarding fault tolerance mechanisms, load balancing, or monitoring can be consistently enforced. Consistently mediating communication has a positive impact on **Interoperability** because for integrating components with each other it is possible to focus on the specific business logic while other aspects can be implemented consistently via the mediating backing services. Many functionalities are available within such backing services, but using them may have a negative impact on **Time-behavior**, because this mediation is not without overhead. However, **Analyzability** is impacted positively, because systems can be analyzed and monitored from a central place in a consistent way.

Factor **Consistently mediated communication**
By mediating communication through additional components, there is no direct dependence on the other communication partner and additional operations can be performed to manage the communication, such as load balancing, monitoring, or the enforcement of policies. By using centralized mediation approaches, such as Service Meshes, management actions can be performed universally and consistently across the components of an application.
Categories: Network Communication
Impacts: ↗ **Interoperability** ↘ **Time-behaviour** ↗ **Analyzability**
Read more:
[120] 3 Sidecar Pattern, Service Mesh Pattern, Service Abstraction Pattern (Proxy communication with services to include service discovery and load balancing)
[60] 10.3
[242] 11.4.2
Measure(s) used for evaluation:
Service mesh usage

4.5. Impacts

Parts of this section have been taken from [174].

The connections between elements of the quality models are formed through the formulation of impacts. They describe how an impacted factor, e.g. a quality aspect, is influenced if the factor from which the impact originates is present in a system. The impact relationship can be either positive (↗) which means that the impacted factor is improved, or it can be negative (↘) which corresponds to the opposite meaning of a factor being worsened. The actual effect of an impact depends on whether the factor from which an impact originates is present in a system or not. If it is present, also the extent to which it is present is important. In general, a proportional effect is assumed, meaning that the more a factor is present, the stronger is the impact effect. It has to be noted that according to the Quamoco metamodel [294] an impact relation is either positive or negative and therefore not invertible in the sense that the non-existence of a factor would have the opposite impact effect. If an impacting factor is not present in a system, thus, simply no impact effect is assumed. It has to be noted that in the initial quality model and also while doing the survey [174], a differentiation in the strength of impacts was made. For the current state of the quality model, however, this differentiation was dropped, because it is in general difficult to argue when a "slightly positive" impact exists or a "positive" one in comparison. Instead, only a differentiation between "negative" and "positive" impacts is made, without further differentiation. Nevertheless, as the survey used this differentiation, it is included in the results presented in the following.

All impacts of the quality model have already been presented together with the relevant product factors in Section 4.4 and are therefore not explicitly repeated in this section. Their initial formulation was done together with the formulation of the product factors based on the reviewed literature. Because the literature describing characteristics of cloud-native applications also discusses the potential effects of these characteristics, the corresponding impact relationships could be derived from this. Due to the significance of the impact relationships as the connecting elements within the quality model, ensuring their validity is important. As described in Section 3.1.3, a validation survey was therefore performed focusing explicitly on the impact relations between the factors of the quality model. The results of this survey [174] are presented in the following, also providing more details on the characteristics of impacts.

There are two main groups of results. On the one hand, the survey aimed to validate existing impacts specified in the quality model in its then current status. And on the other hand, additional impacts should be identified which were not yet part of the quality model. The results for validating existing impacts are shown in Table 4.7, Table 4.8, and Table 4.9. In these tables, results are grouped by product factors to make impacts comparable per product factor. The significant results for the identification of additional impacts are shown in Table 4.10. All tables

show the number of answers per impact strength, ranging from "negative" (--) over "slightly negative" and "no impact" (0) to "slightly positive" (+) and "positive" (++). The "Mean" is calculated by assigning the stated impacts numerical values from -2 (--), -1 (-), over 0 (0) to 1 (+) and +2 (++) and calculating the mean from these values.

A positively hypothesized impact is rated as valid (✓), if $mean > 0.5$ and $p < 0.1$ while a negatively hypothesized impacts is rated as valid if $mean < -0.5$ and $p < 0.1$.

Table 4.7.: Factors with at least one successful validation of initially stated impacts [174]

Factor	Impact	--	-	0	+	++	Mean	Valid	p-Value
Authentication delegation	↗ Authenticity	0	0	0	2	3	1.60	✓	0.0288
Automated Infrastructure	↗ Modifiability	0	2	12	1	5	0.45	✗	0.0224
	↗ Recoverability	0	0	7	3	10	1.15	✓	0.0006
Automated restarts	↗ Recoverability	0	0	2	2	8	1.50	✓	0.0007
Built-in autoscaling	↗ Elasticity	0	0	3	2	12	1.53	✓	0.0000
	↗ Resource utilization	0	0	3	3	11	1.47	✓	0.0000
Cloud vendor abstraction	↗ Adaptability	0	0	4	1	6	1.18	✓	0.0176
Consistent centralized metrics	↗ Analyzability	0	0	1	3	3	1.29	✓	0.0560
	↗ Recoverability	0	0	6	0	1	0.29	(✗)	0.1164
Distributed tracing of invocations	↗ Analyzability	0	0	1	2	5	1.50	✓	0.0122
	↗ Recoverability	0	0	7	0	1	0.25	✗	0.0423
Health and readiness checks	↗ Analyzability	0	0	5	1	3	0.78	(✓)	0.2329
	↗ Recoverability	0	0	5	0	4	0.89	✓	0.0414
Infrastructure abstraction	↗ Modifiability	0	0	5	2	1	0.50	(✓)	0.3182
	↗ Adaptability	0	0	2	4	2	1.00	✓	0.0970
	↗ Recoverability	0	0	6	2	0	0.25	(✗)	0.1055
Managed infrastructure	↗ Simplicity	0	0	11	1	2	0.36	✗	0.0093
	↗ Resource utilization	0	0	7	4	3	0.71	✓	0.0765
Physical service distribution	↗ Availability	0	0	1	1	8	1.70	✓	0.0001
Service replication	↗ Time-behavior	0	0	4	4	6	1.14	✓	0.0140
Use infrastructure as code	↗ Modifiability	0	0	12	3	1	0.31	✗	0.0048
	↗ Installability	0	0	8	1	7	0.94	✓	0.0039
	↗ Recoverability	0	0	8	3	5	0.81	✓	0.0378
Vertical data replication	↗ Time-behavior	0	0	1	0	4	1.60	✓	0.0288

Table 4.7 contains product factors with at least one validated impact (indicated by "✓"). As it shows, this is the case for several initially stated impacts and these impacts were kept in the quality model as-is. Other impacts, however, could not be validated, but individual ratings did support the impacts. For example, this

is the case for the impact from **Infrastructure abstraction** on **Modifiability** which has been rated by some as being positive, while most rated it as having no impact. But it has to be noted here that the value "no impact" (0), was pre-selected by default by the survey tool. A rating of "no impact" therefore can also be the result of a user rating impacts for a product factor, but not considering all potential options. Such impacts which could not be successfully validated were therefore not removed directly, but revised based on these results and the consideration of the literature on which they are based. This is in contrast to results were there are mixed responses, as it is the case for the impact from Automated infrastructure on **Modifiability**. These results indicate a problem with the formulation of the impact and the product factor was therefore revised. The factor Automated infrastructure in-specific was split into two factors which is also discussed later, because of the additional results on impacts that were not previously considered.

Listed in Table 4.8, there are product factors with at least one potentially valid impact. For most of the impacts, a clear tendency towards their validation is recognizable. Tendency means that individual ratings supported these impacts, but there are simply not enough answers to rate the impact as validated based on the used evaluation. This is for example the case for the impact from **Retries for safe invocations** on **Fault tolerance** which is clearly supported by the mean value, but because it was rated only five times, it has overall been evaluated as potentially valid (indicated by "(✓)"). For all impacts listed in Table 4.8 a thorough reconsideration was applied to decide whether to keep it or remove it from the quality model. To do so, also the literature representing the initial reasoning for each impact was reconsidered.

As shown in Table 4.9, several impacts could not be validated or are potentially invalid. A partial explanation for invalidated impacts is that to keep the survey manageable, mediating factors were not considered which could have an explanatory function. When a mediating factor is missing, the indirect impact from a factor on a quality aspect might be less obvious. For example, the impact from **Limited request trace scope** on **Modifiability** is mediated by **Service independence** which was not part of the survey. Therefore, the results needed to be considered on a case-by-case basis to decide whether to remove or keep the impact in the quality model. Those impacts which could clearly not be validated ("✗") were removed from the quality mode. Exceptions are cases where the investigated impact was mediated by a factor not considered in the survey. Here, impacts were kept for further validation if it was considered to be nevertheless relevant. This applies for example to the impact from **Limited request trace scope** on **Modifiability**, mediated by the factor **Service independence**.

Furthermore, there are results that show a mixed picture of stated impacts. These partly reveal factors which can be understood differently and therefore need to be revised. For example, this was the case for the factor **Resource limits** which was described in the survey as: '*For all components the maximum amount of resources a component can consume is limited based on its predicted needs so that resources are*

Table 4.8.: Factors with at least one potentially valid initially stated impact [174]

Factor	Impact	--	-	0	+	++	Mean	Valid	p-Value
Access control management consistency	↗ Integrity	0	0	4	4	2	0.80	(✓)	0.1549
Account separation	↗ Accountability	0	0	4	0	3	0.86	(✓)	0.1244
API-based communication	↗ Interoperability	0	0	6	1	3	0.70	(✓)	0.1655
	↗ Testability	0	0	4	4	2	0.80	(✓)	0.1549
Circuit breaked communication	↗ Fault tolerance	0	0	3	1	4	1.12	(✓)	0.1455
Configuration stored in specialized services	↗ Adaptability	0	0	1	0	1	1.00	(✓)	1.0000
	↗ Availability	0	1	1	0	0	-0.50	(⇄)	1.0000
Consistent centralized logging	↗ Analyzability	0	0	1	2	3	1.33	(✓)	0.1185
	↗ Recoverability	0	0	5	0	1	0.33	(✗)	0.1718
Dynamic scheduling	↗ Modifiability	0	0	6	0	0	0.00	✗	0.0696
	↗ Resource utilization	0	0	2	1	3	1.17	(✓)	0.3158
	↗ Recoverability	0	0	6	0	0	0.00	✗	0.0696
Immutable artifacts	↗ Replaceability	0	0	3	0	1	0.50	(✓)	0.6296
Limited data scope	↗ Modularity	0	0	2	2	1	0.80	(✓)	0.7222
Logical grouping	↗ Co-Existence	0	0	4	0	0	0.00	(✗)	0.2160
	↗ Modifiability	0	0	3	0	1	0.50	(✓)	0.6296
Managed backing services	↗ Simplicity	0	0	4	3	2	0.78	(✓)	0.3169
	↗ Resource utilization	0	0	7	2	0	0.22	✗	0.0433
Mostly stateless services	↗ Modularity	0	0	5	1	3	0.78	(✓)	0.2329
	↗ Elasticity	0	0	4	1	4	1.00	(✓)	0.1325
	↗ Replaceability	0	0	6	2	1	0.44	(✗)	0.2329
Physical data distribution	↗ Availability	0	0	2	1	4	1.29	(✓)	0.1100
Retries for safe invocations	↗ Fault tolerance	0	0	1	2	2	1.20	(✓)	0.2695
Rolling upgrades enabled	↗ Availability	0	0	0	0	3	2.00	(✓)	0.0185
Secrets stored in specialized services	↗ Availability	1	1	4	0	0	-0.50	(⇄)	0.6399
	↗ Confidentiality	0	0	2	2	2	1.00	(✓)	0.4547
Separation by gateways	↗ Reusability	0	0	2	0	0	0.00	(✗)	1.0000
	↗ Availability	0	0	0	2	0	1.00	(✓)	0.1111
	↗ Integrity	0	0	2	0	0	0.00	(✗)	1.0000
Sharded data store replication	↗ Time-behavior	0	0	3	0	1	0.50	(✓)	0.6296

provisioned efficiently. That means a component gets the resources that it needs, but not more than necessary. By making the resource requirements explicit, for example in a configuration file, these limits can be enforced by the infrastructure.' The description includes the implicit assumption that resource requirements of components are

Table 4.9.: Factors with (potentially) invalid and unclear initially stated impacts [174]

Factor	Impact	--	-	0	+	++	Mean	Valid	p-Value
Addressing abstraction	↗ Interoperability	0	0	5	0	1	0.33	(✗)	0.1718
	↗ Modularity	0	0	6	0	0	0.00	✗	0.0696
Asynchronous communication	↗ Modularity	0	0	5	1	1	0.43	(✗)	0.3459
Backing service decentralization	↗ Co-Existence	0	0	6	0	0	0.00	✗	0.0696
	↗ Modifiability	0	0	5	0	1	0.33	(✗)	0.1718
Command query responsibility segregation	↗ Modularity	0	0	5	2	0	0.29	(✗)	0.1430
Communication partner abstraction	↗ Interoperability	0	0	3	0	0	0.00	(✗)	0.4444
	↗ Modularity	0	0	2	1	0	0.33	(✗)	1.0000
	↘ Analyzability	0	0	3	0	0	0.00	(✗)	0.4444
Contract-based links	↗ Interoperability	0	0	5	0	0	0.00	(✗)	0.1101
	↗ Testability	0	0	5	0	0	0.00	(✗)	0.1101
Horizontal data replication	↗ Time-behavior	0	0	7	0	1	0.25	✗	0.0423
Limited request trace scope	↗ Co-Existence	0	0	5	1	0	0.17	(✗)	0.1718
	↗ Modifiability	0	0	6	0	0	0.00	✗	0.0696
Consistently mediated communication	↗ Interoperability	0	0	4	1	0	0.20	(✗)	0.3519
	↗ Modularity	0	0	4	1	0	0.20	(✗)	0.3519
Persistent communication	↗ Modularity	0	0	4	1	0	0.20	(✗)	0.3519
	↗ Recoverability	0	0	3	2	0	0.40	(✗)	0.3519
Resource Limits	↗ Resource utilization	2	2	2	3	2	0.09	(✗)	0.7243
Specialized stateful services	↗ Modularity	0	0	2	0	0	0.00	(✗)	1.0000
	↗ Elasticity	1	0	1	0	0	-1.00	(⇄)	1.0000
	↗ Replaceability	0	0	2	0	0	0.00	(✗)	1.0000
Usage of existing solutions for non-core capabilities	↗ Reusability	0	0	4	3	0	0.43	(✗)	0.1244

known well, so that appropriate resource limits can be set. If appropriate resource limits are set, resource utilization is impacted positively because components can be provided with the resources that they need. And the infrastructure only has to provide the specified resources, not more. But if that assumption does not hold and resource limits are set too strict, a negative impact on resource utilization can result. Components might not get the resources they actually need, even though resources would still be available which are then in turn also not utilized. A similar behavior is imaginable regarding the impact on **Availability**. When appropriate resource limits are set, misbehaving components can not take away shared resources from other components. This protects other components and has a positive im-

pact on **Availability**. However, the availability of individual components might be impacted negatively, if they do not get the resources they need due to too strict resource limits. Thus, the quality model was revised based on the results to better reflect these circumstances. The factor *Resource limits* was refined into a new factor **Enforcement of appropriate resource boundaries** that captures both aspects of providing components with the resources they need, but also enforcing limits.

Each impact showing unclear results ("(✗)" or "(⇄)"), was thoroughly reconsidered to decide whether to keep it or remove it from the quality model. Again, also literature leading to the initial reason for that impact was reconsidered.

In addition to the validation of previously formulated impacts, the second aim of the survey was to identify additional impacts. The large variance of provided answers, however, represents a challenge. To make a selection of which results to consider for incorporation in the quality model, the significance of results was considered as an indicator. Specifically, only the results with the highest significance (that means the lowest p-Value) were selected. In Table 4.10, the additional impacts with the highest significance, with a p-Value of < 0.1, are included.

One product factor with noteworthy results is **Automated infrastructure**. In the originally formulated quality model it was mainly based on the pattern *Automated infrastructure* as described by Reznik et al. [241] in chapter 10. Reznik et al. argue that through automated provisioning of required infrastructure, modifications are simplified and can be done more quickly. The results from the validation survey [174] show an ambiguity for this factor, because several impacts on different quality aspects were reported. There are two possible explanations for this result. One is that factors can be rather abstract and comprise several aspects, which makes their assessment ambiguous. Depending on which aspects of a factor participants think of, their ratings may involve multiple quality aspects. This can be addressed by splitting up factors into (sub-)factors that capture more clearly distinguishable aspects of cloud-native applications and make the formulated impact relationships more precise. Another possible explanation is that the quality aspects themselves are not completely conceptually disjoint. Following this explanation, certain quality aspects overlap conceptually, i.e. they share certain concepts. If a factor in question now impacts specifically such a shared concept, it would be plausible for it to impact both quality aspects simultaneously. The consequence of this explanation would be that the different quality aspects could not only be considered as distinguishable sub-aspects of their respective high-level aspects, but rather as top-level quality aspects themselves.

On the basis of the survey results, the first explanation was given more weight and the factor was therefore refined and split up into two separate factors **Automated infrastructure provisioning** and **Automated infrastructure maintenance**, which impact the respective quality aspects. On the one hand, the positive impact on **Availability** was assigned to the factor **Automated infrastructure maintenance**, because it ensures the correct functioning of components during operation, therefore ensuring **Availability**.

Table 4.10.: Factors with impacts considerable for the quality model (p < 0.1) [174]

Factor	Quality Aspect	--	-	0	+	++	Mean	Impact	p-Value
API-based communication	Modularity	0	1	2	0	7	1.30	↗	0.0022
	Reusability	0	0	4	1	5	1.10	↗	0.0480
	Replaceability	0	0	7	0	3	0.60	↗	0.0377
Automated Infrastructure	Modularity	0	0	13	2	5	0.60	↗	0.0031
	Reusability	0	0	11	2	7	0.80	↗	0.0028
	Testability	0	2	9	3	6	0.65	↗	0.0997
	Capability	0	0	14	1	5	0.55	↗	0.0008
	Elasticity	0	0	11	2	7	0.80	↗	0.0028
	Resource utilization	0	0	13	2	5	0.60	↗	0.0031
	Time-behavior	0	0	13	2	5	0.60	↗	0.0031
	Installability	0	1	8	2	9	0.95	↗	0.0050
	Availability	0	0	7	3	10	1.15	↗	0.0006
	Fault tolerance	0	0	11	1	8	0.85	↗	0.0006
Automated restarts	Availability	0	1	0	3	8	1.50	↗	0.0001
	Fault tolerance	0	0	3	3	6	1.25	↗	0.0137
Built-in autoscaling	Capability	0	0	7	3	7	1.00	↗	0.0095
	Time-behavior	0	0	8	4	5	0.82	↗	0.0292
	Availability	0	0	8	5	4	0.76	↗	0.0292
	Fault tolerance	0	0	8	5	4	0.76	↗	0.0292
Cloud vendor abstraction	Reusability	0	0	6	1	4	0.82	↗	0.0804
	Replaceability	0	0	6	1	4	0.82	↗	0.0804
Command query responsibility segregation	Simplicity	0	4	3	0	0	-0.57	↘	0.0560
Dynamic scheduling	Capability	0	0	1	0	5	1.67	↗	0.0027
Health and readiness checks	Availability	0	0	3	1	5	1.22	↗	0.0414
Managed infrastructure	Elasticity	0	0	7	4	3	0.71	↗	0.0765
	Availability	0	0	4	3	7	1.21	↗	0.0074
	Fault tolerance	0	0	9	2	3	0.57	↗	0.0499
	Maturity	0	0	8	3	3	0.64	↗	0.0765
	Recoverability	0	0	8	3	3	0.64	↗	0.0765
Mostly stateless services	Testability	0	0	2	0	7	1.56	↗	0.0006
Physical data distribution	Fault tolerance	0	0	1	1	5	1.57	↗	0.0095
Physical service distribution	Fault tolerance	0	0	2	3	5	1.30	↗	0.0233
Service replication	Availability	0	0	1	4	9	1.57	↗	0.0000
	Fault tolerance	0	0	5	3	6	1.07	↗	0.0238
Usage of existing solutions for non-core capabilities	Simplicity	0	0	3	4	0	0.57	↗	0.0560
Use infrastructure as code	Reusability	0	0	9	2	5	0.75	↗	0.0242
	Testability	0	0	9	3	4	0.69	↗	0.0404
	Adaptability	0	0	8	6	2	0.62	↗	0.0163
	Replaceability	0	0	10	3	3	0.56	↗	0.0286
	Availability	0	0	9	3	4	0.69	↗	0.0404
	Fault tolerance	0	0	12	0	4	0.50	↗	0.0009

The positive impact on **Installability**, on the other hand, was assigned to the factor **Automated infrastructure provisioning**, because it makes the installation process easier and repeatable. It has to be noted is that also the positive impact from **Automated infrastructure provisioning** on **Modifiability** was kept, although the survey results did not show a clear confirmation (see Table 4.7).

Other factors with noteworthy results are **Managed infrastructure** and **Use infrastructure as code**, because for them impacts on several quality aspects are found to be significant. This is even true for several quality aspects with the same high-level quality aspect. For example, **Use infrastructure as code** has positive impacts on **Replaceability** and **Adaptability** which are both sub-aspects of portability. Additionally, **Testability** and **Reusability** are both sub-aspects of maintainability. These results, where multiple impacts to different quality aspects were found, were analyzed to decide about their inclusion in the quality model. However, for example in the case of **Managed infrastructure** it was decided to not include further impacts in the quality model, because the aspects within the factors which were associated with certain quality aspects are already covered by other more specific factors. For example, the aspect that it has a positive impact on **Availability** is most likely due to the reasoning that the provider takes care of keeping components available through corresponding mechanisms, such as automated updates. But this aspect is already covered by the factor **Automated infrastructure maintenance**. Or the aspect that it has a positive impact on **Elasticity** is due to the reasoning that the provider takes care of scaling decisions. But this aspect is also included and better represented by the factor **Built-in autoscaling**.

Overall, therefore a decision was made on a case-by-case-basis about whether to include newly identified impacts in the quality model or not.

4.6. Measures

4.6.1. Measure Search and Formulation

Quality aspects, product factors and impacts already enable a qualitative evaluation of software architectures. For a quantitative evaluation, however, measures are needed, more specifically architectural measures. Architectural measures enable a quantification of the extent to which product factors are present in a system based on a design time analysis. This means the architecture is considered in a static way, without executing it, by analyzing source code, deployment artifacts, or, as in the case of the presented approach, models. For the measures of the quality model, there are two major sources. On the one hand, the measures identified through the literature search (as described in Section 3.1.2) were assigned to suitable product factors and adapted to the entities (see Section 4.3) of the quality model, if necessary. On the other hand, if for a product factor no measure could be found in literature, one or more new measures were formulated based on the core characteristics of the product factor and the available entities of the quality model. This lead to a total of 193 measures which are applicable within the context of the quality model. However, not all measures are equally relevant. On the one hand, not all measures found in literature could be implemented with the information included in the entities of the quality model. And on the other hand, not all implemented measures are actually included within evaluations for product factors. This is because selections had to be made if multiple measures were assigned to a single factor and because some measures were not found to be informative enough to be used. This leads to three different states for measures within the quality model:

- IN USE: The measure is implemented and used for evaluations.
- IMPLEMENTED: The measure is implemented and calculated for applicable entities, but not used in evaluations.
- UNSUPPORTED: The measure is generally applicable within the context of the quality model, but cannot be implemented with the entities of the quality model.

An interesting result is the distribution of measures per product factor. Figure 4.4 shows this distribution with the addition of how many measures were found in literature and how many were newly formulated while Figure 4.5 includes the implementation status as additional information.

As it can be seen, there are a few product factors with a lot of measures assigned, some product factors with a reasonable amount of found measures and many product factors with a single measure assigned. Specifically those measures which are assigned as single measures to a product factor have mostly been formulated newly, because no suitable measures for the product factor in question

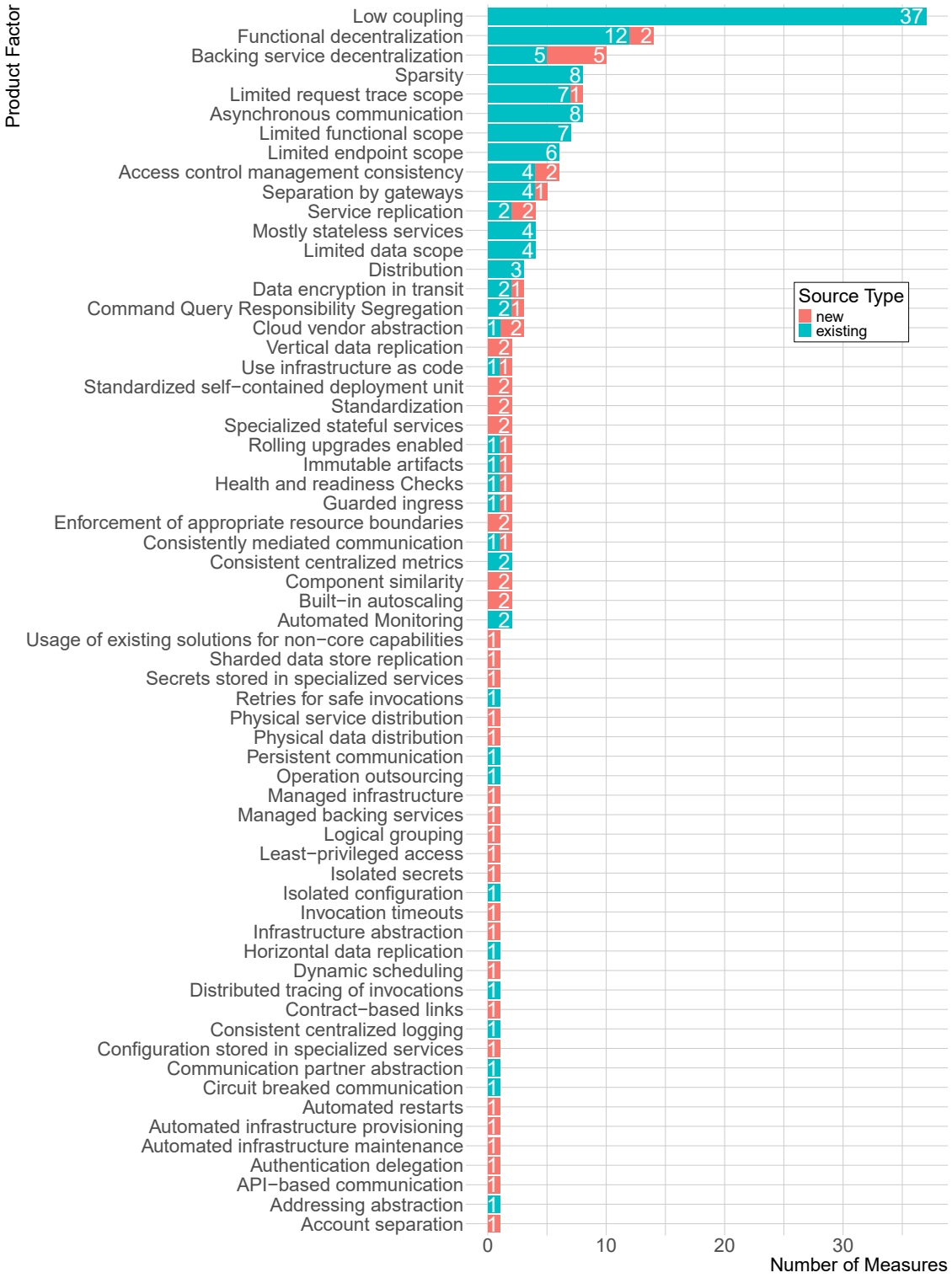


Figure 4.4.: Number of Measures per Product Factor, differentiated by source

could be found in literature. For each product factor, the number of measures which are actually used for the evaluation (Status IN USE) are typically only one or two.

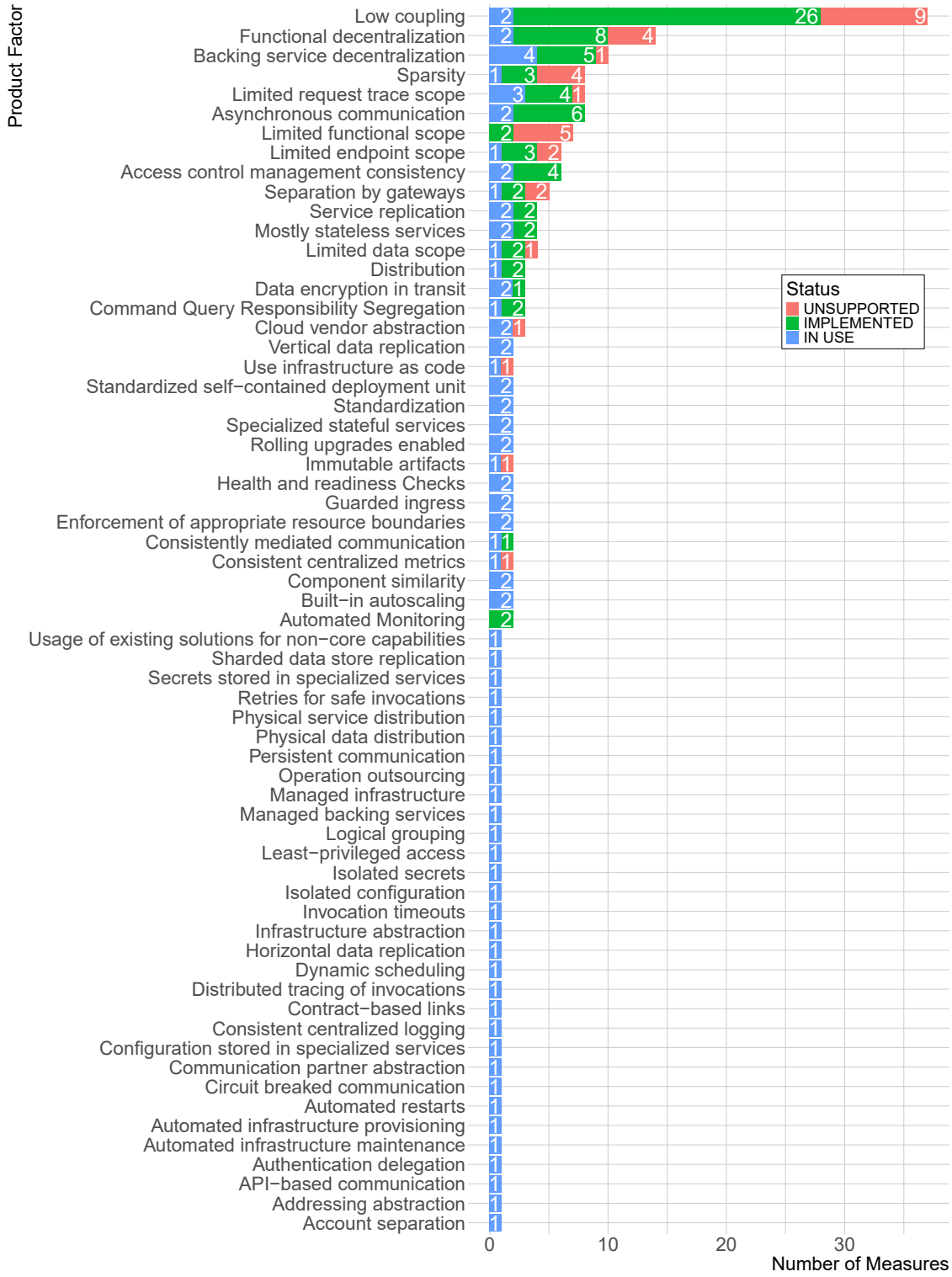


Figure 4.5.: Number of Measures per Product Factor, differentiated by status

The intention is to keep the evaluation understandable and therefore meaningful. The more measures are combined to evaluate a single product factor, the more complex an evaluation becomes, and thus less comprehensible.

Nevertheless, all measures which are supported by the quality model, because their values can be calculated with the available entities and their included information, are implemented (Status IMPLEMENTED) and calculated when a system is evaluated. For measures that are currently not supported (Status UNSUPPORTED), typically required information is not included in the entities. The decision on whether or not to include certain information in the entities was based on how central the information is and how relevant it is for various product factors and quality aspects. For completeness and because these measures are nevertheless relevant, they are kept and referenced in the quality model and may be implemented in a later version, if deemed necessary. One example is the measure **Concurrently available versions complexity** for which it would be necessary to include information about active and passive versions of components and which specific versions are used by other components.

Another perspective on the distribution of measures can be gained when assigning measures to the quality aspects which their product factors ultimately support to evaluate. To assign measures to quality aspects, the impacts starting from the product factor to which a measure is assigned are searched step by step until a quality aspect is found. A measure can therefore be assigned to more than one quality aspect. This perspective is shown in Figures 4.6 and 4.7

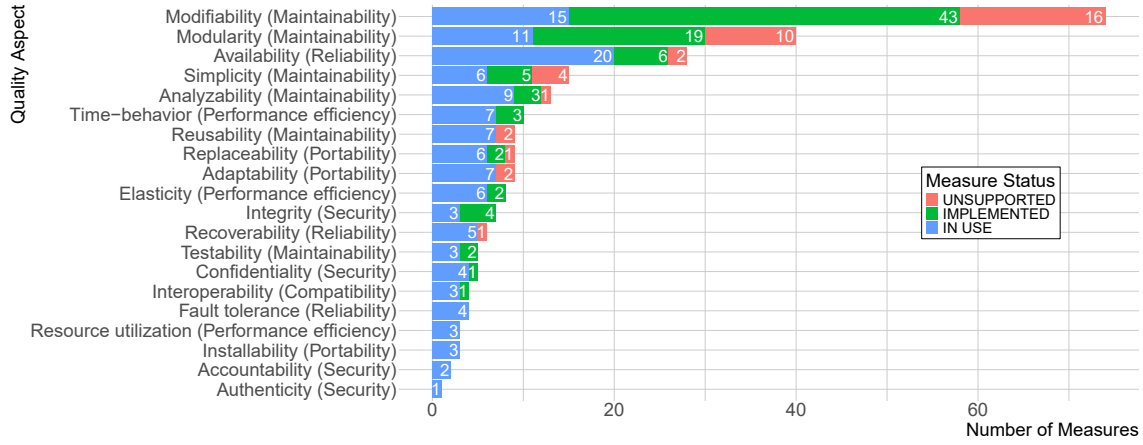


Figure 4.6.: Number of Measures per Quality Aspect, differentiated by status

As it can be seen, most measures, especially those found in literature, are assigned to quality aspects which are sub-aspects of the high-level aspect **maintainability**. Based on the literature search results, it can therefore be stated that **maintainability** is the most considered quality aspect for architectural measures. A focus in literature has more specifically been on cohesion and coupling as core software design characteristics to consider for developing maintainable software [36, 205]. Apart from that, **Availability** and **Time-behavior** are recognizable as having many measures assigned. This can be explained with the importance of these quality aspects as well as their intuitive understanding. Measures associated with sub-aspects of **security** are less numerous and had to be partly newly defined for the

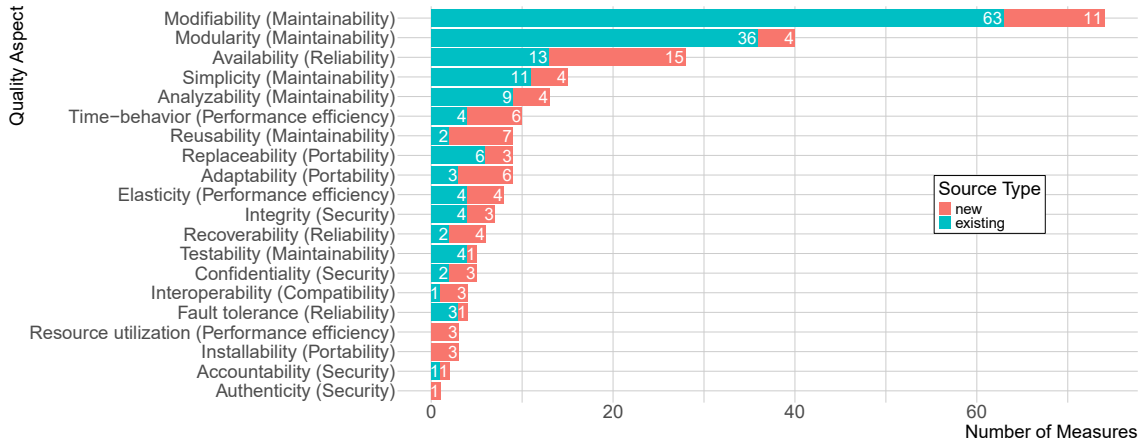


Figure 4.7.: Number of Measures per Quality Aspect, differentiated by source

related product factors. However, in more recent literature, also measures in this area have been proposed [207].

As a final aspect of how measures can be related to the other elements of the quality model, Figure 4.8 shows to which entities measures can be applied. It has to be noted that measures can potentially be calculated for more than one entity, if it is reasonable to aggregate it or apply it to a part of a **system** only. Aggregation can for example be applied to measures calculated on the level of individual **components** by averaging them across all **components** of a **system** (see **Ratio of documented endpoints** as an example). A partial application of a measure can be done for example by applying a system-level measure only to those components that are part of a certain request trace (see **Degree of asynchronous communication** as an example). As Figure 4.8 shows, most measures can be calculated on the level of a system as a whole. This is because they are either directly defined on that level or if not, it is often possible to aggregate measures on a system level.

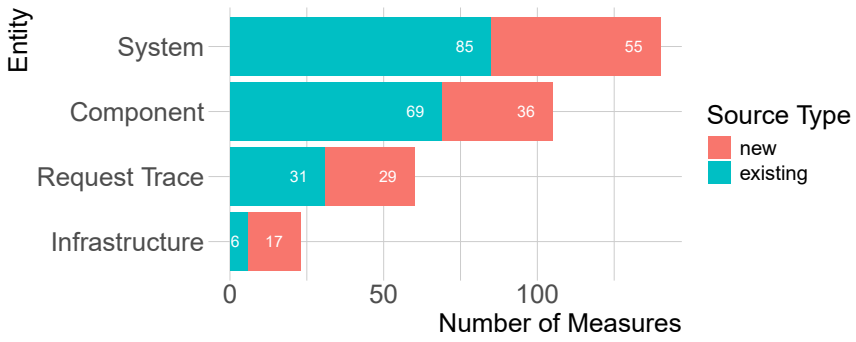


Figure 4.8.: Number of Measures per Entity, differentiated by status

In the current state of the quality model, measures are only implemented and calculated for these four entities: **system**, **component**, **infrastructure**, and **request**

trace. The reason is that during the literature search for measures, these were the main entities based on which measures are defined. Furthermore, enabling the implementation of measures for additional entities would increase the complexity of the approach and not all product factors are relatable to all individual entities. Even for the selected four entities not all product factors are applicable to all entities (for example, **Vertical data replication** cannot be evaluated based on an individual component). Because the selected four entities subsume the other entities, their choice is justified and still enables evaluations on different levels of granularity.

4.6.1.1. Architectural Measures from Literature

In the following, all measures which are currently used for evaluations in the approach are presented in a consistent format. All additional measures can be found in appendix B. Now, firstly, measures found through the literature search are presented and then, secondly, newly defined measures are presented.

For each measure, the specific name is given, and its status is reported in the top right corner. The formula for each measure is based on the entities (4.3) and their respective properties. The variables, especially the capitalized ones, correspond with their names to the different entity abbreviations. To make the formula definitions more concise, some measures with more complex formulas also include helper functions which are used within the formulas. Additionally, "Applicable Entities" captures for which entities a measure can be calculated. The measure formula has to be seen in this context and if, for example, a measure is applicable to a **system** and a **request trace** and it uses the set of components C, then in the case of a **request trace**, only components which are part of the **request trace** are considered while all components are considered when calculating a measure for the whole **system**. The "Associated Factor" is the product factor for which a measure can be used to evaluate it. "Associated Quality Aspects" lists all quality aspects which are at some point influenced by this measure, because an impact relation exists between the associated factor and a quality aspect. Finally, "Literature Sources" lists the source(s) from which a measure has been adapted. If only "new" is reported, then the measure was newly defined for the quality model and not considered in literature so far. The Tables 4.11 to 4.17 present all measures derived from literature.

Table 4.11.: Architectural measures from literature - 1

Degree of asynchronous communication IN USE

Formula:

$$\frac{\sum_{i=1}^{|C|} \frac{|\{ e \mid e \in c_i.providedEndpoints \wedge isAsync(e) \}|}{|\{ e \mid e \in c_i.providedEndpoints \}|}}{|C|}$$

Functions:

$isAsync$:$e \rightarrow (e.kind$="send event"$\vee e.kind$="subscribe")

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Asynchronous communication**
Associated Quality Aspects:	Literature Sources:
Modularity	[232]

Asynchronous communication utilization IN USE

Formula:

$$\frac{|\{ l \mid l \in L \wedge isAsync(l) \}|}{|L|}$$

Functions:

$isAsync$:$l \rightarrow (l.targetEndpoint.kind$="send event"$\vee l.targetEndpoint.kind$="subscribe")

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Asynchronous communication**
Associated Quality Aspects:	Literature Sources:
Modularity	[205]

Ratio of storage backend sharing IN USE

Formula:

$$\frac{\sum_{i=1}^{|SBS|} |\{ s' \mid s' \in S \wedge share(s, s', sbs_i) \}|}{|S| * |SBS|}$$

Functions:

$shares_a,s_b,sbs \rightarrow (\exists l_1,l_2 \subset L(l_1.sourceComponent=s_a \wedge l_2.sourceComponent=s_b \wedge l_1.targetEndpoint \in sbs.providedEndpoints \wedge l_2.targetEndpoint \in sbs.providedEndpoints))$

Applicable Entities:	Associated Factor:
Component	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[133]

Ratio of links with complex failover IN USE

Formula:

$$\frac{|\{ l \mid l \in L \wedge isSync(l.targetEndpoint) \wedge l.circuit\_breaker \neq \text{"none"} \}|}{|\{ l \mid l \in L \wedge isSync(l.targetEndpoint) \}|}$$

Functions:

$isSynce \rightarrow (e.kind$="query"$\vee e.kind$="command")

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Circuit breaked communication**
Associated Quality Aspects:	Literature Sources:
Fault tolerance	[12]

Table 4.12.: Architectural measures from literature - 2

Event sourcing utilization metric IN USE

Formula:

$$\frac{|eventBasedInteractions|}{|eventBasedInteractions| + |\{\ l \mid l \in L \wedge isSync(l)\ \}|}$$

Functions:

$eventBasedInteractions$={ (l_1,l_2) | $t(l_1)$=$e_1 \wedge t(l_2)$=$e_2 \wedge ofLog(e_1) \wedge ofLog(e_2) \wedge isEventBased(e_1,e_2)$ }

t:$l \rightarrow l.targetEndpoint$

$ofLog$:$e \rightarrow e \in bbs.providedEndpoints \wedge bbs \in BBS \wedge bbs.kind$="log"

$isEventBased$:$e_a,e_b \rightarrow e_a.kind$="send event" $\wedge e_b.kind$="subscribe" $\wedge e_a.url = e_b.url$

$isSync$:$l \rightarrow (l.targetEndpoint.kind$="query" $\vee l.targetEndpoint.kind$="command")

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Communication partner abstraction**
Associated Quality Aspects:	Literature Sources:
Analyzability, **Modularity**	[204], [206]

Ratio of components or infrastructure nodes that export logs to a central service IN USE

Formula:

$$\frac{|componentsExportingLogs| + |infrastructureExportingLogs|}{|C| + |I|}$$

Functions:

$linkedToLogger$:$c \rightarrow (\exists l \in L(l.sourceComponent = c \wedge l.targetEndpoint \in ls \wedge ls \in BS \wedge ls.kind$="logging" $) \vee \exists l \in L(l.sourceComponent = ls \wedge ls \in BS \wedge ls.kind$="logging" $\wedge l.targetEndpoint \in c.providedEndpoints))$

$sharesLogs$:$c \rightarrow (\exists bd \in BD(bd.kind$="logs" $\wedge \exists (c,bd) \in c.RBD_C \wedge \exists (ls,bd) \in ls.RBD_C (ls \in BS \wedge ls.kind$="logging" $)))$

$componentsExportingLogs$={ $c \mid c \in C \wedge linkedToLogger(c) \wedge sharesLogs(c)$ }

$infrastructureExportingLogs$={ $i \mid i \in I \wedge sharesLogs(i)$ }

Applicable Entities:	Associated Factor:
Component, **Infrastructure**, **System**	**Consistent centralized logging**
Associated Quality Aspects:	Literature Sources:
Accountability, **Analyzability**	[207]

Ratio of components or infrastructure nodes that export metrics IN USE

Formula:

$$\frac{|componentsExportingMetrics| + |infrastructureExportingMetrics|}{|C| + |I|}$$

Functions:

$linkedToMetrics$:$c \rightarrow (\exists l \in L(l.sourceComponent = c \wedge l.targetEndpoint \in ms \wedge ms \in BS \wedge ms.kind$="metrics" $) \vee \exists l \in L(l.sourceComponent = ms \wedge ms \in BS \wedge ms.kind$="metrics" $\wedge l.targetEndpoint \in c.providedEndpoints))$

$sharesMetrics$:$c \rightarrow (\exists bd \in BD(bd.kind$="metrics" $\wedge \exists (c,bd) \in c.RBD_C \wedge \exists (ms,bd) \in ms.RBD_C (ms \in BS \wedge ms.kind$="metrics" $)))$

$componentsExportingMetrics$={ $c \mid c \in C \wedge linkedToMetrics(c) \wedge sharesMetrics(c)$ }

$infrastructureExportingMetrics$={ $i \mid i \in I \wedge sharesMetrics(i)$ }

Applicable Entities:	Associated Factor:
Component, **Infrastructure**, **System**	**Consistent centralized metrics**
Associated Quality Aspects:	Literature Sources:
Analyzability	[207]

Table 4.13.: Architectural measures from literature - 3

Ratio of secured links IN USE

Formula:

$$\frac{|\{\, l \mid l \in L \wedge l.targetEndpoint.protocol \in SUPPORTS\_TLS \,\}|}{|L|}$$

Functions:

$SUPPORTS\_TLS$={"https","sftp"}

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Data encryption in transit**
Associated Quality Aspects:	Literature Sources:
Confidentiality	[315], [316]

Distributed tracing support IN USE

Formula:

$$\frac{|\{\, c \mid c \in C \wedge \neg(isTracing(c)) \wedge linkedToTracing(c) \,\}|}{|\{\, c \mid c \in C \wedge \neg(isTracing(c)) \,\}|}$$

Functions:

$isTracing$:$c \rightarrow (c \in BS \wedge c.kind$="tracing")

$linkedToTracing$:$c \rightarrow (\exists l \in L(l.sourceComponent = c \wedge l.targetEndpoint \in ts.providedEndpoints \wedge ts \in BS \wedge ts.kind$"tracing"))

Applicable Entities:	Associated Factor:
Component, **System**, **Request Trace**	**Distributed tracing of invocations**
Associated Quality Aspects:	Literature Sources:
Analyzability	[204], [206]

Number of availability zones used by infrastructure IN USE

Formula:

$$|\{\, az \mid \exists i \in I \wedge az \in i.availability\_zone) \,\}|$$

Applicable Entities:	Associated Factor:
System, **Infrastructure**, **Request Trace**	**Distribution**
Associated Quality Aspects:	Literature Sources:
Availability	[110], [23]

Ratio of components whose external ingress is proxied IN USE

Formula:

$$\frac{|\{\, c \mid c \in C \wedge \exists ee \in EE(ee \in c.providedEndpoints) \wedge c.externalIngressProxiedBy \neq \emptyset \,\}|}{|\{\, c \mid c \in C \wedge \exists ee \in EE(ee \in c.providedEndpoints) \,\}|}$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Guarded ingress**
Associated Quality Aspects:	Literature Sources:
Availability	[207]

Ratio of services that provide health endpoints IN USE

Formula:

$$\frac{|\{\, s \mid s \in S \wedge \exists e \in s.providedEndpoints(e.health\_check = true) \,\}|}{|S|}$$

Applicable Entities:	Associated Factor:
Component, **System**, **Request Trace**	**Health and readiness checks**
Associated Quality Aspects:	Literature Sources:
Availability, **Analyzability**, **Recoverability**	[207]

Table 4.14.: Architectural measures from literature - 4

Storage replication level IN USE

Formula:

$$\frac{\sum_{i=1}^{|DM|} dm_i.replicas | dm_i.deployed \in SBS}{|\{\ sbs \mid sbs \in SBS \wedge \exists dm \in DM(dm.deployed = sbs)\ \}|}$$

Applicable Entities:	Associated Factor:
System, **Request Trace**, **Component**	**Horizontal data replication**
Associated Quality Aspects:	Literature Sources:
Availability, **Time-behavior**	[110], [61]

Configuration externalization IN USE

Formula:

$$\frac{|\{\ bd \mid bd \in BD \wedge bd.kind = \text{"config"} \wedge externalized(bd)\ \}|}{|\{\ bd \mid bd \in BD \wedge bd.kind = \text{"config"} \wedge isUsed(bd)\ \}|}$$

Functions:

$externalized$:$bd \rightarrow (\exists (ci,bd) \in ci.RBD(ci \in C \cup I \wedge (ci,bd).usage\_relation = \text{"usage"}) \wedge \exists (ci',bd) \in ci'.RBD(ci' \in C \cup I \wedge (ci',bd).usage\_relation = \text{"persistence"}))$

$isUsed$:$bd \rightarrow (\exists (ci,bd) \in ci.RBD(ci \in C \cup I))$

Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**, **Request Trace**	**Isolated configuration**
Associated Quality Aspects:	Literature Sources:
Adaptability	[12]

Cohesion between endpoints based on data aggregate usage IN USE

Formula:

$$\frac{\sum_{i=1}^{|c.providedEndpoints|} |\{\ (e_i, e_n) \mid n > i \wedge ((e_i.RDA_E \cap e_n.RDA_E) \neq \emptyset)\ \}|}{\frac{|c.providedEndpoints|}{|\{\ da \mid \exists e \in c.providedEndpoints(\exists (e, da) \in e.RDA_E)\ \}|}}$$

Applicable Entities:	Associated Factor:
Component, **System**	**Limited data scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[223]

Service interface usage cohesion IN USE

Formula:

$$\frac{\sum_{i=1}^{|C|} |\{\ e \mid e \in c.providedEndpoints \wedge linked(c_i, e)\ \}|}{|\{\ c' \mid c' \in C \wedge connected(c', c)\ \}| \times |c.providedEndpoints|}$$

Functions:

$linked$:$c,e \rightarrow (\exists l \in L(l.sourceComponent = c \wedge l.targetEndpoint = e))$

$connected$:$c_a,c_b \rightarrow \exists l \in L(l.sourceComponent = c_a \wedge l.targetEndpoint \in c_b.providedEndpoints)$

Applicable Entities:	Associated Factor:
Component, **System**	**Limited endpoint scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[36], [224], [136]

Table 4.15.: Architectural measures from literature - 5

Maximum number of services within a request trace IN USE

Formula:

$$(|rt.nodes|)|\forall rt'(rt' \in RT \wedge rt' \neq rt \wedge rt'.nodes \leq rt.nodes$$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Limited request trace scope**
Associated Quality Aspects:	Literature Sources:
Modifiability	[12]

Average complexity of request traces IN USE

Formula:

$$\frac{\sum_{i=1}^{|RT|} \sum_{j=1}^{|rt.involvedLinks|} |rt.involvedLinks_j.links|}{|RT|}$$

Applicable Entities:	Associated Factor:
System	**Limited request trace scope**
Associated Quality Aspects:	Literature Sources:
Modifiability	[320]

Number of components a component is linked to relative to the total amount of components IN USE

Formula:

$$\frac{|\{ c' \mid c' \in C \wedge \exists l \in L(l.sourceComponent = c \wedge l.targetEndpoint \in c'.providedEndpoints) \}|}{|C| - 1}$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[238], [237], [317]

Degree of coupling in a system IN USE

Formula:

$$\frac{\sum_{i=1}^{|C|} |\{ c' \mid c' \in C \wedge \exists l \in L(l.sourceComponent = c_i \wedge l.targetEndpoint \in c'.providedEndpoints) \}|}{|C|^2 - |C|}$$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[238], [237], [115], [317]

Ratio of stateless components IN USE

Formula:

$$\frac{|\{ c \mid c \in C \wedge c.stateless = true \}|}{|C|}$$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Mostly stateless services**
Associated Quality Aspects:	Literature Sources:
Testability, **Modularity**, **Replaceability**, **Elasticity**	[114]

Table 4.16.: Architectural measures from literature - 6

Degree to which components are linked to stateful components IN USE

Formula:

$$\frac{\sum_{i=1}^{|C|} \frac{|\{ c' \mid c' \in C \wedge isLinkedTo(c_i, c') \wedge c'.stateless = false \}|}{|\{ c' \mid c' \in C \wedge isLinkedTo(c_i, c') \}|}}{|C|}$$

Functions:

$isLinkedTo{:}c_a,c_b \rightarrow (\exists l \in L(l.sourceComponent = c_a \wedge l.targetEndpoint \in c_b.providedEndpoints)$

Applicable Entities:	Associated Factor:
Component, **System**, **Request Trace**	**Mostly stateless services**
Associated Quality Aspects:	Literature Sources:
Testability, **Modularity**, **Replaceability**, **Elasticity**	[232]

Ratio of provider-managed components and infrastructure IN USE

Formula:

$$\frac{|\{ c \mid c \in C \wedge c.managed = true \}| + |\{ i \mid i \in I \wedge isManaged(i) \}|}{|C| + |I|}$$

Functions:

$isManaged{:}i \rightarrow (i.environment\_access = \text{"limited"} \vee i.environment\_access = \text{"none"})$

Applicable Entities:	Associated Factor:
System	**Operation outsourcing**
Associated Quality Aspects:	Literature Sources:
Simplicity	[313]

Service interaction via backing service IN USE

Formula:

$$\frac{connectionsViaBroker}{connectionsViaBroker + |\{ l \mid l \in L \wedge serviceInteration(l) \wedge isSync(l.targetEndpoint) \}|}$$

Functions:

$connectionsViaBroker = |\{ (l_1,l_2) \mid l_1,l_2 \in L \wedge viaBackingService(l_1,l_2) \}|$

$viaBackingService{:}l_a,l_b \rightarrow l_a.sourceComponent \in S \wedge l_a.targetEndpoint \in bbs.providedEndpoints \wedge bbs \in BBS \wedge (bbs.kind = \text{"queue"} \vee bbs.kind = \text{"topic"}) \wedge l_a.targetEndpoint.kind = \text{"send event"} \wedge l_b.sourceComponent \in S \wedge l_b.targetEndpoint \in bbs.providedEndpoints \wedge l_b.targetEndpoint.kind = \text{"subscribe"} \wedge l_a.targetEndpoint.url\_path = l_b.targetEndpoint.url\_path$

$serviceInteration{:}l \rightarrow (l.sourceComponent \in S \wedge l.targetEndpoint \in s'.providedEndpoints \wedge s' \in S)$

$isSync{:}e \rightarrow (e.kind = \text{"query"} \vee e.kind = \text{"command"})$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Persistent communication**
Associated Quality Aspects:	Literature Sources:
Fault tolerance	[205], [204], [206]

Ratio of links with retry logic IN USE

Formula:

$$\frac{|\{ l \mid l \in L \wedge isSync(l.targetEndpoint) \wedge l.retries > 0 \}|}{|\{ l \mid l \in L \wedge isSync(l.targetEndpoint) \}|}$$

Functions:

$isSync{:}e \rightarrow (e.kind = \text{"query"} \vee e.kind = \text{"command"})$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Retries for safe invocations**
Associated Quality Aspects:	Literature Sources:
Fault tolerance	[12]

Table 4.17.: Architectural measures from literature - 7

Rolling update option	IN USE
Formula: $\frac{\lvert \{ i \mid i \in I \wedge deploysComponent(i) \wedge supportsRollingUpdate(i) \} \rvert}{\lvert \{ i \mid i \in I \wedge deploysComponent(i) \} \rvert}$	
Functions: $deploysComponent{:}i \rightarrow (\exists dm \in DM(dm.host = i \wedge dm.deployed \in C))$ $supportsRollingUpdate{:}i \rightarrow (\text{"rolling"} \in i.supported\_update\_strategies \vee \text{"blue-green"} \in i.supported\_update\_strategies)$	
Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Rolling upgrades enabled**
Associated Quality Aspects:	Literature Sources:
Availability	[271]

Amount of redundancy	IN USE
Formula: $\frac{\sum_{i=1}^{\lvert C \rvert} \lvert \{ dm \mid dm \in DM \wedge dm.deployed = c_i \} \rvert}{\lvert \{ c \mid c \in C \wedge \exists dm \in DM(dm.deployed = c) \} \rvert}$	
Applicable Entities:	Associated Factor:
System, **Request Trace**, **Component**	**Service replication**
Associated Quality Aspects:	Literature Sources:
Availability, **Time-behavior**	[320]

Service replication level	IN USE
Formula: $\frac{\sum_{i=1}^{\lvert DM \rvert} dm_i.replicas \lvert dm_i.deployed \in S}{\lvert \{ s \mid s \in S \wedge \exists dm \in DM(dm.deployed = s) \} \rvert}$	
Applicable Entities:	Associated Factor:
System, **Request Trace**, **Component**	**Service replication**
Associated Quality Aspects:	Literature Sources:
Availability, **Time-behavior**	[110], [61]

Number of components	IN USE
Formula: $\lvert C \rvert$	
Applicable Entities:	Associated Factor:
System	**Sparsity**
Associated Quality Aspects:	Literature Sources:
Simplicity	[262], [291], [259], [320]

4.6.1.2. Newly defined Architectural Measures

New measures had to be formulated for all product factors for which no measures were found in literature and for all product factors where no suitable or implementable measures had been found. To formulate suitable measures, the product factor description was evaluated thoroughly to derive the main aspects of an architecture which need to be analyzed for determining the extent to which a product factor is present (see also Section 2.2.3.2). These main aspects were then compared with the available entities and the information available through their properties. Based on that, one or more measures were formulated that capture these aspects. Furthermore, the recommendations by Becker et al. [27] and respectively Erl et al. [80] were followed: According to them, measures need to be *quantifiable*, *repeatable*, *comparable*, and *easily obtainable*. Quantifiability, repeatability, and easy obtainability are achieved through relying only on the entities of the quality model and their respective properties. Because modeled architectures can be saved and do not change, unless explicitly modified, measures can be calculated repeatedly. Comparability is somewhat more difficult to achieve, but can be ensured by limiting measure values to a certain value range, preferably values between 0 and 1. Through this, measure values are better comparable when considering different software architectures. The Tables 4.18 to 4.30 present all measures which were newly defined for the quality model. It has to be noted that the measure formulation process and the process of specifying properties for the entities were done iteratively, extending the properties of entities where necessary to formulate a certain measure.

Table 4.18.: Architectural measures, newly formulated - 1

Consistency of supported authentication methods of external endpoints — IN USE

Formula:

$$\frac{\sum_{i=1}^{|EE|}\sum_{j=i+1}^{|EE|}\begin{cases}0 & \text{if } none(ee_i, ee_j)\\ similarity(ee_i, ee_j) & \text{else}\end{cases}}{\frac{|EE| * (|EE|-1)}{2}}$$

Functions:

$none{:}ee_a,ee_b \rightarrow (|ee_a.supported\_authentication\_methods \cup ee_b.supported\_authentication\_methods| = 0)$

$similarity{:}ee_a,ee_b \rightarrow (\frac{|ee_a.supported\_authentication\_methods \cap ee_b.supported\_authentication\_methods|}{|ee_a.supported\_authentication\_methods \cup ee_b.supported\_authentication\_methods|})$

Applicable Entities:	Associated Factor:
System, **Component**	**Access control management consistency**
Associated Quality Aspects:	Literature Sources:
Integrity	new

Ratio of unique account usage — IN USE

Formula:

$$\frac{|\{\, account \mid \exists ci \in C \cup I(account \in ci.identities)\,\}|}{|C| + |I|}$$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Account separation**
Associated Quality Aspects:	Literature Sources:
Accountability	new

Ratio of documented endpoints — IN USE

Formula:

$$\frac{|\{\, e \mid e \in E \wedge e.documentedBy \neq \emptyset\,\}|}{|E|}$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**API-based communication**
Associated Quality Aspects:	Literature Sources:
Interoperability, **Testability**	new

Ratio of delegated authentication — IN USE

Formula:

$$\frac{|\{\, c \mid c \in C \wedge notAuthService(c) \wedge c.authenticationBy \neq \emptyset\,\}|}{|\{\, c \mid c \in C \wedge notAuthService(c)\,\}|}$$

Functions:

$notAuthService{:}c \rightarrow (c \notin BS \vee c.providedFunctionality \neq \text{"}authentication/authorization\text{"})$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Authentication delegation**
Associated Quality Aspects:	Literature Sources:
Authenticity	new

Table 4.19.: Architectural measures, newly formulated - 2

Ratio of automatically maintained infrastructure IN USE

Formula:

$$\frac{|\{\, i \mid i \in I \wedge (i.maintenance = \text{"automated"} \vee i.maintenance = \text{"transparent"})\,\}|}{|I|}$$

Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Automated infrastructure maintenance**
Associated Quality Aspects:	Literature Sources:
Availability, **Recoverability**	new

Ratio of automatically provisioned infrastructure IN USE

Formula:

$$\frac{|\{\, i \mid i \in I \wedge isAutomated(i.provisioning)\,\}|}{|I|}$$

Functions:

$isAutomated{:}provisioning \rightarrow (provisioning = \text{"automated-coded"} \vee provisioning = \text{"automated-inferred"} \vee provisioning = \text{"transparent"})$

Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Automated infrastructure provisioning**
Associated Quality Aspects:	Literature Sources:
Modifiability, **Installability**	new

Deployments with restart IN USE

Formula:

$$\frac{|\{\, dm \mid dm \in DM \wedge automatedRestart(dm)\,\}|}{|DM|}$$

Functions:

$automatedRestart{:}dm \rightarrow (dm.automated\_restart\_policy = \text{"onProcessFailure"} \vee dm.automated\_restart\_policy = \text{"onHealthFailure"})$

Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**	**Automated restarts**
Associated Quality Aspects:	Literature Sources:
Recoverability	new

Ratio of broker backend sharing IN USE

Formula:

$$\frac{\sum_{i=1}^{|BBS|} |\{\, s' \mid s' \in S \wedge share(s, s', bbs)\,\}|}{|S| * |BBS|}$$

Functions:

$shares_a, s_b, bbs \rightarrow (\exists l_1, l_2 \subset L(l_1.sourceComponent = s_a \wedge l_2.sourceComponent = s_b \wedge l_1.targetEndpoint \in bbs.providedEndpoints \wedge l_2.targetEndpoint \in bbs.providedEndpoints))$

Applicable Entities:	Associated Factor:
Component	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Table 4.20.: Architectural measures, newly formulated - 3

Average weighted broker backend sharing IN USE

Formula:

$$\frac{\sum_{i=1}^{|BBS|} \frac{\sum_{j=1}^{|S|} \begin{cases} 1/n & \text{if } transitiveUse(s_j, bbs_i) \\ 0 & \text{else} \end{cases}}{|S|}}{|BBS|}$$

Functions:

$transitiveUse{:}s,bbs \rightarrow (\exists(l_1,...,l_n) \subset L(l_1.sourceComponent = s \wedge l_n.targetEndpoint \in bbs.providedEndpoints))$

Applicable Entities:	Associated Factor:
System	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Average weighted storage backend sharing IN USE

Formula:

$$\frac{\sum_{i=1}^{|SBS|} \frac{\sum_{j=1}^{|S|} \begin{cases} 1/n & \text{if } transitiveUse(s_j, sbs_i) \\ 0 & \text{else} \end{cases}}{|S|}}{|SBS|}$$

Functions:

$transitiveUse{:}s,sbs \rightarrow (\exists(l_1,...,l_n) \subset L(l_1.sourceComponent = s \wedge l_n.targetEndpoint \in sbs.providedEndpoints))$

Applicable Entities:	Associated Factor:
System	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Deployed entities autoscaling IN USE

Formula:

$$\frac{|\{ i \mid i \in I \wedge \exists dm \in DM(dm.host = i \wedge dm.deployed \in C) \wedge autoscale(i) \}|}{|\{ i \mid i \in I \wedge \exists dm \in DM(dm.host = i \wedge dm.deployed \in C) \}|}$$

Functions:

$autoscalei \rightarrow (i.deployed\_entities\_scaling = "automated-built-in" \vee i.deployed\_entities\_scaling = "automated-separate")$

Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**	**Built-in autoscaling**
Associated Quality Aspects:	Literature Sources:
Availability, **Elasticity**	new

Infrastructure autoscaling IN USE

Formula:

$$\frac{|\{ i \mid i \in I \wedge selfscale(i) \}|}{|I|}$$

Functions:

$selfscalei \rightarrow (i.self\_scaling = "automated-built-in" \vee i.self\_scaling = "automated-separate")$

Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Built-in autoscaling**
Associated Quality Aspects:	Literature Sources:
Availability, **Elasticity**	new

Table 4.21.: Architectural measures, newly formulated - 4

Non-provider-specific infrastructure artifacts IN USE

Formula:

$$\frac{|\{\, i \mid i \in I \wedge \forall a \in i.artifacts(a.provider\_specific = false) \,\}|}{|I|}$$

Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Cloud vendor abstraction**
Associated Quality Aspects:	Literature Sources:
Adaptability, **Reusability**	new

Non-provider-specific component artifacts IN USE

Formula:

$$\frac{|\{\, c \mid c \in C \wedge \forall a \in c.artifacts(a.provider\_specific = false) \,\}|}{|C|}$$

Applicable Entities:	Associated Factor:
System, **Component**	**Cloud vendor abstraction**
Associated Quality Aspects:	Literature Sources:
Adaptability, **Reusability**	new

Read write separation for data aggregates IN USE

Formula:

$$\frac{\sum_{i=1}^{|DA|} \begin{cases} 1 & \text{if } separated(da_i) \\ 0 & else \end{cases}}{|DA|}$$

Functions:

$persists$:$c,da \rightarrow (\exists (c,da) \in c.RDA_C((c,da).usage_relation=$"persistence"))

$separatedRead$:$da \rightarrow (\exists c_1,c_2 \in C(c_1 \neq c_2 \wedge persists(c_1,da) \wedge persists(c_2,da) \wedge \exists (e_1,da)(e_1 \in c_1.providedEndpoints \wedge \exists (e_1,da) \in e_1.RDA_E \wedge e_1.kind=$"query"$\wedge \nexists (e_2,da)(e_2 \in c_1.providedEndpoints \wedge (e_2,da) \in e_2.RDA_E \wedge e_2.kind=$"command"$)) \wedge \exists (e_3,da)(e_3 \in c_2.providedEndpoints \wedge \exists (e_3,da) \in e_3.RDA_E \wedge e_3.kind=$"command"$)))$

$separatedWrite$:$da \rightarrow (\exists c_1,c_2 \in C(c_1 \neq c_2 \wedge persists(c_1,da) \wedge persists(c_2,da) \wedge \exists (e_1,da)(e_1 \in c_1.providedEndpoints \wedge \exists (e_1,da) \in e_1.RDA_E \wedge e_1.kind=$"command"$\wedge \nexists (e_2,da)(e_2 \in c_1.providedEndpoints \wedge (e_2,da) \in e_2.RDA_E \wedge e_2.kind=$"query"$)) \wedge \exists (e_3,da)(e_3 \in c_2.providedEndpoints \wedge \exists (e_3,da) \in e_3.RDA_E \wedge e_3.kind=$"query"$)))$

$separated$:$da \rightarrow (separatedRead(da) \vee separatedWrite(da))$

Applicable Entities:	Associated Factor:
System, **Component**	**Command query responsibility segregation**
Associated Quality Aspects:	Literature Sources:
Simplicity, **Modularity**	new

Component artifacts similarity IN USE

Formula:

$$\frac{\sum_{i=1}^{|C|} \sum_{j=i+1}^{|C|} \begin{cases} 0 & \text{if } |artifactTypes(c_i) \cup artifactTypes(c_j)| = 0 \\ \frac{|artifactTypes(c_i) \cap artifactTypes(c_j)|}{|artifactTypes(c_i) \cup artifactTypes(c_j)|} & \text{else} \end{cases}}{\frac{|C| * (|C|-1)}{2}}$$

Functions:

$artifactTypes$:$c \rightarrow \{\, t \mid t=a.type \wedge a \in c.artifacts \,\}$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Component similarity**
Associated Quality Aspects:	Literature Sources:
Reusability	new

Table 4.22.: Architectural measures, newly formulated - 5

Infrastructure artifacts similarity IN USE

Formula:

$$\frac{\sum_{k=1}^{|I|}\sum_{l=k+1}^{|I|}\begin{cases}0 & \text{if } |artifactTypes(i_k) \cup artifactTypes(i_l)| = 0\\ \frac{|artifactTypes(i_k)\cap artifactTypes(i_l)|}{|artifactTypes(i_k)\cup artifactTypes(i_l)|} & \text{else}\end{cases}}{\frac{|I| * (|I|-1)}{2}}$$

Functions:

$artifactTypes{:}i\rightarrow\{\ t \mid t{=}a.type\wedge a{\in}i.artifacts\ \}$

Applicable Entities:	Associated Factor:
System	**Component similarity**
Associated Quality Aspects:	Literature Sources:
Reusability	new

Configuration stored in config service IN USE

Formula:

$$\frac{|\{\ bd \mid bd \in BD \wedge bd.kind = \text{"config"} \wedge storedInConfigService(bd)\ \}|}{|\{\ bd \mid bd \in BD \wedge bd.kind = \text{"config"}\ \}|}$$

Functions:

$storedInConfigService{:}bd\rightarrow(\exists bs{\in}BS(bs.providedFunctionality{=}\text{"configuration"})\wedge\exists(bs,bd){\in}$
$bs.RBD_C((bs,bd).usage\_relation{=}\text{"persistence"}))$

Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**	**Configuration stored in specialized services**
Associated Quality Aspects:	Literature Sources:
Adaptability	new

Service mesh usage IN USE

Formula:

$$\frac{\sum_{i=1}^{|C|}\begin{cases}0.5 & \text{if } iProxiedByMesh(c_i)\\ 0 & \text{else}\end{cases}+\begin{cases}0.5 & \text{if } eProxiedByMesh(c_i)\\ 0 & \text{else}\end{cases}}{|C|}$$

Functions:

$iProxiedByMesh{:}c\rightarrow(c.ingressProxiedBy{=}p\wedge p{\in}PBS\wedge p.kind{=}\text{"Service Mesh"})$

$eProxiedByMesh{:}c\rightarrow(c.egressProxiedBy{=}p\wedge p{\in}PBS\wedge p.kind{=}\text{"Service Mesh"})$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Consistently mediated communication**
Associated Quality Aspects:	Literature Sources:
Interoperability, **Time-behavior**, **Analyzability**	new

Ratio of endpoints covered by a contract IN USE

Formula:

$$\frac{|\{\ e \mid e \in E \wedge e.documentedBy \neq \emptyset \wedge isContract(e.documentedBy)\ \}|}{|E|}$$

Functions:

$isContract{:}a\rightarrow(a.type{=}\text{"}Spring.CloudContract\text{"}\vee a.type{=}\text{"}Pact.Contract\text{"})$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Contract-based links**
Associated Quality Aspects:	Literature Sources:
Interoperability, **Adaptability**	new

Table 4.23.: Architectural measures, newly formulated - 6

Ratio of external endpoints that support TLS	IN USE
Formula: $\frac{\lvert \{ ee \mid ee \in EE \wedge ee.protocol \in SUPPORTS\_TLS \} \rvert}{\lvert EE \rvert}$	
Functions: $SUPPORTS\_TLS$={"https","sftp"}	
Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Data encryption in transit**
Associated Quality Aspects:	Literature Sources:
Confidentiality	new

Ratio of components deployed dynamically	IN USE
Formula: $\frac{\lvert \{ dm \mid dm \in DM \wedge dm.deployed \in C \wedge isDynamic(dm.host) \} \rvert}{\lvert \{ dm \mid dm \in DM \wedge dm.deployed \in C \} \rvert}$	
Functions: $isDynamic_i \rightarrow (i.kind$="software-platform"$\vee i.kind$="cloud-service")	
Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Dynamic scheduling**
Associated Quality Aspects:	Literature Sources:
Resource utilization	new

Ratio of infrastructure entities enforcing resource boundaries	IN USE
Formula: $\frac{\lvert \{ i \mid i \in I \wedge t.enforced\_resource\_bounds = true \} \rvert}{\lvert I \rvert}$	
Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Enforcement of appropriate resource boundaries**
Associated Quality Aspects:	Literature Sources:
Resource utilization, **Availability**	new

Ratio of deployment mappings with stated resource requirements	IN USE
Formula: $\frac{\lvert \{ dm \mid dm \in DM \wedge t.resource\_requirements \neq$ "unstated" $\} \rvert}{\lvert DM \rvert}$	
Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**	**Enforcement of appropriate resource boundaries**
Associated Quality Aspects:	Literature Sources:
Resource utilization, **Availability**	new

Table 4.24.: Architectural measures, newly formulated - 7

Data aggregate spread IN USE

Formula:

$$\frac{\sum_{i=1}^{|DA|} \frac{\sum_{j=1}^{|S|} \begin{cases} 1 & \text{if } \exists (c_j, da_i) \in c_j.RDA_C \\ 0 & \text{else} \end{cases}}{|S|}}{|DA|}$$

Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Request trace similarity based on included components IN USE

Formula:

$$\frac{\sum_{i=1}^{|RT|} \sum_{j=i+1}^{|RT|} \frac{|rt_i.nodes \cap rt_j.nodes|}{|rt_i.nodes \cup rt_j.nodes|}}{|RT| * (|RT| - 1)}$$

Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Ratio of rate limiting endpoints IN USE

Formula:

$$\frac{|\{ e \mid e \in E \wedge e.rate\_limiting \neq \text{"none"} \}|}{|E|}$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Guarded ingress**
Associated Quality Aspects:	Literature Sources:
Availability	new

Ratio of services that provide readiness endpoints IN USE

Formula:

$$\frac{|\{ s \mid s \in S \wedge \exists e \in s.providedEndpoints(e.readiness\_check = true) \}|}{|S|}$$

Applicable Entities:	Associated Factor:
Component, **System**, **Request Trace**	**Health and readiness checks**
Associated Quality Aspects:	Literature Sources:
Availability, **Analyzability**, **Recoverability**	new

Table 4.25.: Architectural measures, newly formulated - 8

Replacing deployments	IN USE
Formula: $\frac{\lvert \{ dm \mid dm \in DM \wedge dm.update\_strategy \neq "in-place" \} \rvert}{\lvert DM \rvert}$	
Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**	**Immutable artifacts**
Associated Quality Aspects:	Literature Sources:
Replaceability	new
Ratio of abstracted hardware	IN USE
Formula: $\frac{\lvert \{ i \mid i \in I \wedge (i.kind = "software-platform" \vee i.kind = "cloud-service") \} \rvert}{\lvert I \rvert}$	
Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Infrastructure abstraction**
Associated Quality Aspects:	Literature Sources:
Adaptability	new
Ratio of links with timeout	IN USE
Formula: $\frac{\lvert \{ l \mid l \in L \wedge l.timeout > 0 \} \rvert}{\lvert L \rvert}$	
Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Invocation timeouts**
Associated Quality Aspects:	Literature Sources:
Fault tolerance	new
Secrets externalization	IN USE
Formula: $\frac{\lvert \{ bd \mid bd \in BD \wedge bd.kind = "\text{secret}" \wedge externalized(bd) \} \rvert}{\lvert \{ bd \mid bd \in BD \wedge bd.kind = "\text{secret}" \wedge \exists (ci, bd) \in ci.RBD(ci \in C \cup I) \} \rvert}$	
Functions: $externalized{:}bd \rightarrow (\exists (ci,bd) \in ci.RBD(ci \in C \cup I \wedge (ci,bd).usage\_relation = "\text{usage}") \wedge \exists (ci',bd) \in ci'.RBD(ci' \in C \cup I \wedge (ci',bd).usage\_relation = "\text{persistence}"))$	
Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**, **Request Trace**	**Isolated secrets**
Associated Quality Aspects:	Literature Sources:
Confidentiality	new

Table 4.26.: Architectural measures, newly formulated - 9

Access restricted to callers IN USE

Formula:

$$\frac{\sum_{i=1}^{|E|} 1 - \frac{e_i.allow\_acces\_to \setminus \{\, acc \mid acc \in e_i.allow\_acces\_to \wedge noCall(acc, e_i) \,\}}{|e_i.allow\_acces\_to|}}{|E|}$$

Functions:

$noCall{:}acc,e \rightarrow (\nexists l \in L(acc \in l.sourceComponent.identities \wedge l.targetEndpoint = e))$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Least-privileged access**
Associated Quality Aspects:	Literature Sources:
Integrity	new

Request trace complexity IN USE

Formula:

$$\sum_{i=1}^{|rt.involvedLinks|} |rt.involvedLinks_i.links|$$

Applicable Entities:	Associated Factor:
Request Trace	**Limited request trace scope**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Namespace separation IN USE

Formula:

$$\frac{\sum_{i=1}^{|C|} 1 - \frac{\sum_{j=1}^{|C \setminus c_i|} \begin{cases} 1 & \text{if } c_i.namespace = c_j.namespace \\ 0 & \text{else} \end{cases}}{|C \setminus c_i|}}{|C|}$$

Applicable Entities:	Associated Factor:
System, **Component**	**Logical grouping**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Ratio of managed backing services IN USE

Formula:

$$\frac{|\,\{\, bs \mid bs \in BS \cup SBS \cup PBS \cup BBS \wedge bs.managed = true) \,\}\,|}{|\,\{\, bs \mid bs \in BS \cup SBS \cup PBS \cup BBS \,\}\,|}$$

Applicable Entities:	Associated Factor:
System, **Component**	**Managed backing services**
Associated Quality Aspects:	Literature Sources:
Simplicity	new

Table 4.27.: Architectural measures, newly formulated - 10

Ratio of fully managed infrastructure IN USE

Formula:

$$\frac{|\{\, i \mid i \in I \wedge fullyManaged(i) \,\}|}{|I|}$$

Functions:

$fullyManaged$:$i \rightarrow ((i.environment\_access$="none"$\vee i.environment\_access$="limited"$) \wedge i.maintenance$="transparent"$)$

Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Managed infrastructure**
Associated Quality Aspects:	Literature Sources:
Simplicity	new

Number of availability zones used by storage services IN USE

Formula:

$$|\{\, az \mid \exists i \in I(\exists dm \in DM(dm.host = i \wedge dm.deployed \in SBS) \wedge az \in i.availability\_zone) \,\}|$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Physical data distribution**
Associated Quality Aspects:	Literature Sources:
Availability	new

Number of availability zones used by services IN USE

Formula:

$$|\{\, az \mid \exists i \in I(\exists dm \in DM(dm.host = i \wedge dm.deployed \in S) \wedge az \in i.availability\_zone) \,\}|$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Physical service distribution**
Associated Quality Aspects:	Literature Sources:
Availability	new

Rolling updates IN USE

Formula:

$$\frac{|\{\, dm \mid dm \in DM \wedge (dm.update\_strategy = \text{"rolling"} \vee dm.update\_strategy = "blue-green") \,\}|}{|DM|}$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Rolling upgrades enabled**
Associated Quality Aspects:	Literature Sources:
Availability	new

Secrets stored in vault IN USE

Formula:

$$\frac{|\{\, bd \mid bd \in BD \wedge bd.kind = \text{"secret"} \wedge storedOnlyInVault(bd) \,\}|}{|\{\, bd \mid bd \in BD \wedge bd.kind = \text{"secret"} \,\}|}$$

Functions:

$storedOnlyInVault$:$bd \rightarrow (\exists vault \in BS(vault.kind$="vault"$\wedge \exists (vault,bd) \in vault.RBD_C((vault,bd).usage\_relation=$"persistence"$)) \wedge \nexists (c,bd)((c,bd) \in c.RBD_C \wedge (c \notin BS \vee c.kind \neq$"vault"$) \wedge (c,bd).usage\_relation=$"persistence"$)))$

Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**	**Secrets stored in specialized services**
Associated Quality Aspects:	Literature Sources:
Confidentiality	new

Table 4.28.: Architectural measures, newly formulated - 11

Degree of separation by gateways IN USE

Formula:

$$\frac{1}{\frac{|\{\, s \mid s \in S \wedge s.externalIngressProxiedBy = g \wedge g \in PBS \wedge g.kind = \text{"API Gateway"} \,\}|}{|\{\, g \mid g \in PBS \wedge \wedge g.kind = \text{"API Gateway"} \,\}|}}$$

Applicable Entities:	Associated Factor:
Component, **System**, **Request Trace**	**Separation by gateways**
Associated Quality Aspects:	Literature Sources:
Modularity, **Availability**	new

Level of sharding across storage backing services IN USE

Formula:

$$\frac{\sum_{i=1}^{|SBS|} sbs_i.shards}{|SBS|}$$

Applicable Entities:	Associated Factor:
Component, **System**, **Request Trace**	**Sharded data store replication**
Associated Quality Aspects:	Literature Sources:
Time-behavior	new

Ratio of specialized stateful services IN USE

Formula:

$$\frac{|\{\, bs \mid bs \in BS \cup SBS \cup BBS \wedge bs.stateless = false \,\}|}{|\{\, c \mid c \in C \wedge c.stateless = false \,\}|}$$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Specialized stateful services**
Associated Quality Aspects:	Literature Sources:
Modularity, **Replaceability**, **Elasticity**	new

Ratio of suitably replicated stateful services IN USE

Formula:

$$\frac{|\{\, bs \mid bs \in BS \cup SBS \cup BBS \wedge bs.stateless = false \wedge suitablyReplicated(bs) \,\}|}{|\{\, bs \mid bs \in BS \cup SBS \cup BBS \wedge bs.stateless = false \wedge replicated(bs) \,\}|}$$

Functions:

$suitablyReplicated bs \rightarrow (\exists dm \in DM(dm.deployed = bs \wedge dm.replicas > 1) \wedge bs.replication\_strategy \neq \text{"none"})$

$replicated: bs \rightarrow (\exists dm \in DM(dm.deployed = bs \wedge dm.replicas > 1))$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Specialized stateful services**
Associated Quality Aspects:	Literature Sources:
Modularity, **Replaceability**, **Elasticity**	new

Table 4.29.: Architectural measures, newly formulated - 12

Ratio of standardized artifacts	IN USE
Formula: $\frac{\lvert \{ a \mid a \in A \wedge a.based\_on\_standard = true \} \rvert}{\lvert \{ a \mid a \in A \} \rvert}$	
Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**, **Request Trace**	**Standardization**
Associated Quality Aspects:	Literature Sources:
Reusability	new
Ratio of entities providing standardized artifacts	IN USE
Formula: $\frac{\lvert \{ ci \mid ci \in C \cup I \wedge \exists a \in ci.artifacts(a.based\_on\_standard = true) \} \rvert}{\lvert C \rvert + \lvert I \rvert}$	
Applicable Entities:	Associated Factor:
System, **Component**, **Infrastructure**, **Request Trace**	**Standardization**
Associated Quality Aspects:	Literature Sources:
Reusability	new
Standardized deployments	IN USE
Formula: $\frac{\lvert \{ dm \mid dm \in DM \wedge dm.deployed \in C \wedge dm.deployment\_unit.based\_on\_standard = true \} \rvert}{\lvert \{ dm \mid dm \in DM \wedge dm.deployed \in C \} \rvert}$	
Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Standardized self-contained deployment unit**
Associated Quality Aspects:	Literature Sources:
Installability	new
Self-contained deployments	IN USE
Formula: $\frac{\lvert \{ dm \mid dm \in DM \wedge dm.deployed \in C \wedge dm.deployment\_unit.self\_contained = true \} \rvert}{\lvert \{ dm \mid dm \in DM \wedge dm.deployed \in C \} \rvert}$	
Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Standardized self-contained deployment unit**
Associated Quality Aspects:	Literature Sources:
Installability	new
Ratio of non-custom backing services	IN USE
Formula: $\frac{\lvert \{ bs \mid bs \in BS \cup SBS \cup PBS \cup BBS \wedge bs.software\_type \neq \text{"custom"} \} \rvert}{\lvert \{ bs \mid bs \in BS \cup SBS \cup PBS \cup BBS \} \rvert}$	
Applicable Entities:	Associated Factor:
System, **Component**	**Usage of existing solutions for non-core capabilities**
Associated Quality Aspects:	Literature Sources:
Simplicity	new

Table 4.30.: Architectural measures, newly formulated - 13

Ratio of infrastructure with IaC artifact IN USE

Formula:

$$\frac{|\{\, i \mid i \in I \wedge \exists a \in i.artifacts(isIaC(a)) \,\}|}{|I|}$$

Applicable Entities:	Associated Factor:
System, **Infrastructure**	**Use infrastructure as code**
Associated Quality Aspects:	Literature Sources:
Modifiability, **Adaptability**, **Reusability**, **Recoverability**	new

Ratio of cached data aggregates IN USE

Formula:

$$\frac{\sum_{i=1}^{|C|} \sum_{j=1}^{|c_i.RDA_C|} \begin{cases} 1 & \text{if } c_i.rda_j.usage\_relation = \text{"}cached\text{-}usage\text{"} \\ 0 & \text{else} \end{cases}}{\sum_{i=1}^{|C|} \sum_{j=1}^{|c_i.RDA_C|} \begin{cases} 1 & \text{if } c_i.rda_j.usage\_relation = \text{"}cached\text{-}usage\text{"} \vee c_i.rda_j.usage\_relation = \text{"usage"} \\ 0 & \text{else} \end{cases}}$$

Applicable Entities:	Associated Factor:
Component, **System**	**Vertical data replication**
Associated Quality Aspects:	Literature Sources:
Analyzability, **Availability**, **Time-behavior**	new

Data replication along request trace IN USE

Formula:

$$\frac{replicationAlong(rt) + replicationInEndpoint(rt)}{\sum_{i=1}^{|rt.involvedLinks|} |rt.involvedLinks_i.RDA_E| + |rt.referencedEndpoint.RDA_E|}$$

Functions:

$$replicationAlong{:}rt \rightarrow \left(\sum_{i=1}^{|rt.involvedLinks|} replicatedDAs(rt.involvedLinks_i) \right)$$

$$replicatedDAs{:}link \rightarrow \left(\sum_{j=1}^{|link.targetEndpoint.RDA_E|} \begin{cases} 1 & \text{if } replicated(link.targetEndpoint.rda_j) \\ 0 & \text{else} \end{cases} \right)$$

$$replicationInE{:}rt \rightarrow \left(\sum_{i=1}^{|rt.referencedEndpoint.RDA_E|} \begin{cases} 1 & \text{if } replicated(rt.referencedEndpoint.rda_i) \\ 0 & \text{else} \end{cases} \right)$$

$$replicated{:}rda \rightarrow (rda.usage\_relation = \text{"}cached\text{-}usage\text{"} \vee rt.involvedLinks_i.rda_j.usage\_relation = \text{"persistence"})$$

Applicable Entities:	Associated Factor:
Request Trace	**Vertical data replication**
Associated Quality Aspects:	Literature Sources:
Analyzability, **Availability**, **Time-behavior**	new

4.6.2. Measure Validation

Parts of this section have been taken from [172].

The applicability of both, the measures adapted from literature and the newly formulated measures, needs to validated. The validation should cover the applicability of measures to evaluate the product factors with which they are associated as well as their usefulness to quantitatively evaluate the resulting impacts from the corresponding product factors. A validation based on an experiment as described in Section 3.1.5 has been done for a selected set of measures. Only a selection was used, because of the breadth of the quality model and the difficulty to validate certain measures in a controlled and measurable context. The selected measures and corresponding product factors are:

- Service replication level for the product factor Service replication
- Storage replication level for the product factor Horizontal data replication
- Ratio of cached data aggregates and Data replication along request trace for the product factor Vertical data replication

To validate the usefulness to quantitatively evaluate resulting impacts of these product factors, the following hypotheses and corresponding null hypothesis were formulated as a starting point for the experiment:
H1: *A higher Service replication has a positive impact on Time-behavior.*
$H1_0$: *Service replication has no impact or a negative impact on Time-behavior.*

H2: *A higher Service replication has a positive impact on Availability.*
$H2_0$: *Service replication has no impact or a negative impact on Availability.*

H3: *A higher Horizontal data replication has a positive impact on Time-behavior.* **$H3_0$**: *Horizontal data replication has no impact or a negative impact on Time-behavior.*

H4: *A higher Vertical data replication has a positive impact on Time-behavior.*
$H4_0$: *Vertical data replication has no impact or a negative impact on Time-behavior.*

H5: *A higher Vertical data replication has a positive impact on Availability.*
$H5_0$: *Vertical data replication has no impact or a negative impact on Availability.*
Each hypothesis describes a relationship between a factor (measurable by architectural measures) and a quality aspect (measurable by runtime measures). The aim of the validation experiment is to investigate whether the null hypotheses can be rejected. If a null hypothesis can be rejected, a relationship between the architectural measure and the runtime measure is assumed, and thus the corresponding alternative hypothesis is considered to be supported. In that case the architectural measures have shown their applicability.

Because the quality aspects in consideration are **Time-behavior** and **Availability**, runtime measures which can be used to evaluate these quality aspects are required. For the experiments the following two runtime measures were used:

Runtime Measure: **Response Time (rt)**:
The time it takes for a client from sending a request until it has received a full response.
Applicable entities: *System, Request Trace*
Used for quality aspect: Time-behavior
Literature source: [148]

The response time [148] is captured for every invocation by a client and can be aggregated in different ways: It can either be aggregated based on a system as a whole, but also per request trace, since different request traces can provide different functionalities and thus require more or less processing. Furthermore, it can be aggregated using either the mean, the median, or percentiles. The median or percentiles are more robust against outliers, so they should be considered at least in addition to a simple mean, if not preferred [30].

Runtime Measure: **Success rate**:
The ratio of successful requests over all requests within a specific time frame.
Applicable entities: *System, Request Trace*
Used for quality aspect: Availability
Literature source: [30]

The success rate (or successability [30]) measures the ratio of successful requests over all sent requests. What a successful request is needs to be specified. In the context of the experiment an HTTP status code of 200 as a response is considered to be successful. To use the success rate as a measure for **Availability**, only failures caused by the server side should be considered as unsuccessful requests. Therefore, only responses with a status code 500 are considered unsuccessful.

For each hypothesis experiment runs with a range of architectural variations of the TeaStore application were performed. The architectural variations differ in certain implementation and deployment aspects so that these differences are reflected in the architectural measures captured for the variations.

In the following, the results from the experiment runs are reported structured by the stated hypotheses. Each plot covers a certain architectural measure and its differing values for a range of architectural variations on the x-axis. On the y-axis the values for one of the runtime measures are plotted. By relating the gathered values for the design time measures with the values for the runtime measures through a linear regression model, their relation can be investigated. A relation between measures is considered to exist, if the relation expressed by the linear regression model is significant (based on the calculated p-values). Although for the shown results, non-linear regression models might have provided an overall better fit (considering R^2), these were not considered further in favor of the more intuitive linear regression models.

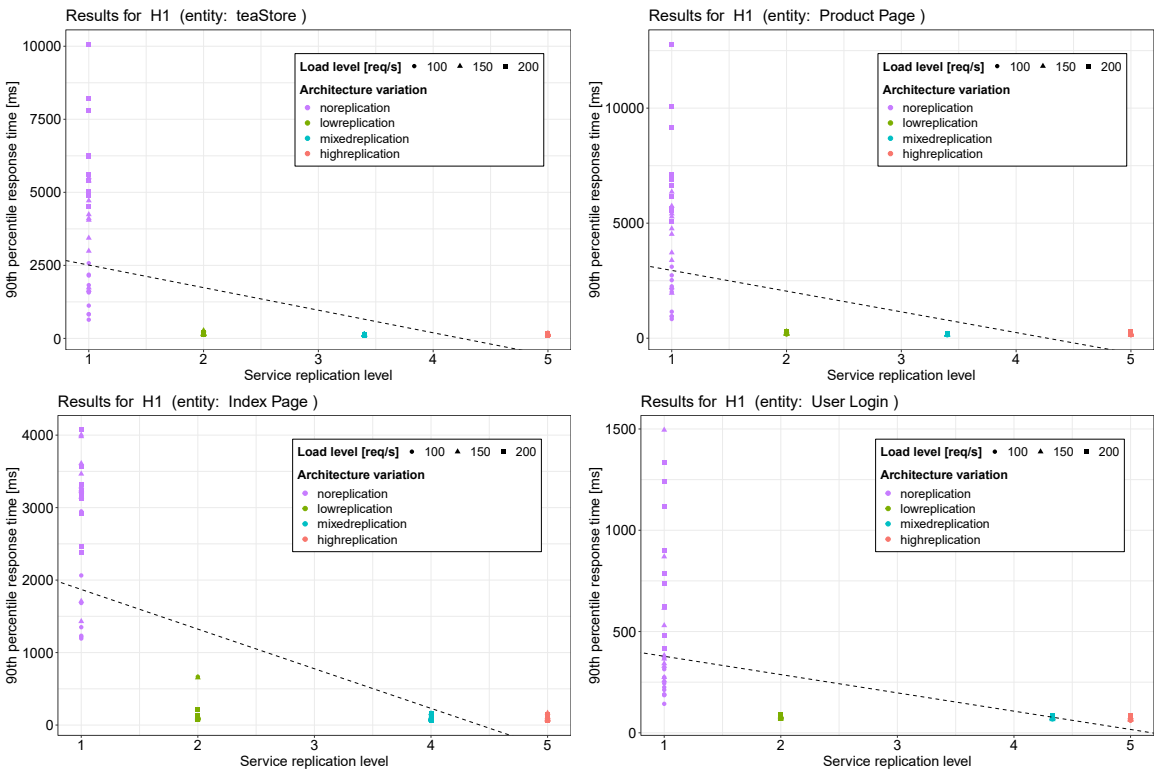


Figure 4.9.: Selected results for H1 with dashed regression lines (see Table 4.31). A decreasing response time corresponds to a better time-behavior.

Table 4.31.: Linear regression models for H1, see Figure 4.9

	teaStore	Product Page	Index Page	User Login
(Intercept)	3282.774***	3847.234***	2417.631***	468.060***
	(323.561)	(381.905)	(177.373)	(44.449)
Service replication level	−772.457***	−900.205***	−546.896***	−90.315***
	(100.380)	(118.481)	(52.304)	(12.732)
Num.Obs.	120	120	120	120
R2	0.334	0.329	0.481	0.299
R2 Adj.	0.329	0.323	0.477	0.293

Note: + p <0.1, * p <0.05, ** p <0.01, *** p <0.001

The data presented in Figure 4.9 and Table 4.31 show results relevant for **H1**, claiming that a higher **Service replication** has a positive impact on **Time-behavior**. As it can be seen for both the **teaStore** system as a whole and the request traces: **Product Page**, **Index Page**, and **User Login**, all variations with replication (*Service replication level* > 1) lead to a better performance than the *noreplication* variation. However, the differences between `lowreplication`, `mixedreplication`, and `highreplication` are small. This indicates that service replication only improves **Time-behavior** up to a certain limit. The relationship between *Service replication level* and the 90th percentile response time is significant (see Table 4.31). However, the linear regression model is not the best fit here because instead of a

linear improvement of response time, an improvement is possible only up to an inherent limit.

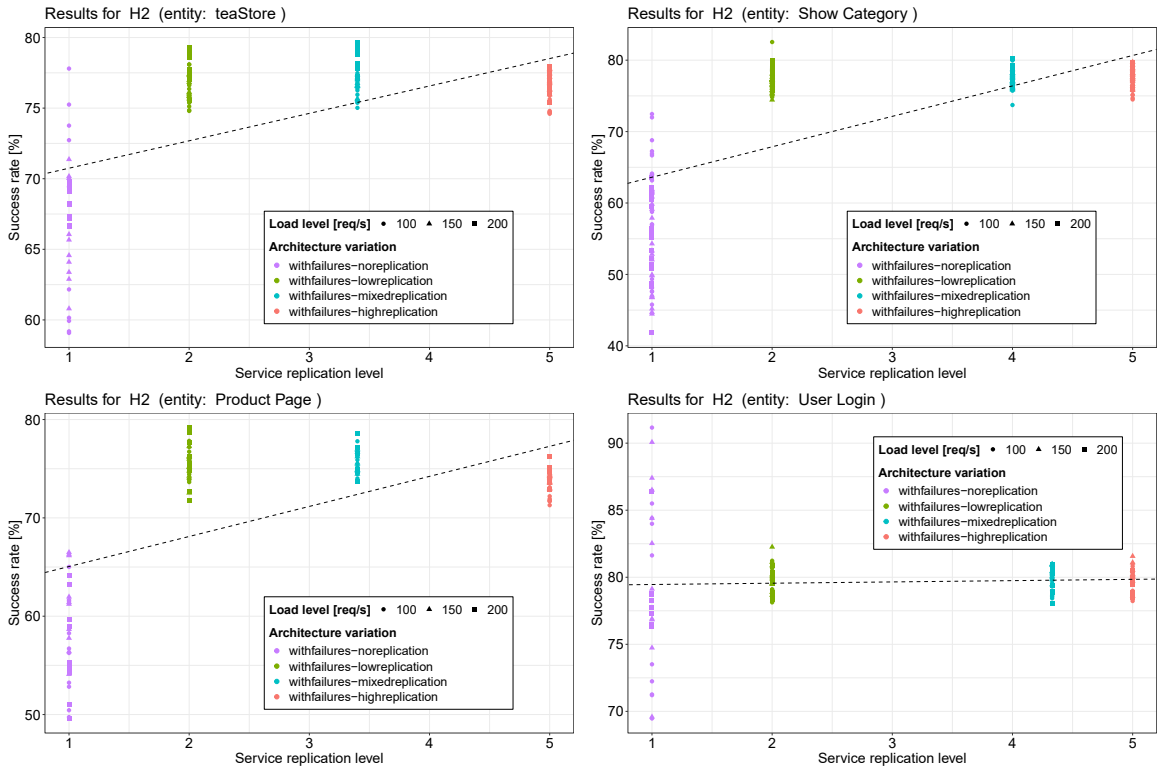


Figure 4.10.: Selected results for H2 with dashed regression lines (see Table 4.32) An increasing success rate corresponds to a better availability.

Table 4.32.: Linear regression models for H2, see Figure 4.10

	teaStore	Show Category	User Login	Product Page
(Intercept)	68.803***	59.349***	79.364***	61.996***
	(0.794)	(1.031)	(0.594)	(1.231)
Service replication level	1.941***	4.262***	0.096	3.056***
	(0.246)	(0.304)	(0.170)	(0.382)
Num.Obs.	120	240	120	120
R2	0.345	0.452	0.003	0.352
R2 Adj.	0.339	0.450	−0.006	0.346

Note: + p <0.1, * p <0.05, ** p <0.01, *** p <0.001

The data in Figure 4.10 and Table 4.32 focusing on **H2**, claiming that **Service replication** has a positive impact on **Availability**, shows a less clear result: A differentiation can be made between request traces, based on the number of services they span. For the **Show Category** and **Product Page** request traces, various data is integrated from multiple services, also from the image service. For these request traces, the **Service replication** improves the success rate because the risk that all instances of a service within the request trace are unavailable at the same time is reduced. However, the success rate is limited by the expected 80% resulting

from how failures are simulated in the services (for 8 out of 10 seconds, a service instance is available) and because requests were not retried by JMeter.

The **User Login** request trace depends on fewer data and services, and thus **Service replication** has a smaller impact. Notable, however, is the approximation of the expected 80% success rate as an average value, not as a limit. For the teaStore system as a whole, the impact is significant, but not entirely clear. The variations with a *Service replication level* of 2 and 3 show somewhat better results than the variation with a *Service replication level* of 5. This can only be explained with an unfortunate timing of simulated unavailability during executions and the influence of the results from the less complex request traces in the aggregated results for the whole system.

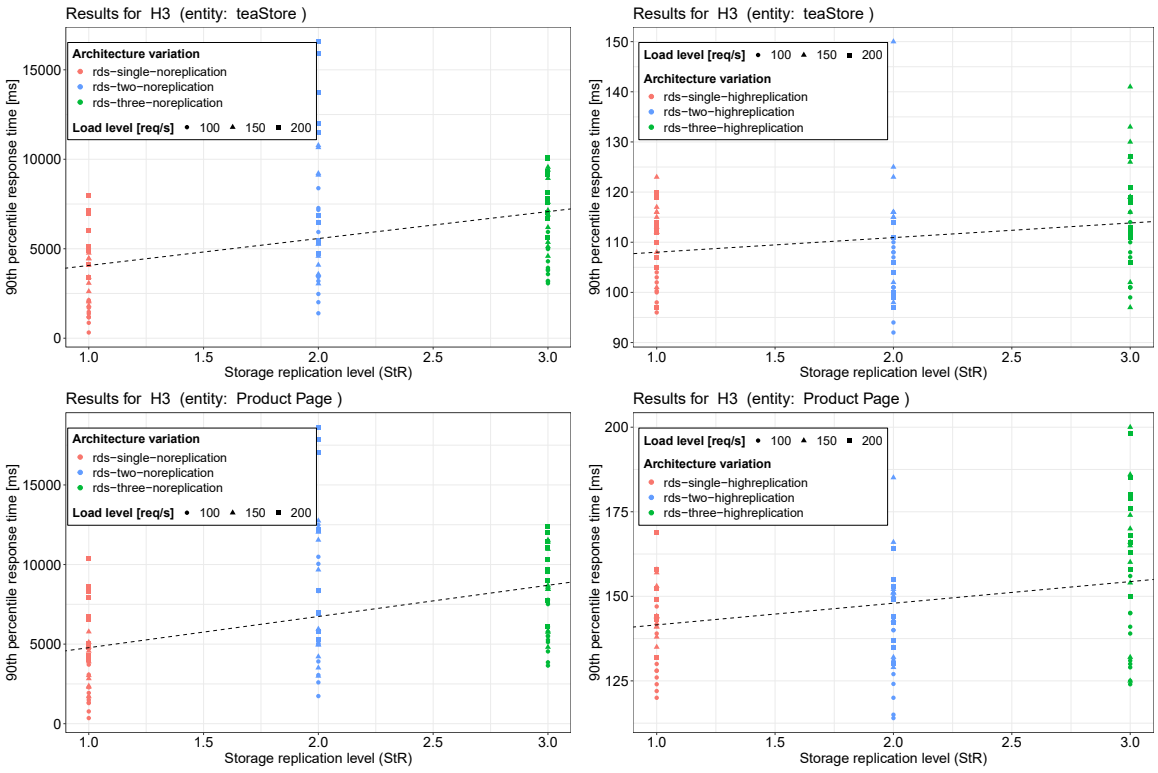


Figure 4.11.: Selected results for H3 with dashed regression lines (see Table 4.33) A decreasing response time would correspond to a better time-behavior.

Table 4.33.: Linear regression models for H3, see Figure 4.11

	teaStore	Product Page	teaStore	Product Page
(Intercept)	2562.150**	135.158***	2562.150**	135.158***
	(860.591)	(5.168)	(860.591)	(5.168)
Storage replication level	1504.468***	6.397**	1504.468***	6.397**
	(398.376)	(2.392)	(398.376)	(2.392)
Num.Obs.	90	90	90	90
R2	0.139	0.075	0.139	0.075
R2 Adj.	0.130	0.065	0.130	0.065

Note: + p <0.1, * p <0.05, ** p <0.01, *** p <0.001

Taking a look at the data for the investigation of **H3** in Figure 4.11 and Table 4.33, the hypothesis that a higher **Horizontal data replication** leads to a better **Time-behavior** is not supported. Instead, in the case of `noreplication`, where a single instance for each service was used, the response time is even worse for the variation with three database instances. The explanation for why there is no improvement is that the database is neither the bottleneck in the `noreplication` variation nor in the `highreplication` scenario. Therefore, the addition of database replicas does not improve the performance. The most likely explanation of why the performance is worse is that a newly added replica did not have the time yet to build up query caches, and the applied warm-up time was not enough. But especially in the `highreplication` scenario, the differences in response times are negligible. This leads to the result that the **Horizontal data replication** has no impact on **Time-behavior**, at least for the conducted experiments.

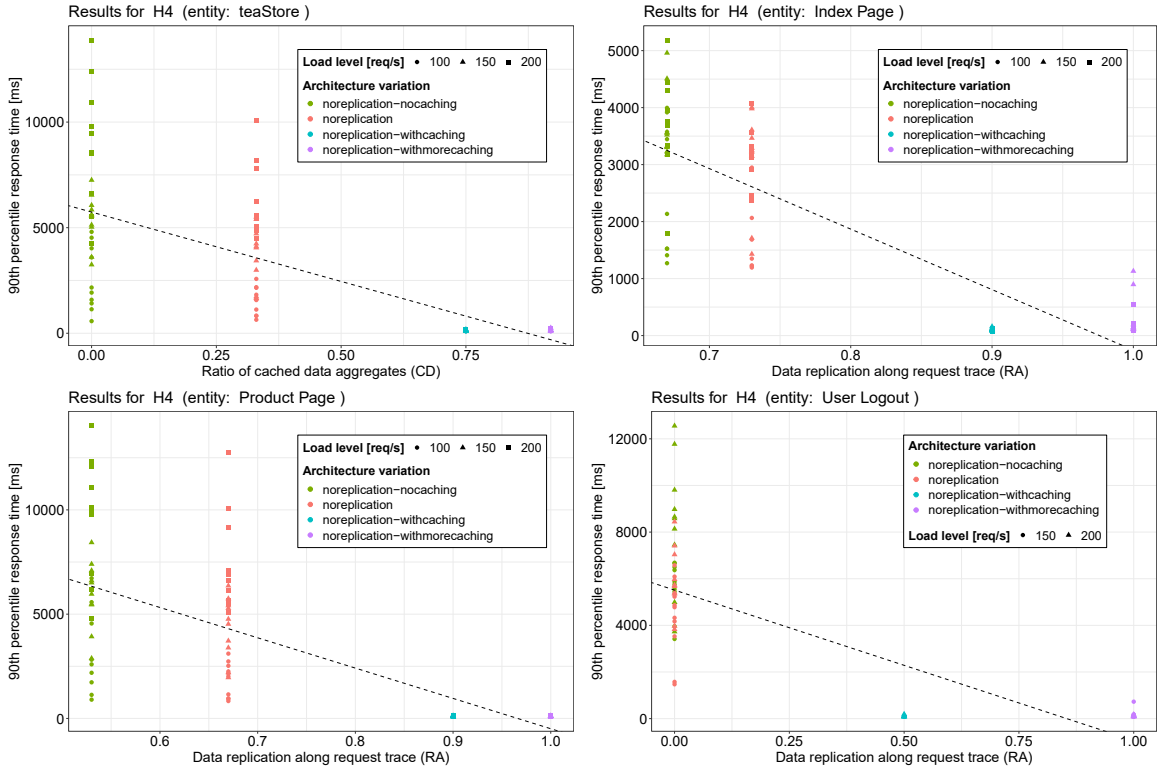


Figure 4.12.: Selected results for H4 with dashed regression lines (see Table 4.34) A decreasing response time corresponds to a better time-behavior.

H4 claims that a higher **Vertical data replication** has a positive impact on **Time-behavior**. The data in Figure 4.12 and Table 4.34 supports this claim. Response time consistently decreases with a higher **Vertical data replication**. This is the case for both measures, *Ratio of cached data aggregates* at the system level and *Data replication along request trace* at the level of individual request traces. Especially when caching is used at the Webui service, directly where requests come in, (`-withcaching` and `-withmorecaching`), the response time drops significantly. Overall, however, similar to H1, the linear regression model is not the best fit in this case. One exception is the increase of response time for the request trace **In-**

Table 4.34.: Linear regression models for H4, see Figure 4.12

	teaStore	Product Page	Index Page	User Logout
(Intercept)	5738.738***	14 039.265***	10 360.671***	5512.225***
	(325.733)	(899.789)	(467.750)	(317.089)
Ratio of cached data aggregates	−6569.883***			
	(528.792)			
Data replication along request trace		−14 532.519***	−10 615.647***	−6453.935***
		(1129.201)	(559.889)	(567.225)
Num.Obs.	120	120	120	80
R2	0.567	0.584	0.753	0.624
R2 Adj.	0.563	0.580	0.751	0.619

Note: + p <0.1, * p <0.05, ** p <0.01, *** p <0.001

dex Page with more caching. However, these seem to be single outliers which may be due to some unusual invocations for entities that were not cached.

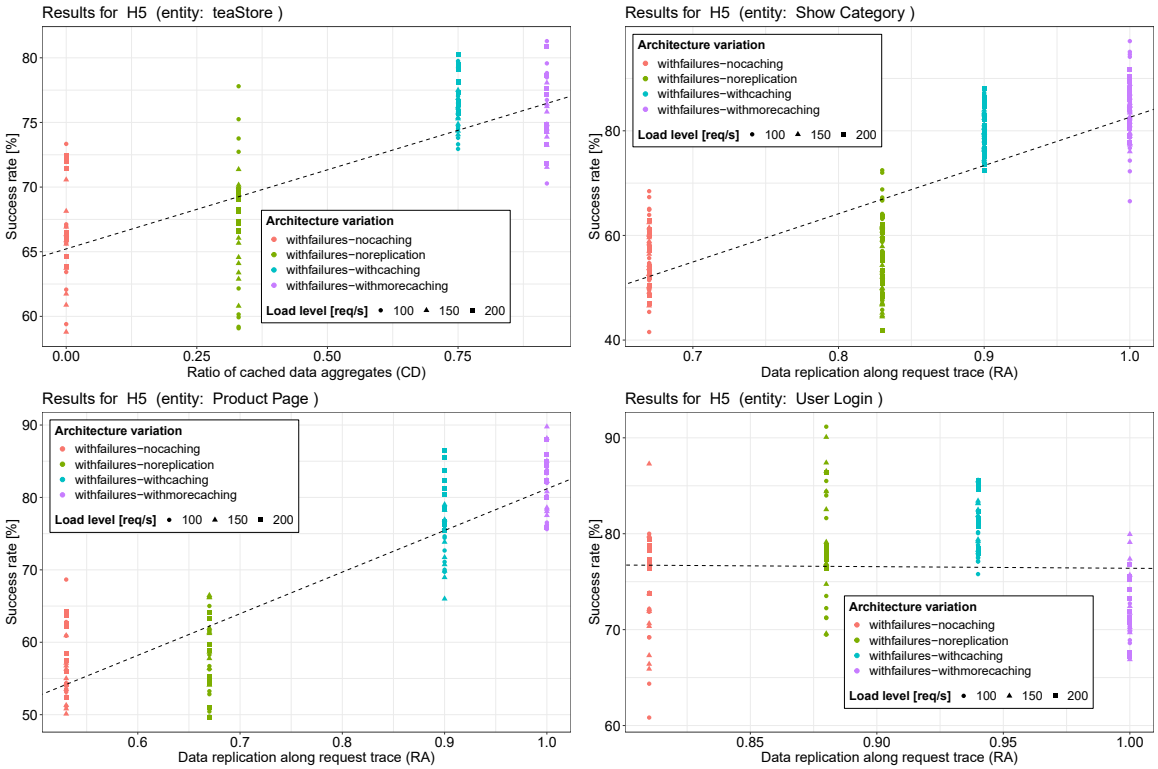


Figure 4.13.: Selected results for H5 with dashed regression lines (see Table 4.35) An increasing success rate corresponds to a better availability.

Finally, the claim of **H5** that a higher **Vertical data replication** also has a positive impact on **Availability** is considered. The data in Figure 4.13 and Table 4.35 provides a clear result only for certain cases. Considering the **teaStore** as a whole, the improvement is significant. Similar to the consideration of H2, however, a

Table 4.35.: Linear regression models for H5, see Figure 4.13

	teaStore	User Login	Product Page	Show Category
(Intercept)	65.218***	78.072***	23.714***	−9.630*
	(0.601)	(6.874)	(2.072)	(4.029)
Ratio of cached data aggregates	12.235***			
	(0.976)			
Data replication along request trace		−1.673	57.474***	92.215***
		(7.552)	(2.601)	(4.694)
Num.Obs.	120	120	120	240
R2	0.571	0.000	0.805	0.619
R2 Adj.	0.567	−0.008	0.804	0.617

Note: + p <0.1, * p <0.05, ** p <0.01, *** p <0.001

differentiation needs to be made between request traces combining data from several services and simpler ones. The improvement in the success rate is evident for the **Show Category** and **Product Page** request traces, which combine data from the persistence service and the recommender service. For other request traces, the results are less clear. For example, for the **User Login** request trace there is a slight improvement in the success rate, but it is not consistent for all cases.

As a summary, Table 4.36 includes all validation results with a ✓, if the linear regression models for the considered measures per entity were significant, and with a ✗, if this was not the case. As it can be seen, most relations were found to be significant except for H3 and several request traces in the case of H5. For H3, it has already been discussed that this is due to the database not being the bottleneck and therefore a replication of it not impacting **Time-behavior**. And for H5, the explanation is that especially for the less complex request traces, the effect of caching on **Availability** is not as significant, also because of the simulated unavailability approach applying equally to all services. The measures *Service replication level*, *Ratio of cached data aggregates*, and *Data replication along request trace* are thus seen as validated. The measure *Storage replication level* could not be validated successfully based on the experiment. It is nevertheless kept in the quality model, because of the mentioned results interpretation and because the concept of using database replication for performance improvements being a common approach, also studied in literature [265].

Apart from the validation of the specific measures in consideration, the results from the validation experiment also show in general an approach for validating measures used in the context of quality evaluation approaches. The measures formulated to support the concepts derived from literature during the formulation of the quality model, are validated from a quantitative perspective by measuring them and the impacts of their variations in a use case. It is one possible approach for validation that relies on quantitative data and an experiment. This might, however, not be possible to use for all measures and product factors of a quality model.

Table 4.36.: Summarized Hypotheses evaluation per entity

	Rejection of null hypothesis				
Entity	**$H1_0$**	**$H2_0$**	**$H3_0$**	**$H4_0$**	**$H5_0$**
teaStore	✓	✓	✗	✓	✓
Index Page	✓	✓	✗	✓	✗
Show Category	✓	✓	✗	✓	✓
User Login	✓	✗	✗	✓	✗
Product Page	✓	✓	✗	✓	✓
Add Product To Cart	✓	✓	✗	✓	✗
User Logout	✓	✗	✗	✓	✗

Especially for measures used for factors in the context of maintainability, experiments would have to be conducted over a longer time-frame. This is because quantitative measures for maintainability can typically only be observed over time (for example, the number of faults detected over time [264]). Conducting an experiment over a long time-frame requires more effort. Another approach to validate such measures, could therefore be, for example, to quantitatively analyzing architectural variations based on expert opinions. Experts rate certain quality aspects of a system based on their personal experience and these ratings can then be compared with architectural measures covering the same quality aspects through the product factors to which they are assigned.

4.7. Evaluations

As the last type of elements of the quality model, evaluations connect all other elements of a quality model to provide quality evaluation results specific to a modeled software architecture. Evaluations use measure values, calculated for the entities of an architectural model, as an input. Based on these, factors are quantified and rated. This in turn also enables and evaluation of impacted factors and quality aspects by combining individual evaluation results through the specified impact relationships. Accordingly, for each factor of the quality model (product factor or quality aspect) an evaluation function must be defined that takes measure values and impacting factors as an input and provides a corresponding output for that factor. Therefore, it must be clear how to interpret the calculated measure values and how to derive evaluation results for factors (see also Section 2.2.4). Also, aggregation functions are required, if multiple measures are used for a factor or if a factor is impacted by multiple other factors. The evaluation approach used for the quality model is detailed in the following sections. Firstly, general considerations for factor evaluations are described in Section 4.7.1. Then the chosen approaches for evaluating leaf factors (Section 4.7.2), intermediate factors (Section 4.7.3), and quality aspects (Section 4.7.4) are described. And finally, exemplary evaluations based on applications serving as use cases are presented in Section 4.7.5.

4.7.1. General Factor Evaluation Considerations

In this section, general aspects of the chosen evaluation approach are presented. The way how specifically factors are evaluated in the sense of how measures are interpreted and in which form evaluation results are captured is not entirely prescribed by the Quamoco approach [294]. It is stated that an evaluation should assign a value between 0 and 1 to a factor, in line with the concept that a product factor should be evaluable according to "*the extent that a factor is present in a software product*" [294]. But an additional interpretation scale should be used to present the result. Therefore, the definition of such an interpretation scale is custom to the specific quality model at hand. For the quality model presented in this work, it was decided to use a consistent evaluation result approach for all product factors. But instead of a numerical value, it was decided to use an ordinal value that is more intuitive to understand. The more detailed results, specifically measure values, are nevertheless available together with details on how these values were interpreted to come up with the ordinal value. Furthermore, factors can be evaluated on the level of different entities, if applicable. Thus, evaluations are not only available on the level of a system as a whole, but also on the level of individual components and request traces. This provision of evaluations for different entities together with the used measure values is intended to make up for the potential loss of information through using ordinal values for evaluations (see Section 2.2.4).

The ordinal scale for product factor evaluation results used in the quality model includes the following values for an `evaluationResult`:

- **n/a**: the factor could not be evaluated, because of missing information (e.g., if the evaluation requires information from entities which were simply not modeled (yet) in a software architecture)
- **none**: the factor could be evaluated, but it is non-existent in a system (or another entity)
- **low**: the factor was found to be present to a low extent
- **moderate**: the factor was found to be present to a moderate extent
- **high**: the factor was found to be present to a high extent

For the evaluation of each product factor, thus, an evaluation function is needed which evaluates a factor by assigning it a value of this ordinal scale based on a set of input parameters. This general evaluation function has the following form:

$$evaluate : \texttt{factor}, \texttt{entity}, \texttt{measures}, \texttt{impacts} \rightarrow \texttt{evaluationResult} \quad (4.1)$$

As inputs, this function takes the product `factor` to evaluate, the `entity` based on which a product factor should be evaluated, the values of the assigned architectural `measures` calculated for that `entity`, and potentially already evaluated `impacts` from other factors that impact the factor to evaluate. The evaluation function is unique for each product factor. However, there are some general approaches on which evaluation functions can rely and customize when needed. These approaches can be differentiated based on the types of factor to evaluate, as shown in the following sections.

4.7.2. Evaluating Leaf Product Factors

Leaf product factors are not impacted by any other factors. Thus, their evaluation is based entirely on corresponding measures. The calculated measure values need to be interpreted in order to assign an evaluation result to a leaf factor. Especially when measure values range from 0 to 1, they can be interpreted according to a set of pre-defined mappings. These are shown in the form of plots in Figure 4.14.

The basic interpretation would be the **linear** evaluation mapping with which a measure between 0 and 1 is assigned to an ordinal `evaluationResult` value. For cases in which a measure interpretation should be more "sensitive", meaning that even lower values of a measure should be interpreted as a factor being present, an **exponential** evaluation mapping can be used. And, in contrast to that, for cases in which a measure interpretation should be less "sensitive", a **square rooted** mapping can be used. If more than one measure is used for the evaluation,

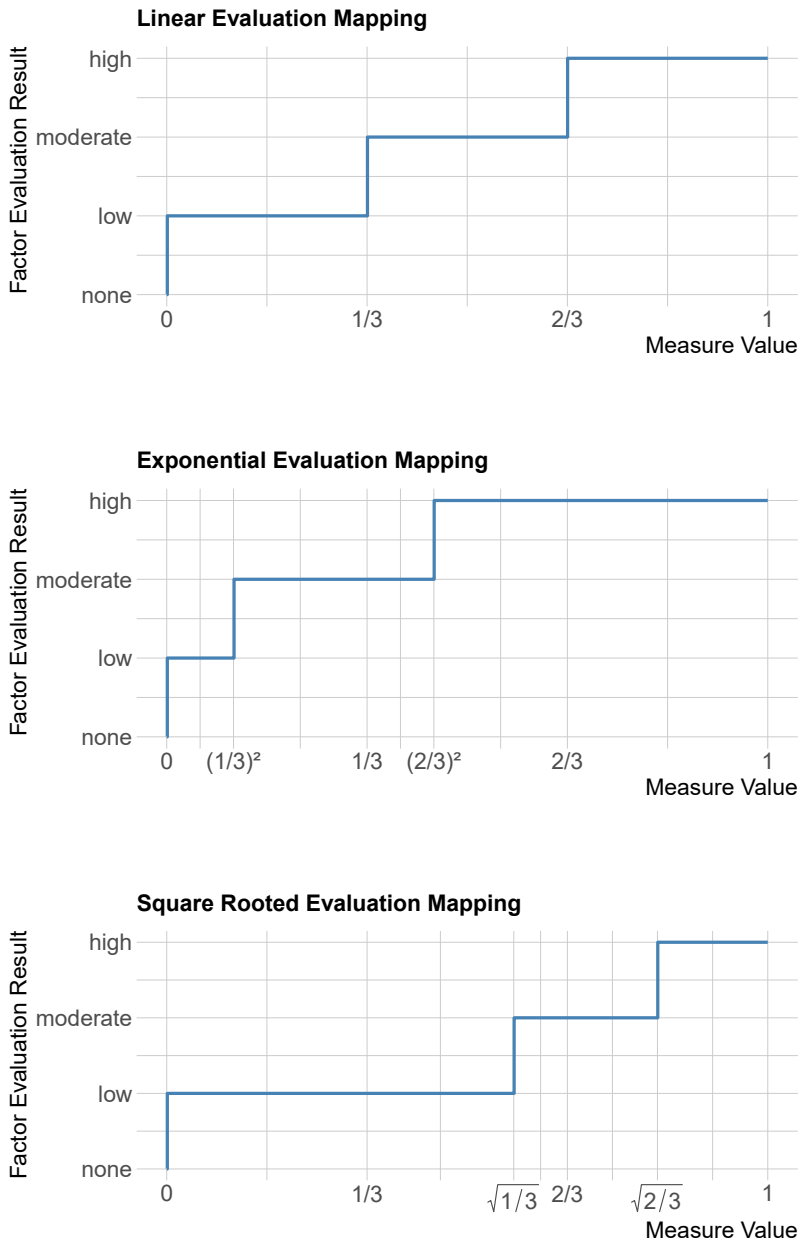


Figure 4.14.: Options for mapping measure values [0:1] to factor evaluation results

they can be combined in different ways, depending on the specific interpretation. For example, their values could be averaged, their values could be summed up or multiplied, or different weights could be used for their aggregation. For measures which do not range in values from 0 to 1, custom evaluations need to be specified, typically in the form of thresholds. Defining these thresholds, however, can be challenging, because they might not be generally assignable and instead depend on the specific application at hand.

The specific evaluation approaches for leaf factors of the quality model are described in more detail in the tables 4.37, 4.38, 4.39, and 4.40. For all leaf factors, it is shown for which entities these factors can be evaluated, which measures are

used for which entities and how these measures are interpreted. The mentioned evaluation functions refer to those presented in Figure 4.14. As it can be seen, most evaluation functions interpret the measure values with a linear mapping. But, for example, the measure *Ratio of non-custom backing services* is interpreted with a square rooted mapping. This is because custom backing services should really be reduced to a minimum, since every custom backing service for a functionality that has already been implemented by the community represents an additional effort. For the measure *Event sourcing utilization metric* an exponential mapping was chosen, because already a few interactions designed this way can have an impact. Furthermore, some evaluation functions are based on thresholds. For example, the measure *Service replication level* is an absolute value for which an interpretation is needed. A reasonable replication level depends on the application at hand and while a *Service replication level* of 1 is clearly mappable to a **Service replication** of "none", thresholds for the other extents are difficult to choose in a generally applicable way. Therefore, a set of thresholds was used which would be reasonable for medium-sized applications, but these might not be reasonable for all kinds of applications. The fact that the context of an application influences the interpretation of measures applies in a similar way to the measures *Maximum number of services within a request trace*, *Request trace complexity*, *Number of components*, *Storage replication level*, and *Level of sharding across storage backing services*. Also for these measures a reasonable set of thresholds for medium-sized applications was chosen. Other additional measures requiring a set of thresholds are *Number of availability zones used by services* and *Number of availability zones used by storage services*. In general, an availability zone is expected to fail rarely. Thus, using a second availability zone as a backup already improves availability and a third enables a faster recovery [200]. But using even more availability zones has only little benefits. Therefore, threshold values for these measures are set rather conservatively.

Table 4.37.: Evaluation overview for leaf factors - I

Product Factor	Entity	Measures used	Evaluation function
Data encryption in transit	System, Request Trace	Ratio of external endpoints that support TLS, Ratio of secured links	linear mapping of averaged measure values
Isolated secrets	Component, Infrastructure, System, Request Trace	Secrets externalization	linear mapping of measure value
Secrets stored in specialized services	Component, Infrastructure, System	Secrets stored in vault	linear mapping of measure value
Least-privileged access	Component, System	Access restricted to callers	linear mapping of measure value
Access control management consistency	System, Component	Consistency of supported authentication methods of endpoints, Consistency of supported authentication methods of external endpoints	linear mapping of averaged measure values
Account separation	System, Request Trace	Ratio of unique account usage	square rooted mapping of measure value
Authentication delegation	System, Request Trace	Ratio of delegated authentication	square rooted mapping of measure value
Limited data scope	Component, System	Cohesion between endpoints based on data aggregate usage	linear mapping of measure value
Limited endpoint scope	System, Component	Service interface usage cohesion	linear mapping of measure value
Command query responsibility segregation	System, Component	Read write separation for data aggregates	exponential mapping of measure value
Separation by gateways	Component, System, Request Trace	Degree of separation by gateways	linear mapping of measure value
Mostly stateless services	System, Request Trace	Ratio of stateless components, Degree to which components are linked to stateful components (inverse)	linear mapping of average measure values
Specialized stateful services	System, Request Trace	Ratio of specialized stateful services, Ratio of suitably replicated stateful services	linear mapping of measure values product
Asynchronous communication	System, Request Trace	Degree of asynchronous communication, Asynchronous communication utilization	linear mapping of averaged measure values
Communication partner abstraction	System, Request Trace	Event sourcing utilization metric	exponential mapping of measure value
Persistent communication	System, Request Trace	Service interaction via backing service, Event sourcing utilization metric	linear mapping of weighted average measure values
Usage of existing solutions for non-core capabilities	System, Component	Ratio of non-custom backing services	square rooted mapping of measure value
Standardization	System, Component, Infrastructure, Request Trace	Ratio of standardized artifacts, Ratio of entities providing standardized artifacts	linear mapping of weighted average measure values
Component similarity	System	Component artifacts similarity, Infrastructure artifacts similarity	linear mapping of averaged measure values
	Request Trace	Component artifacts similarity	linear mapping of measure value

Table 4.38.: Evaluation overview for leaf factors - II

Product Factor	Entity	Measures used	Evaluation function
Consistent centralized logging	Component, System, Infrastructure	Ratio of components or infrastructure nodes that export logs to a central service	linear mapping of measure value
Consistent centralized metrics	Component, Infrastructure, System	Ratio of components or infrastructure nodes that export metrics	linear mapping of measure value
Distributed tracing of invocations	Component, System, Request Trace	Distributed tracing support	linear mapping of measure value
Health and readiness checks	Component, System, Request Trace	Ratio of services that provide health endpoints, Ratio of services that provide readiness endpoints	linear mapping of averaged measure values
Automated infrastructure provisioning	System, Infrastructure	Ratio of automatically provisioned infrastructure	linear mapping of measure value
Use infrastructure as code	System, Infrastructure	Ratio of infrastructure with IaC artifact	linear mapping of measure value
Dynamic scheduling	System, Component, Request Trace	Ratio of components deployed dynamically	linear mapping of measure value
Low coupling	System	Degree of coupling in a system (inverse)	square rooted mapping of measure value
	Component	Number of components a component is linked to relative to the total amount of components (inverse)	linear mapping of measure value
Functional decentralization	System	Data aggregate spread (inverse), Request trace similarity based on included components (inverse)	square rooted mapping of averages measure values
Limited request trace scope	System	Average complexity of request traces, Maximum number of services within a request trace	*m*=maximum of values $m <= 1 \rightarrow high$ $1 < m <= 3 \rightarrow moderate$ $3 < m <= 6 \rightarrow low$
	Request Trace	Request trace complexity, Maximum number of services within a request trace	*m*=maximum of values $m <= 1 \rightarrow high$ $1 < m <= 3 \rightarrow moderate$ $3 < m <= 6 \rightarrow low$
Logical grouping	System, Component	Namespace separation	square rooted mapping of measure value
Backing service decentralization	System	Average weighted storage backend sharing (inverse), Average weighted broker backend sharing (inverse)	square rooted mapping of averaged measure values
	Component	Ratio of storage backend sharing (inverse), Ratio of broker backend sharing (inverse)	square rooted mapping of averaged measure values
Addressing abstraction	System, Request Trace	Service discovery usage	linear mapping of measure value
Sparsity	System	Number of components	*m*=measure value $m < 3 \rightarrow high$ $3 <= m < 6 \rightarrow moderate$ $6 <= m < 10 \rightarrow low$
Managed infrastructure	System, Infrastructure	Ratio of fully managed infrastructure	square rooted mapping of measure value
Managed backing services	System, Component	Ratio of managed backing services	square rooted mapping of measure value

Table 4.39.: Evaluation overview for leaf factors- III

Product Factor	Entity	Measures used	Evaluation function
Service replication	System, Component, Request Trace	Service replication level, Amount of redundancy	*m*=measure value $m >= 3 \rightarrow high$ $1.5 <= m < 3 \rightarrow moderate$ $1 < m < 1.5 \rightarrow low$ $m <= 1 \rightarrow none$
Horizontal data replication	System, Request Trace, Component	Storage replication level	*m*=measure value $m >= 3 \rightarrow high$ $1.5 <= m < 3 \rightarrow moderate$ $1 < m < 1.5 \rightarrow low$ $m <= 1 \rightarrow none$
Vertical data replication	System	Ratio of cached data aggregates	linear mapping of measure value
	Request Trace	Data replication along request trace	linear mapping of measure value
	Component	Ratio of cached data aggregates	linear mapping of measure value
Sharded data store replication	System, Request Trace, Component	Level of sharding across storage backing services	*m*=measure value $m >= 3 \rightarrow high$ $1.5 <= m < 3 \rightarrow moderate$ $1 < m < 1.5 \rightarrow low$ $m <= 1 \rightarrow none$
Enforcement of appropriate resource boundaries	System, Infrastructure	Ratio of infrastructure entities enforcing resource boundaries, Ratio of deployment mappings with stated resource requirements	linear mapping of averaged measure values
	Component	Ratio of deployment mappings with stated resource requirements	linear mapping of measure value
Built-in autoscaling	System, Infrastructure	Deployed entities autoscaling, Infrastructure autoscaling	linear mapping of averaged measure values
	Component	Deployed entities autoscaling	linear mapping of measure value
Infrastructure abstraction	System, Infrastructure	Ratio of abstracted hardware	linear mapping of measure value
Cloud vendor abstraction	System	Non-provider-specific infrastructure artifacts, Non-provider-specific component artifacts	linear mapping of averaged measure values
	Component	Non-provider-specific component artifacts	linear mapping of measure value
	Infrastructure	Non-provider-specific infrastructure artifacts	linear mapping of measure value
Isolated configuration	System, Component, Infrastructure, Request Trace	Configuration externalization	linear mapping of measure value
Configuration stored in specialized services	System, Component, Infrastructure	Configuration stored in config service	linear mapping of measure value
Contract-based links	System, Component, Request Trace	Ratio of endpoints covered by contract	linear mapping of measure value
Standardized self-contained deployment unit	System, Component, Request Trace	Standardized deployments, Self-contained deployments	linear mapping of averaged measure values

Table 4.40.: Evaluation overview for leaf factors- IV

Product Factor	Entity	Measures used	Evaluation function
Immutable artifacts	System, Component, Infrastructure	Replacing deployments	linear mapping of measure value
Guarded ingress	System, Component, Request Trace	Ratio of components whose external ingress is proxied, Ratio of rate limiting endpoints	linear mapping of averaged measure values
Physical data distribution	System, Component, Request Trace	Number of availability zones used by storage services	m=measure value $m >= 4 \rightarrow high$ $3 <= m < 4 \rightarrow moderate$ $1 < m < 3 \rightarrow low$ $m <= 1 \rightarrow none$
Physical service distribution	System, Component, Request Trace	Number of availability zones used by services	m=measure value $m >= 4 \rightarrow high$ $3 <= m < 4 \rightarrow moderate$ $1 < m < 3 \rightarrow low$ $m <= 1 \rightarrow none$
Rolling upgrades enabled	System	Rolling update option, Rolling updates	linear mapping of averaged measure values
	Infrastructure	Rolling update option	linear mapping of measure value
	Component, Request Trace	Rolling updates	linear mapping of measure value
Automated restarts	System, Component, Infrastructure	Deployments with restart	square rooted mapping of measure value
Automated infrastructure maintenance	System, Infrastructure	Ratio of automatically maintained infrastructure	linear mapping of measure value
Invocation timeouts	System, Component, Request Trace	Ratio of links with timeout	square rooted mapping of measure value
Retries for safe invocations	System, Component, Request Trace	Ratio of links with retry logic	square rooted mapping of measure value
Circuit breaked communication	System, Component, Request Trace	Ratio of links with complex failover	square rooted mapping of measure value
API-based communication	System, Component, Request Trace	Ratio of documented endpoints	linear mapping of measure value
Consistently mediated communication	System, Component, Request Trace	Service mesh usage	linear mapping of measure value

When it is possible to set evaluation results for factors based on measure values, the next point is how to quantify the impacts resulting from an evaluated factor. As described in Section 4.5, impacts can be either positive or negative. Following the concept that impacts are effective if the factor from which they are resulting is present, the extent to which a factor is present also determines the strength or weight of an impact. Although this determination of an impact weight based on a factor evaluation result can be customized, a conservative default approach has been chosen. This approach is shown in Figure 4.15. Conservative in this case means, that an impact is only considered to be effective if the factor from which the impact originates is at least evaluated as being present **moderate**ly. For example, if an impact is positive and the factor evaluation result is moderate, the impact has a **slightly positive** effect.

4.7.3. Evaluating Intermediate Factors

When `impacts` are available, the `evaluationResult` can also be determined solely from these. This is the case for intermediate factors which only have incoming impacts without additional measures. The evaluation function then needs to aggregate the incoming impact weights for assigning an `evaluationResult` to such a factor. Again, this aggregation can be customized per factor, but some predefined aggregation functions can be used. For all aggregation functions, the incoming impact weights are firstly mapped to numerical values in the following way:

`negative` $\rightarrow -1$, `slightly negative` $\rightarrow -0.5$, `neutral` $\rightarrow 0$,
`slightly positive` $\rightarrow +0.5$, and `positive` $\rightarrow +1$.

The mapped values can then be aggregated by calculating the `mean`, the `median`, choosing the `lowest` or `highest` value or also a `custom` function. Furthermore, it can be specified which precondition should be fulfilled for doing an aggregation, depending on how many impact weights should be present. It might be enough that `at-least-one` impact is present or that a `majority` is needed, or that `all` impacts need to be present.

The chosen aggregation functions and preconditions for all intermediate factors of the quality model are shown in Table 4.41. For most intermediate factor evaluations it was decided, that `at-least-one` impact is sufficient to provide an evaluation, because the impacting factors cover somewhat different aspects. For the factors **Service-orientation** and **Service independence**, however, a `majority` of impacts should be evaluated, because several aspects need to be considered in combination for a meaningful evaluation. For the aggregation, mostly the `median` was chosen, so that weaker impacts are not suppressed by stronger impacts. But, for example, for **Access restriction**, the `lowest` impact was chosen to be the deciding one, because in the context of security, all aspects need to be considered.

For cases where `measures` and `impacts` need to be considered in combination, a custom evaluation function is required. Such cases, however, currently do not exist in the quality model.

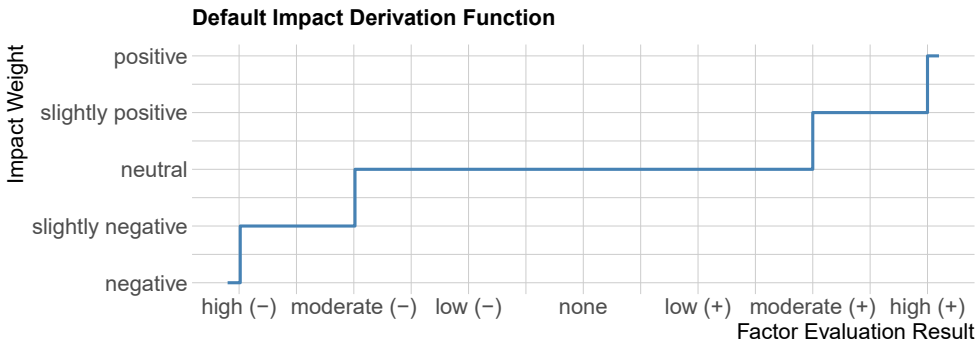


Figure 4.15.: Default mapping of factor evaluation results to impact strengths (+ describes a positive impact, while - describes a negative impact).

Table 4.41.: Evaluation overview for intermediate factors

Product Factor	Entity	Impacts Aggregation	Precondition
Secrets management	Component	mean	at-least-one
	Infrastructure	mean	at-least-one
	System	mean	at-least-one
	Request Trace	mean	at-least-one
Access restriction	Component	lowest	at-least-one
	System	lowest	at-least-one
Service-orientation	System	median	majority
	Component	median	majority
Limited functional scope	Component	median	at-least-one
	System	median	at-least-one
Isolated state	System	lowest	at-least-one
	Request Trace	lowest	at-least-one
Loose coupling	System	median	at-least-one
	Request Trace	median	at-least-one
Automated monitoring	System	median	at-least-one
	Component	median	at-least-one
	Infrastructure	median	at-least-one
	Request Trace	median	at-least-one
Service independence	System	mean	majority
	Component	mean	majority
Operation outsourcing	System	mean	at-least-one
	Component	median	at-least-one
	Infrastructure	median	at-least-one
Replication	System	mean	at-least-one
	Component	mean	at-least-one
	Request Trace	mean	at-least-one
Configuration management	System	median	at-least-one
	Component	median	at-least-one
	Infrastructure	median	at-least-one
	Request Trace	median	at-least-one
Distribution	System	median	at-least-one
	Component	median	at-least-one
	Infrastructure	mean	at-least-one
	Request Trace	median	at-least-one
Seamless upgrades	System	median	at-least-one
	Infrastructure	median	at-least-one
	Component	median	at-least-one
	Request Trace	median	at-least-one
Autonomous fault handling	System	median	at-least-one
	Component	median	at-least-one
	Request Trace	median	at-least-one

4.7.4. Evaluating Quality Aspects

Lastly, quality aspects also need to be evaluated by assigning them an evaluation result. Their evaluation is solely based on the incoming impacts and their respective weights. Thus, their evaluation can be formalized in the following form:

$$evaluate : \texttt{qualityAspect}, \texttt{entity}, \texttt{impacts} \rightarrow \texttt{qaEvaluationResult} \quad (4.2)$$

To also evaluate quality aspects consistently, it was decided to use the following scale for `qaEvaluationResult` values:

- **n/a** The quality aspect cannot be evaluated, because no incoming impacts have weights assigned
- **neutral** The quality aspect is evaluated, but it is overall impacted in a neutral way.
- **mixed** The quality aspect is evaluated, but no overall evaluation is done, because the impact weights conflict (e.g. there are negative and positive impacts).
- **negative** The quality aspect is evaluated as being negative.
- **slightly negative** The quality aspect is evaluated as being slightly negative.
- **slightly positive** The quality aspect is evaluated as being slightly positive.
- **positive** The quality aspect is evaluated as being positive.

In the current state of the quality model it was decided to use the same evaluation function for all quality aspects to have a consistent default approach. For providing insights on how an architecture could be changed or improved, the product factors provide more meaningful information anyway and the impacted quality aspects are more important for the overall structure of an evaluation and to be able to focus on a certain area. An aggregated evaluation result for a quality aspect provides more of a general overview rather than specific aspects which could be tackled in an architecture. The chosen evaluation approach basically uses the same functions as the evaluation approach for intermediate factors where also a number of incoming impacts need to be aggregated. For all quality aspect **at-least-one** impact should be effective for an evaluation and as an aggregation function the **median** was chosen. A difference, however, is that a quality aspect can also be evaluated as having **mixed** impacts if there are positive as well as negative impacts. This possibility is in line with the reasoning that the evaluations of individual product factors and their resulting impacts on quality aspects are more meaningful for providing actionable insights than a somehow aggregated quantitative score for a quality aspect.

4.7.5. Validation through the Evaluation of Use Cases

The presented evaluations have been formulated based on the understanding of the product factors from literature as well as knowledge of and experience with the topic. Thus, to validate their suitability, the quality evaluation approach was applied to a set of applications serving as use cases [173], as described in Section 3.1.6.

To present this validation, firstly the extracted key characteristics of the applications are listed and secondly the resulting evaluations of the applications are reviewed. The created architectural models of the applications are not considered in detail here, but used as examples in Section 5.2.5 and Section 5.3.2.

4.7.5.1. Key Characteristics of Applications

The **TeaStore** [293] is a reference application for microservices-based architectures (**Tea1**). It was primarily built for conducting performance benchmarking experiments to investigate performance aspects in microservices-based architectures. It consists of five services providing the business functionality. Accordingly, the product factors **Service-orientation** and **Service independence**, as well as the corresponding factors impacting them, represent characteristics that a microservices-based architecture should follow. For enabling communication between these services a registry backing service is included, thus providing **Addressing abstraction** (**Tea2**). Services retrieve available instances of other services from the registry backing service and perform client-side load balancing. Thus, services can also be replicated in any manner which is covered by the factor **Service replication** (**Tea3**). To avoid sending traffic to unhealthy instances, the services provide health endpoints through which **Health and readiness checks** can be performed (**Tea4**). All communication is based on synchronous RESTful HTTP calls for which services use timeouts and retries when acting as clients to implement **Autonomous fault handling** (**Tea5**). The persistence service which is connected to the storage backing service uses caching, providing **Vertical data replication** for request traces that query data from the database at some point (**Tea6**). For the deployment, the TeaStore offers container images and Kubernetes deployment descriptions which means **Standardized self-contained deployment units** should be available (**Tea7**). A deployment on Kubernetes was assumed and as a platform it supports **Built-in autoscaling** (**Tea8**), the deployment of **Immutable artifacts** (**Tea9**), **Automated restarts** (**Tea10**), and it can have **Rolling upgrades enabled** (**Tea11**). For the deployment of the Kubernetes cluster as an infrastructure, Terraform[50] was used which relates to the product factor **Use infrastructure as code** (**Tea12**).

The **LakesideMutual** application was created to investigate API patterns for microservices-based systems [270, 323]. Thus, also the product factors **Service-orientation** and **Service independence** are relevant (**Lak1**). A core focus of the application, however, is on the endpoints, in contrast to, for example, deployment aspects. Several patterns are described by the creators of the application [323] to characterize

[50] `https://developer.hashicorp.com/terraform`, visited 2025-10-20

the endpoints of an application regarding their roles and responsibilities. This is, however, not representable by the modeling approach apart from modeling the endpoints themselves with the properties relevant to the evaluation approach. One specific pattern built into the application that is representable to some extent, is API Key [270] which is related to the product factor **Access control management consistency** (**Lak2**). Another built-in characteristic is using rate limiting [270] for endpoints which contributes to the factor **Guarded ingress** (**Lak3**). Regarding communication between services, the LakesideMutual application also includes messaging-based **Asynchronous communication** (**Lak4**). **Health and readiness checks** are implemented using the Spring Boot Actuator[51] library (**Lak5**). And the endpoints of the services within the application are documented using OpenAPI[52], which supports the factor **API-based communication** (**Lak6**). For the deployment of the LakesideMutual application, it was decided to follow the suggestion of using a managed Kubernetes service. Thus, similar to the TeaStore, **Standardized self-contained deployment units** are available (**Lak7**) which can be used as **Immutable artifacts** (**Lak8**). Furthermore, Kubernetes supports **Built-in autoscaling** (**Lak9**), **Automated restarts** (**Lak10**), and there are **Rolling upgrades enabled** (**Lak11**).

Finally, the **Spring PetClinic Cloud** application is an extended fork of the original Spring PetClinic application. While the original PetClinic application was built to showcase features of the Spring Framework, the PetClinic Cloud application was built to showcase technologies and features for implementing distributed, cloud-based applications with Spring Cloud[53]. It can also be seen as a microservices-based application, thus also laying a focus on the factors **Service-orientation** and **Service independence** (**Pet1**). All three core services, Customers service, Vets service, and Visits service have their own data store, in conformance with the factor **Backing service decentralization** (**Pet2**). In addition, a configuration backing service based on Spring Cloud Config[54] enables **Configuration management** (**Pet3**). **Distributed tracing of invocations** is implemented in the application based on zipkin[55] (**Pet4**) together with a prometheus[56] backing service to which **Consistent centralized metrics** are exported (**Pet5**). Furthermore, service discovery is implemented with Eureka[57] to enable **Addressing abstraction** (**Pet6**). These are existing projects and thus the factor **Usage of existing solutions for non-core capabilities** is covered by their inclusion as well (**Pet7**). **Autonomous fault handling** is implemented through timeouts, retries and the circuit breaker pattern in links (**Pet8**). Finally, for the PetClinic application, a deployment using Docker Compose was

[51] `https://www.baeldung.com/spring-boot-actuators`, visited 2025-10-20
[52] `https://www.openapis.org/`, visited 2025-10-20
[53] `https://javaetmoi.com/2018/10/architecture-microservices-avec-spring-cloud/`, visited 2025-10-20
[54] `https://docs.spring.io/spring-cloud-config/docs/current/reference/html/`, visited 2025-10-20
[55] `https://zipkin.io/`, visited 2025-10-20
[56] `https://prometheus.io/`, visited 2025-10-20
[57] `https://cloud.spring.io/spring-cloud-netflix/reference/html/`, visited 2025-10-20

chosen which is covered by the factors **Standardized self-contained deployment unit** (**Pet9**) and **Immutable artifacts** (**Pet10**).

4.7.5.2. Evaluation Results

The data basis for the evaluation of the applications are their modeled software architectures. These have been created manually based on the source code and other available documentation or deployment artifacts. The models are available online[58] and used as examples for the graphical representation of the modeling approach in Section 5.3.2. Based on the models, evaluations have been done based on the presented quality model and the corresponding tooling. These evaluation results are presented in the following. The tables 4.42 to 4.50 contain the evaluation results for all three applications on a system entity level with the results for all product factors and corresponding measures of the quality model. They are ordered by the top-level quality aspects to which they can be assigned. If factors impact different factor hierarchies, they are included in the tables each time and can thus be duplicated. In addition, the key characteristics of the applications, as described in Section 4.7.5.1, are added to the corresponding product factors by their code.

As it can be seen in Table 4.42, several aspects regarding security are not taken into account by the applications. At least for **Data encryption in transit**, the reason why it is not supported is based on the assumed deployment model for the applications. It could be changed comparatively easily by adding a TLS certificate to the web server in each component and activating Hypertext Transfer Protocol Secure (HTTPS). Secrets, however, are mostly added to the components directly, if they are needed, which can be explained with the nature of the applications. They are all exemplary applications used in research or to showcase certain features, and they should, therefore, be easy to run and investigate and need not be operated securely. For the TeaStore and LakeSideMutual applications, however, some user management has been implemented and certain endpoints are therefore protected and can only be invoked by authorized users. The fact that API keys are used by LakesideMutual (Lak2) has also been recognized in the form that for all non-external endpoints that use authentication, the same approach, namely API key, is used. An access restriction is implemented for some endpoints by TeaStore and LakesideMutual. But the PetClinic application does not use access restriction at all. Although the TeaStore application includes an authentication service, it is not modeled as an explicit authentication backing service, because it also includes additional functionalities such as the shopping cart management. Therefore, the factor **Authentication delegation** is evaluated as none.

Regarding **Service-orientation**, all three applications show an evaluation as moderate (Tea1,Lak1,Pet1). Especially relevant for that factor is the sub-factor **Limited functional scope** with its two sub-factors **Limited data scope** and **Limited endpoint**

[58] `https://github.com/r0light/clounaq-evaluation`, visited 2025-10-20

Table 4.42.: Use case evaluation results in the context of security

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Confidentiality	neutral	neutral	neutral
	Data encryption in transit	none	none	none
	Ratio of external endpoints that support TLS	0.000	0.000	0.000
	Ratio of secured links	0.000	0.000	0.000
	Secrets management	none	none	n/a
	Secrets stored in specialized services	none	none	n/a
	Secrets stored in vault	0.000	0.000	n/a
	Isolated secrets	none	none	n/a
	Secrets externalization	0.000	0.000	n/a
	Integrity	neutral	neutral	neutral
	Access restriction	none	none	none
	Least-privileged access	low	low	none
	Access restricted to callers	0.202	0.069	0.000
Lak2	Access control management consistency	high	high	n/a
	Consistency of supported authentication methods of endpoints	1.000	1.000	n/a
	Consistency of supported authentication methods of external endpoints	1.000	1.000	n/a
	Accountability	neutral	neutral	neutral
	Account separation	low	low	low
	Ratio of unique account usage	0.182	0.077	0.071
	Consistent centralized logging	none	none	none
	Ratio of components or infrastructure nodes that export logs to a central service	0.000	0.000	0.000
	Authenticity	neutral	neutral	neutral
	Authentication delegation	none	none	none
	Ratio of delegated authentication	0.000	0.000	0.000

scope. Although the overall evaluations for the systems are mostly moderate, a more detailed evaluation is beneficial in this case as there are differences regarding the individual services. For example, within the TeaStore application, the Imageprovider service has an evaluation result of high for the factor **Limited data scope** (*Cohesion between endpoints based on data aggregate usage*: 1), while the Webui service is evaluated only as low (*Cohesion between endpoints based on data aggregate usage*: 0.315). This can be explained by considering that the Imageprovider service is, in fact, limited only to the management of images and has thus a higher cohesion, while the Webui service covers almost all data aggregates used in the system and presents the overall User Interface (UI) to all functionalities of the system, and thus it also uses all data aggregates. For the factor **Limited endpoint scope** the Auth service is evaluated as high (*Service interface usage cohesion*: 0.875), while the Persistence service is evaluated as low (*Service interface usage cohesion* 0.25). The measure calculates the interface cohesion based on how a service is invoked by other services And since the Auth service is only invoked by the Webui service, whereas the Persistence service is invoked by almost all other services, the differ-

ences in the cohesion can be explained. The factor **Isolated state** is implemented by the TeaStore and the PetClinic applications, but not by LakeSideMutual. The reason is that no explicit storage backing services are included, instead state is managed in-memory in the business services. This is indicated by Specialized stateful services being evaluated as none due to the *Ratio of specialized stateful services* being 0. This is nevertheless well documented in the application, and it is acknowledged that for an actual deployment separate data stores should be used.

However, LakeSideMutual is the only application with **Asynchronous communication** (Lak4) as indicated by the factor being evaluated as low. For the whole application, the measure *Asynchronous communication utilization* is rather low. Therefore, it is helpful to evaluate the different request traces to identify those with **Asynchronous communication**. These request traces are "Request Insurance Quote", "Respond to Insurance Quote" and "Accept Insurance Quote" (*Asynchronous communication utilization* has a value of 1 for these request traces).

In the context of **Reusability**, only for the TeaStore **Use infrastructure as code** is fulfilled, because only for it Terraform scripts were available with which the infrastructure could be provisioned in an automated way (Tea12). By evaluating the different parts of the infrastructure, it can be seen that *Ratio of infrastructure with IaC artifact* is 1 for the "Managed EC2 Node Group" and the "AWS EKS Cluster", but 0 for "AWS Load Balancing". This is because the last infrastructure entity cannot be provisioned through IaC, but is provisioned transparently by the cloud provider. It has to be noted, that for the other applications it would also be possible to add IaC scripts.

Further, considering the **Analyzability** of the applications (see Table 4.44), it can be seen that only the PetClinic application includes backing services for capturing **Consistent centralized metrics** (Pet5) and using **Distributed tracing of invocations** (Pet4). Which services include **Distributed tracing of invocations** can be evaluated more in detail by considering the evaluation results for the different components. Namely, the "Customer Service", the "Visits Service" and the "Vets Service" support it (*Distributed tracing support*: 1), while the storage backing services do not (*Distributed tracing support*: 0). However, **Health and readiness checks** are implemented by all three applications (Tea4, Lak5).

For the second product factor, specifically relevant for microservices-based architectures, **Service independence** (Tea1, Lak1, Pet1), it can be seen that coupling is highest (and thus **Low coupling** is evaluated only as moderate) for the TeaStore application. This is because often all services are required when the Webui service serves requests which is also visible by the consideration of the *Maximum number of services within a request trace* and *Average complexity of request traces* measures. In contrast, the PetClinic application shows the highest **Backing service decentralization** because each of the business services has its own dedicated storage backing service (Pet2).

Table 4.43.: Use case evaluation results in the context of maintainability - 1

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Modularity	neutral	neutral	neutral
Tea1,Lak1,Pet1	Service-orientation	moderate	moderate	moderate
	Limited functional scope	moderate	moderate	moderate
	Limited data scope	moderate	moderate	moderate
	Cohesion between endpoints based on data aggregate usage	0.568	0.437	0.459
	Limited endpoint scope	moderate	high	moderate
	Service interface usage cohesion	0.548	0.673	0.525
	Command query responsibility segregation	none	moderate	none
	Read write separation for data aggregates	0.000	0.400	0.000
	Separation by gateways	n/a	n/a	n/a
	Degree of separation by gateways	n/a	n/a	n/a
	Isolated state	moderate	none	high
	Mostly stateless services	moderate	moderate	high
	Ratio of stateless components	0.625	0.636	0.692
	Degree to which components are linked to stateful components	0.480	0.889	0.194
	Specialized stateful services	high	none	high
	Ratio of specialized stateful services	0.667	0.000	1.000
	Ratio of suitably replicated stateful services	n/a	n/a	n/a
	Loose coupling	none	none	none
Lak4	Asynchronous communication	none	low	none
	Degree of asynchronous communication	0.000	0.065	0.000
	Asynchronous communication utilization	0.000	0.220	0.000
	Communication partner abstraction	none	none	none
	Event sourcing utilization metric	0.000	0.000	0.000
	Reusability	positive	positive	positive
	Standardization	moderate	moderate	high
	Ratio of standardized artifacts	0.280	0.412	0.765
	Ratio of entities providing standardized artifacts	0.636	0.769	0.929
	Component similarity	moderate	moderate	high
	Component artifacts similarity	0.679	0.458	0.769
	Infrastructure artifacts similarity	0.333	n/a	n/a
Tea12	Use infrastructure as code	high	none	none
	Ratio of infrastructure with IaC artifact	0.667	0.000	0.000
	Cloud vendor abstraction	moderate	high	high
	Non-provider-specific infrastructure artifacts	0.500	n/a	n/a
	Non-provider-specific component artifacts	0.875	0.909	1.000

The factor **Addressing abstraction** is evaluated as high for all three applications, because service discovery is either provided by explicit services as in the case of TeaStore and PetClinic, or by Kubernetes as in the case of LakeSideMutual.

Table 4.44.: Use case evaluation results in the context of maintainability - 2

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Analyzability	neutral	neutral	mixed
	Communication partner abstraction	none	none	none
	Event sourcing utilization metric	0.000	0.000	0.000
	Automated monitoring	none	none	low
	Consistent centralized logging	none	none	none
	Ratio of components or infrastructure nodes that export logs to a central service	0.000	0.000	0.000
Pet5	Consistent centralized metrics	none	none	low
	Ratio of components or infrastructure nodes that export metrics	0.000	0.000	0.154
Pet4	Distributed tracing of invocations	n/a	n/a	moderate
	Distributed tracing support	n/a	n/a	0.333
Tea4,Lak5	Health and readiness checks	moderate	moderate	high
	Ratio of services that provide health endpoints	0.800	0.800	1.000
	Ratio of services that provide readiness endpoints	0.800	0.000	0.000
Tea6	Vertical data replication	low	none	moderate
	Ratio of cached data aggregates	0.308	0.000	0.500
	Consistently mediated communication	none	none	none
	Service mesh usage	0.000	0.000	0.000
	Modifiability	positive	slightly positive	slightly positive
	Automated infrastructure provisioning	high	high	none
	Ratio of automatically provisioned infrastructure	1.000	1.000	0.000
Tea12	Use infrastructure as code	high	none	none
	Ratio of infrastructure with IaC artifact	0.667	0.000	0.000
Tea1,Lak1,Pet1	Service independence	low	moderate	moderate
	Low coupling	moderate	high	high
	Degree of coupling in a system	0.214	0.127	0.179
	Functional decentralization	low	low	low
	Data aggregate spread	0.429	0.440	0.500
	Request trace similarity based on included components	0.594	0.500	0.556
	Limited request trace scope	low	moderate	moderate
	Maximum number of services within a request trace	6.000	2.000	3.000
	Average complexity of request traces	3.529	0.833	2.000
	Logical grouping	low	none	none
	Namespace separation	0.429	0.000	0.000
Pet2	Backing service decentralization	low	n/a	moderate
	Average weighted storage backend sharing	0.600	n/a	0.375
	Average weighted broker backend sharing	n/a	n/a	n/a
Tea2,Pet6	Addressing abstraction	high	high	high
	Service discovery usage	1.000	0.980	1.000

As shown in Table 4.45, LakeSideMutual is the only application that is evaluated different than none for the factor **API-based communication**. This is because it is an application focusing on API patterns and therefore the interfaces of the services are documented by OpenAPI[59] descriptions to a large extent (Lak6). The reason why *Ratio of documented endpoints* is not 1 can be found when evaluating the components individually. It can then be seen that the backend services provide documented APIs while the frontend components do not. For the factor **Usage of existing solutions for non-core capabilities**, PetClinic has the highest evaluation (Pet7), because it is a showcase application especially for backing services that are open-source and can be used within the Spring environment. The registry backing service in the TeaStore application is a custom implementation (*Ratio of non-custom backing services*: 0) which is why overall the system is evaluated only as moderate in this context.

Table 4.45.: Use case evaluation results in the context of maintainability - 3

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Testability	neutral	positive	slightly positive
	Mostly stateless services	moderate	moderate	high
	Ratio of stateless components	0.625	0.636	0.692
	Degree to which components are linked to stateful components	0.480	0.889	0.194
Lak6	API-based communication	none	high	none
	Ratio of documented endpoints	0.000	0.696	0.000
	Simplicity	neutral	mixed	neutral
	Command query responsibility segregation	none	moderate	none
	Read write separation for data aggregates	0.000	0.400	0.000
Pet7	Usage of existing solutions for non-core capabilities	moderate	none	high
	Ratio of non-custom backing services	0.667	0.000	1.000
	Sparsity	low	none	none
	Number of components	8.000	11.000	13.000
	Operation outsourcing	low	low	none
	Ratio of provider-managed components and infrastructure	0.364	0.154	0.000
	Managed infrastructure	low	low	none
	Ratio of fully managed infrastructure	0.333	0.500	0.000
	Managed backing services	low	none	none
	Ratio of managed backing services	0.333	0.000	0.000

Because for the TeaStore and LakeSideMutual applications a cloud based deployment was chosen, only for them, **Managed infrastructure** is evaluated as not none. However, the *Ratio of fully managed infrastructure* measure is slightly higher for the LakeSideMutual application because the managed Google Kubernetes Engine was used.

Continuing with product factors in the context of performance efficiency as shown in Table 4.46, it can be seen that **Service replication** is only fulfilled by the

[59] `https://www.openapis.org/`, visited 2025-10-20

Table 4.46.: Use case evaluation results in the context of performance efficiency

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Time-behavior	neutral	neutral	neutral
	Replication	low	none	low
Tea3	Service replication	moderate	none	none
	Amount of redundancy	1.000	1.000	1.000
	Service replication level	3.400	1.000	1.000
	Horizontal data replication	none	n/a	none
	Storage replication level	1.000	n/a	1.000
Tea6	Vertical data replication	low	none	moderate
	Ratio of cached data aggregates	0.308	0.000	0.500
	Sharded data store replication	none	n/a	none
	Level of sharding across storage backing services	1.000	n/a	1.000
	Consistently mediated communication	none	none	none
	Service mesh usage	0.000	0.000	0.000
	Resource utilization	positive	positive	positive
	Dynamic scheduling	high	high	high
	Ratio of components deployed dynamically	1.000	1.000	1.000
	Enforcement of appropriate resource boundaries	moderate	low	none
	Ratio of infrastructure entities enforcing resource boundaries	0.333	0.500	0.000
	Ratio of deployment mappings with stated resource requirements	0.778	0.000	0.000
	Elasticity	positive	positive	slightly positive
	Isolated state	moderate	none	high
	Mostly stateless services	moderate	moderate	high
	Ratio of stateless components	0.625	0.636	0.692
	Degree to which components are linked to stateful components	0.480	0.889	0.194
	Specialized stateful services	high	none	high
	Ratio of specialized stateful services	0.667	0.000	1.000
	Ratio of suitably replicated stateful services	n/a	n/a	n/a
Tea8,Lak9	Built-in autoscaling	high	high	none
	Deployed entities autoscaling	1.000	1.000	0.000
	Infrastructure autoscaling	0.333	0.500	0.000

TeaStore application where a replicated deployment was possible (Tea3) and thus also chosen. The database, however, is not replicated (*Storage replication level*: 1). For the factor **Vertical data replication** the *Ratio of cached data aggregates* is slightly higher for the PetClinic application. Although caching is applied in a similar way in these applications, the TeaStore application overall has a larger range of data aggregates which cannot be cached in all contexts. It is beneficial in this case to analyze individual request traces. For example, one of the core request traces "Index Page" in the TeaStore application does have a *Data replication along request trace* of 0.733 which is actually evaluated as high (Tea6). The factors **Enforcement**

of appropriate resource boundaries and **Built-in autoscaling** are evaluated as supported, although to a different extent, only by the TeaStore and the LakeSideMutual applications, because their deployment is based on Kubernetes as a platform (Tea8,Lak9).

Table 4.47.: Use case evaluation results in the context of portability

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Adaptability	positive	neutral	positive
Tea12	Use infrastructure as code	high	none	none
	Ratio of infrastructure with IaC artifact	0.667	0.000	0.000
	Infrastructure abstraction	high	high	high
	Ratio of abstracted hardware	0.667	1.000	1.000
	Cloud vendor abstraction	moderate	moderate	moderate
	Non-provider-specific infrastructure artifacts	0.000	0.000	0.000
	Non-provider-specific component artifacts	0.875	0.909	1.000
Pet3	Configuration management	moderate	none	high
	Isolated configuration	high	none	high
	Configuration externalization	1.000	0.000	1.000
	Configuration stored in specialized services	none	none	high
	Configuration stored in config service	0.000	0.000	1.000
	Contract-based links	none	none	none
	Ratio of endpoints covered by contract	0.000	0.000	0.000
	Installability	positive	positive	positive
	Automated infrastructure provisioning	high	high	none
	Ratio of automatically provisioned infrastructure	1.000	1.000	0.000
Tea7,Lak7,Pet9	Standardized self-contained deployment unit	moderate	moderate	high
	Standardized deployments	0.000	0.000	1.000
	Self-contained deployments	0.875	0.900	0.923
	Replaceability	positive	neutral	positive
	Isolated state	moderate	none	high
	Mostly stateless services	moderate	moderate	high
	Ratio of stateless components	0.625	0.636	0.692
	Degree to which components are linked to stateful components	0.480	0.889	0.194
	Specialized stateful services	high	none	high
	Ratio of specialized stateful services	0.667	0.000	1.000
	Ratio of suitably replicated stateful services	n/a	n/a	n/a
Tea2,Pet6	Addressing abstraction	high	high	high
	Service discovery usage	1.000	0.980	1.000
Tea9,Lak8,Pet10	Immutable artifacts	high	high	high
	Replacing deployments	0.889	1.000	1.000

Regarding factors in the context of portability (see Table 4.47), the factor **Configuration management** is evaluated the highest for the PetClinic application. It externalizes configuration and includes a specialized configuration service (Pet3)

where configuration data is stored and can be updated. TeaStore at least stores configuration externally in the Kubernetes environment. For the LakeSideMutual application, the factor is evaluated as none, because configuration is directly included in the services for simplicity.

All three applications, however, use containers for the deployment, thus supporting **Standardized self-contained deployment units**. But only the PetClinic application uses the standardized (OCI) container images directly for deployment (Pet9) while Kubernetes pods are used for the deployment in TeaStore and LakeSideMutual (inside the pods containers are nevertheless used as well)(Tea7,Lak7). Containers and pods are both **Immutable artifacts** if deployed accordingly on Docker or Kubernetes. Thus, they are replaced when updated instead of applying changes to the deployed artifacts (Tea9,Lak8,Pet10).

Table 4.48 covers factors in the context of portability with **Availability** as a quality aspect with several impacting factors. To ensure **Seamless upgrades**, rolling upgrades can be used as described by the factor **Rolling upgrades enabled**. These need to be available from the infrastructure on which components are running which is the case for Kubernetes (Tea11,Lak11), but not when using Docker directly. Apart from the infrastructure supporting rolling upgrades (*Rolling update option*), it also needs to be actively used for components, e.g. in their deployment artifacts (*Rolling updates*). It was not assumed to be used for the evaluation, but could nevertheless be added. Several factors that have been considered in the context of **Time-behavior** already, are also relevant for **Availability**, but are therefore not considered in detail again. One additional factor, however, is **Guarded ingress**. By guarding the ingress of an application, it can be protected from malicious input that may, for example, overload the application and thus its Availability can be ensured. The basis is that external ingress is proxied and thus a central place exists where protection mechanisms can be applied. This is the case for TeaStore and LakeSideMutual, where Kubernetes services are only reachable via a load balancer from the outside. LakeSideMutual in addition uses a specific mechanism for guarding, namely rate limiting (Lak3). Furthermore, **Availability** can be impacted positively by deploying an application across different availability zones (corresponding to different data centers). In the current state, multiple availability zones are used for both services and storage backing services only in the case of TeaStore. But by adapting the infrastructure, this could be enabled for the other applications as well. Also focusing on the infrastructure, the factor **Automated infrastructure maintenance** is evaluated as high for the TeaStore and LakeSideMutual applications. This is because managed virtual machines are used for running the corresponding Kubernetes clusters. In the case of the TeaStore with a managed node group and in the case of LakeSideMutual with a completely managed Kubernetes cluster.

Also in the context of reliability, evaluations for the quality aspects **Fault tolerance** and **Recoverability** are shown in Table 4.49. **Autonomous fault handling** is specifically considered by the TeaStore and the PetClinic, because timeouts and re-

Table 4.48.: Use case evaluation results in the context of reliability - 1

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Availability	positive	neutral	neutral
	Seamless upgrades	none	none	none
	Separation by gateways	n/a	n/a	n/a
	Degree of separation by gateways	n/a	n/a	n/a
Tea11,Lak11	Rolling upgrades enabled	low	low	none
	Rolling update option	0.500	0.500	0.000
	Rolling updates	0.000	0.000	0.000
Tea4,Lak5	Health and readiness checks	moderate	moderate	high
	Ratio of services that provide health endpoints	0.800	0.800	1.000
	Ratio of services that provide readiness endpoints	0.800	0.000	0.000
Tea3	Service replication	moderate	none	none
	Amount of redundancy	1.000	1.000	1.000
	Service replication level	3.400	1.000	1.000
	Horizontal data replication	none	n/a	none
	Storage replication level	1.000	n/a	1.000
Tea6	Vertical data replication	low	none	moderate
	Ratio of cached data aggregates	0.308	0.000	0.500
	Enforcement of appropriate resource boundaries	moderate	low	none
	Ratio of infrastructure entities enforcing resource boundaries	0.333	0.500	0.000
	Ratio of deployment mappings with stated resource requirements	0.778	0.000	0.000
Tea8,Lak9	Built-in autoscaling	high	high	none
	Deployed entities autoscaling	1.000	1.000	0.000
	Infrastructure autoscaling	0.333	0.500	0.000
Lak3	Guarded ingress	moderate	moderate	none
	Ratio of components whose external ingress is proxied	1.000	0.875	n/a
	Ratio of rate limiting endpoints	0.000	0.214	0.000
	Distribution	moderate	none	none
	Physical data distribution	moderate	n/a	none
	Number of availability zones used by storage services	3.000	n/a	1.000
	Physical service distribution	moderate	low	none
	Number of availability zones used by services	3.000	2.000	1.000
	Automated infrastructure maintenance	high	high	none
	Ratio of automatically maintained infrastructure	1.000	1.000	0.000

tries are used for service invocations (Tea5, Pet8), although not for all invocations within the PetClinic application. LakeSideMutual also includes an example of a circuit breaker, but only in the "Customer Self-Service Backend service" (*Ratio of links with complex failover*: 0.250 for this service). **Automated restarts** are supported by Kubernetes and thus in TeaStore and LakeSideMutual (Tea10,Lak10). In

the TeaStore application, the measure *Deployments with restart* is slightly lower, because it considers all deployment mappings and the deployment mapping from the infrastructure entity "AWS EKS Cluster" on the infrastructure entity "Managed EC2 Node Group". The "Managed EC2 Node Group", however, does not include an automated restart policy for the Kubernetes processes running on its associated virtual machines.

Table 4.49.: Use case evaluation results in the context of reliability - 2

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Fault tolerance	positive	neutral	neutral
	Persistent communication	none	none	none
	Service interaction via backing service	0.000	0.000	0.000
	Event sourcing utilization metric	0.000	0.000	0.000
Tea5,Pet8	Autonomous fault handling	high	none	none
	Invocation timeouts	high	none	low
	Ratio of links with timeout	1.000	0.000	0.268
	Retries for safe invocations	high	none	low
	Ratio of links with retry logic	1.000	0.000	0.268
	Circuit breaked communication	none	low	low
	Ratio of links with complex failover	0.000	0.026	0.268
	Recoverability	positive	positive	neutral
Tea12	Use infrastructure as code	high	none	none
	Ratio of infrastructure with IaC artifact	0.667	0.000	0.000
	Automated infrastructure maintenance	high	high	none
	Ratio of automatically maintained infrastructure	1.000	1.000	0.000
Tea10,Lak10	Automated restarts	high	high	none
	Deployments with restart	0.889	1.000	0.000
Tea4,Lak5	Health and readiness checks	moderate	moderate	high
	Ratio of services that provide health endpoints	0.800	0.800	1.000
	Ratio of services that provide readiness endpoints	0.800	0.000	0.000

Finally, in the context of compatibility (see Table 4.50) only the factor **API-based communication** is evaluated as not none. It was already considered in the context of **Testability** where only the LakeSideMutual application provides documentation via OpenAPI.

To summarize the results of the evaluations, it can be stated that the key characteristics of the applications are recognizable through the quality evaluation approach. Although the measures reported in the presented tables only evaluate the applications on the level of complete systems, this already provides an overview. For a more detailed evaluation, the possibility to evaluate individual components or request traces is essential, as shown for selected evaluation results.

The evaluation approach, however, is not able to cover all details of the applications. For example, the characteristic that the TeaStore application relies on client-side load balancing [293] is currently not representable with the evaluation ap-

Table 4.50.: Use case evaluation results in the context of compatibility

Characteristics	Factor/*Measure*	TeaStore	LakeSideMutual	PetClinic
	Interoperability	neutral	neutral	neutral
	Contract-based links	none	none	none
	Ratio of endpoints covered by contract	0.000	0.000	0.000
Lak6	API-based communication	none	high	none
	Ratio of documented endpoints	0.000	0.696	0.000
	Consistently mediated communication	none	none	none
	Service mesh usage	0.000	0.000	0.000

proach. Or, as another example, specific details about message or payload data that is sent through links is also not evaluable with the approach, although for example several patterns regarding message characteristics are implemented in the LakeSideMutual application [270]. These aspects are abstracted in order to keep the evaluation approach focused on those factors found to be relevant for cloud-native software architectures from the literature.

Thus, also aspects were application architectures could be improved according to certain quality aspects are uncovered in the evaluation. **Configuration management** and especially **Secrets management** are factors based on which application architectures could be refactored. It is, however, acknowledged that the chosen applications are exemplary applications with specific purposes for learning or research and thus production readiness was not a major concern. Furthermore, there is potential to improve the factor **Automated monitoring**. For example, by including a centralized state-of-the-art logging approach [149] for all three applications or adding distributed tracing support for the TeaStore and the LakeSideMutual applications. If performance aspects are important, the factors **Horizontal data replication** or **Sharded data store replication** could be considered to evolve the way storage backing services are used in the applications. Finally, more specific aspects such as **Communication partner abstraction** through event sourcing, **Command query responsibility segregation**, or **Consistently mediated communication** through introducing a service mesh can be considered for the applications, if they fit the domain of the corresponding applications and could bring desired benefits. However, these last aspects would represent major refactorings. In the end, the reported evaluation should function as an information basis for the further evolution of software architectures. Specific decisions and implementations need to be made by those developing applications who are also familiar with the business goals and requirements of applications.

> Reuse of existing software development assets is an essential part of any software development process
>
> *Tony Wasserman [297]*

5. A Modeling Approach for Cloud-native Software Architectures

Parts of this chapter have been taken from [71, 169].

In this chapter, hypothesis 3 *("An architectural model can be used to express cloud-native application characteristics.")* is supported.

In this chapter the modeling approach for representing cloud-native software architectures is presented in detail. As described in Section 3.1.4, the aims of the modeling approach are to (1) represent software architectures aligned to the entities of the quality model, so that the quality model can be applied to the corresponding architectural models, (2) provide a structured textual format for automated processing and storage, (3) provide a graphical format for intuitive modeling and understanding, and, if possible, (4) enable an integration with other modeling, development, or evaluation approaches. To formulate a suitable modeling approach, existing modeling languages were reviewed in order to identify a suitable approach to build on. The results of this review are discussed in Section 5.1 based on which TOSCA (see Section 2.3.5) was chosen as a modeling basis. The thus developed TOSCA profile is presented in Section 5.2. It is the structured textual format for representing software architectures in the context of the quality model. Furthermore, the chosen graphical representation in accordance with the textual modeling approach is presented in Section 5.3.

5.1. Modeling Language Evaluation Results

To identify a suitable modeling approach to build on, the results from a structured review on existing ADLs for cloud-based applications were used [70, 71]. This review searched and compared existing modeling approaches based on characteristics explicitly formulated to be aligned with the aims of the quality evaluation approach. Based on the search results a selection of the most suitable languages was made. These languages are TOSCA [208], CloudML [87], CAMEL [2], LEMMA [234], and Context Mapper [132] (see also Section 2.3.4). For this selection, a more detailed comparison was made which is summarized in Table 5.1. The ADLs were compared according to criteria focusing on general characteristics, characteristics specific to the quality model and its entities that need to be represented. Additional aspects cover the usage of the language for modeling and evaluation.

As it can be seen in Table 5.1 the *purposes* of the languages cover a range of different aspects with deployment nevertheless being the most commonly covered purpose. TOSCA has an application centric *focus* meaning that cloud deployment options can be found and modeled that explicitly match with specific application characteristics. CloudML and CAMEL have a multi-cloud focus. That means their goal is to model and deploy applications consistently while also supporting multiple different cloud providers. LEMMA and Context Mapper were specifically developed to support microservices-based architectures and thus put an additional focus on how individual services can be designed and how they interact with each other. All languages have at least a textual *syntax*, with TOSCA, CloudML, and CAMEL also including a graphical syntax. The textual syntaxes are based on an *abstract syntax* which is mostly described using Ecore (MOF)[60].

The *semantic* criterion describes how meaning is given to the elements of a specific model. This can be simply through *prose*, *operational* if a corresponding tool processes a model and performs actions based on it, for example, deploying an application, or *translational*, if the model is translated into a different form, such as executable source code. For TOSCA, CloudML, and CAMEL the main semantic is operational, because deployments are realized based on the models. For LEMMA and Context Mapper, a translation to source code for the modeled services is the main use case. Because architectures of distributed cloud applications can become complex, possibilities to handle this *complexity* are beneficial. So far, only TOSCA and Context Mapper provide mechanisms to substitute or refine elements of a model in order to adjust the level of detail which is in consideration. *Reusability* is evaluated based on the possibilities to refer to whole models or parts of other models so that certain parts of architectures do not need to be modeled again and again, but can be reused. Concluding the general criteria, the *typing mechanism* captures how instances and new types can be specified within a language. *Linguistic* typing refers to the possibility of typing model instances based on a metamodel for a language while *ontological* typing refers to the possibility of typing model instances based on other model instances. If both typing mechanisms are available, a language is more flexible in terms of extensibility.

More relevant to the quality model, the criteria *Application Structure* and *Cloud Environment* describe which kind of entities can be modeled with the specific language. Overall, the languages cover a similar set of entities. Differences are that, for example, CloudML does not have an entity specifically for storage or Context Mapper does not cover infrastructure entities and hosting relationships. *Modeling Support* and *Analysis Support* cover the functionalities of the tooling available with the languages and how it is possible to model and analyze architectures modeled with the respectively languages using these tools. With the criterion *Number of Entities* it is counted how many entities of the quality model could be represented at the time of the evaluation. The value in parentheses includes also entities that can only be represented partly. The thirteen entities at the time of the evaluation were

[60] `https://www.omg.org/spec/MOF`, visited 2025-10-20

Table 5.1.: Summary of the ADL review [71]

Criterion	TOSCA (v1.3)	CloudML (v2.1)	CAMEL (v2.5)	LEMMA (v0.8.5)	Context Mapper (v6.0.0)
Purpose	deployment & provisioning	deployment & provisioning	entire application lifecycle	development & deployment	design & integration
Focus	appl.-centric	multi-cloud	multi-cloud	MSA MDD	MSA DDD
Abstract Syntax	XML Schema	Ecore (MOF)	Ecore (MOF)	Xcore (MOF) | UML	Ecore (MOF) | UML
Concrete Syntax	textual (YAML, XML) | graphical	textual (JSON, XMI) | graphical	textual (XML) | graphical	textual (DSL, XText)	textual (XMI, DSL, XText)
Semantic	english prose, operational	operational	english prose, operational	english prose, translational	english prose, translational
Complexity Reduction	Node Template Substitution (✓)	✗	✗	✗	BoundedContext Refinement (○)
Reusability	medium	high	high	medium	medium
Typing Mechanism	ontological & linguistic	ontological & linguistic	ontological & linguistic	linguistic	linguistic
Application Structure	C, E, R, S	C, E, R	C, E, R, S, D	C, E, D	C, E, R, D
Cloud Environment	ES, I, H	ES, I, H	ES, I, H	ES, I, H	ES
Modeling Support	graphical	textual & graphical	textual	textual	textual
Analysis Support	deployment	deployment & monitoring	deployment & monitoring	design	design
Number of Entities	9 (10) / 13	7 (9) / 13	9 (10) / 13	8 (9) / 13	6 / 13
Grouping Possibilities	✓	✗	○	○	✗
Other Evaluations	○	○	✗	✓	○
Metrics	○	✗	Metric Model	Static Analyzer	✗

✗ not included | ○ partly included | ✓ fully included
(C) Component, (E) Endpoint, (R) Relationship, (S) Storage, (D) Data, (ES) External Services, (I) Infrastructure, (H) Hosting

System, Component, Service, Endpoint, External Endpoint, Backing Service, Storage Backing Service, Link, Infrastructure, Deployment Mapping, Request Trace, Data Aggregate, and Backing Data. Furthermore, *grouping mechanisms* are also relevant in the context of handling complexity, a full support, however, was only found for TOSCA which provides the possibility to assign entities to an explicit group entity.

Finally, *Other Evaluations* and *Metrics* review whether the languages were already used in contexts similar to the aim of the quality evaluation approach considered in this work.

To summarize the ADL evaluation results, it can be said that all the selected languages could be used as a basis for the modeling approach needed in the con-

text of this work. TOSCA and CAMEL, however, provide the most suitable basis, also for example regarding the number of representable entities from the quality model. Languages such as TOSCA, CloudML, and CAMEL focus more on the level of cloud infrastructure due to their focus on the deployment of applications, while providing fewer possibilities to represent details of individual component or on how components communicate. LEMMA and Context Mapper, in contrast, focus more on individual components and their interactions. Because the formulated quality model covers both the infrastructure level and the communication between components in the form of links and request traces, an extension of the chosen language is necessary, independently of which language is chosen.

Based on the evaluation results, it was decided to use TOSCA as a basis for the modeling approach in this work. Although CAMEL might have been a reasonable choice as well, TOSCA was chosen, because it is a standard supported by a larger community and because of its extensibility that is implicitly built in through the possibility of creating own TOSCA types. It has to be noted, that the evaluation was based on the versions of the languages available at that time, as also specified in Table 5.1. Further developments and aspects introduced with later versions of the languages might also influence the evaluation results, but could not be considered for this evaluation.

5.2. CNA Modeling TOSCA Profile

The initial TOSCA extension as a modeling approach for the entities of the quality model [71] was developed based on the TOSCA Simple Profile in YAML Version 1.3[61]. This profile included predefined Node Types for cloud applications, such as `tosca.nodes.Compute` which were reused where possible or from which new types were derived to specify types for the entities of the quality model. The idea was to enable an interoperability with tools who rely on the same basic types. However, during the work on the modeling approach, TOSCA 2.0 was pushed forward as the new standard. With the 2.0 version, the predefined Node Types of 1.3 were deprecated, because they were formulated with a focus on cloud resources assignable to the IaaS service model and showed to be not well adapted to newer cloud resources. Therefore, the initially formulated TOSCA extension was migrated to TOSCA 2.0 which meant formulating a custom TOSCA profile for the representation of the quality model entities (see Section 2.3.5). Overall, as described in Section 3.1.4, the profile is the result of the initial formulation [71], refined by a use case-based evaluation [145], and refined to enable the application of the evaluation approach [173]. This profile is available online[62] and is in part presented in the following.

[61] `https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html`, visited 2025-10-20

[62] `https://github.com/r0light/cna-modeling-tosca-profile`, visited 2025-10-20

The profile contains TOSCA type definitions which enable the modeling of software architectures according to the entities of the quality model (see Section 4.3). The possibility of defining `properties` was used to include more detailed information about entities in the form of the entity properties described in tables 4.2 to 4.6. First, the capability types are described in Section 5.2.1, because they are needed for formulating relationships between nodes. Then, as the largest part of the profile, node types are presented in Section 5.2.2 and relationship types in Section 5.2.3. Finally, artifact types are presented in Section 5.2.4.

5.2.1. Capability Types

In total, seven capability types are defined in the profile. These are:

- **cna-modeling.capabilities.Endpoint** to indicate that an endpoint is offered which can be invoked by others.
- **cna-modeling.capabilities.Host** to indicate that a node can host another node.
- **cna-modeling.capabilities.DataUsage** to indicate that data is used by another node.
- **cna-modeling.capabilities.Proxy** to indicate that a node can proxy communication of another node.
- **cna-modeling.capabilities.AdressResolution** to indicate that a node can provide address resolution to another node.
- **cna-modeling.capabilities.Authentication** to indicate that a node can provide authentication functions to another node.
- **cna-modeling.capabilities.Linkable** to indicate that a node can be linked to from another node.

Most of the capability types are just used as a marker in nodes so that relationships can be formulated with these nodes as target nodes. Thus, their definition is rather short, as in the example of the `Host` capability shown in Listing 5.1.

Listing 5.1: The Host capability definition

```
1 cna-modeling.capabilities.Host:
2  description:  >-
3  Indicates that the node can provide hosting.
```

However, capability types can also be extended with additional `properties` to include further information. This is done if a property is very specific to a certain capability, as shown in Listing 5.2 for the `AddressResolution` capability with the `address_resolution_kind` property. This differentiation of how properties are assigned, however, is only done in TOSCA, while in the formal specification of the entities in Section 4.3, the property of the capability is assigned to the entity which corresponds to the node type which has this capability.

Listing 5.2: The AddressResolution capability definition

```
cna-modeling.capabilities.AdressResolution:
 description:  When included, the Node is able to provide address resolution,
   meaning that symbolic names can be translated into specific IP addresses
 properties:
address_resolution_kind:
 description:  If address resolution is provided, state the kind here, otherwise
   leave as 'none'
 type:  string
 validation:  { $valid_values: [ $value, [ "none", "discovery", "DNS", "other" ]]}
 default:  none
```

5.2.2. Node Types

The node types are used to model the entities of the quality model. In total, there are 13 node types defined in the profile:

- **cna-modeling.entities.Component** to represent the Component entity.
- **cna-modeling.entities.Service** to represent the Service entity. It derives from `Component`.
- **cna-modeling.entities.BackingService** to represent the Backing Service entity. It derives from `Component`.
- **cna-modeling.entities.ProxyBackingService** to represent the Proxy Backing Service entity. It derives from `Component`.
- **cna-modeling.entities.BrokerBackingService** to represent the Broker Backing Service entity. It derives from `Component`.
- **cna-modeling.entities.StorageBackingService** to represent the Storage Backing Service entity. It derives from `Component`.
- **cna-modeling.entities.Infrastructure** to represent the Infrastructure entity.
- **cna-modeling.entities.Endpoint** to represent the Endpoint entity.
- **cna-modeling.entities.Endpoint.External** to represent the External Endpoint entity. It derives from `Endpoint`.
- **cna-modeling.entities.BackingData** to represent the Backing Data entity.
- **cna-modeling.entities.DataAggregate** to represent the Data Aggregate entity.
- **cna-modeling.entities.RequestTrace** to represent the Request Trace entity.
- **cna-modeling.entities.Network** to represent the Network entity.

When a node type is derived from another node type, it adopts its `properties`, `capabilities`, and `requirements`. This is especially relevant for the `Component` type and those deriving from it, because most things are defined once in the `Component` type definition and then simply adopted by the others through derivation.

Some entities are defined as node types, although they would not be considered as such from the perspective of TOSCA. In TOSCA nodes are components of a system that are deployed and managed by a TOSCA orchestrator, e.g. a container with a web service. However, the entities endpoint, external endpoint, backing data, and data aggregate are not components which are deployed, but instead they are part of a component and they would be managed together with a component by a TOSCA orchestrator. Endpoints, for example, were modeled only as capabilities in the TOSCA Simple Profile. But capabilities cannot be uniquely identified which would make it impossible to model a link relationship from a component to a specific endpoint, but only from one component to another. Therefore, node types for endpoint and external endpoint entities were formulated and the fact that an endpoint is part of a component needs to be modeled additionally.

Listing 5.3 shows an excerpt of the `Component` node type definition.
For `properties`, TOSCA offers the possibility to define a data `type`, a `default` value and a specification whether the property is `required` or not.

Listing 5.3: The Component node type (excerpt)

```
cna-modeling.entities.Component:
 description: Node Type to model Component entities
 properties:
software_type:
type: string
 description: The type of the software in the sense of who developed it. If it is
  a self-written component use "custom", if it is an existing open-source solution
   which is not customized (apart from configuration) use "open-source". If it is
  licensed proprietary software, use "proprietary".
 default: custom
 required: true
 validation: { $valid_values: [ $value, [ "custom", "open-source", "proprietary"
  ]]}
stateless:
type: boolean
 description: True if this component is stateless, that means it requires no disk
  storage space where data is persisted between executions. That means it can
  store data to disk, but should not rely on this data to be available for
  following executions. Instead, it should be able to restore required data after
  a restart in a different environment.
 default: true
 required: true
   [...]
 requirements:
- host:
 capability: cna-modeling.capabilities.Host
 node: cna-modeling.entities.Infrastructure
 relationship: cna-modeling.entities.HostedOn.DeploymentMapping
# Allow the definition of endpoints provided by this Component
- provides_endpoint:
 capability: cna-modeling.capabilities.Endpoint
 relationship: cna-modeling.relationships.Provides.Endpoint
count_range: [0, UNBOUNDED]
 [...]
```

Furthermore, there are advanced possibilities for defining a `validation` function. For `string` values, this can be used, for example, to limit the potential

values to an enumeration of values, as done for the `software_type` property in Listing 5.3. Two exemplary `requirements` are shown for the `Component` typed definition. Based on these it is possible to model `relationships` to other nodes by matching a `requirement` with a `capability` of another node as described in the `requirement` definition. For example, for the `host` requirement a node of type `cna-modeling.entities.Infrastructure` is needed that can fulfill this requirement with a capability of type `cna-modeling.capabilities.Host` (see Listing 5.1). The relationship can then be modeled, as shown in Listing 5.6, with a `cna-modeling.entities.HostedOn.DeploymentMapping` relationship type.

The corresponding host capability definition is shown in Listing 5.4 for the `Infrastructure` node type. It can be seen that the `Component` node type can be hosted on an `Infrastructure`. But also an `Infrastructure` node can be hosted on another `Infrastructure` node. This is enabled by also specifying a `host` requirement for the `Infrastructure` node type.

Listing 5.4: The Infrastructure node type (excerpt)

```
cna-modeling.entities.Infrastructure:
 description: Node Type to model Infrastructure entities
 properties:
kind:
type: string
 description: The kind of infrastructure. Possible kinds are "physical hardware",
   "virtual hardware", "software platform", or "cloud service".
 required: true
 validation: { $valid_values: [ $value, [ "physical-hardware", "virtual-hardware",
   "software-platform", "cloud-service"]]}
 default: virtual-hardware
environment_access:
type: string
 description: Describes the extent of available access to the environment in which
    the infrastructure is operated. With full access, one can control all aspects
   of the infrastructure. Limited access means that infrastructure is under control
    of a provider and only certain things are allowed, such as configuration. With
   no access infrastructure is completely managed by a cloud provider.
 validation: { $valid_values: [ $value, ["full", "limited", "none"]]}
 default: full
   [...]
 capabilities:
host:
type: cna-modeling.capabilities.Host
valid_source_node_types:
- cna-modeling.entities.Component
- cna-modeling.entities.Service
- cna-modeling.entities.BackingService
- cna-modeling.entities.StorageBackingService
- cna-modeling.entities.ProxyBackingService
- cna-modeling.entities.BrokerBackingService
- cna-modeling.entities.Infrastructure
[...]
 requirements:
# Allow the deployment on another Infrastructure entity
- host:
 capability: cna-modeling.capabilities.Host
node: cna-modeling.entities.Infrastructure
 relationship: cna-modeling.entities.HostedOn.DeploymentMapping
count_range: [0, UNBOUNDED]
# Allow the definition of Backing Data usage
- uses_backing_data:
 capability: cna-modeling.capabilities.DataUsage
node: cna-modeling.entities.BackingData
 relationship: cna-modeling.relationships.AttachesTo.BackingData
count_range: [0, UNBOUNDED]
[...]
```

As a last example for a node type definition, Listing 5.5 shows an excerpt of the `BackingService` node type definition. Here, an important aspect is the keyword `derived_from` which specifies that the `BackingService` node type also has all `properties`, `requirements`, and `capabilities` of the `Component` node type. The `properties` and `capabilities` defined in the `BackingService` node type are added to them.

Listing 5.5: The BackingService node type (excerpt)

```
cna-modeling.entities.BackingService:
derived_from: cna-modeling.entities.Component
 description:  Node Type to model Backing Service entities
 properties:
providedFunctionality:
type: string
 description:  "The functionality provided by this backing service"
 required:  true
 validation:  { $valid_values: [ $value, [ "naming/addressing", "configuration", "
   authentication/authorization", "logging", "metrics", "tracing", "vault", "other"
    ]]}
 default:  other
replication_strategy:
type: string
 description:  The strategy used to replicate data, if multiple replicas of this
   backing service are deployed.
 required:  true
 default:  none
 validation:  { $valid_values: [ $value, [ "none", "read-only-replication", "active
   -active-replication" ]]}
 capabilities:
address_resolution:
type: cna-modeling.capabilities.AdressResolution
valid_source_node_types:
- cna-modeling.entities.Component
- cna-modeling.entities.Service
- cna-modeling.entities.BackingService
- cna-modeling.entities.StorageBackingService
- cna-modeling.entities.ProxyBackingService
- cna-modeling.entities.BrokerBackingService
[...]
```

5.2.3. Relationship Types

A TOSCA model is a graph where nodes are the vertices and relationships are the edges. Relationship types are thus necessary to specify the edges of the graph. The profile contains ten relationship types, listed here:

- **cna-modeling.relationships.ConnectsTo.Link** to represent the link entity, connecting a component to an endpoint.
- **cna-modeling.relationships.HostedOn.DeploymentMapping** to represent the deployment mapping entity, connecting a component or infrastructure entity to an infrastructure entity.
- **cna-modeling.relationships.Provides.Endpoint** to specify that a component provides an endpoint.
- **cna-modeling.relationships.AttachesTo.DataAggregate** to specify that a component uses a data aggregate.

- **cna-modeling.relationships.AttachesTo.BackingData** to specify that a component or infrastructure uses a backing data.
- **cna-modeling.relationships.ProxiedBy.ProxyBackingService** to specify that a component is proxied by a proxy backing service.
- **cna-modeling.relationships.LinksTo** to specify that a component or infrastructure is linked to a network.
- **cna-modeling.relationships.UseAddressResolution** to specify that a component uses a backing service, network, or infrastructure for address resolution.
- **cna-modeling.relationships.DelegateAuthentication** to specify that a component uses a backing service to delegate authentication functions.
- **cna-modeling.relationships.PartOf** to specify that an external endpoint is part of a request trace.

The first two relationship types `Link` and `DeploymentMapping` correspond to the entities link and deployment mapping while the other relationships are needed to model the facts that components can provide endpoints , use data aggregates or backing data and to specify other specific relationships to backing services. Relationships can also be extended with `properties`, which is shown, for example, in Listing 5.6 with the properties `deployment` and `replicas`.

With `valid_capability_types` and `valid_target_node_types` it is possible to limit the targets of the relationship types to make them specific for certain use cases. As it can be seen in Listing 5.7, both `cna-modeling.entities.Endpoint` and `cna-modeling.entities.Endpoint.External` are valid target node types for the `Link` relationship, meaning that links are formulated from components to endpoints.

Listing 5.6: The DeploymentMapping relationship definition

```
1  cna-modeling.relationships.HostedOn.DeploymentMapping:
2   description:  Relationship Type to model DeploymentMapping entities
3   properties:
4  deployment:
5  type:  string
6   required:  true
7   description:  How this deployment mapping is described or ensured. Valid values
      can be manual, automated imperative, or automated declarative
8   validation:  { $valid_values: [ $value, [ "manual", "automated-imperative", "
      automated-declarative", "transparent"]]}
9   default:  manual
10 replicas:
11 type:  integer
12  description:  The minimum number of replicated instances for the deployed
      component when it is running
13  required:  true
14  default:  1
15    [...]
16 valid_source_node_types: [ cna-modeling.entities.Component, cna-modeling.entities
      .Service, cna-modeling.entities.BackingService, cna-modeling.entities.
      StorageBackingService, cna-modeling.entities.ProxyBackingService, cna-modeling
      .entities.BrokerBackingService, cna-modeling.entities.Infrastructure ]
17 valid_target_node_types: [ cna-modeling.entities.Infrastructure ]
18 valid_capability_types: [ cna-modeling.capabilities.Host ]
```

Listing 5.7: The Link relationship definition for the Link entity (excerpt)

```
cna-modeling.relationships.ConnectsTo.Link:
 description: Relationship Type to model Link entities
 properties:
retries:
type: integer
 description: Number of times this invocation is retried, if it fails.
 default: 0
 required: true
   [...]
valid_source_node_types: [ cna-modeling.entities.Component, cna-modeling.entities
    .Service, cna-modeling.entities.BackingService, cna-modeling.entities.
    StorageBackingService, cna-modeling.entities.ProxyBackingService, cna-modeling
    .entities.BrokerBackingService ]
valid_target_node_types: [ cna-modeling.entities.Endpoint, cna-modeling.entities.
    Endpoint.External ]
valid_capability_types: [ cna-modeling.capabilities.Endpoint ]
```

5.2.4. Artifact Types

As the last part of the TOSCA profile, artifact types are included which can be used to model the specific artifacts that are provided for the entities of an application. An initial set of artifacts types is defined which represent typically used deployment resources and which are relevant for factors of the quality model (e.g., **Standardized self-contained deployment unit** or **API-based communication**). The following artifact types are included in the profile, but this set is intended to be extended over time:

- **CNA.Artifact**
- Kubernetes.Resource
- Kubernetes.Resource.Service
- Kubernetes.Resource.Deployment
- Implementation
- Implementation.Bash
- Implementation.Java
- Implementation.Python
- Implementation.JavaScript
- Image.Container
- Image.Container.OCI
- Image.VM
- Azure.Resource
- Azure.ResourceManagerTemplate
- Terraform.Script
- Pulumi.Script
- Ansible.Script
- Chef.Script
- Puppet.Script
- CloudFormation.Script
- AWS.Resource
- AWS.EKS.Cluster
- AWS.EC2.Instance
- AWS.EC2.LoadBalancer
- AWS.EC2.NodeGroup
- AWS.Beanstalk.Application
- AWS.RDS.Instance
- GCP.Resource
- OpenAPI
- Spring.CloudContract
- Pact.Contract

All artifact types derive from `CNA.Artifact` which is shown in Listing 5.8.

Listing 5.8: The CNA.Artifact type definition

```
CNA.Artifact:
 description: The root artifact type for artifacts used in the cna modeling
   application
 properties:
provider_specific:
type: boolean
 description: "Whether this artifact is (cloud) provider-specific or not."
 required: true
 default: false
based_on_standard:
type: text
 description: If the artifact is based on a standard, specify it here.
 required: true
 validation: { $valid_values: [ $value, ["none", "OCI", "OpenAPI", "other"]]}
 default: "none"
self_contained:
type: boolean
 description: "Whether this artifact is self-contained or not, that means whether
   it needs additional resources explicitly added to it to be used."
 required: true
 default: false
```

The derivation is important to include the defined `properties` in all artifact types, because this information is needed for the quality evaluation approach. Furthermore, the types are structured in the way that there are some general types like `Implementation`, `Azure.Resource`, or `AWS.Resource` which can be made more specific by making their name more explicit. This is also intended to enable extensibility in the way that more specific artifacts can be added to suitable general artifacts.

5.2.5. Exemplary Architectural Models

To show how the TOSCA profile can be used to model application architectures, exemplary models are presented in this section. These models represent the applications used for the evaluation of the quality evaluation approach, presented in Section 4.7.5 [173]. The complete models are available online[63].

Starting with the TeaStore application, Listing 5.9 shows how the Webui service can be modeled. It is of type `cna-modeling.entities.Service` and `properties` and `artifacts` are simply included as attributes in it. Because it is implemented as a Java web service, one artifact which is used for the deployment is the `webui-war`. In the `requirements` section, with the `provides_endpoint` requirement the connections to the endpoints of the service can be specified. The referenced node in this case is the endpoint node (see Listing 5.10) which is on the same hierarchical level as the service as far as TOSCA is concerned. However, the endpoint is not independent and always needs to be attached to the service which provides it. With the `uses_data` requirement, a relationship to the Product data aggregate is specified. And with the `host` requirement, it is stated how the service is deployed, namely with the `aws_eks_cluster_hosts_webui` deployment

[63] `https://github.com/r0light/clounaq%2Devaluation`, visited 2025-10-20

mapping. Lastly, the `endpoint_link` requirement represents the outgoing communication from this service to the endpoint of another service, in this case the `get_categories` endpoint.

Listing 5.9: Representation of the webui service (TeaStore) (excerpt)

```
webui:
type: cna-modeling.entities.Service
metadata:
label: Webui
properties:
managed: false
software_type: custom
stateless: true
namespace: teastore-namespace
artifacts:
webui-war:
type: Implementation.Java
file: tools.descartes.teastore.webui.war
description: The war file containing the service
properties:
- provider_specific: false
- based_on_standard: none
- self_contained: false
requirements:
- provides_endpoint:
capability: cna-modeling.capabilities.Endpoint
node: isready_b
relationship: cna-modeling.relationships.Provides.Endpoint
- provides_external_endpoint:
capability: cna-modeling.capabilities.Endpoint
node: index
relationship: cna-modeling.relationships.Provides.Endpoint
- uses_data:
node: product
relationship: webui_uses_product
- host:
node: aws_eks_cluster
relationship: aws_eks_cluster_hosts_webui
- endpoint_link:
node: get_categories
relationship: webui_linksTo_get_categories
[...]
```

Listing 5.10 shows the `index` external endpoint of the Webui service. It has an `endpoint` capability and an `external_endpoint` capability.

Listing 5.10: Representation of the index endpoint (TeaStore)

```
index:
type: cna-modeling.entities.Endpoint.External
metadata:
label: Index
properties:
kind: query
method_name: GET
protocol: HTTP
url_path: /index
rate_limiting: none
idempotent: true
readiness_check: false
health_check: false
allow_access_to: []
documented_by: []
capabilities:
endpoint:
properties: {}
external_endpoint:
properties: {}
requirements:
- uses_data: category
- uses_data: image
```

Listing 5.11 shows the `webui_linksTo_get_categories` relationship which is also referenced by the `endpoint_link` requirement in Listing 5.9. Within the relationship, the properties specific to the link entity are set.

Listing 5.11: Representation of the `webui_linksTo_get_categories` relationship (TeaStore)

```
webui_linksTo_get_categories:
type: cna-modeling.relationships.ConnectsTo.Link
properties:
relation_type: ''
timeout: 130000
retries: 2
circuit_breaker: none
```

The relationships representing link entities can in turn be referenced by request traces, for example, the index_page request trace as shown in Listing 5.12. A particularity here is that the `involved_links`, that means the links which are part of the request trace, are stored as a property. The reason is that for request traces an ordering of the links is important. Furthermore, it is possible that certain links are invoked in parallel while others are invoked sequentially. Therefore, the links of a request trace are stored in a sequence of sequences which would not be possible using `requirements`. The outer sequence provides the ordering relation and all links in an inner sequence are invoked in parallel. In the example shown in Listing 5.12, there are four links which are invoked strictly sequentially, because they are all synchronous requests performed by the Webui service to serve requests made to the `index` external endpoint. The `nodes` property includes all components which are part of the request trace as a result of the involved links. It is not meant to be a freely editable property, but instead its value depends on the `involved_links` property and should be derived from that. A single external endpoint is the starting point for a request trace. It is, therefore, referenced via a `requirement` and the corresponding `cna-modeling.relationships.PartOf` relationship.

Listing 5.12: Representation of the index page request trace (TeaStore)

```
index_page:
type: cna-modeling.entities.RequestTrace
metadata:
label: Index Page
properties:
nodes:
- webui
- persistence
- db
- auth
- imageprovider
involved_links:
- - webui_linksTo_get_categories
- - persistence_linksTo_sql_query
- - webui_linksTo_isloggedin
- - webui_linksTo_getwebimages
requirements:
- external_endpoint:
node: index
relationship: cna-modeling.relationships.PartOf
```

As another example, parts of the modeled architecture of the LakeSide Mutual application are shown in the following. Listing 5.13 shows a modeled infrastructure entity, in this case a managed Google Kubernetes Engine Cluster. It is set up to run in a single region `us-east1` and using two availability zones from that region `us-east1-b` and `us-east1-c`. Because it is managed, `environment_access` is `limited` which means that certain configurations for the cluster nodes can be made, but otherwise the actual nodes are not accessible. It is maintained in an `automated` way by the cloud provider and also provisioned in a `transparent` way.

Listing 5.13: Representation of the Kubernetes Cluster infrastructure (LakeSide Mutual) (excerpt)

```
google_kubernetes_engine_cluster:
type: cna-modeling.entities.Infrastructure
metadata:
label: Google Kubernetes Engine Cluster
properties:
kind: software-platform
environment_access: limited
maintenance: automated
provisioning: transparent
supported_artifacts:
- Kubernetes.Resource
- Kubernetes.Resource.Service
- Kubernetes.Resource.Deployment
availability_zone: us-east1-b,us-east1-c
region: us-east1
supported_update_strategies:
- replace
- rolling
- blue-green
deployed_entities_scaling: automated-built-in
self_scaling: manual
enforced_resource_bounds: true
identities: {}
capabilities:
address_resolution:
properties:
address_resolution_kind: DNS
```

One component of the LakeSide Mutual application that is deployed on the Kubernetes cluster is the Spring Boot Admin Backing Service. Its modeled representation is shown in Listing 5.14. Because it is deployed by the application developers, it is not managed. And since the source code of the Spring Boot Admin Service is publicly available and was used in the application as-is, the property `software_type` is set to `open-source`. The `providedFunctionality` property, which is specific to the type `cna-modeling.entities.BackingService`, is set to `other`, because the Spring Boot Admin service provides a range of functionalities to monitor and manage other services based on Spring Boot and it cannot be assigned to any single functionality of those specified for the `BackingService` type (see Listing 5.5). Three artifacts are associated with the Spring Boot Admin service. At its core, it is a Java application. Within the LakeSideMutual application an OCI-conformant container image is provided as well as a Kubernetes Deployment description. OCI is a standard, and thus set in the `based_on_standard` property. Furthermore, the container image and the Kubernetes Deployment description are `self_contained`, meaning that all required software is included to deploy and execute it. The JAR file is not `self_contained`, because it needs a

Java Runtime Environment (JRE) to be actually executed. None of the artifacts is specific to a certain cloud provider. The deployment of the Spring Boot Admin service is specified through the `host` requirement where it is stated that the node is deployed on the `google_kubernetes_engine_cluster` node and the deployment mapping is represented and further specified through the relationship called `google_kubernetes_engine_cluster_hosts_spring_boot_admin`.

Listing 5.14: Representation of the Spring Boot Admin Service (LakeSide Mutual) (excerpt)

```
spring_boot_admin:
type: cna-modeling.entities.BackingService
metadata:
label: Spring Boot Admin
properties:
managed: false
software_type: open-source
stateless: true
load_shedding: false
identities: {}
namespace: default
providedFunctionality: other
replication_strategy: none
artifacts:
admin-service-jar:
type: Implementation.Java
file: spring-boot-admin-0.0.2-SNAPSHOT.jar
properties:
- provider_specific: false
- based_on_standard: none
- self_contained: false
admin-service-image:
type: Image.Container.OCI
file: ''
properties:
- provider_specific: false
- based_on_standard: OCI
- self_contained: true
kubernetes deployment:
type: Kubernetes.Resource.Deployment
file: kubernetes/manifests/spring-boot-admin.yaml
properties:
- provider_specific: false
- based_on_standard: none
- self_contained: true
requirements:
- host:
node: google_kubernetes_engine_cluster
relationship: google_kubernetes_engine_cluster_hosts_spring_boot_admin
[...]
```

Listing 5.15 shows the mentioned deployment mapping relationship for the deployment of the Spring Boot Admin service. The type of the `deployment` is stated as `automated-declarative`, because with Kubernetes resource descriptions the deployment is declarative. In the resource description it is described how the deployment should look like and Kubernetes then takes care of the actual deployment. Because an artifact of type `Kubernetes.Resource.Deployment` is available for the Spring Boot Admin service, it can be used as a `deployment_unit`. With Kubernetes, the `update_strategy` for pods is `replace`, because updates are not performed within pods. Instead, if a new version is deployed, pods with the old version are deleted and replaced with pods of the new version. More advanced strategies are not explicitly used in the LakeSide Mutual application, but would be available as shown by the `supported_update_strategies` property in List-

ing 5.13. The restart policy is `onProcessFailure`, meaning that a pod is automatically restarted if the process of the container inside it fails for some reason. Finally, no explicit resource requirements, regarding the required amount of Central Processing Unit (CPU) shares or memory, are stated in the deployment description, therefore this property is set to `unstated`.

Listing 5.15: Representation of the Deployment Mapping for the Spring Boot Admin Service (LakeSide Mutual)

```
google_kubernetes_engine_cluster_hosts_spring_boot_admin:
type: cna-modeling.relationships.HostedOn.DeploymentMapping
 properties:
deployment: automated-declarative
deployment_unit: Kubernetes.Resource.Deployment
replicas: 1
update_strategy: replace
automated_restart_policy: onProcessFailure
assigned_account: default-account
resource_requirements: unstated
```

To also show the modeling of data aggregates and backing data entities and how they are related to other components, an example from the PetClinic application is shown in Listing 5.16. It shows the representation of the `visit` data aggregate and the representation of the `visits_service_config` backing data. The `DataAggregate` type does not have any additional properties itself and just one `capability` to enable formulating relationships to it. The `BackingData` in contrast has a property `included_data` to include details on the included data and the same capability as the `DataAggregate` type. Because the backing data entity is intended to represent different kind of data, that is not core to the application domain, but nevertheless needed, the `kind` property allows for a differentiation. In this case, it is about configuration data and thus the `config` value was set.

Listing 5.16: Representation of the Visit data aggregate and Visit Config backing data (PetClinic)

```
visit:
type: cna-modeling.entities.DataAggregate
 capabilities:
provides_data: {}
 [...]
visits_service_config:
type: cna-modeling.entities.BackingData
 capabilities:
provides_data: {}
 properties:
kind: config
included_data:
spring.zipkin.baseUrl: http://tracing-server:9411
server.port: '8082'
eureka.client.serviceUrl.defaultZone: http://discovery-server:8761/eureka/
 [...]
```

The `visit` data aggregate and the `visit_service_config` backing data entities are used by the Visits service, as it is modeled in Listing 5.17. This is done through two requirements, `uses_data` and `uses_backing_data` which reference the nodes for the `visit` data aggregate and the `visits_service_config` backing data entity. Furthermore, two explicit relationships are required which are referenced by their names in the requirements.

Listing 5.17: Representation of the Visits Service (PetClinic) (excerpt)

```
visits_service:
type: cna-modeling.entities.Service
metadata:
label: Visits Service
properties:
managed: false
software_type: custom
stateless: true
load_shedding: false
identities: {}
namespace: default
requirements:
- uses_data:
node: visit
relationship: visits_service_uses_visit
- uses_backing_data:
node: visits_service_config
relationship: visits_service_uses_visits_service_config
[...]
```

The relationships between the Visits service and the data entities that it uses are shown in Listing 5.18. Additional information about how components use certain data is modeled within these relationships using properties. The important property here is `usage_relation` which describes whether a component just uses data, uses it, but also caches it or whether it persists it. As it can be seen, both `visit` and `visits_service_config` are cached by the Visits service.

Listing 5.18: Representation of relationships between Visits service and the used data (PetClinic)

```
visits_service_uses_visit:
type: cna-modeling.relationships.AttachesTo.DataAggregate
metadata:
label: Visit
properties:
usage_relation: cached-usage
sharding_level: 0
visits_service_uses_visits_service_config:
type: cna-modeling.relationships.AttachesTo.BackingData
metadata:
label: Visits Service Config
properties:
usage_relation: cached-usage
```

To summarize, the shown excerpts of architectural models showcase how architectures can be represented in a structured textual way with the TOSCA-based modeling approach. The modeling approach is extensible, and in some aspects explicitly intended to be further developed, for example, regarding the available artifact types (5.2.4) or regarding the values that certain `properties` can have. Nevertheless, the textual models can become quite large which is also why mostly only excerpts were included in this section.

5.3. Graphical Representation

In an addition to the textual representation for cloud-native software architectures presented in the previous Section 5.2, also a graphical representation format was developed (see also Section 2.3.3). The textual representations can become quite long and complex, because a relationship might be referenced in a component, but its actual description is placed farther away from it. Navigating in a long text document for understanding the structure of a system can thus be difficult and hinder the usefulness of an architectural model. In contrast, with a graphical approach, a better overview can be achieved, because information can be presented in a more compact way. Since also the TOSCA-based textual model is essentially a graph, also for the graphical model, a graph-based approach was taken. The components of the system are the vertices while the relationships in the form of links and deployment mappings are the edges. To develop the graphical representation, a distinct shape was used for each entity to include semantics in the visual representations [195]. These shapes are presented in detail in Section 5.3.1. Examples for how modeled architectures look like are presented in Section 5.3.2 which partly align with the textual models shown in Section 5.2.5. Finally, further aspects considering how complexity of models can be handled with the graphical approach are presented in Section 5.3.3.

5.3.1. Entity Shapes and Connections

To each entity a unique shape was assigned to enable a differentiation based on geometric characteristics without relying on colors. An overview of all entity shapes is provided in Figure 5.1 which also repeats how the different entities are related. The shapes and their connections are explained in the following.

A basic rectangular shape (▭) represents components which fits to their usage of modeling parts of a system in a more abstract way or where the other, more specific, entities do not fit. These more specific entities have specific shapes, such as a hexagon (⬡) for representing services which is also used elsewhere in literature [242, 293]. Backing Services are represented with a rhombus (◇). Storage Backing Services are modeled with a cylinder (⛁), a common symbol for databases [60, 242]. A rotated cylinder (▯) represents broker backing services. This was also adopted from other uses in literature [116, 242]. For proxy backing services a slight variation of the rhombus was used (▷) where one side is rectangular.

Endpoints are represented by circles (○) which are not filled, while external endpoints are represented by circles that are filled (●). The similarity is intentional, because external endpoints are a specialization of endpoints. In a comparable way, data aggregates are represented by ellipses (⬭) while backing data entities are represented by ellipses with a cut-off side (▭). The way that the relationships between components as well as its derived entities and endpoints, external endpoints, data aggregates and backing data entities are visualized is that these entities (○, ●, ⬭,

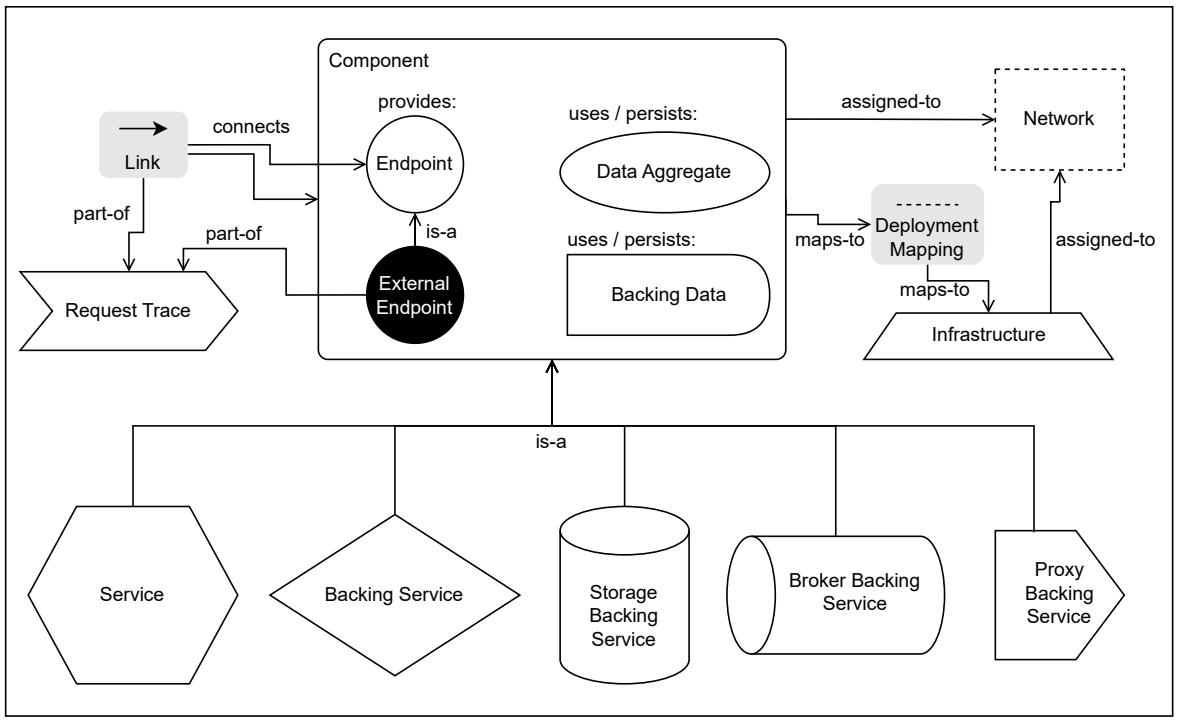


Figure 5.1.: Entity Shapes Overview

▭) are included in component entities in the sense that they are drawn within the boundaries of those entities if they are part of it. For example, if an endpoint is provided by a service, the endpoint shape is drawn within the service shape.

Furthermore, infrastructure entities are shown as trapezoids (⏢) which may also be drawn with a differing width to visualize the relation to other entities in the sense that they provide the foundation for the other entities and also to visualize stacks of infrastructure. Network entities are drawn as rectangular shapes with dashed borders (⬚) so they could be used as wrappers around other entities, but they can also be drawn to the side of other entities. Similarly, request traces are drawn in an arrow shape (▱) and are also modeled to the side of other entities.

Finally, as the connecting elements, links are simply arrows (⟶) that point from an invoking component to the invoked endpoint. And deployment mappings are represented as dashed lines (- - - -) which connect components or infrastructure entities to the infrastructure entities on which they are deployed.

5.3.2. Exemplary Graphical Models

To show how the graphical modeling approach can be applied to actual application architectures, examples from the applications used in Section 4.7.5 and Section 5.2.5 are used in the following. The models were created with the Clounaq tool[64] and can also be imported and modified with it. Figure 5.2 shows the complete model of the TeaStore application.

[64] `https://clounaq.de`, visited 2025-10-20

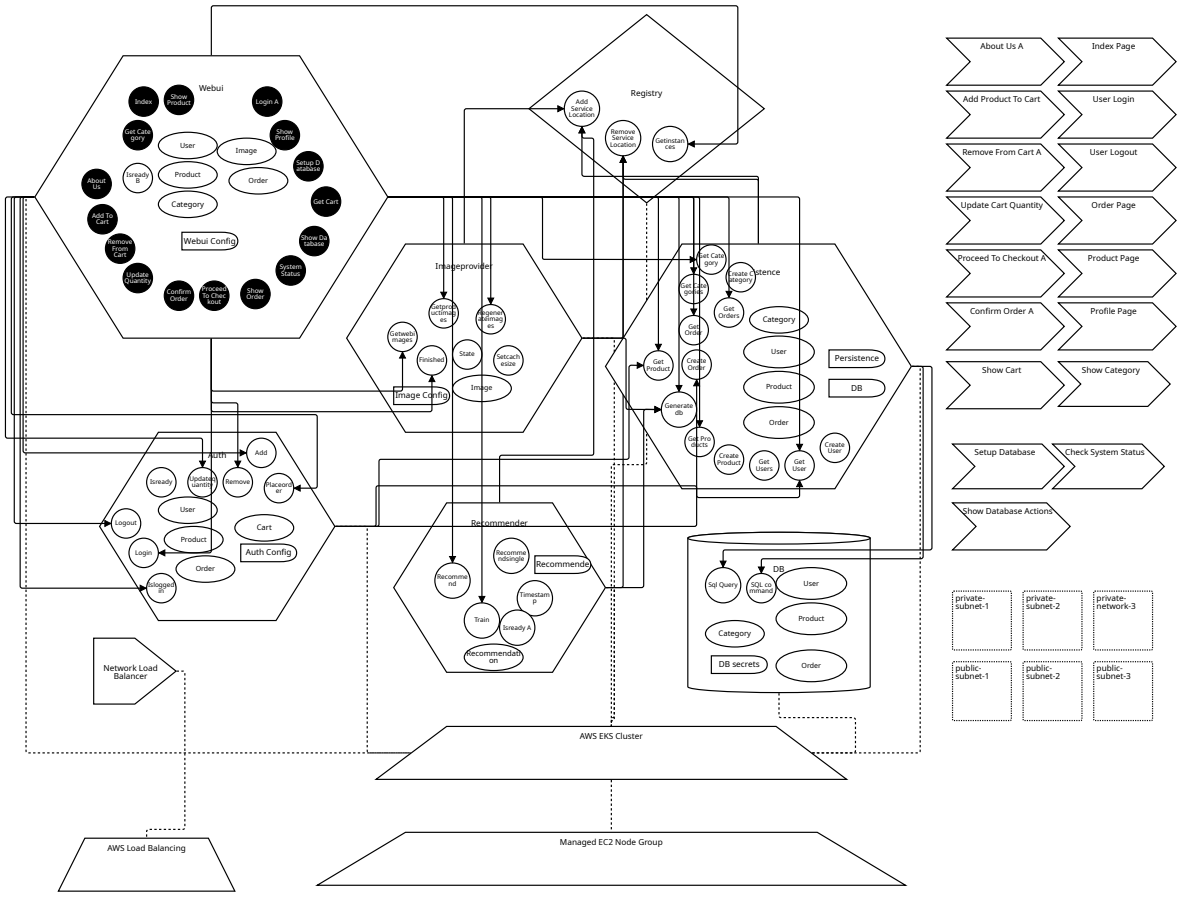


Figure 5.2.: TeaStore system overview

The different entities can be identified by their shapes (5.3.1). Only links (⟶) and deployment mappings (⇢) are visible as relationships while others, for example, that the *Network Load Balancer* acts as a proxy for the Webui service, are not explicitly shown in the graphical representation. The reasoning for this is that those two relationships are explicit entities themselves while the other relationships are not. As it can be seen, such an architectural model can become quite complex. Therefore, it's recommended to view the model within the Clounaq tool instead of the static way presented here. Furthermore, possibilities for complexity reduction for such models are presented in Section 5.3.3.

The excerpt in Figure 5.3 shows the Webui service and the Persistence service in more detail and graphically represents the information from Listing 5.9. The `webui_linksTo_get_categories` link (Listing 5.11) is visualized as an arrow (⟶) from the Webui service (⬡) to the *Get Categories* endpoint (○) of the Persistence service (⬡). Furthermore, the `uses_data` requirement for the `product` data aggregate is visualized by the Product data aggregate (⬭) being drawn within the Webui service. And the `provides_endpoint` requirement is visualized by the Index external endpoint (●) also being drawn within the Webui service.

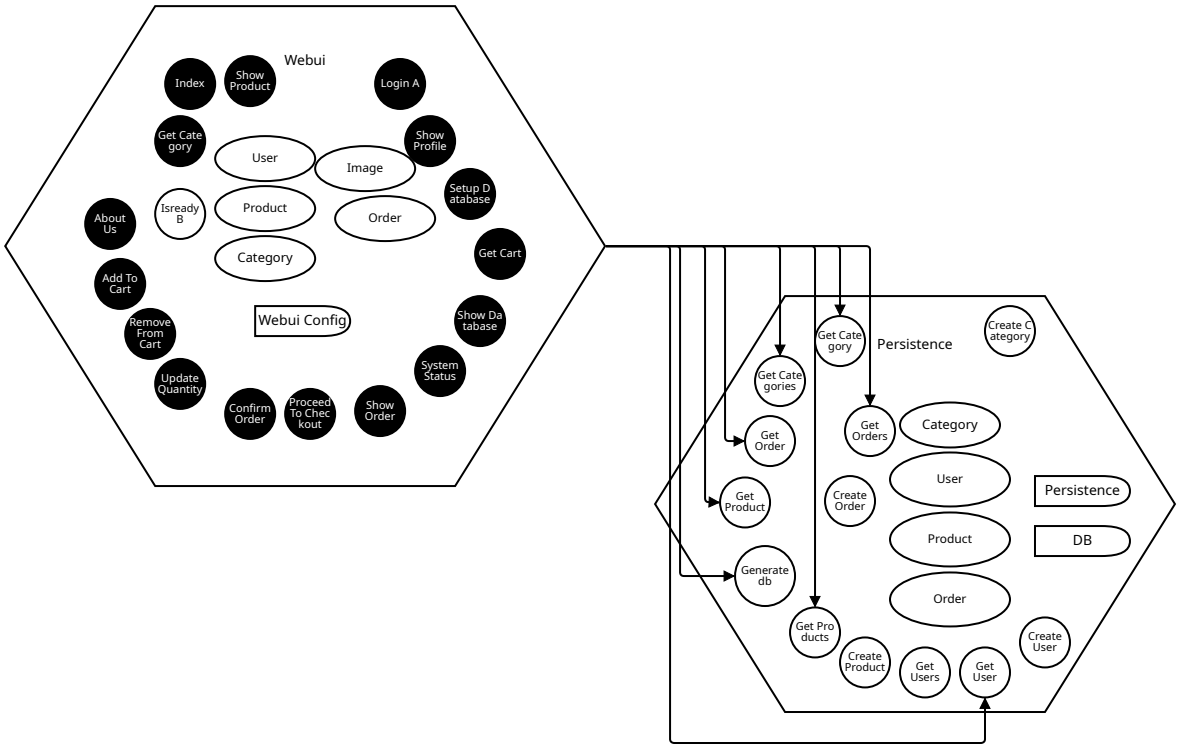


Figure 5.3.: TeaStore Webui and Persistence services with links

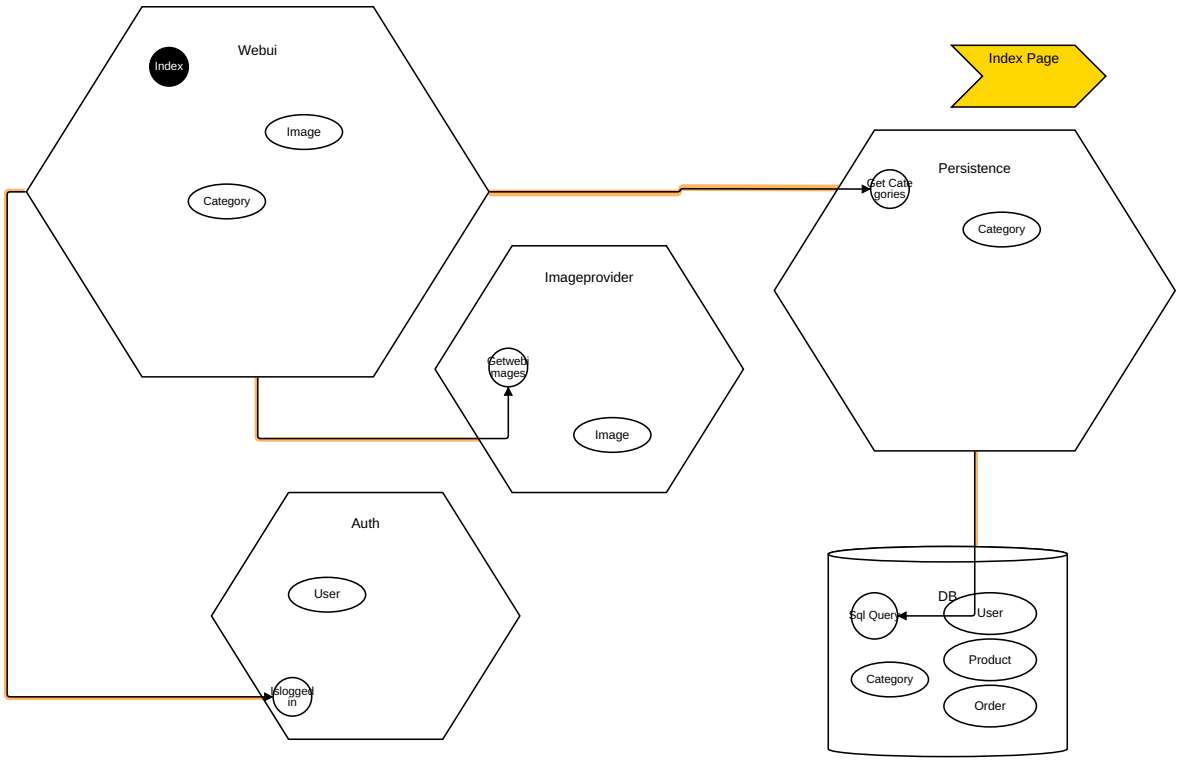


Figure 5.4.: TeaStore Index Page request trace

Figure 5.4 shows the Index Page request trace (▱) as it is also described in Listing 5.12. This specific view furthermore includes the endpoints (Consistency of supported authentication methods of endpoints), links (⟶), and data aggregates (⬭) that are part of this request trace. The links are explicitly included in the request traces. The endpoints are those which are the target endpoints of the involved links. And the data aggregates are those which are used by the set of identified endpoints. The ordering of the links within the request trace cannot be visualized in the graphical models explicitly. For viewing or editing the included links and their ordering, the request trace details need to be accessed.

It can, however, be seen that the invocation of the link from the Persistence service to the database (⛁) endpoint happens as a result of the invocation of the link from the Webui service to the *Get Categories* endpoint of the Persistence service. But for the outgoing links from the Webui service it cannot be derived in which order they are invoked or whether they may be invoked concurrently.

The second exemplary application is the LakeSideMutual application, shown as the overall system in Figure 5.5. In contrast to the TeaStore application, the frontend is not included in a service, but there are Single-Page Applications (SPAs) as frontend components which call the services actually providing the functionalities of the application. It was decided to model SPAs as general component entities (▭), because no explicit entity for frontend components is included in the modeling approach (yet) and it is difficult to differentiate endpoints of a SPA. The frontend is loaded to the client's browser once over the network and afterwards interaction with the SPA works locally in the browser with requests over the network being performed to the endpoints of the backend services. Thus, there is a single external endpoint modeled for the SPA frontends which represents the initial loading of the frontend and then it is modeled which external endpoints of the backend services are invoked by the frontends. Although the LakeSide Mutual application is the only modeled application with asynchronous communication (see Section 4.7.5.2), there is no explicit broker backing service entity modeled (▭). The reason is, that the used ActiveMQ[65] message broker is bundled within the *Policy Management Backend* service. The modeling approach, however, does not support a nesting of component entities and thus asynchronous endpoints of the message broker are considered as part of the service. After all, the service is also a single unit of deployment. It is noted by the developers, however, than in a more realistic setup the broker should be run independently which would then also be reflected in the architectural model. In a similar way, no storage backing services (⛁) are modeled for the LakeSideMutual application, because these are also included in the *Customer Management Backend* service, *Customer Self-Service Backend* service, *Customer Core* service, and *Policy Management Backend* service.

[65] `https://activemq.apache.org/`, visited 2025-10-20

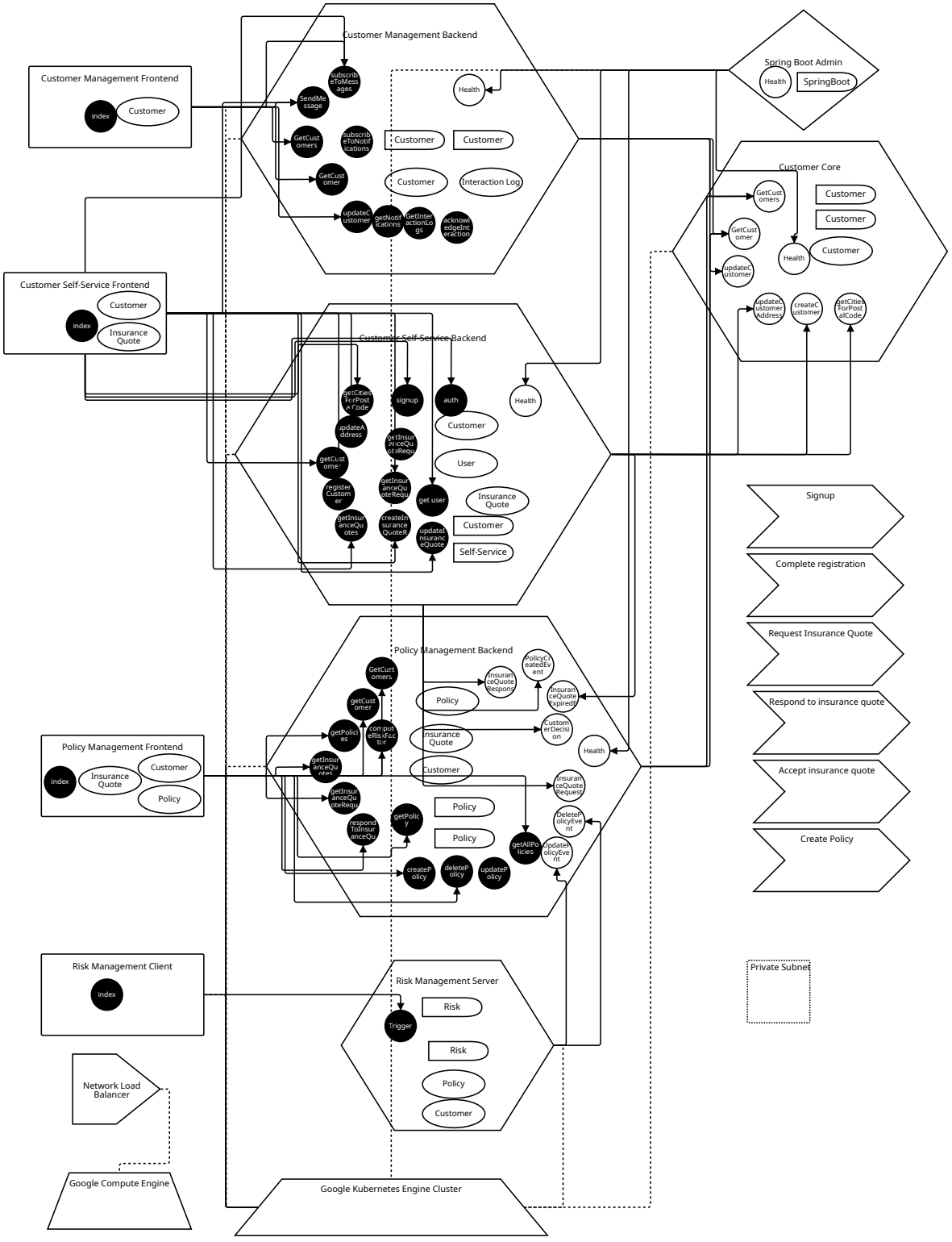


Figure 5.5.: LakeSideMutual System overview

Figure 5.6 shows a more detailed extract of the LakeSideMutual application to highlight how deployment mappings are modeled. The shown extract corresponds to the Listings 5.13, 5.14, and 5.15 regarding the Spring Boot Admin backing service (◇), the Google Kubernetes Engine Cluster infrastructure (⏢) and its connecting deployment mapping (┈┈). In contrast to the model of the TeaStore application, only the *Google Kubernetes Engine Cluster* infrastructure is modeled, because it is used as a managed service with no details about the underlying infrastructure for providing cluster nodes. For the TeaStore application, a *Managed EC2 Node Group* was also modeled as an infrastructure on which the cluster is deployed, because it was explicitly configured for the setup. To showcase an additional deployment mapping, also the deployment of the *Network Load Balancer* used as a load balancer for the Kubernetes cluster is included in Figure 5.6.

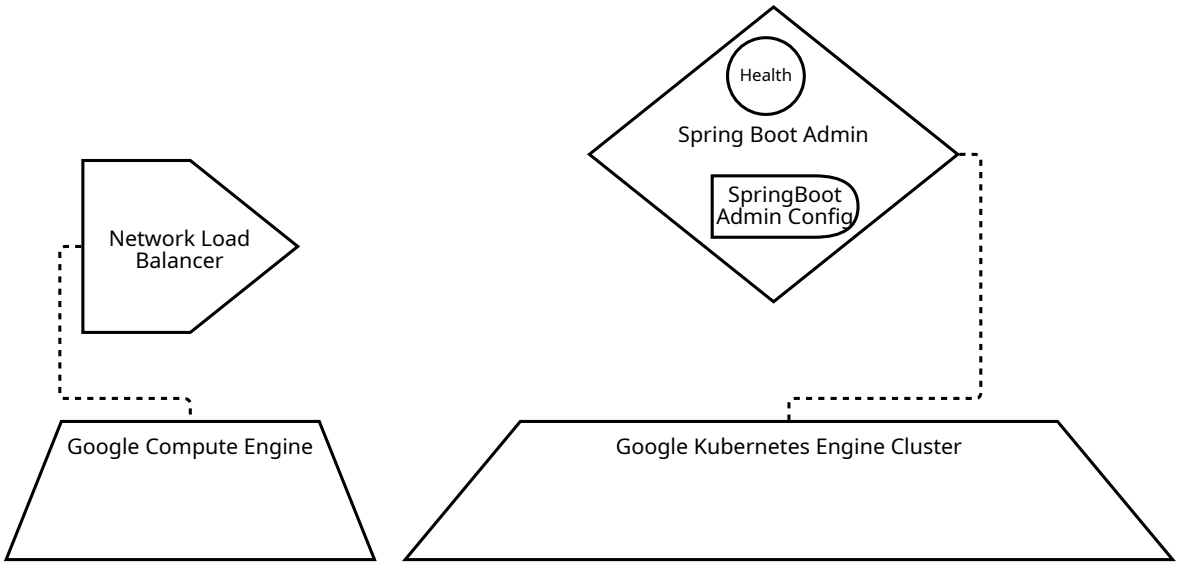


Figure 5.6.: LakeSideMutual Admin service deployment mapping

As a final example, an extract from the model of the PetClinic application is included in Figure 5.7. It focuses on the modeling of data aggregates and backing data entities, in this case the *Visit* data aggregate (⬭, highlighted in yellow) and the *Visits Service Config* backing data entity (▭, highlighted in green). In contrast to the textual representation, data aggregates and backing data entities are not modeled as independent entities in the graphical format, but always within the context of components. In the textual representation, the *Visit* data aggregate and the *Visits Service Config* backing data entity were first modeled as nodes (see Listing 5.16) and then relationship to them were modeled (see Listing 5.18), for example for the *Visits* service (see Listing 5.17). In the graphical representation, however, the relationships are expressed by drawing the data aggregates and backing data entities directly within the components that use or persist them. The fact that the different drawn representations (⬭or ▭) represent the same entity can be configured in the entity details.

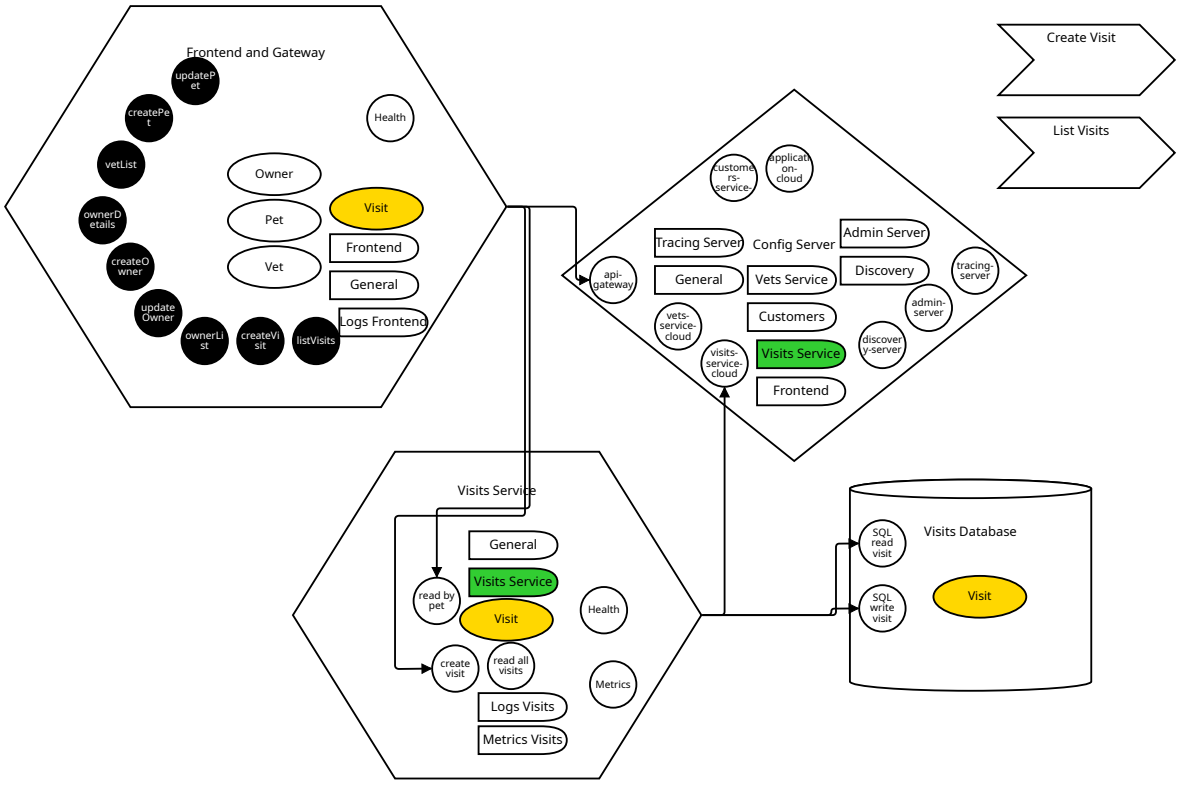


Figure 5.7.: PetClinic view on Visit data aggregate and Visits Service Config backing data

5.3.3. Handling Complexity in Architectural Models

As already stated, textual representations of architectural models can become quite large and thus complex to understand. Having a graphical representation as such, therefore, is a first step to better handle the complexity. But also graphical models can become complex for real-world applications, as it can be seen in Figure 5.2 or Figure 5.5. Hence, further means to handle the complexity in models are discussed in the following. One option is the possibility to hide details of components to set the focus on their overall structure and relationships. Details are in this case the entities which are part of other entities. For example, endpoints and data aggregates are part of components or derived entities. Thus, a focus can be set on a service as such by temporarily hiding the specific endpoints of that service. In the graphical modeling approach, detailed entities are: endpoints (○), external endpoints (●), data aggregates (⬭), and backing data entities (▭).

Furthermore, because cloud-native applications cover a broad range of aspects, another option is to temporarily set a focus on certain aspects while hiding others. This can be done in graphical models by providing filters that hide or show certain types of elements within a model. In specific, the following *views* are differentiated in the graphical modeling approach which cover certain types of entities:

- **Deployment View**: Infrastructure entities, Deployment Mappings
- **Communication View**: Endpoints, External Endpoints, Links, Request Traces

- **Backing Services View**: Backing Services, their Endpoints and communication to or from them
- **Data View**: Data Aggregates, Backing Data entities
- **Request Trace View**: Request Traces

By activating or deactivating a *view*, the corresponding entities are either hidden or shown in the model. So for example, if a focus should be set on the deployment of components, the communication and data views might not be that important and could therefore be hidden. This is shown in Figure 5.8 where only the *Deployment View* of the TeaStore application is visualized. In contrast to the model in Figure 5.2, fewer entities are shown. Thus, a focus can be set on analyzing and editing the deployments of components on corresponding infrastructure. Because in the shown model the other entities, such as endpoints, are simply hidden, the services take up more space than necessary, but this could be changed by also performing a resizing action when focusing on this view.

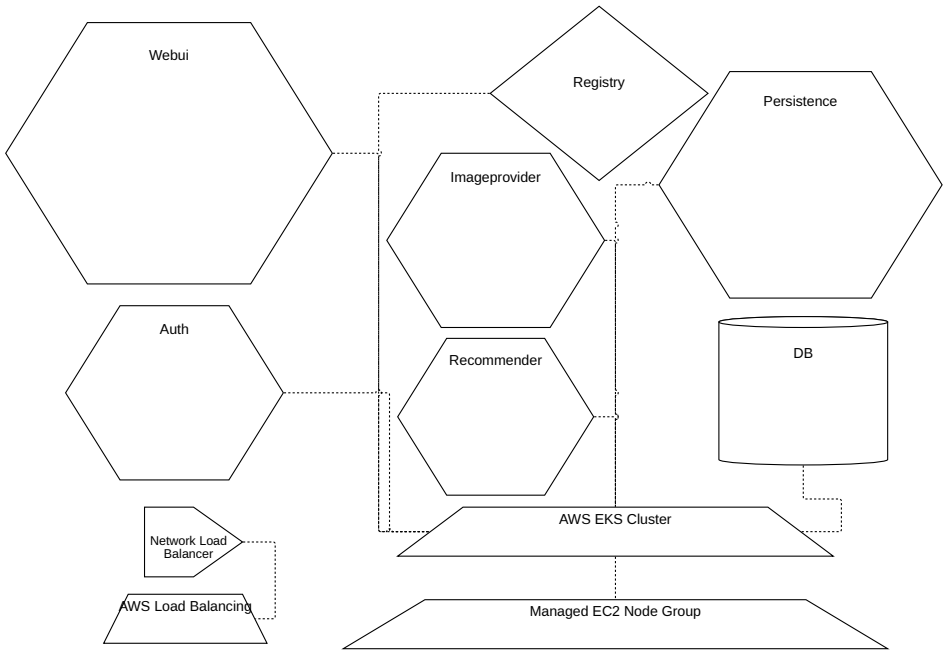


Figure 5.8.: TeaStore system Deployment View

As another example, Figure 5.9 shows just the *Communication View*, where deployment mappings and infrastructure entities are hidden. This view is beneficial when the communication between components should be investigated and potentially edited. In the model, the *Data View* has been deactivated, but it could be activated again, if it is also of interest how data is used in the communication.

Apart from the mentioned views, another possibility to handle the complexity of models through focusing only on parts of an application is to focus only on entities related to a specifically selected entity. This has already been shown, for example, in Figure 5.4. A specific request trace entity, *Index Page*, is selected and viewed in

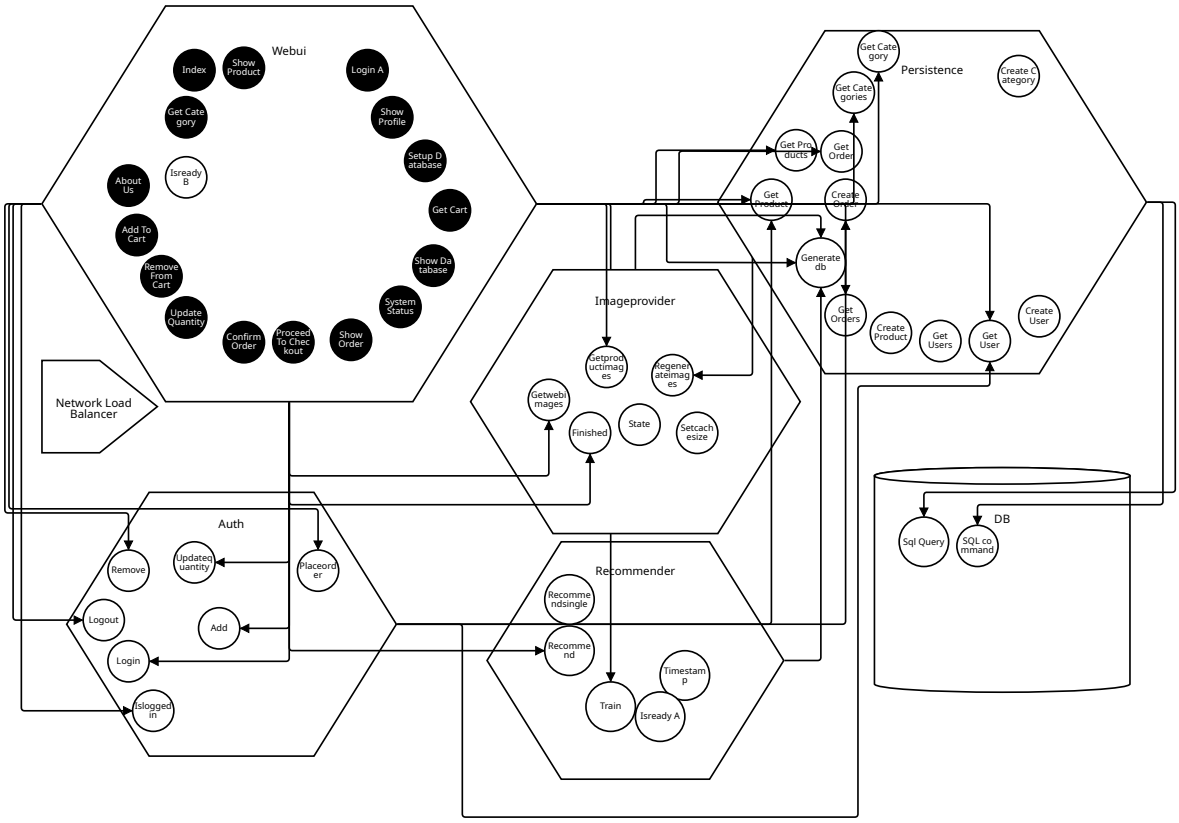


Figure 5.9.: TeaStore System Communication View

detail, meaning that only the links which are part of the request trace are shown in addition to the endpoints that are targeted by these links and the components to which they belong. Further entities for which this could be applied are for example network and infrastructure entities. In the case of a network entity, only those entities which are assigned to this network could be shown if it is selected. Or, if an infrastructure entity is selected, only those components which are deployed on that infrastructure could be shown.

All in all, the presented modeling approach fulfills the aims listed at the beginning of Chapter 5. The entities of the quality model can be represented (1) both textually (2) and graphically (3). An integration with other approaches is possible through the usage of TOSCA (4). Although the types of the presented TOSCA profile are custom to the quality evaluation approach, for an integration with other approaches, for example, a deployment using a TOSCA orchestrator can be achieved through *substitution mappings* which are also a part of TOSCA. Essentially, a type from one profile is mapped to a type of another profile and thus, models using these types can be transformed in either one form or the other. An architectural model of an application that should be evaluated using the presented approach could, for example, be transformed into another model which is supported by a TOSCA orchestrator that could then deploy and manage the application based on the transformed model. The *substitution mappings*, however, need to be formulated first specific to the, potentially provider-specific, other profile.

Never send a human to do a machine's job

Agent Smith (in The Matrix, 1999)

6. Clounaq: A Practical Tool for Quality Evaluations

Parts of this chapter have been taken from [168].

The Clounaq (**Clou**d-**n**ative **a**rchitectural **q**uality) tool[66] [168] is the practical implementation of the quality evaluation approach presented in this work. It has been developed throughout the overall research effort based on the approach described in Section 3.2. In the following, the tool and its implementation are presented in more detail, specifically focusing on the provided functionality (6.1) and how the functionality is implemented internally (6.2). To show how the tool can be applied to a use case and how it supports the overall quality evaluation approach is finally demonstrated in Section 6.3.

6.1. Provided Functionality

The Clounaq tool provides functionalities to perform quality evaluations using the quality model for cloud-native software architectures as described in Chapter 4. To do so, three major parts of the tool are differentiated which can be accessed through a tab menu, thus enabling continuous access to all functionalities. The three parts are described in the following with an integrated description of their intended usage in Section 6.1.4.

6.1.1. Quality Model Tab

Apart from the initial **Home** tab (🏠), the first major tab (**Quality Model**) presents the quality model as an interactive version. All factors are presented as a graph with the product factors (see 4.4) drawn in the middle and quality aspects (see 4.2) on the surrounding borders. They are connected with arrows that represent the impacts (see 4.5) between them. This way, product factors impacting only one quality aspect are drawn close to it while product factors that impact multiple quality aspects are drawn more in the middle of the graph. Factors can be dragged

Icons used in this chapter stem from FontAwesome (`https://github.com/FortAwesome/Font-Awesome`), made available under the CC BY 4.0 License (`https://creativecommons.org/licenses/by/4.0/deed.en`)

[66] `https://clounaq.de`, visited 2025-10-20

and moved around. By selecting a factor, its details (description, related literature, applicable entities, and relevant measures 4.6) are shown on the side. This can be seen in the screenshot in Figure 6.1 where the product factor *Mostly stateless services* is selected. The related literature is also linked so that a user can directly access it for further reading.

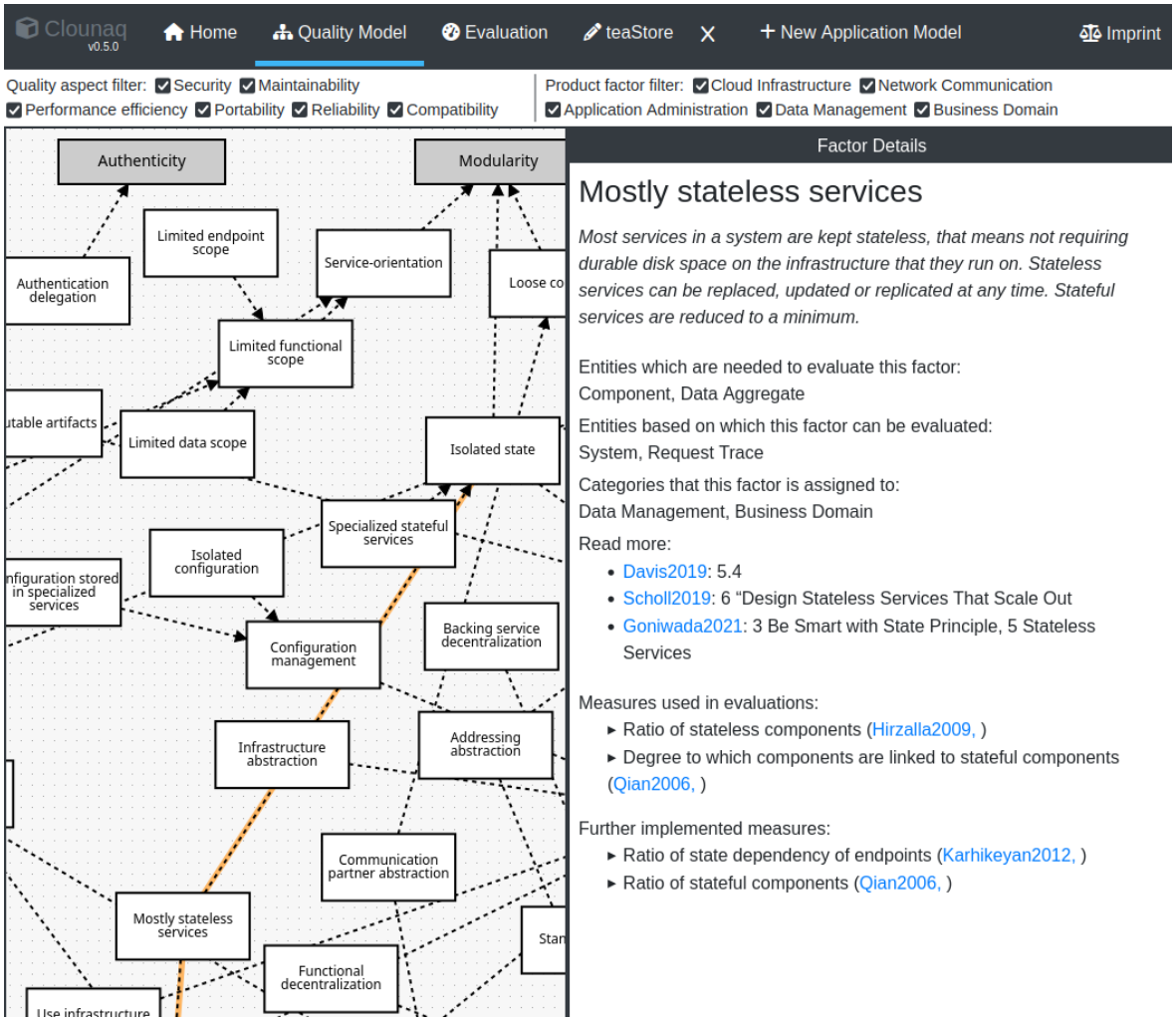


Figure 6.1.: Screenshot of the Quality Model tab with opened factor details

Furthermore, the quality model can be filtered based on quality aspects or product factor categories (see 4.4) as shown on the top in Figure 6.1. Below the presentation of the quality model itself, the formal specification of the quality model entities (see 4.3) is included to provide all necessary information for understanding the measures used for evaluations.

6.1.2. Modeling Tab(s)

The **Modeling** tab (✎) enables the modeling of software architectures. It can be entered either through creating a new, empty model or through importing an existing model from a file. Multiple models can be edited in parallel by creating a new modeling tab for each individual model.

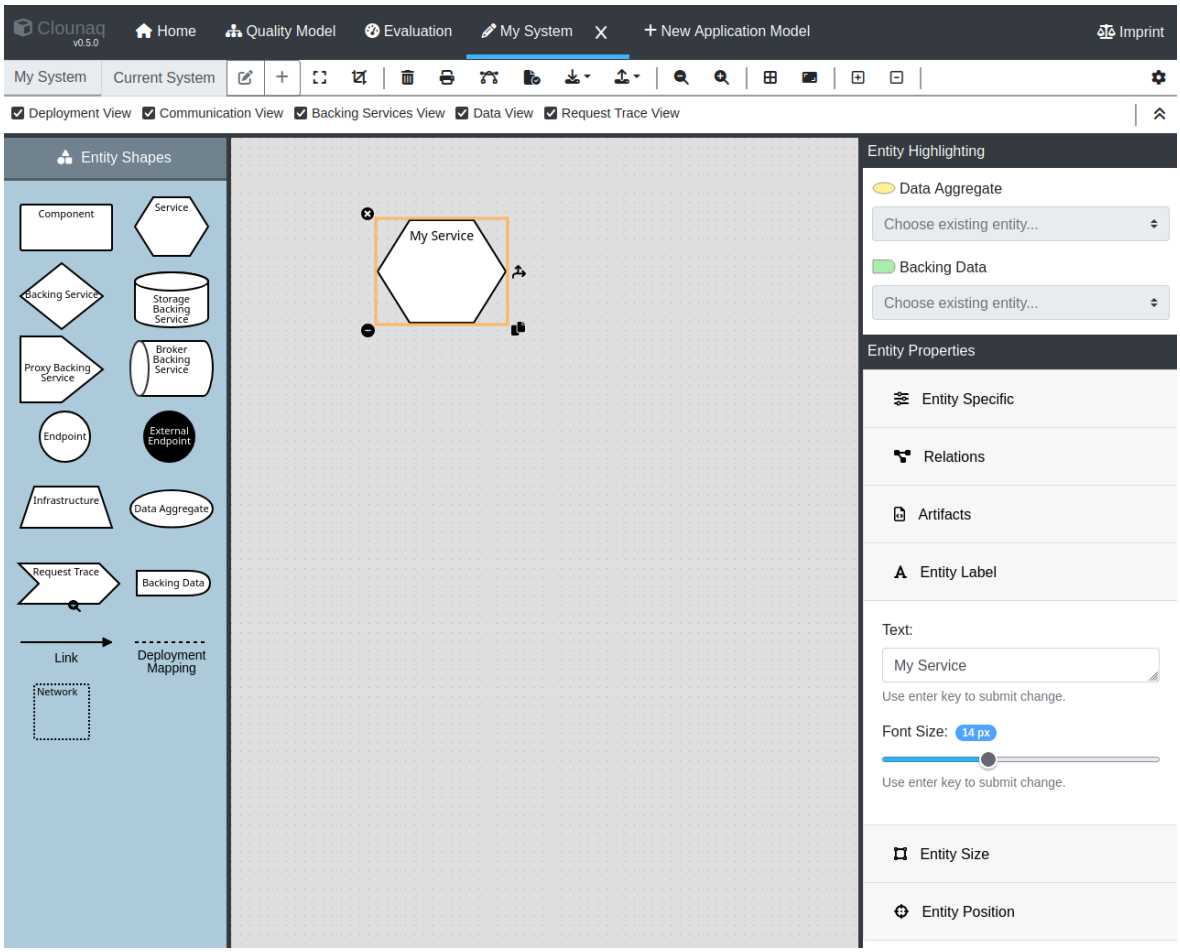


Figure 6.2.: Screenshot of the Modeling Tab

As shown in Figure 6.2, the core part of the modeling tab is a canvas on which architectural models, represented using the graphical representation format (see 5.3), can be edited. To do so, on the right side the different entity shapes are presented and on the left side a details view is included. Per drag and drop, entities can be created and moved around the canvas. Links and deployment mappings can be created by connecting shapes through dragging a connection from one shape to another. Entities can be copied and details can be shown or hidden. On the right side, details of the selected entity can be edited, especially the entity properties and relations to other entities (see 4.3). Also, functionalities for changing the position and size of shapes are available.

In the upper toolbar additional functionalities for the model as a whole are included. Among other things, the model can be zoomed in, zoomed out, or exported as a Scalable Vector Graphic (SVG). As a core part, models can be exported into a textual format. A custom JSON format is available as well as TOSCA, using the presented TOSCA profile (see 5.2). The TOSCA format is the preferred option for persisting architectural models. These can then be imported again. But the TOSCA format also enables an integration with other tools, such as Winery [147]. Finally, to apply the means for handling complexity

presented in Section 5.3.3, it is possible to filter the shown entities based on the different views. These can be activated or deactivated in the second toolbar. Also, the possibilities to focus on a certain request trace or highlighting the usage of certain data aggregate or backing data entities are implemented.

6.1.3. Evaluation Tab

If architectural models are open within the tool, they can be evaluated according to the quality model within the **Evaluation Tab** (). In the background, all applicable architectural measures are calculated for an architectural model, if it is selected. And based on the measure values, the product factors, impacts, and quality aspects are evaluated. The evaluation results are reported and visualized in the evaluation tab with the help of sub-graphs from the quality model graph. These sub-graphs are created either *per Product Factor* or *per Quality Aspect*. If an evaluation *per Product Factor* is done, that means that all product factors that can be connected through impact relations are grouped together. This is shown exemplary in Figure 6.3 for the TeaStore system product factors evaluation results in the context of Configuration management (compare to Table 4.47).

If sub-graphs are created *per Quality Aspect*, that means all factors which impact a certain quality aspect, either directly or via intermediate factors, are grouped together. This is shown exemplary in Figure 6.4, again, with the evaluation results of the TeaStore system for the quality aspect Fault tolerance (compare to Table 4.49).

The user can select the evaluation viewpoint, either *per Product Factor* or *per Quality Aspect*. Below the visualized sub-graphs, the specific product factor evaluation results and measure values can be viewed in detail for a better understanding of the evaluation results. Within these details, the factor descriptions and the calculation of the measure values are repeated.

As a default, the system entity is evaluated when selecting an architectural model for evaluation. By using drop-down boxes, either a specific component, an infrastructure entity, or a request trace can be selected to show its individual evaluation results. While the evaluation tab is mainly intended for an interactive exploration of the evaluation results, the results can also be exported in a structured way to either persist them or for processing them with other tools. There is an export option for exporting all calculated measures (not only those used for evaluations) for all entities as a Comma-separated values (CSV) file with the following columns: `measureKey`, `measureName`, `systemName`, `entityName`, `entityType`, `entityId`, `value`. And there is an export option for exporting the evaluation results for all factors as a CSV file with the following basic columns: `entity`, `hierarchy`, `type`, `key`, `name` and an additional column for each entity. A row is included for each entity type and factor combination which contains all evaluation results for applicable entities.

Finally, the evaluation results can also be filtered by the same approach used in the quality model tab (see Section 6.1.1). By default, all quality aspects and factor

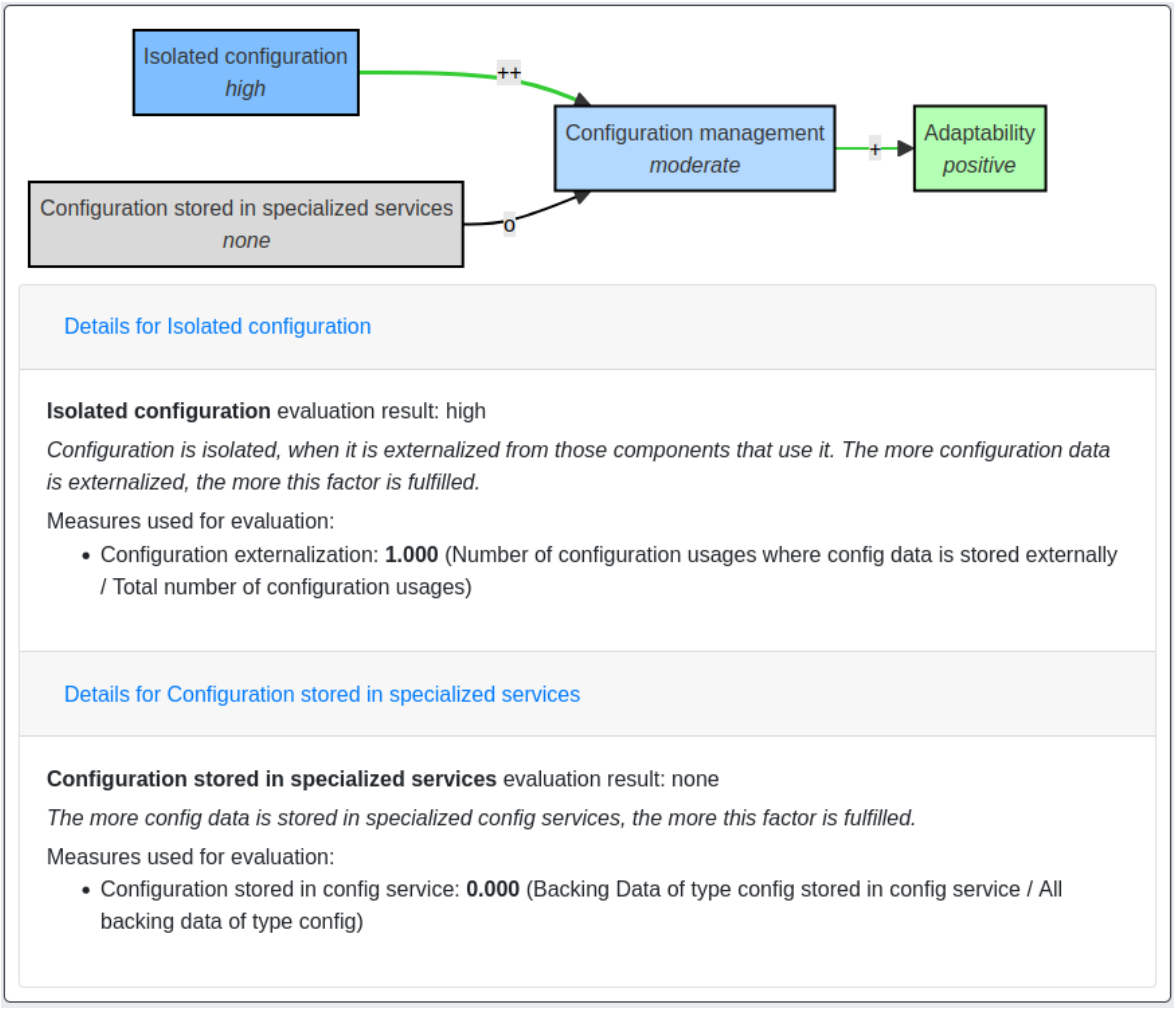


Figure 6.3.: Screenshot of Evaluation Tab Results visualized per Product Factor

categories are selected. By deselecting quality aspects of categories, corresponding evaluation results are removed from the view, thus enabling a more focused view on specific aspects, if desired.

6.1.4. Intended Usage

The intended usage of all provided functionalities together is depicted in Figure 6.5.

A user can start with informing herself about the quality model (, Section 6.1.1) to understand the individual factors and considered quality aspects. The main effort will then (A) be spent on creating an architectural model (, Section 6.1.2) for the application that should be investigated. This needs to be done manually, because of the specific modeling approach. An automated creation of models could potentially be implemented as well, but requires approaches specific to certain technologies which are used to implement cloud-native applications. If an architectural model has previously been created, it can be imported as well. When at least one architectural model is available, the user can proceed (B) to the evaluation tab (, Section 6.1.3) and view the evaluation results for the selected architectural

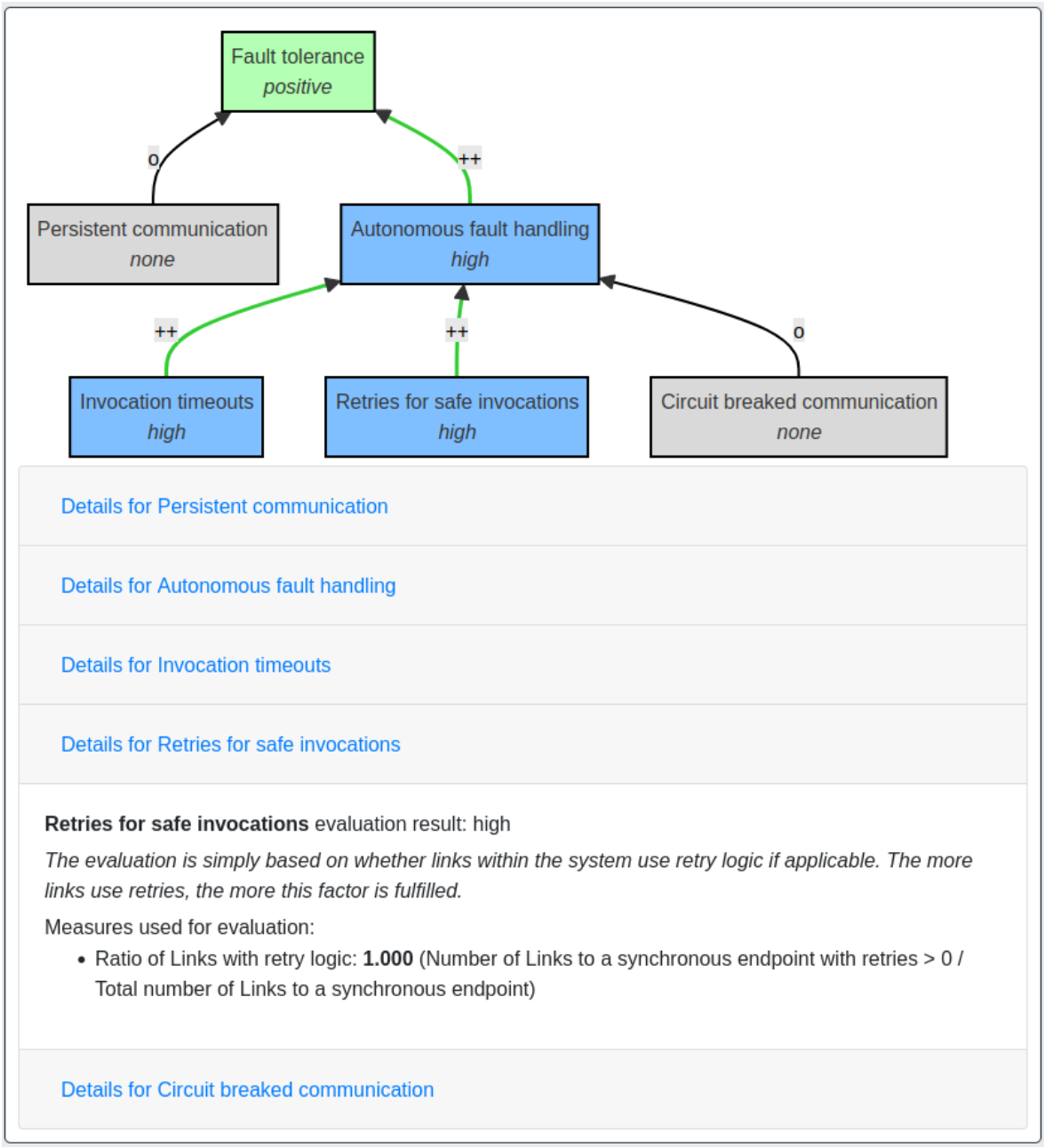


Figure 6.4.: Screenshot of Evaluation Tab Results visualized per Quality Aspect

model. The evaluation results are calculated automatically based on the modeled architecture(s).

Based on the evaluation results, a user then has several options. It's possible to get more information about the quality model and its elements (C_1) to better understand the evaluation results. If a user is working on the design of a new application, changes to the model can be made (C_2) to see how these changes affect the evaluation results. Or if a user decides to change the implementation based on the gained information from the quality evaluation, this can also be done (C_3), but would be out of the scope of the tool. If changes to the actual architecture

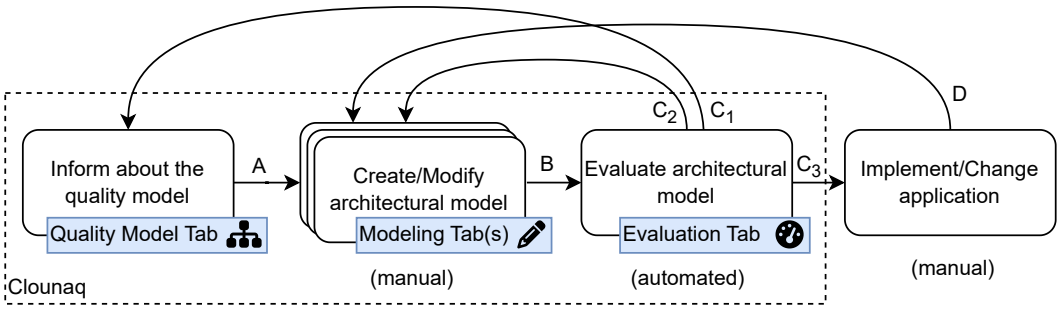


Figure 6.5.: Intended usage of the Clounaq tool functionalities

are made, naturally, these also need to be reflected in the architectural model (D) for an up-to-date view.

6.2. Tool Implementation

This section describes how the Clounaq tool is implemented. An overview on the internal architecture and how it supports the different functionalities is given in Section 6.2.1. Furthermore, the tool has been developed with extensibility in mind. How this was achieved is described in Section 6.2.2.

6.2.1. Internal Architecture

The Clounaq tool is a web application, mainly written in TypeScript[67] and using the following main frameworks for its functionalities:

- VueJS[68], as the overall framework for the application structure and provisioning.
- JointJS[69], as the diagramming library for working on architectural models and viewing the quality model
- Mermaid[70], as the diagramming library for visualizing evaluation results.

The tool is a SPA which runs entirely in the browser of the user without an additional backend. This design is intentional to simplify hosting (it is currently hosted using GitHub Pages[71]) and support the reproducibility of our approach, because no additional setup is required to run and use the tool. It is open-source software, made available at GitHub[72], so that it can be reviewed and used freely. The decision for using JointJS as a core framework was based on a review of available

[67] `https://www.typescriptlang.org/`, visited 2025-10-20
[68] `https://vuejs.org/`, visited 2025-10-20
[69] `https://www.jointjs.com/`, visited 2025-10-20
[70] `https://mermaid.js.org/`, visited 2025-10-20
[71] `https://docs.github.com/en/pages`, visited 2025-10-20
[72] `https://github.com/r0light/cna-quality-tool`, visited 2025-10-20

diagramming libraries [70]. Another noteworthy option was mxgraph[73] which at the time of the review, however, already was in an archived state with no further development. An updated version was announced, but not available yet at that time, which is why the decision for JointJS was made. At the time of writing, the successor maxGraph[74] has been released, and it represents a suitable alternative which might have been chosen if it had been available as a stable release at the time of the development of the tool.

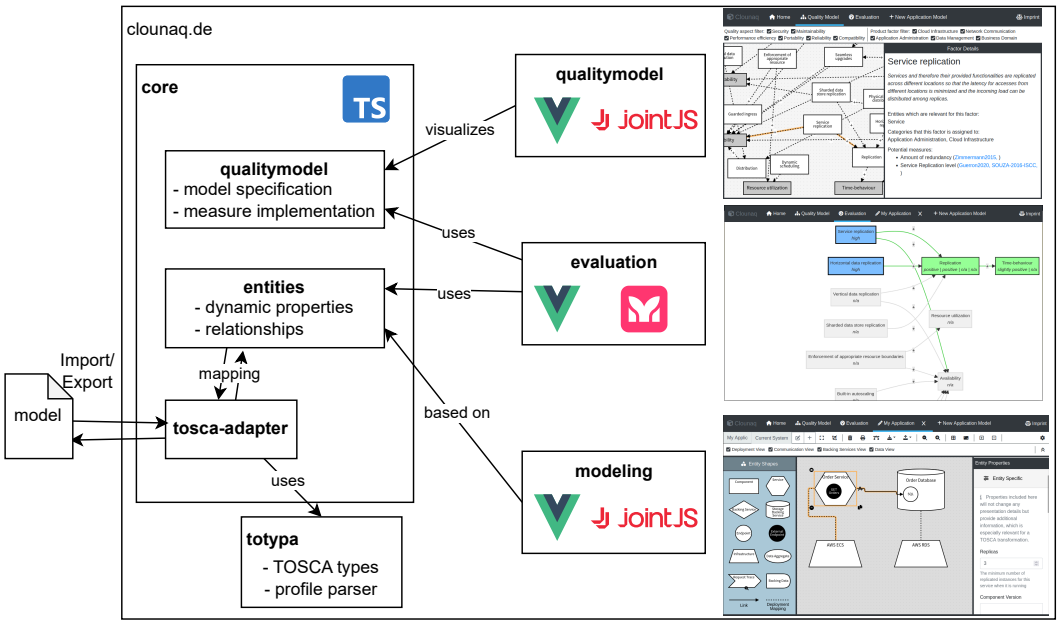


Figure 6.6.: The internal design of the Clounaq tool and corresponding views. [168]

Internally, the application is structured in five main parts represented by folders underneath `src` (see Figure 6.6): The heart of the application is implemented in the folder **`core`** using only TypeScript and none of the mentioned diagramming libraries to keep the core logic independent of specific frameworks. In `core/entities`, the entities (see Section 4.3) are implemented as plain TypeScript classes, but with a flexible `properties` attribute that is dynamically filled based on the entity properties defined in the included TOSCA profile (see Section 5.2). In `core/qualitymodel`, the quality model elements are implemented with several classes that enable a comfortable usage of the elements, for example, to perform an evaluation of an architecture. Also, the calculation of measures (see Section 4.6) is implemented here, solely based on the TypeScript classes for the entities. For each measure calculation, at least one test is included (in `tests`). The tests are implemented using Vitest[75].

In **`qualitymodel`**, the quality model tab (🖧) is implemented using VueJS and JointJS. It solely depends on the specification of the quality model in `core` and changes made to the quality model are automatically reflected in the quality model

[73] `https://github.com/jgraph/mxgraph`, visited 2025-10-20
[74] `https://github.com/maxGraph/maxGraph`, visited 2025-10-20
[75] `https://vitest.dev/`, visited 2025-10-20

view. Similarly, **evaluation** includes the code for the evaluation tab (), depending solely on the code in core to evaluate entities based on the specified quality model, implemented measures, and implemented evaluations. The specific visualizations for the *per Product Factor* and *per Quality Aspect* views are implemented using VueJS and Mermaid. The folder **modeling** provides the modeling tab () implementation based on VueJS and JointJS. The major connecting class is modeling/systemEntityManager.ts which includes code to transform entity objects into the diagramming elements specific to JointJS and vice versa. Within the modeling implementation, thus, entities are represented and modified in an additional form specific to JointJS. Within this specific form, each entity has a diagramming shape and a so-called model in which the property values are stored. The properties can be edited in the property details view which dynamically generates the editor fields based on the entity properties and their values.

Finally, for the import and export to the textual modeling format, the folder **totypa** (**to**sca **ty**pescript **pa**rser) includes a complete representation of the TOSCA standard using TypeScript type definitions. These are used by a parser script which can parse arbitrary TOSCA profiles to generate TypeScript objects that can then be used in applications. This is used to parse the TOSCA profile (presented in Section 5.2) so that it can be used for exporting and importing architectural models into and from TOSCA models.

6.2.2. Design for Extensibility

The way the tool was designed internally has been explicitly chosen to enable extensibility. It is extensible specifically in the following dimensions:

- Integrate changes to the quality model (create or change factors, impacts, measures, or evaluations)
- Add additional literature sources
- Update entity properties
- Add additional representation formats (textual or graphical)

To allow for an integration of changes to the quality model elements, the quality model is defined completely in JSON with no dependencies on other parts of the application. The JSON structure is stored within several constants in the file src/core/qualitymodel/specifications/qualitymodel.ts. For the different quality model elements, TypeScript type definitions for their specification are included to guide the modification through type checking. An example of such a type definition is shown in Listing 6.1 which shows the type definition for specifying product factors. The values for relevantEntities and applicableEntities are limited to the entity names which actually exist in the system. By requiring the type MeasureKey for specifying the measures relevant to a product factor, it can

be ensured that only measures can be referenced which are actually specified in the quality model.

Listing 6.1: Type definition for defining product factors

```
export type ProductFactorSpec = {
 name: string,
 description: string,
 categories: FactorCategoryKey[],
 relevantEntities: `${ENTITIES}`[],
 applicableEntities: (ENTITIES.SYSTEM | ENTITIES.COMPONENT | ENTITIES.
     INFRASTRUCTURE | ENTITIES.REQUEST_TRACE)[],
 sources: SourceSpec[],
 measures: MeasureKey[],
 evaluations: ProductFactorEvaluationSpec[]
}
```

For specifying product factors, the shown type definition needs to be satisfied. This is shown exemplary for the factor **Mostly stateless services** in Listing 6.2. The information about the product factors is then, for example, used for the presentation of the quality model, as shown in Figure 6.1.

Listing 6.2: Specification of the factor Mostly stateless services

```
"mostlyStatelessServices": {
 "name": "Mostly stateless services",
 "description": "Most services in a system are kept stateless, that means not
     requiring durable disk space on the infrastructure that they run on. Stateless
     services can be replaced, updated or replicated at any time. Stateful services
     are reduced to a minimum.",
 "categories": ["dataManagement", "businessDomain"],
 "relevantEntities": [ENTITIES.COMPONENT, ENTITIES.DATA_AGGREGATE],
 "applicableEntities": [ENTITIES.SYSTEM, ENTITIES.REQUEST_TRACE],
 "sources": [{ "key": "Davis2019", "section": "5.4" }, { "key": "Scholl2019", "
     section": "6 "Design Stateless Services That Scale Out" }, { "key": "
     Goniwada2021", "section": "3 Be Smart with State Principle, 5 Stateless
     Services" }],
 "measures": ["ratioOfStateDependencyOfEndpoints", "ratioOfStatefulComponents", "
     ratioOfStatelessComponents", "
     degreeToWhichComponentsAreLinkedToStatefulComponents"],
 "evaluations": [{
  "targetEntities": [ENTITIES.SYSTEM, ENTITIES.REQUEST_TRACE],
  "evaluation": "mostlyStatelessServices",
  "measures": ["ratioOfStatelessComponents", "
      degreeToWhichComponentsAreLinkedToStatefulComponents"],
  "reasoning": "This factor is fulfilled if the ratio of stateless services is
      rather high and if in addition the degree to which components are linked to
      stateful components is rather low. These two measures are aggregated."
 }]
},
```

If the quality model evolves further, for example, new product factors just need to be added in the shown format into the quality model specification, and they will be automatically rendered in the quality model tab (🖧) without requiring changes to the code that creates the quality model view. The same applies for most elements of the quality model: quality aspects, product factors, impacts, measures, evaluations.

When adding new measures to the quality model, also implementations for their calculation need to be provided. This requires more effort, but can be done entirely in the `core`. To add measure calculation implementations, the calculation must be implemented in TypeScript relying on the implemented entity classes. Different files for the different entities for which measures can be calculated are included. For example, the file `componentMeasures.ts` contains all implementa-

tions of measures for components. An extract is shown in Listing 6.3. It shows the implementation of the measure *Ratio of documented endpoints* for component entities.

Listing 6.3: Implementation of the measure Ratio of documented endpoints for components

```
export const ratioOfDocumentedEndpoints: Calculation = (parameters:
    CalculationParameters<Component>) => {
 let allComponentEndpoints = parameters.entity.getEndpointEntities.concat(
     parameters.entity.getExternalEndpointEntities);

 if (allComponentEndpoints.length === 0) {
  return "n/a";
 }

 let documented = allComponentEndpoints.filter(endpoint => endpoint.getDocumentedBy
     .length > 0);

 return documented.length / allComponentEndpoints.length;
}
```

The `parameters` object which is provided to the measure calculation includes two attributes: `entity` representing the entity for which the measure should be calculated and `system` representing the current system entity. Through the `system` object, all other entities can be accessed, if required for the calculation. Thus, when extending the tool by adding new measures, a single function for each applicable entity needs to be added which can rely on all model information through the provided parameters. The return type of the calculation function is specified as:

```
type MeasureValue = number | string | "n/a";
```

The value `"n/a"` should be used, if a calculation of the measure is not possible due to missing information.

Additional literature to reference for product factors or measures, can be added in the file `literature.ts`, using the type definition shown in Listing 6.4.

Listing 6.4: Type definition for defining literature sources

```
type LiteratureSpec = {
 title: string,
 year: number,
 authors: string,
 publisher: string,
 url: string
}
```

A `url` should always be included here so that a hyperlink can be offered in the tool to guide users to further information on product factors and measures. A unique key needs to be assigned to each literature source which is used, for example, in the `sources` attribute of a product factor specification (see Listing 6.2).

Changes to the entities of the quality model are more difficult to introduce, because that would require changes to both the textual representation format and the graphical representation format. Changes to the graphical representation format would in turn also needed to be introduced into the modeling tab (✎) of the tool. For example, a new shape would have to be added for a new entity. What is supported more easily, however, is introducing changes to the entity properties. The

way the tool is built, changes to entity properties only need to be done in the corresponding TOSCA profile (see Section 5.2). By parsing the changed profile, changes to properties are automatically introduced into the tool. This can be done with a prepared `npm` script, that can be started from the root of the tool folder using:

```
$ npm run parseProfiles
```

If an existing type is used for a new property, the property editor section in the modeling tab (✎) can automatically provide a corresponding input field.

Finally, through the architecture of the tool as presented in Section 6.2.1, an extension of the tool with new textual and graphical representation formats is generally possible. This is due to the strict separation of the core quality model specification from the functionalities for exporting or importing a model into TOSCA and from the graphical modeling functionality implemented with JointJS. If, for example, an additional textual representation format should be supported, an adapter in parallel to the **tosca-adapter** (see Figure 6.6) needs to be integrated. That adapter should transform models of the new textual format into the corresponding TypeScript entity classes. The evaluation and modeling functionalities could remain largely untouched, because they only rely on the core TypeScript entity classes. Thus, an extension of the tool to support textual models of specific technologies, such as IaC tools, is generally possible. In turn, if a new graphical representation format should be introduced, a new modeling functionality would have to be implemented which relies again on the core TypeScript entity classes. That way, the evaluation functionality would still function, even with a new graphical representation format and also the export and import to the TOSCA-based textual format would be possible. Because the tool is open source and available on GitHub, anybody can create a fork of it and introduce her own specific modifications or extensions and furthermore easily provision it locally or as a new static website.

6.3. Applying Clounaq to a Use Case

In this section, hypothesis 4 *("By applying architectural measures to a modeled software architecture, impacts on quality aspects can be evaluated")* is supported.

To showcase the intended usage of the Clounaq tool to support the overall quality evaluation approach, a small use case is presented in the following. It is explicitly kept simple to keep the consequences comprehensible. The use case builds on the already presented evaluation of the LakeSideMutual application [173]. Following the actions shown in Figure 6.5, the steps of getting informed about the quality model, modeling the application architecture (A), and performing a first evaluation (B) have already been done. The graphical representation of the architectural model is shown in Figure 5.5 and the evaluation results on a system level are presented in Section 4.7.5.2. From the evaluation results it can be seen that the factor **Isolated state** is evaluated as *none* which is mainly because also the factor **Specialized stateful services** is evaluated as *none*. To improve the impact on **Elasti-**

city for a production deployment of the application, a user of the Clounaq tool could therefore decide to introduce explicit databases instead of in-memory databases directly included in the backend services.

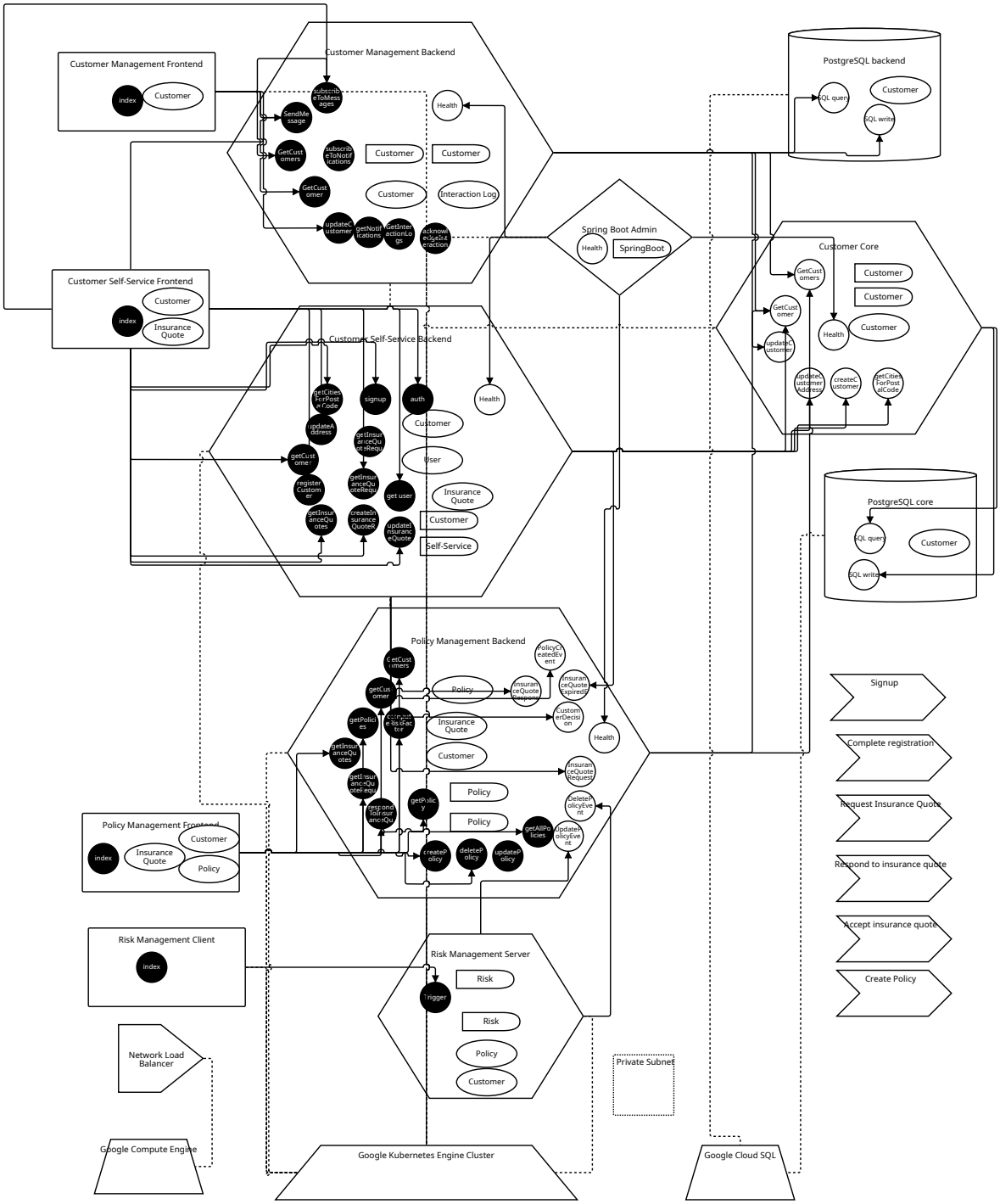


Figure 6.7.: LakeSideMutual System overview after change

Because the LakeSideMutual application is already deployed in the Google Cloud, a reasonable choice could thus be to use Google Cloud SQL[76] instances. To investigate the impact of this change, the model can be modified in the Clounaq

[76] `https://cloud.google.com/sql`, visited 2025-10-20

application (C_2). This is shown in Figure 6.7 where explicit databases have been introduced for the *Customer Backend Service* and the *Customer Core Service*. As a result, these two service can now be characterized as stateless.

When evaluating the changed application architecture again (B), the effects become obvious. This is summarized in Table 6.1 where the evaluation results for selected factors are presented once for the **original** application architecture and once for the **changed** application architecture.

Table 6.1.: LakeSideMutual evaluation results before and after change

Factor/*Measure*	**LakeSideMutual original**	**LakeSideMutual changed**
Time-behavior	neutral	neutral
Replication	none	none
Service replication	none	none
Amount of redundancy	1.000	1.000
Service replication level	1.000	1.000
Horizontal data replication	n/a	none
Storage replication level	n/a	1.000
Vertical data replication	none	none
Ratio of cached data aggregates	0.000	0.000
Sharded data store replication	n/a	none
Level of sharding across storage backing services	n/a	1.000
Elasticity	positive	positive
Isolated state	none	**moderate**
Mostly stateless services	moderate	moderate
Ratio of stateless components	0.636	0.692
Degree to which components are linked to stateful components	0.889	0.500
Specialized stateful services	none	**moderate**
Ratio of specialized stateful services	0.000	**0.500**
Ratio of suitably replicated stateful services	n/a	n/a
Built-in autoscaling	high	high
Deployed entities autoscaling	1.000	0.846
Infrastructure autoscaling	0.500	0.667
Simplicity	mixed	mixed
Usage of existing solutions for non-core capabilities	none	**moderate**
Ratio of non-custom backing services	0.000	**0.750**
Sparsity	none	none
Number of components	11.000	13.000
Operation outsourcing	low	low
Ratio of provider-managed components and infrastructure	0.154	**0.375**
Managed infrastructure	low	moderate
Ratio of fully managed infrastructure	0.500	0.667
Managed backing services	none	**moderate**
Ratio of managed backing services	0.000	**0.750**

The factor **Isolated state** is now evaluated as **moderate** in the changed application architecture, because the factor **Specialized stateful services** is now evaluated

as **moderate**. While there were no specialized stateful services included in the application before, two of the backend services have been made stateless and the state was moved to specialized storage backing services. The change has only been applied to two services, while *Customer Self-Service Backend* and *Policy Management Backend* are still stateful services because of their included in-memory databases. Therefore, the measure *Ratio of specialized stateful services* has the value 0.5 for the changed architecture. This could be changed further by also introducing explicit databases for these remaining services.

Another difference that becomes visible is that the *Ratio of non-custom backing services* has increased, because the added databases represent proven existing solutions instead of a custom data storage approach. And also the *Ratio of managed backing services* has increased, because managed database instances were chosen. This, in turn, increases the *Ratio of provider-managed components and infrastructure*, but not to the extent that **Operation outsourcing** is evaluated different than **low**.

This small presented use case demonstrates how the Clounaq tool can be used for the overall quality evaluation approach and for evolving application architectures driven by such evaluations. The used product factors and architectural measures can quantify architectural decisions to support the decision-making on implementation choices. Whether to actually apply changes to an application based on these evaluations, however, needs to be decided by the responsible architects.

Part III.

Implications and Conclusion

> Sir Bedevere: ...and that, my liege, is how we know the Earth to be banana-shaped.
> King Arthur: This new learning amazes me, Sir Bedevere. Explain again how sheep's bladders may be employed to prevent earthquakes.
>
> *Monty Python and the Holy Grail, 1975*

7. Discussion

In this chapter, the results presented in Part II are discussed to reconsider the initially stated hypotheses (7.1), derive implications for theory and practice (7.2), highlight limitations of the current state of the approach (7.3), and present opportunities for future work (7.4).

7.1. Evaluation of Hypotheses

The initially formulated hypotheses stated in Section 1.2 can be evaluated using the results presented in Part II.

Hypothesis 1 ("*Cloud-native spans a broad range of aspects*") is supported by the review of cloud-native as a topic in Section 2.1 which serves as a foundation for the results of this work. It shows that cloud-native as a topic spans a broad range of aspects, including application design, technology selection, and deployment infrastructure. The fact that these aspects need to be considered in combination is substantiated by the quality model presented in Chapter 4. For example, to evaluate the elasticity of an application architecture, it is on the one hand important to consider the chosen deployment infrastructure to support autoscaling [255] and on the other hand to also consider the application design, specifically regarding the distribution of state [118, 255]. Regarding modifiability, services of an application should be kept independent by decentralizing functionalities and thus reduce impacts of changes in one services on other services [108]. But this also requires automation [241] of the chosen infrastructure, since a provisioning of multiple decentralized services would otherwise lead to increased operational effort. As a consequence of hypothesis 1, also the factor categories (see Section 4.4) have been formulated to highlight the fact that various aspects are covered.

The quality model presented in Chapter 4 substantiates **Hypothesis 2** ("*A hierarchical quality model is able to express cloud-native architectural design decisions.*"). The application characteristics are covered by the product factors of the quality model while the consequences of design decisions become traceable through the impact relationships, at least for the considered quality aspects. Through the hierarchical approach, aspects of cloud-native application characteristics can be viewed and evaluated from different dimensions which helps in handling the complexity.

The modeling approach presented in Chapter 5 as the realization of the formal specification of entities in Section 4.3 is the basis for supporting **Hypothesis 3** ("*An architectural model can be used to express cloud-native application characteristics.*"). By the modeling of exemplary applications [173] as shown in Section 5.2.5 and Sec-

tion 5.3.2 with this approach, the hypothesis is validated. It has to be noted, however, that a modeling approach specific to the quality model had to be developed which can only partly rely on existing modeling approaches and open industry standards, i.e. TOSCA. Application characteristics are on the one hand expressible through the usage of different types of entities together with their relationship and on the other hand through the specification of more detailed entity properties.

Finally, **Hypothesis 4** ("*By applying architectural measures to a modeled software architecture, impacts on quality aspects can be evaluated*") is supported through the measure-based quantification of software architectures as shown in Section 4.7.5.2 and Section 6.3. An important aspect here is that the applied quantification approach is static, focusing on the modeled components and their relationships. This was a conscious decision, as the inclusion of an additional quantification of runtime aspects would have meant an escalation of complexity. It has been found, however, that quantification should not be constrained to the architecture of a system as a whole, because aggregation of quantification often masks important details. The quantification and evaluation approach thus also enables the partial evaluation of a system architecture through a focus on different types of entities.

7.2. Implications

From the results of this work, the following five implications can be drawn, with both consequences for theory and practice.

A certain level of abstraction needs to be chosen for the analysis of software architectures according to cloud-native characteristics.

As an overall implication from the formulation process of the quality model, it can be stated that a decision had to be made about which level of abstraction to use for the quality model. This is mainly because of the inherent complexity of the topic of cloud-native, and it applies specifically to the entities of the quality model and their properties. While the product factors as such can be formulated on the basis of an arbitrary level of abstraction, enabling their quantification requires a decision about which information to include about an architecture and which not. Based on this decision, entities are formulated, and their properties are defined. In the case of the presented quality model, the level of abstraction was determined during the iterative formulation of the quality model factors (3.1.1) and entities (3.1.4). More specifically, the chosen level of abstraction is tailored to the factors of the quality model which excludes certain more detailed information. This applies, for example, to message content which was not considered, although certain measures for cohesion rely on it [133]. Or, as another example, detailed scaling policies are not represented. Such policies can be evaluated in detail, as done by Ilyushkin et al. [119], but for the factor **Built-in autoscaling**, the entities merely contain whether autoscaling is used or not. Furthermore, detailed service intern-

als are abstracted, for example, information about internal implementations of functionalities which can be used for measuring cohesion in a service [224].

As a theoretical implication, it can be stated that a quality model for evaluating cloud-native software architectures should clearly define the used level of abstraction. It can be defined before the formulation process or derived as a result during the formulation process. For practice, this means, before applying a quality evaluation approach, the desired evaluation goals should be clear and the evaluation approach should be aligned with these goals regarding the used level of abstraction. If a more detailed level is desired, a more detailed approach might be more suitable which then however might ignore details on certain other aspects.

Formulating meaningful quality models requires a structured approach and continuous validation.

To formulate a quality model that is meaningful, it must be clearly understandable which significance its elements have and how the statements made within the quality model are justified. By relying on an existing approach or metamodel for formulating quality models, the types of elements of the quality model are clearly defined, facilitating the formulation of elements based on these types. The Quamoco metamodel [294] has proven its usefulness in this regard in this work. However, it is specific to hierarchical quality models and other types of quality models might require a different kind of theoretical basis.

A quality *model* explains real-world circumstances from a certain point of view, potentially on the basis of assumptions. Therefore, the statements made by a quality model need initial justification and additional validation. The initial justification is based on the structured process applied in this work for formulating the quality model (see Section 3.1). Regarding validation, different types of approaches exist [170] and should be applied throughout the formulation process, if suitable. Additionally, a quality model should be continuously validated through its usage and application.

The approach shown in this work explicitly considers validation throughout the formulation process. Validation steps have been performed for the presented quality model considering impacts (see Section 3.1.3 and Section 4.5), considering measures (see Section 3.1.5 and Section 4.6.2), and considering evaluations (see Section 3.1.6 and Section 4.7.5). However, regarding the measure validation, only a few selected measures were validated in detail, due to the required effort to do so. Nevertheless, as a theoretical contribution, the general approach to measure validation is shown and can be continued or applied in other work.

To support an evaluation of quality in a general way, quantification must be applied with caution.

While the aim of the presented quality model is to enable quality evaluations of software architectures regarding cloud-native characteristics in a general way, a de-

tailed quantitative evaluation of quality always needs to be in relation to the quality requirements of an individual application (see Section 2.2). Including quantification into the quality model, while necessary for enabling actual evaluations, can thus only be done to a certain extent. This can also be seen from the results of the measure validation experiment [172] (see Section 4.6.2). The positive impact of **Service replication** on **Time-behavior** is evident, but deriving a specific quantitative functional relationship between the factor and the quality aspect would only apply to an individual application, not in a general way.

Thus, for the presented quality model, the decision was made to include quantification only for the used architectural measures to enable an evaluation of individual factors. But for the derived impact relationships and specifically the evaluation of quality aspects, only discrete values are used to avoid providing a more detailed aggregated result which might not be meaningful. This also results in the possibility of evaluating a quality aspect as having a *mixed* result (see Section 4.7.4).

As an implication for theory, for quality models and quality evaluation approaches, it should be made explicit to what extent quantification is included and meaningful. To use quantification to a larger extent, also the quality requirements of an application need to be stated in a quantitative way.

For practice, this aspect needs to be kept in mind as well when performing quality evaluations of a software architecture. For an approach as presented in this work, which aims to evaluate quality aspects in a general way, individual quality requirements need to be included in the decision-making process as well, while the evaluation serves as an informational basis.

Deriving actions from quality evaluations also requires a consideration of cost

An aspect which has been paid little attention to so far, but which is of great importance in the context of cloud computing, specifically in practice, is cost [183]. The reason that it has not been addressed in detail yet, is that it would add additional complexity to the quality model, and there is the question of how to properly integrate it. In general, "low cost" could be seen as an additional quality aspect, potentially in terms of *performance efficiency*. But, also given the context of the previous implication, an evaluation of whether the cost of operating an application in the cloud is low or high, is relative to the individual application requirements at hand.

It is nevertheless a fact, that the different factors of the presented quality model can also have an impact on the cost of operating an application. This has become apparent during the formulation of the product factors. For making decisions about how to evolve an architecture, cost thus needs to be considered as an influencing factor.

To address this, the product factors of the quality model have been additionally evaluated regarding their impact on cost. For each factor, it was evaluated whether the presence of the factor has an impact on (1) the **cloud resource cost** (called *service*

charge by Martens et al. [183]) and (2) the **operational effort** (aggregating *implementation, configuration, integration, and migration* and *maintenance and modification* as defined by Martens et al. [183]). The **cloud resource cost** refers to the amount of money which has to be paid to a cloud provider for the used resources. The **operational effort** refers to the expenditure of time required for the product factor.

The evaluation is presented in Table 7.1 where for each leaf product factor (those factors which are not impacted by other factors) an assessment is given whether cloud resource cost or operational effort **increase** or **decrease** when the factor is present in a system. If no impact is expected, a factor is rated as none. For several aspects, an impact **depends on** additional considerations. If this is the case, an impact is stated as **potential** and the reason is provided in the **depends on** column. What needs to be considered in addition can be either **make or buy** or **setup**. **Make or buy** refers to the decision of whether additional cloud services are bought to realize the product factor or it is realized through an own implementation. For example, to realize the factor **Consistent centralized logging** either a logging service offered and managed by a cloud provider can be used (buy) or it can be implemented in a custom way (make). Although both possibilities will probably result in higher cloud resource cost, if assuming the custom implementation is also deployed in the cloud, there is a difference in the amount of increase. The operational effort will increase more, if an own implementation is used.

Setup refers to the fact, that factors can be realized in different ways which can impact cost in different ways. Actual impacts depend on whether new components are introduced or not, how such new components are configured (e.g., how many resources are allocated), and fundamentally whether there are alternative implementations which do not impact costs. An example for this, is how the factor **Asynchronous communication** is realized. If an additional cloud message broker service is added to an architecture, this significantly increases the cloud resource cost. If a broker is added within an existing component (as also done in the LakeSideMutual sample application(see Section 4.7.5.1)), the impact on additional cloud resource cost may be negligible.

The theoretical implication is that cost aspects should be considered when formulating and working on quality models and quality evaluation approaches, especially in the cloud context. The presented quality model admittedly does not cover cost-related aspects in depth to focus on the core technical characteristics of cloud-native applications. But to account for that, the evaluation from Table 7.1 can be a basis for including it as an aspect in the quality model. For practitioners, it should be clear that aiming for a higher quality of application architectures often comes with increased cost. When using the presented quality evaluation approach, cost aspects should be considered as well, before actually creating or making changes to a software architecture. Again the presented cost evaluation of product factors can provide guidance in that regard.

Table 7.1.: Assessment of leaf factor impacts on cost and effort

Product Factor	Cloud resource cost	Operational effort	depends on
Data encryption in transit	none	none	
Isolated secrets	none	none	
Secrets stored in specialized services	potential increase	slight increase	make or buy
Least-privileged access	none	none	
Access control management consistency	none	none	
Account separation	none	none	
Authentication delegation	potential increase	potential increase	make or buy
Limited data scope	none	none	
Limited endpoint scope	none	none	
Command query responsibility segregation	potential increase	increase	setup
Separation by gateways	increase	increase	
Mostly stateless services	none	none	
Specialized stateful services	potential increase	potential increase	setup
Asynchronous communication	potential increase	potential increase	setup
Communication partner abstraction	potential increase	potential increase	setup
Persistent communication	slight increase	slight increase	
Usage of existing solutions for non-core capabilities	potential increase	potential decrease	make or buy + setup
Standardization	none	potential decrease	
Component similarity	none	potential decrease	
Consistent centralized logging	potential increase	potential increase	make or buy
Consistent centralized metrics	potential increase	potential increase	make or buy
Distributed tracing of invocations	potential increase	potential increase	make or buy
Health and readiness checks	none	none	
Automated infrastructure provisioning	none	decrease	
Use infrastructure as code	none	potential decrease	setup
Dynamic scheduling	potential decrease	none	setup
Low coupling	none	none	
Functional decentralization	potential increase	potential increase	setup
Limited request trace scope	none	none	
Logical grouping	none	none	
Backing service decentralization	potential increase	increase	setup
Addressing abstraction	none	none	
Sparsity	potential decrease	potential decrease	setup
Managed infrastructure	increase	decrease	
Managed backing services	increase	decrease	
Service replication	increase	potential increase	setup
Horizontal data replication	increase	potential increase	setup
Vertical data replication	potential increase	increase	setup
Sharded data store replication	potential increase	increase	setup
Enforcement of appropriate resource boundaries	potential decrease	none	setup
Built-in autoscaling	potential de- or increase	potential de- or increase	setup
Infrastructure abstraction	none	none	
Cloud vendor abstraction	none	none	
Isolated configuration	none	none	
Configuration stored in specialized services	potential increase	slight increase	make or buy
Contract-based links	none	none	
Standardized self-contained deployment unit	none	none	
Immutable artifacts	none	none	
Guarded ingress	potential increase	potential increase	make or buy
Physical data distribution	increase	increase	
Physical service distribution	increase	increase	
Rolling upgrades enabled	slight increase	slight increase	
Automated infrastructure maintenance	potential increase	decrease	setup
Invocation timeouts	none	none	
Retries for safe invocations	none	none	
Circuit breaked communication	none	none	
API-based communication	none	none	
Consistently mediated communication	increase	increase	

Technological heterogeneity is a major challenge for automated modeling and evaluation of software architectures.

As already stated in the introduction (Section 1.2), technological heterogeneity is a major challenge for research on and analysis of cloud-native software architectures. This becomes especially evident in the context of finding a suitable representation format for cloud-native software architectures (Chapter 5). For practical applicability and repeatability, automation in all steps of a quality evaluation approach is desirable. But for automation, it must be possible to programmatically capture information about a software architecture. If the automated capturing of architectural information is implemented for one technology stack, it works for that, but not other technology stacks. If instead a manual capturing (in this case modeling) approach is chosen, different technologies can be supported, but for an automated capturing each technology stack would have to be supported individually. In this work, a choice was nevertheless made for a more abstract manual modeling approach, so that it can be used for various technology stacks. This still allows for a possible automation of modeling through the implementation of adapters specific to certain technologies.

In essence, researchers need to decide which approach to follow regarding their specific research goals. Relying on existing tools and assets can often be beneficial, if available. For practitioners, the choice of a certain technology stack may influence also the availability and applicability of such analysis and evaluation approaches. This can be a factor in technology selection decisions.

7.3. Limitations

For a genuine discussion of the presented quality model and quality evaluation approach, also its limitations need to be named. These are separated into quality model limitations (7.3.1), methodological limitations (7.3.2), and limitations regarding practical applicability (7.3.3).

7.3.1. Quality Model Limitations

Quality model limitations consider the quality model and its elements as such. One limitation is connected to the already stated implication that quantification needs to be applied with caution. Quality is considered by the quality model in a general way, although in the end it is a relative perspective. A relative perspective on quality includes the specific requirements of an application. In the current form of the quality model, it is not possible to state requirements of an application, partly because this is also not explicitly included in the Quamoco metamodel [294]. The other part of the reason is that the evaluation results are intended to serve as an information basis to decide on potential changes to an architecture, not as hard quality goals that need to be met. Considering Quamoco, requirements of

an application could be indirectly included in the evaluation functions for product factors (4.7.1) and quality aspects (4.7.4). From a general perspective, it would be possible to provide a possibility for stating the importance of certain factors and quality aspects during a quality evaluation.

Another limitation is that, to keep the evaluations comprehensible, evaluation results are discrete markings of a comparatively small range (none, low, moderate, high). Because continuous measure values are translated into these discrete values, small modifications might be hidden in evaluation results [198], because they do not lead to significant measure value changes. To account for this, specific measure values are always available together with the evaluation results.

The impacts of the quality model are currently formulated in a uni-directional way, meaning that an impact only has an effect, if the corresponding factor is present. But the non-presence of a factor does not have an inverse effect on the impacted factor. This is also a result of using Quamoco as a basis. A valid question would be whether it could make evaluations more meaningful if a bidirectional impact is possible. However, to not further increase the complexity of evaluations, it was decided to keep the uni-directional way. For some factors, however, it could make sense to extend the quality model in this regard, for example **Low coupling**.

Finally, a motivation for the development of the quality model was also to highlight trade-offs between quality aspects [286]. This is generally possible, because impacts can be positive or negative. However, there are currently very few negative impacts included in the quality model. This is the result of the chosen approach for formulating the quality model factors which relied on literature describing mostly how to design cloud-native application to gain positive impacts on quality aspects. The quality model could thus be evolved further to better depict trade-offs, but this should be done in a structured approach.

7.3.2. Methodological Limitations

Methodological limitations consider the applied research methods and how limitations regarding them may impact the research results. Regarding the formulation of the initial quality model factors, a selected set of practitioner books was chosen (3.1.1). Thus, the resulting factors are impacted by this selection and other source might have lead to a different outcome. However, with the top-down approach of starting with available definitions of cloud-native it was ensured that a broad perspective is taken without missing important aspects. The selected practitioner books then added knowledge to the quality model within the provided context of the definitions. Through the continuous validation, it can furthermore be ensured that a potential bias introduced by the chosen literature is corrected.

Considering the impact validation survey (3.1.3) in specific also limitations need to be reported. The participants were self-recruited, meaning that although the survey was distributed in a controlled way, it was open to anyone interested. To mitigate the risk of participants being insufficiently qualified, the intended target

group was described in detail at the beginning of the survey to make participants aware. Due to the number of participants, they are not representative for the whole target group, and the results are thus not fully generalizable. To ensure construct validity, that means whether the factors are understandable and relevant, a pilot study was used for feedback on the individual factors. They were reformulated when necessary. Nevertheless, due to the novelty of the topic of cloud-native, no existing pre-validated factor descriptions were available. Thus, factor descriptions may contain ambiguities and could have been understood in different ways.

The measure validation (3.1.5) only considered a small selection of measures from the quality model. Further validation studies should consider the remaining measures, especially the newly formulated ones. However, for certain quality aspects, such as those related to maintainability, a validation of measures requires a higher effort. Ideally, software architectures should be observed over a longer time span to evaluate impacts of different measure values on quality aspects.

Similarly, the overall validation of the approach, specifically including evaluations needs a continuous effort. So far the evaluation approach has been applied to three exemplary application architectures (3.1.6). The evaluation results are thus based on the specific characteristics of these applications. It was ensured that the selected applications are relevant within the context of cloud-native, typically microservices-based, applications. They are, however, not representative for all kinds of applications for which the evaluation approach should be applicable.

7.3.3. Limitations regarding Practical Applicability

Practical limitations consider mostly how the quality evaluation approach is intended to be used and the Clounaq tool as such.

One major limitation has been mentioned in the implications already: Modeling of software architectures is currently a manual process. It would significantly facilitate the usage of tooling, if models could be created automatically, based on artifacts such as source code or deployment descriptors. But the challenge is the already mentioned heterogeneity of technologies available in the context of cloud-native applications to which tooling would need to be adapted. Ntentos et al. [207] have taken this approach of generating models automatically from IaC artifacts, but had to start with one specific technology to then continuously include further technologies. For the quality evaluation process as presented in this work, the focus was set on formulating the quality model itself and enabling quality evaluations in a broad context. Therefore, automating model generation has not been in focus, but would be desirable. Through using TOSCA as a basis, a foundation for this is laid. If TOSCA profiles for specific technologies are available and architectures are modeled with these, substitution mappings could be used to map from technology-specific node types to the node types of the presented modeling approach (5.2). Thus, profiles could be automatically transformed and then used in the quality evaluation approach.

Regarding the Clounaq tool itself, a focus has been set on the core functionalities of creating, modifying, importing, and exporting architectural models. There are, however, some limitations regarding the comfortable use of the modeling functionality (6.1.2). It is, for example, not possible to undo an action which is unpleasant if an entity was deleted by accident. Or a multi-select for moving several entities simultaneously is also not available. The tool could thus be further enhanced with typical diagramming functionalities to make it more user-friendly.

Finally, considering the applicability of the quality evaluation approach itself, in its current form, only the evaluation results are reported. But it is up to the user to understand the results and derive potential actions for the application at hand. In general, it would be possible to formulate recommended actions on the basis of the evaluation results which also include explanations on how to change an architecture in order to fulfill certain factors, if desired.

7.4. Future Work

There are several possibilities for extending the presented work in the future. As already mentioned as an implication, to keep the quality model relevant and meaningful, continuous validation is important. With the current state of the quality model and the Clounaq tool being available now, a major path for future work is the application of the evaluation approach to additional, more complex, real-world applications (similar to the approach presented in Section 3.1.6). Through this, the statements made by the quality model can be validated and potentially evolved further. The modeling approach can be further validated regarding its ability to capture characteristics of diverse applications. And the evaluation approach and results can be validated further regarding their helpfulness in deriving actions to evolve software architectures. By additionally including potential users (software architects and software developers) of the Clounaq tool, its ease of use and helpfulness can be improved via direct user feedback.

In this regard, also the extension of the modeling approach and the automation of model creation and modification is another major path for future work. Through the adaptation to specific technologies, the overall applicability of the modeling and evaluation approach can be validated and the potential for evaluating heterogeneous architectures can be improved. This could be done iteratively through starting with commonly used technologies and cloud providers and supporting further technologies step by step.

With a further validated quality model and data from additional use cases, the next step is to fully implement a software architecture optimization method [7]. On top of representing and evaluating the current state of a software architecture, this means also identifying and implementing actions that lead to an optimized architecture. To do so, the current limitations need to be addressed and the suggestions from the two previously mentioned paths for future work should be realized.

Hey, hey, I've seen this one.
I've seen this one.

Marty McFly (in Back to the Future, 1985)

8. Related Work

Parts of this chapter have been taken from [171].

Relations to other work can on the one hand be drawn based on the topical proximity, considering other work focusing on cloud-native software architectures or subsumed topics, such as microservices-based architectures. And on the other hand, work focusing on the development of approaches for quality analysis and improvement of software architectures can be seen as related. Especially, regarding the second area of work, the presented approach can be seen in the overall context of architecture optimization approaches for which a taxonomy has been developed and used by Aleti et al. [7]. By using their taxonomy, this work can be set in relation to others, which has already been done in previous work [171]. The taxonomy has also been used already to review architecture optimization approaches regarding cloud applications in general [93]. By integrating results from these previous works and updating the comparison based on the current state of the Clounaq approach, it can be compared in a structured way to such related work. This is shown in Table 8.1.

The taxonomy by Aleti et al. characterizes software architecture optimization approaches according to the categories **problem**, **solution**, and **validation**. For each category, subcategories are defined which cover specific aspects and enable a classification of approaches according to them.

The overall scope is set by the **problem** category which characterizes the problem tackled by an approach. It can be differentiated based on the *domain*, the *phase* within the software development lifecycle in which it is used, the *quality attributes* which are to be optimized, and *constraints* which are specified upfront and limit the potential solution space. Furthermore, with the *dimensionality* it can be expressed whether there is a single objective or there are multiple quality dimensions considered simultaneously, potentially conflicting with each other.

From the perspective of targeted domains, approaches considering *Cloud deployment* [2, 54, 57] and *Microservices* [206, 267] are the most closely related approaches to Clounaq. In a way, Clounaq combines these two domains in its approach. The approach by Mayr et al. [184] has been selected for the comparison, because it also uses a hierarchical quality model. The use of a hierarchical quality model leads to both approaches being *multi-objective*. While most of the other approaches also consider multiple objectives, their quality attributes and how these are impacted are not structured in an explicit quality model. Nearly all approaches only target the *design time*, as typically an architecture is optimized before its deployment.

Table 8.1.: The Clounaq approach compared to other work using the taxonomy by Aleti et al. [7] (in parts presented previously [93, 171])

Category	Clounaq	Mayr et al. [184]	Achilleos et al. [2]	Soldani et al. [267]	Ntentos et al. [206]	Ciavotta et al. [54]	Cortellessa et al. [57]
				Problem			
Domain	Cloud-native	Embedded systems	Cloud deployment	Microservices	Microservices	Cloud deployment	Cloud deployment
Phase	Design time	Design time	Design- and runtime	Design time	Design time	Design time	Design time
Quality Attribute	multiple	multiple, based on requirements	SLOs focusing on performance, cost, security	Design principles	Design principles	Cost	Cost, performance, power consumption
Dimensionality	multi-objective	multi-objective	multi-objective	multi-objective	multi-objective	single-objective	multi-objective
Constraint	implicit	implicit	custom, e.g. cost, hardware	external	external	performance, resource limits	performance, cost, power consumption
				Solution			
Architecture Representation	TOSCA Profile	Source code	CAMEL	μTOSCA	CodeableModels	Palladio	UML
Quality Evaluation	model-based	model-based	model-based	model-based	model-based	model-based	model-based
Degrees of Freedom	Component types, interactions and deployment	Source Code changes	Deployment configuration	Component interactions and configuration	Components, component interactions	Deployment configuration, component configuration	Deployment configuration, component configuration
Optimization Strategy	problem-specific, approximate	problem-specific, exact	problem-specific, approximate	problem-specific, approximate	problem-specific, approximate	linear integer programming, approximate	genetic algorithm, approximate
Constraint Handling	general	penalty	prohibit	general	general	prohibit	prohibit
				Validation			
Approach Validation	Survey, experiment, use cases	Expert judgment	Use cases and survey	Use cases	Use cases	Use case	Use case
Optimization Validation	none	Expert judgment	Use cases	Use cases	Use cases	Use case	none

While Clounaq targets multiple quality attributes (of the ISO 25000 standard [122]) and Mayr et al. [184] focus on requirements derived from expert knowledge, the approaches focusing on *Cloud deployment* are typically interested in *cost* [2, 54, 57] and *performance* [2, 57]. In the domain of *microservices*, core *design principles* of this architectural style are in the focus and evaluated by the approaches [206, 267]. Constraints are included *implicitly* in hierarchical quality models [184], based on the conflicting quality attributes in the context of deployments [2, 54, 57], and can be formulated *externally* for the approaches focusing on microservices design principles [206, 267]. This means, external constraints can be formulated to allow violations against design principles to a certain extent, if explicitly required by application developers.

Within the **solution** category, it becomes apparent, that all approaches are *model-based*. The specific models which are used, however, differ significantly. Source code is used directly by Mayr et al. [184], while custom *architectural representation* formats are used by Achilleos et al.[77] [2] and Ntentos et al.[78] [206]. The alternative is to rely on already existing representation formats, namely TOSCA, Palladio[79], and UML. It is noteworthy, however, that all approaches relying on an existing format [54, 57, 267], had to extend it with custom additions. *Degrees of Freedom* cover the actions available to an approach for changing an architecture and reach the optimization goal. As it can be seen, these are based on the corresponding used representation formats and while the deployment-focused approaches change deployment configurations [2, 54, 57], the microservices-focused approaches also consider changes to how components interact [206, 267]. The employed *optimization strategy* is mostly *problem-specific*, that means a custom approach for improving an architecture is implemented. And it is typically *approximate*, meaning that there is no globally optimal solution. The approach of Mayr et al. [184] can be categorized as being more exact, because of the focus on requirements which allows for an optimization in terms of either fulfilling requirements or not. While also being *approximate*, the approaches from Ciavotta et al. [54] and Cortellessa et al. [57] employ existing proven mathematical approaches for optimization problems. In this regard formulated *constraints* are strict in the sense that their violation is *prohibited* [2, 54, 57]. Another possibility is to allow for constraint violation, but reacting to them with a *penalty* [184]. *General* covers all other options for handling constraints. In the considered works rated as *general* [206, 267], constraint violations can be tolerated, but are then reported or have additional consequences.

Finally, a comparison is possible regarding how an approach is **validated** and whether an optimization done through the approach has been validated. All approaches perform general validations, with use cases being the most commonly applied method. Except for Cortellessa et al. [57] and the approach in this work, the

[77] `https://camel-dsl.org/`, visited 2025-10-20
[78] `https://github.com/uzdun/CodeableModels`, visited 2025-10-20
[79] `https://www.palladio-simulator.com/`, visited 2025-10-20

compared approaches also perform a validation of results gained from performing optimizations, although these are typically integrated with the general validation.

With the shown comparison, the Clounaq approach is set in the broader context of architecture optimization approaches. A selection of other work was used for the comparison, but based on the taxonomy of Aleti et al. [7] it is possible to continuously compare the approach to other, future, work. It becomes clear that despite the common goal of evaluating and improving architectural aspects of software, a large variety of problems and solutions exists. For closely related approaches, there is a potential for synergies, as exemplified by Clounaq also adopting measures from the approach of Ntentos et al. [206]. The domain of microservices is closely related, and apart from the works included in the comparison, other work exists that focuses on maintainability aspects [12, 47], reliability aspects [11] or security aspects of microservices [21, 207, 315] and from which knowledge can be integrated in the presented approach. Furthermore, in the context of microservices, the analysis of anti-patterns [230, 278] or patterns [58, 286] together with an application to microservices architectures has been investigated.

Considering the focus of cloud deployments, in addition to the approaches included in the comparison, further work applies a focus on patterns used in the context of cloud deployments [251, 269, 313] and how these can be used to detect problems or compare different deployment options. These works also provide aspects that are integrable in the approach of this work. All considered approaches differ regarding their specific domain, formulation of quality attributes, architectural representations, and degrees of freedom. The more closely related works were included in the presented, but as it can be seen, also to these there are significant differences. To summarize, the approach presented in this work is novel and can be differentiated from other existing work specifically based on:

- the considered domain: Because it is covered both how components of cloud-native applications interact and how they are deployed
- the consideration of multiple quality aspects in a hierarchical model: Because the aim is to cover cloud-native broadly
- the abstraction from specific technology: Because technological heterogeneity is taken into account through corresponding abstraction

> Pass on what you have learned. Strength, mastery.
> But weakness, folly, failure also.
> Yes: failure, most of all. The greatest teacher, failure is.
>
> *Yoda (in Star Wars: The Last Jedi, 2017)*

9. Conclusion

Cloud-native, as a topic, aims at developing and operating applications in a way that takes advantage of the benefits of cloud computing while at the same time withstanding the drawbacks of cloud computing. With the increasing prevalence of cloud environments, the topic is of great importance for software architects and developers who are confronted with the task of designing and developing enterprise applications that should run in the cloud. To build applications in a cloud-native way, a range of aspects, including application design, technology selection, deployment infrastructure, and operation need to be considered. However, not all quality aspects that can be impacted through cloud-native characteristics are of the same importance for different applications. And different quality aspects may even be opposed to each other.

This work, therefore, presents a quality model for cloud-native software architectures (Contribution C1, Section 1.3) It provides a comprehensive understanding for how characteristics of cloud-native software architectures impact quality aspects. It can be used to gain a better understanding of cloud-native application characteristics and for a qualitative evaluation of applications to identify potential factors for improvement. Together with the corresponding software architecture modeling approach and the provided architectural measures, applications can also be evaluated in a quantitative way, enabling a more structured, consistent, and partly automated approach. This is implemented in the Clounaq tool (Contribution C2, Section 1.3) which provides an easily accessible way for exploring the quality model, modeling software architectures, and evaluating them according to the quality model. The tool is intended to be used by both researchers and practitioners who want to analyze software architectures in a structured way or investigate possibilities for benefiting from cloud-native characteristics.

The main results of this work substantiate the initially formulated hypotheses of cloud-native covering a broad range of aspects and how these aspects can be quantitatively evaluated for modeled software architectures. By relying on literature, data from a questionnaire-based survey, data from experiments, and the application to use cases, the gained results are supported from different perspectives. And while there are certain limitations to the quality model and the evaluation approach, they could mostly be resolved in future work through extending and broadening the presented approaches. For example, the quality evaluation approach could be applied to additional, real-world applications serving as use cases or experiment objects to iteratively refine the modeling approach, the architectural

measure definition and calculation, the evaluation functions, as well as the impact relationships.

Apart from the quality model and the evaluation approach as the core contributions of this work, there is also a methodological contribution. Using the Quamoco metamodel as a foundation, it has been shown how a quality model can be formulated and validated and which methods can be applied to do so. While the presented methods have been used in this work to formulate a quality model for cloud-native software architectures, they can also be applied in different contexts for formulating and validating quality models for a different domain.

Furthermore, as a practical contribution, the Clounaq tool and its architectural measure calculation implementations provide a basis for modeling and quantifying software architectures. Software architects and developers can use it to design and evaluate application architectures with a focus on non-functional requirements and how these are impacted by cloud-native characteristics and patterns. Researchers may use and extend the measure implementations as a basis for further research on cloud-native, microservices-based application architectures. After all, many of the implemented measures stem from other research where measures have been described, but often without tooling for their automated calculation.

All in all, cloud-native is a relevant topic for research and practice that requires ongoing investigation accompanying the continuous evolution and adoption of cloud computing technologies. With this work, the topic is investigated at its current state from a conceptual perspective that views cloud-native characteristics in the context of quality aspects. And it is examined from a practical perspective by demonstrating how relevant aspects of cloud-native software architectures can be modeled and quantitatively evaluated.

Through the publication of this work, I hope to support others who are interested in the topic of cloud-native, either as practitioners or researchers.

Part IV.

Appendix

A. Additional Impact Validation Results

Table A.1.: Survey results for factors that do not provide a clear or significant result - 1

Factor	Quality Aspect	--	-	0	+	++	Mean	Impact	p-Value
Access control management consistency	Authenticity	0	0	6	2	2	0.60	(↗)	0.2302
	Confidentiality	0	0	4	2	4	1.00	(↗)	0.1549
Account separation	Confidentiality	0	0	4	2	1	0.57	(↗)	0.5499
	Integrity	0	0	2	1	4	1.29	(↗)	0.1100
Addressing abstraction	Modifiability	0	0	4	1	1	0.50	(↗)	0.6399
	Simplicity	0	0	3	3	0	0.50	(↗)	0.1322
	Replaceability	0	0	3	2	1	0.67	(↗)	0.6399
API-based communication	Simplicity	0	1	5	2	2	0.50	(↗)	0.6684
Asynchronous communication	Elasticity	0	0	5	0	2	0.57	(↗)	0.1430
	Resource utilization	0	0	3	2	2	0.86	(↗)	0.5499
	Fault tolerance	0	0	4	2	1	0.57	(↗)	0.5499
Backing service decentralization	Elasticity	0	0	4	1	1	0.50	(↗)	0.6399
	Availability	0	0	3	1	2	0.83	(↗)	0.6399
	Fault tolerance	0	0	3	0	3	1.00	(↗)	0.1322
Circuit breaked communication	Recoverability	0	0	5	1	2	0.62	(↗)	0.3182
Cloud vendor abstraction	Interoperability	0	0	6	3	2	0.64	(↗)	0.1908
	Installability	0	1	3	3	4	0.91	(↗)	0.2344
Command query responsibility segregation	Elasticity	0	0	4	2	1	0.57	(↗)	0.5499
	Availability	0	0	4	1	2	0.71	(↗)	0.5499
	Fault tolerance	0	0	4	2	1	0.57	(↗)	0.5499
Consistent centralized logging	Testability	0	0	4	0	2	0.67	(↗)	0.3158
	Accountability	0	0	3	2	1	0.67	(↗)	0.6399
Contract-based links	Adaptability	0	0	3	0	2	0.80	(↗)	0.3519
Dynamic scheduling	Testability	1	1	4	0	0	-0.50	(↘)	0.6399
	Elasticity	0	0	3	0	3	1.00	(↗)	0.1322
	Time-behavior	0	0	4	0	2	0.67	(↗)	0.3158
	Adaptability	0	0	4	1	1	0.50	(↗)	0.6399

Table A.2.: Survey results for factors that do not provide a clear or significant result - 2

Factor	Quality Aspect	--	-	0	+	++	Mean	Impact	p-Value
Health and readiness checks	Testability	0	0	6	1	2	0.56	(↗)	0.2329
	Fault tolerance	0	0	3	4	2	0.89	(↗)	0.1325
Horizontal data replication	Elasticity	0	0	5	0	3	0.75	(↗)	0.1055
	Availability	0	0	2	3	3	1.12	(↗)	0.1455
	Fault tolerance	0	0	4	2	2	0.75	(↗)	0.4863
Infrastructure abstraction	Interoperability	0	0	5	2	1	0.50	(↗)	0.3182
	Simplicity	0	0	4	3	1	0.62	(↗)	0.2798
	Installability	0	0	5	1	2	0.62	(↗)	0.3182
	Replaceability	0	0	4	2	2	0.75	(↗)	0.4863
Limited data scope	Analyzability	0	0	2	3	0	0.60	(↗)	0.1924
	Testability	0	0	3	1	1	0.60	(↗)	0.8457
Limited request trace scope	Analyzability	0	1	2	1	2	0.67	(↗)	0.8868
	Time-behavior	0	0	3	3	0	0.50	(↗)	0.1322
Managed backing services	Availability	0	1	4	2	2	0.56	(↗)	0.8024
Managed infrastructure	Co-Existence	0	1	8	2	3	0.50	(↗)	0.2434
	Installability	0	1	8	2	3	0.50	(↗)	0.2434
Consistently mediated communication	Time-behavior	0	3	2	0	0	-0.60	(↘)	0.1924
	Replaceability	0	0	3	1	1	0.60	(↗)	0.8457
Mostly stateless services	Analyzability	0	1	4	2	2	0.56	(↗)	0.8024
	Reusability	0	0	4	3	2	0.78	(↗)	0.3169
	Resource utilization	0	0	4	2	3	0.89	(↗)	0.3169
	Recoverability	0	0	6	1	2	0.56	(↗)	0.2329
Persistent communication	Elasticity	0	0	3	1	1	0.60	(↗)	0.8457
	Availability	0	0	2	2	1	0.80	(↗)	0.7222
	Fault tolerance	0	0	1	2	2	1.20	(↗)	0.2695
Physical data distribution	Resource utilization	1	4	2	0	0	-0.86	(↘)	0.1100
	Recoverability	0	1	3	1	2	0.57	(↗)	0.9040
Physical service distribution	Maturity	0	0	6	1	3	0.70	(↗)	0.1655
	Recoverability	0	2	3	2	3	0.60	(↗)	0.6204
Retries for safe invocations	Availability	0	0	2	1	2	1.00	(↗)	0.7222
	Recoverability	0	0	3	0	2	0.80	(↗)	0.3519
Secrets stored in specialized services	Accountability	0	0	4	0	2	0.67	(↗)	0.3158
	Authenticity	0	0	3	1	2	0.83	(↗)	0.6399
	Integrity	0	0	1	2	3	1.33	(↗)	0.1185
Usage of existing solutions for non-core capabilities	Interoperability	0	0	4	2	1	0.57	(↗)	0.5499
Vertical data replication	Analyzability	0	3	2	0	0	-0.60	(↘)	0.1924
	Availability	0	0	3	0	2	0.80	(↗)	0.3519
	Fault tolerance	0	0	3	1	1	0.60	(↗)	0.8457

B. Additional Architectural Measures from Literature

The measures listed in the following tables were also extracted from literature. The status IMPLEMENTED means that they are included in the quality evaluation approach and calculated during architectural evaluations, but they are not actively used for the evaluation of factors. The status UNSUPPORTED means that the measures are relevant to the quality model, but cannot be supported with the current modeling approach, for example because they require a level of detail or additional information that is not included in the architectural modeling approach.

Table B.1.: Additional architectural measures - 1

Ratio of endpoints that support API Keys IMPLEMENTED

Formula:

$$\frac{|\{ e \mid e \in E \wedge \text{"API-Key"} \in e.supported\_authentication\_methods \}|}{|E|}$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Access control management consistency**
Associated Quality Aspects:	Literature Sources:
Integrity	[207]

Ratio of endpoints that support plaintext authentication IMPLEMENTED

Formula:

$$\frac{|\{ e \mid e \in E \wedge \text{"basic\_authentication"} \in e.supported\_authentication\_methods \}|}{|E|}$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Access control management consistency**
Associated Quality Aspects:	Literature Sources:
Integrity	[207], [315], [316]

Ratio of endpoints that are included in an single-sign-on approach IMPLEMENTED

Formula:

$$\frac{|\{ e \mid e \in E \wedge \text{"Single Sign-On"} \in e.supported\_authentication\_methods \}|}{|E|}$$

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Access control management consistency**
Associated Quality Aspects:	Literature Sources:
Integrity	[207]

Number of asynchronous endpoints offered by a service IMPLEMENTED

Formula:

$$|\{ e \mid e \in c.providedEndpoints \wedge isAsync(e) \}|$$

Functions:

$isAsync$:$e \rightarrow (e.kind = \text{"send event"} \vee e.kind = \text{"subscribe"})$

Applicable Entities:	Associated Factor:
Component	**Asynchronous communication**
Associated Quality Aspects:	Literature Sources:
Modularity	[259], [58]

Number of synchronous outgoing links IMPLEMENTED

Formula:

$$|\{ l \mid l \in L \wedge l.sourceComponent = c \wedge isSync(l) \}|$$

Functions:

$isSync$:$l \rightarrow (l.targetEndpoint.kind = \text{"query"} \vee l.targetEndpoint.kind = \text{"command"})$

Applicable Entities:	Associated Factor:
Component	**Asynchronous communication**
Associated Quality Aspects:	Literature Sources:
Modularity	[12], [58]

Table B.2.: Additional architectural measures - 2

Number of asynchronous outgoing links IMPLEMENTED

Formula:

$$|\{ l \mid l \in L \wedge l.sourceComponent = c \wedge isAsync(l) \}|$$

Functions:

$isAsync{:}l \rightarrow (l.targetEndpoint.kind = \text{"send event"} \vee l.targetEndpoint.kind = \text{"subscribe"})$

Applicable Entities:	Associated Factor:
Component	**Asynchronous communication**
Associated Quality Aspects:	Literature Sources:
Modularity	[12], [58]

Ratio of asynchronous outgoing links IMPLEMENTED

Formula:

$$\frac{|\{ l \mid l \in L \wedge l.sourceComponent = c \wedge isAsync(l) \}|}{|\{ l \mid l \in L \wedge l.sourceComponent = c \}|}$$

Functions:

$isAsync{:}l \rightarrow (l.targetEndpoint.kind = \text{"send event"} \vee l.targetEndpoint.kind = \text{"subscribe"})$

Applicable Entities:	Associated Factor:
Component	**Asynchronous communication**
Associated Quality Aspects:	Literature Sources:
Modularity	[133]

Number of synchronous endpoints IMPLEMENTED

Formula:

$$|\{ e \mid e \in E \wedge (e.kind = \text{"query"} \vee e.kind = \text{"command"}) \}|$$

Applicable Entities:	Associated Factor:
System	**Asynchronous communication**
Associated Quality Aspects:	Literature Sources:
Modularity	[259]

Number of asynchronous endpoints IMPLEMENTED

Formula:

$$|\{ e \mid e \in E \wedge (e.kind = \text{"send event"} \vee e.kind = \text{"subscribe"}) \}|$$

Applicable Entities:	Associated Factor:
System	**Asynchronous communication**
Associated Quality Aspects:	Literature Sources:
Modularity	[259]

Ratio of infrastructure nodes that support monitoring IMPLEMENTED

Formula:

$$\frac{|\{ i \mid i \in I \wedge \exists bd \in BD(\exists (i, bd) \in i.BDA \wedge bd.kind = \text{"metrics"} \wedge bd.kind = \text{"logs"}) \}|}{|I|}$$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Automated monitoring**
Associated Quality Aspects:	Literature Sources:
Analyzability	[207], [315]

Table B.3.: Additional architectural measures - 3

Ratio of components that support monitoring IMPLEMENTED

Formula:

$$\frac{|\{\, c \mid c \in C \wedge \exists bd \in BD(\exists (c, bd) \in c.BDA \wedge bd.kind = \text{"metrics"} \wedge bd.kind = \text{"logs"}) \,\}|}{|C|}$$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Automated monitoring**
Associated Quality Aspects:	Literature Sources:
Analyzability	[207], [315]

Degree of storage backend sharing IMPLEMENTED

Formula:

$$|\{\, c' \mid c' \in C \wedge \exists l \in L(l.sourceComponent = c' \wedge l.targetEndpoint \in sbs.providedEndpoints) \,\}|$$

Applicable Entities:	Associated Factor:
Component	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[245]

Shared storage backing service interactions UNSUPPORTED

Formula:

$$n/a$$

Applicable Entities:	Associated Factor:
System	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[204], [205], [206]

Database type utilization IMPLEMENTED

Formula:

$$\frac{|\{\, sbs \mid sbs \in SBS \wedge |\{\, c' \mid connected(c', sbs) \,\}| = 1 \,\}|}{|SBS|}$$

Functions:

$connected{:}c,sbs \rightarrow (\exists l \in L(l.sourceComponent = c \wedge l.targetEndpoint \in sbs.providedEndpoints))$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[205]

Number of services connected to a storage backing service IMPLEMENTED

Formula:

$$|\{\, s \mid s \in S \wedge linkedToBackingService(s) \,\}|$$

Functions:

$linkedToBackingService{:}s \rightarrow (\exists l \in L(l.sourceComponent = s \wedge l.targetEndpoint \in sbs.providedEndpoints \wedge sbs \in SBS))$

Applicable Entities:	Associated Factor:
System	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[58]

Table B.4.: Additional architectural measures - 4

Service portability	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
Component	**Cloud vendor abstraction**
Associated Quality Aspects:	Literature Sources:
Adaptability, **Reusability**	[110], [263]

Number of read endpoints provided by a service	IMPLEMENTED
Formula: $\lvert \{ e \mid e \in c.providedEndpoints \wedge e.kind = \text{"query"} \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Command query responsibility segregation**
Associated Quality Aspects:	Literature Sources:
Simplicity, **Modularity**	[58]

Number of write endpoints provided by a service	IMPLEMENTED
Formula: $\lvert \{ e \mid e \in c.providedEndpoints \wedge (e.kind = \text{"command"} \vee e.kind = \text{"send event"}) \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Command query responsibility segregation**
Associated Quality Aspects:	Literature Sources:
Simplicity, **Modularity**	[58]

Ratio of components or infrastructure nodes that enable performance analytics	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
System	**Consistent centralized metrics**
Associated Quality Aspects:	Literature Sources:
Analyzability	[207]

Ratio of components whose egress is proxied	IMPLEMENTED
Formula: $\frac{\lvert \{ c \mid c \in C \wedge c.egressProxiedBy \neq \emptyset \} \rvert}{\lvert \{ c \mid c \in C \} \rvert}$	
Applicable Entities:	Associated Factor:
System	**Consistently mediated communication**
Associated Quality Aspects:	Literature Sources:
Interoperability, **Time-behavior**, **Analyzability**	[207]

Table B.5.: Additional architectural measures - 5

Ratio of endpoints that support SSL — IMPLEMENTED

Formula:

$$\frac{|\{\, e \mid e \in E \wedge e.protocol \in SUPPORTS\_TLS \,\}|}{|E|}$$

Functions:

$SUPPORTS\_TLS$={"https","sftp"}

Applicable Entities:	Associated Factor:
System, **Component**, **Request Trace**	**Data encryption in transit**
Associated Quality Aspects:	Literature Sources:
Confidentiality	[207]

Component density — IMPLEMENTED

Formula:

$$\frac{|\{\, c \mid c \in C \wedge \exists dm \in DM(dm.deployed = c) \,\}|}{|\{\, i \mid i \in I \wedge \exists dm \in DM(dm.host = i \wedge dm.deployed \in C) \,\}|}$$

Applicable Entities:	Associated Factor:
System	**Distribution**
Associated Quality Aspects:	Literature Sources:
Availability	[110], [243]

Number of services hosted on one infrastructure entity — IMPLEMENTED

Formula:

$$|\{\, s \mid s \in S \wedge \exists dm \in DM(dm.deployed = s \wedge dm.host = i) \,\}|$$

Applicable Entities:	Associated Factor:
Infrastructure	**Distribution**
Associated Quality Aspects:	Literature Sources:
Availability	[58]

Conceptual modularity quality based on data aggregate cohesion and coupling — UNSUPPORTED

Formula:

$$n/a$$

Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[41], [128]

Cyclic communication — IMPLEMENTED

Formula:

$$\exists (l_1, l_2, ..., l_n) \in L(l_1.sourceComponent = c \wedge l_n.targetEndpoint \in c.providedEndpoints)$$

Applicable Entities:	Associated Factor:
Component	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[12], [205]

Table B.6.: Additional architectural measures - 6

Number of synchronous cycles IMPLEMENTED

Formula:

$$|\{ (l_1, l_2, ..., l_n) \mid \forall l_i \in L(isSync(l_i)) \wedge chain(l_n, l_1) \}|$$

Functions:

$isSync$:$l \rightarrow (l.targetEndpoint.kind$="query"$\vee l.targetEndpoint.kind$="command"$)$

$chain$:$l_a, l_b \rightarrow (l_a.targetEndpoint \in l_b.sourceComponent.providedEndpoints)$

Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[74]

Relative importance of the service IMPLEMENTED

Formula:

$$\frac{|\{ c' \mid c' \in C \wedge \exists l \in L(l.sourceComponent = c' \wedge l.targetEndpoint \in c.providedEndpoints) \}|}{|C|}$$

Applicable Entities:	Associated Factor:
Component	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[317]

Extent of aggregation components UNSUPPORTED

Formula:

$$n/a$$

Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[115]

System's centraliZation UNSUPPORTED

Formula:

$$n/a$$

Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[115]

Density of aggregation IMPLEMENTED

Formula:

$$\sum_{i=1}^{|\{ c \mid c \in C \wedge inAndOut(c) \}|} \ln\left(\frac{|\{ l \mid l \in L \wedge out(l, c_i) \}|}{|\{ l \mid l \in L \wedge inOrOut(c_i, l) \}| * 2}\right)$$

Functions:

$inAndOut$:$c \rightarrow (\exists l_1, l_2 \in L(l_1.sourceComponent = c \wedge l_2.targetEndpoint \in c.providedEndpoints))$

out:$c, l \rightarrow (l.sourceComponent = c)$

$inOrOut$:$c, l \rightarrow (l.sourceComponent = c \vee l.targetEndpoint \in c.providedEndpoints)$

Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[115]

Table B.7.: Additional architectural measures - 7

Aggregator centraliZation	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[115]
Data aggregate convergence across components	IMPLEMENTED
Formula: $\frac{\sum_{i=1}^{\lvert C\rvert} \lvert c_i.RDA_C\rvert}{\lvert C\rvert} + \frac{\sum_{i=1}^{\lvert DA\rvert} \lvert \{ c \mid \exists c.RDA_C(c.RDA_C.da = da_i) \} \rvert}{\lvert DA\rvert}$	
Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[137], [180]
Service criticality	IMPLEMENTED
Formula: $\lvert \{ c' \mid c' \in C \wedge \exists l(l \in L \wedge l.sourceComponent = c' \wedge l.targetEndpoint \in c.providedEndpoints) \} \rvert * \lvert \{ c' \mid c' \in C \wedge \exists l \in L(l.sourceComponent = c \wedge l.targetEndpoint \in c'.providedEndpoints) \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[36], [248]
Ratio of cyclic request traces	IMPLEMENTED
Formula: $\frac{\lvert \{ rt \mid rt \in RT \wedge hasCycle(rt) \}}{\lvert RT\rvert}$	
Functions: $hasCycle{:}rt \rightarrow (\exists (l_1,l_2,...l_n) \in rt.involvedLinks(l_n.targetEndpoint \in l_1.sourceComponent.providedEndpoints))$	
Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[103]
Number of potential cycles in a system	IMPLEMENTED
Formula: $\lvert \{ (l_1, l_2, ..., l_n) \mid \forall l_i \in L \wedge l_n.targetEndpoint \in l_1.sourceComponent.providedEndpoints \} \rvert$	
Applicable Entities:	Associated Factor:
System	**Functional decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	[223]

Table B.8.: Additional architectural measures - 8

Number of deployment target environments	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
System	**Immutable artifacts**
Associated Quality Aspects:	Literature Sources:
Replaceability	[12]

Data aggregate scope	UNSUPPORTED
Formula: $\|SYS.DA\| or \|C.RDA_C\|$	
Applicable Entities:	Associated Factor:
Component, **System**	**Limited data scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[259], [320]

Service interface data cohesion	IMPLEMENTED
Formula: $\frac{\|\{ e \mid e \in c.providedEndpoints \wedge \exists e' \in c.providedEndpoints(\wedge((e.RDA_E \cap e'.RDA_E) \neq \emptyset)) \}\|}{\|c.RDA_C\|}$	
Applicable Entities:	Associated Factor:
Component	**Limited data scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[36], [224], [136], [41], [128], [127], [17], [18], [37]

Data aggregate count	IMPLEMENTED
Formula: $\|s.RDA_C\|$	
Applicable Entities:	Associated Factor:
Component	**Limited data scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[15]

Number of provided synchronous and asynchronous endpoints	IMPLEMENTED
Formula: $\|\{ e \mid e \in c.providedEndpoints \}\|$	
Applicable Entities:	Associated Factor:
Component	**Limited endpoint scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[12], [74], [259], [41], [128], [58]

Table B.9.: Additional architectural measures - 9

Number of synchronous endpoints offered by a service	IMPLEMENTED
Formula: $\lvert \{ e \mid e \in c.providedEndpoints \wedge (e.kind = \text{"query"} \vee e.kind = \text{"command"}) \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Limited endpoint scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[259]
Distribution of synchronous calls	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
Component	**Limited endpoint scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[74]
Cohesion of endpoints based on invocation by other services	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
Component	**Limited endpoint scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[223]
Unused endpoint count	IMPLEMENTED
Formula: $\lvert \{ e \mid e \in c.providedEndpoints \wedge \nexists l (l \in L \wedge l.targetEndpoint = e) \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Limited endpoint scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[15]
Total service interface cohesion	IMPLEMENTED
Formula: $\frac{ServiceInterfaceDataCohesion + ServiceInterfaceUsageCohesion}{2}$	
Applicable Entities:	Associated Factor:
Component	**Limited functional scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[36], [224]

Table B.10.: Additional architectural measures - 10

Cohesiveness of service	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
Component	**Limited functional scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[211], [156]
Cohesion of a service based on other endpoints called	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
Component	**Limited functional scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[223]
Lack of cohesion of a service	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
Component	**Limited functional scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[6]
Average system lack of cohesion of a service	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
System	**Limited functional scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[6]
Size of a service	IMPLEMENTED
Formula: $\lvert c.RDA_C \rvert + \lvert \{ c' \mid \exists l \in L(l.sourceComponent = c' \wedge l.targetEndpoint \in c.providedEndpoints) \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Limited functional scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[15]
Unreachable endpoint count	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
Component	**Limited functional scope**
Associated Quality Aspects:	Literature Sources:
Modularity	[15]

Table B.11.: Additional architectural measures - 11

Maximum length of service link chain per request trace IMPLEMENTED

Formula:

$$(\sum_{i=1}^{|rt.involvedLinks|} |rt.involvedLinks.links|) | \forall rt' (rt' \in RT \wedge rt' \neq rt \wedge \sum_{i=1}^{|rt'.involvedLinks|} |rt'.involvedLinks.links| \leq \sum_{i=1}^{|rt.involvedLinks|} |rt.involvedLinks.links|$$

Applicable Entities:	Associated Factor:
System	**Limited request trace scope**
Associated Quality Aspects:	Literature Sources:
Modifiability	[12], [74], [245]

Number of request traces IMPLEMENTED

Formula:

$$|RT|$$

Applicable Entities:	Associated Factor:
System	**Limited request trace scope**
Associated Quality Aspects:	Literature Sources:
Modifiability	[320]

Request trace length IMPLEMENTED

Formula:

$$|rt.involvedLinks|$$

Applicable Entities:	Associated Factor:
Request Trace	**Limited request trace scope**
Associated Quality Aspects:	Literature Sources:
Modifiability	[223], [97]

Number of cycles in request traces IMPLEMENTED

Formula:

$$|\{ (l_1, ..., l_n) \mid l_i \in L \wedge hasCycle((l_1, ..., l_n), rt) \}|$$

Functions:

$hasCycle{:}(l_1,...,l_n),rt \rightarrow (\forall l_i \in (l_1,...,l_n) \exists rt.involvedLinks(l_i \in rt.involvedLinks.links \wedge chain(l_n,l_1)))$

$chain{:}l_a,l_b \rightarrow (l_a.targetEndpoint \in l_b.sourceComponent.providedEndpoints)$

Applicable Entities:	Associated Factor:
Request Trace	**Limited request trace scope**
Associated Quality Aspects:	Literature Sources:
Modifiability	[223], [97]

Ratio of request traces containing a frontend component UNSUPPORTED

Formula:

$$n/a$$

Applicable Entities:	Associated Factor:
System	**Limited request trace scope**
Associated Quality Aspects:	Literature Sources:
Modifiability	[316]

Table B.12.: Additional architectural measures - 12

Number of links per component IMPLEMENTED

Formula:

$$|\{\, l \mid l \in L \wedge (l.sourceComponent = c \vee l.targetEndpoint \in c.providedEndpoints) \,\}|$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[320], [280], [245]

Number of consumed endpoints IMPLEMENTED

Formula:

$$|\{\, e \mid e \in E \wedge \exists l \in L(l.sourceComponent = s \wedge l.targetEndpoint = e) \,\}|$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[12], [97], [225]

Incoming outgoing ratio of a component IMPLEMENTED

Formula:

$$\frac{|\{\, l \mid l \in L \wedge l.sourceComponent = c \,\}|}{|\{\, l \mid l \in L \wedge l.targetEndpoint \in c.providedEndpoints \,\}|}$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[280]

Ratio of outgoing links of a service IMPLEMENTED

Formula:

$$\frac{|\{\, l \mid l \in L \wedge l.sourceComponent = c \,\}|}{|\{\, l \mid l \in L \wedge (l.sourceComponent = c \vee l.targetEndpoint \in c.providedEndpoints) \,\}|} * 100$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[227]

Interaction density based on components IMPLEMENTED

Formula:

$$\frac{|L|}{|C|}$$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[280]

Table B.13.: Additional architectural measures - 13

Coupling degree based on potential coupling IMPLEMENTED

Formula:

$$\frac{(|C| * (|C|-1) * (|C|-1)) - \sum_{i=1}^{|C|}\sum_{j=1}^{|C|} minPath(c_i, c_j)}{(|C| * (|C|-1) * (|C|-1)) - (|C| * (|C|-1))}$$

Functions:

$minPath{:}c_a,c_b \rightarrow |(l_1,...,l_n)|\,|(l_1.sourceComponent{=}c_a \wedge l_n.targetEndpoint{\in}c_b.providedEndpoints) \vee (l_1.sourceComponent{=}c_b \wedge l_n.targetEndpoint{\in}c_a.providedEndpoints) \wedge \min(n)$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[227]

Interaction density based on links IMPLEMENTED

Formula:

$$\frac{|L|}{|C| * (|C|-1)}$$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[280], [133]

Indirect Interaction density of a system IMPLEMENTED

Formula:

$$\frac{\sum_{i=1}^{|C\setminus c|} \begin{cases} 0 & \text{if } directlyLinked(c, c_i) \\ 1 & \text{if } indirectlyLinked(c, c_i) \end{cases}}{|\{\, c' \mid linked(c, c') \,\}|}$$

Functions:

$directlyLinked{:}c_a,c_b \rightarrow (\exists l{\in}L(l.sourceComponent{=}c_a \wedge l.targetEndpoint{\in}c_b.providedEndpoints))$

$indirectlyLinked{:}c_a,c_b \rightarrow (\exists (l_1,...,l_n){\subset}L(l_1.sourceComponent{=}c_a \wedge l_n.targetEndpoint{\in}c_b.providedEndpoints \wedge n{>}1))$

$linked{:}c_a,c_b \rightarrow (\exists (l_1,...,l_n){\subset}L(l_1.sourceComponent{=}c_a \wedge l_n.targetEndpoint{\in}c_b.providedEndpoints))$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[133]

Service coupling based on endpoint entropy IMPLEMENTED

Formula:

$$\frac{\sum_{i=1}^{|c.providedEndpoints|} -\log\left(\frac{1}{|\{\, l \mid l \in L \wedge l.targetEndpoint = e_i \,\}| + 1}\right)}{|c.providedEndpoints|}$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[130]

Table B.14.: Additional architectural measures - 14

System coupling based on endpoint entropy IMPLEMENTED

Formula:

$$\sum_{j=1}^{|C|}\left(\frac{\sum_{i=1}^{|c.providedEndpoints|} -\log\left(\frac{1}{|\{\, l \mid l \in L \wedge l.targetEndpoint = e_i \,\}| + 1}\right)}{|c.providedEndpoints|}\right)$$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[130]

Modularity quality based on cohesion and coupling UNSUPPORTED

Formula:

$$n/a$$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[41], [128]

Combined metric for indirect dependency IMPLEMENTED

Formula:

$$\frac{"IndirectInteractiondensityofasystem" + "IndirectDependencybecauseofshareddatarepository"}{2}$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[133]

Services interdependence in the system IMPLEMENTED

Formula:

$$|\{\, (c_1, c_2) \mid c_1, c_2 \in C \wedge bidirectionalLink(c_1, c_2) \,\}|$$

Functions:

$bidirectionalLink{:}c_a,c_b \rightarrow (\exists l \in L(l.sourceComponent = c_a \wedge l.targetEndpoint \in c_b.providedEndpoints) \wedge \exists l \in L(l.sourceComponent = c_b \wedge l.targetEndpoint \in c_a.providedEndpoints))$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[36], [248]

Average number of directly connected services IMPLEMENTED

Formula:

$$\frac{|\{\, c' \mid c' \in C \wedge linked(c, c') \,\}| + |\{\, c' \mid c' \in C \wedge linked(c', c) \,\}|}{|C|}$$

Functions:

$linked{:}c_a,c_b \rightarrow (\exists l \in L(l.sourceComponent = c_a \wedge l.targetEndpoint \in c_b.providedEndpoints))$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[259]

Table B.15.: Additional architectural measures - 15

Number of components that are linked to a component	IMPLEMENTED
Formula: $\lvert \{ c' \mid c' \in C \wedge \exists l \in L(l.sourceComponent = c' \wedge l.targetEndpoint \in c.providedEndpoints) \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[36], [248], [259], [317], [15], [97], [225]
Number of components a component is linked to	IMPLEMENTED
Formula: $\lvert \{ c' \mid c' \in C \wedge \exists l \in L(l.sourceComponent = c \wedge l.targetEndpoint \in c'.providedEndpoints) \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[36], [248], [74], [259], [238], [237], [115], [227], [317]
Number of links between two services	UNSUPPORTED
Formula: $\lvert \{ l \mid l \in L \wedge l.sourceComponent = c \wedge l.targetEndpoint \in c'.providedEndpoints \} \rvert$	
Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[115]
Aggregate system metric to measure service coupling	IMPLEMENTED
Formula: $\frac{\sum_{i=1}^{\lvert S \rvert} \lvert \{ s' \mid s' \in S \wedge linked(s_i, s') \} \rvert}{\lvert S \rvert * (\lvert S \rvert - 1)}$	
Functions: $linked{:}s_a,s_b \rightarrow (\exists l \in L(l.sourceComponent = s_a \wedge l.targetEndpoint \in s_b.providedEndpoints))$	
Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[115], [97]
Service coupling based on data exchange complexity	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[137], [180]

Table B.16.: Additional architectural measures - 16

Simple degree of coupling in a system IMPLEMENTED

Formula:

$$\frac{\sum_{i=1}^{|C|} |\{ c' \mid c' \in C \wedge \exists l \in L(l.sourceComponent = c_i \wedge l.targetEndpoint \in c'.providedEndpoints) \}|}{|C|}$$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[232]

Direct service sharing IMPLEMENTED

Formula:

$$\frac{\frac{|\{ s \mid s \in S \wedge differentIncomingLinksS(s) \}|}{|S|} + \frac{|\{ e \mid e \in E \wedge differentIncomingLinksE(e) \}|}{|L|}}{2}$$

Functions:

$differentIncomingLinksS: s \rightarrow (\exists (l_1, l_2) \subset L(l_1.sourceComponent = s' \wedge l_1.targetEndpoint \in s.providedEndpoints \wedge l_2.sourceComponent = s'' \wedge l_2.targetEndpoint \in s.providedEndpoints))$

$differentIncomingLinksE: e \rightarrow (\exists (l_1, l_2) \subset L(l_1.sourceComponent = s' \wedge l_1.targetEndpoint = e \wedge l_2.sourceComponent = s'' \wedge l_2.targetEndpoint = e))$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[205]

Transitively shared services IMPLEMENTED

Formula:

$$\frac{\frac{|\{ c \mid c \in C \wedge transitiveReach(c) \}|}{|C|} + \frac{|\{ e \mid e \in E \wedge transitiveReachE(e) \}|}{|L|}}{2}$$

Functions:

$transitiveReachC: c \rightarrow (\exists (l_1, l_2) \subset L(l_1.targetEndpoint \in l_2.sourceComponent.providedEndpoints \wedge l_2.targetEndpoint \in c.providedEndpoints \wedge l_1.sourceComponent \neq c))$

$transitiveReachE: e \rightarrow (\exists (l_1, l_2) \subset L(l_1.targetEndpoint \in l_2.sourceComponent.providedEndpoints \wedge l_2.targetEndpoint = e \wedge e \notin l_1.sourceComponent.providedEndpoints))$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[205]

Ratio of shared non-external components to non-external components IMPLEMENTED

Formula:

$$\frac{|\{ s \mid s \in S \wedge twoDifferentInLinks(s) \}|}{|S|}$$

Functions:

$twoDifferentInLinks: s \rightarrow (\exists s', s'' \in S(s' \neq s'' \wedge \exists l \in L(l.sourceComponent = s' \wedge l.targetEndpoint \in s.providedEndpoints) \wedge \exists l' \in L(l'.sourceComponent = s'' \wedge l'.targetEndpoint \in s.providedEndpoints)))$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[314]

Table B.17.: Additional architectural measures - 17

Ratio of shared dependencies of non-external components to possible dependencies — IMPLEMENTED

Formula:

$$\frac{\sum_{i=1}^{|C|} |\{ (c', c'') \mid sharedDependency(c', c'', c_i) \}|}{|C|}$$

Functions:

$sharedDependency{:}c_a,c_b,c_c \rightarrow (c_a,c_b \neq c_c \wedge \exists l \in L(l.sourceComponent = c_a \wedge l.targetEndpoint \in c_a.providedEndpoints) \wedge \exists l' \in L(\wedge l'.sourceComponent = c_b \wedge l'.targetEndpoint \in c_c.providedEndpoints))$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[314]

Degree of dependence on other components — UNSUPPORTED

Formula:

$$n/a$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[211], [156], [210], [227]

Average system coupling — IMPLEMENTED

Formula:

$$\frac{\sum_{i=1}^{|L|} \begin{cases} 0.1 & \text{if } isSend(l_i) \\ 0.5 & \text{if } isCommand(l_i) \\ 0.2 & \text{else} \end{cases} + \frac{\sum_{j=1}^{|l.targetEndpoint.RDA_E|} \begin{cases} 0.1 & \text{if } persists(l_i.targetEndpoint.rda_j) \\ 0.5 & \text{if } caches(l_i.targetEndpoint.rda_j) \\ 0.25 & \text{else} \end{cases}}{|l.targetEndpoint.RDA_E|}}{|C|}$$

Functions:

$isSend{:}l \rightarrow (l.targetEndpoint.kind = \text{"send event"})$

$isCommand{:}l \rightarrow (l.targetEndpoint.kind = \text{"command"})$

$persists{:}rda \rightarrow (rda.usage\_relation = \text{"persistence"})$

$caches{:}rda \rightarrow (rda.usage\_relation = \text{"cached-usage"})$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[88]

Coupling of services based on used data aggregates — UNSUPPORTED

Formula:

$$\frac{|\{ da \mid da \in DA \wedge \exists (c_x, da) \in c_x.RDA_C \} \cap \{ da \mid da \in DA \wedge \exists (c_y, da) \in c_y.RDA_C \}|}{|\{ da \mid da \in DA \wedge \exists (c_x, da) \in c_x.RDA_C \} \cup \{ da \mid da \in DA \wedge \exists (c_y, da) \in c_y.RDA_C \}|}$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[223]

Table B.18.: Additional architectural measures - 18

Coupling of services based services which call them — UNSUPPORTED

Formula:

$$\frac{|\{ c \mid c \in C \wedge linked(c, c_x) \} \cap \{ c \mid c \in C \wedge linked(c, c_y) \}|}{|\{ c \mid c \in C \wedge linked(c, c_x) \} \cup \{ c \mid c \in C \wedge linked(c, c_y) \}|}$$

Functions:

$linked{:}c_a,c_b \rightarrow (\exists l \in L(l.sourceComponent = c_a \wedge l.targetEndpoint \in c_b.providedEndpoints))$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[223]

Coupling of services based services which are called by them — UNSUPPORTED

Formula:

$$\frac{|\{ c \mid c \in C \wedge linked(c_x, c) \} \cap \{ c \mid c \in C \wedge linked(c_y, c) \}|}{|\{ c \mid c \in C \wedge linked(c_x, c) \} \cup \{ c \mid c \in C \wedge linked(c_y, c) \}|}$$

Functions:

$linked{:}c_a,c_b \rightarrow (\exists l \in L(l.sourceComponent = c_a \wedge l.targetEndpoint \in c_b.providedEndpoints))$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[223]

Coupling of services based on amount of request traces that include a specific link — UNSUPPORTED

Formula:

$$\max\left(\frac{|\{ rt \mid rt \in RT \wedge linkInRT(rt, c_x, c_y) \}|}{|\{ rt \mid rt \in RT \wedge c_y \in rt.nodes \}|}, \frac{|\{ rt \mid rt \in RT \wedge linkInRT(rt, c_y, c_x) \}|}{|\{ rt \mid rt \in RT \wedge c_x \in rt.nodes \}|}\right)$$

Functions:

$linkInRT{:}rt,c_a,c_b \rightarrow (\exists l \in rt.involvedLinks(l.sourceComponent = c_a \wedge l.targetEndpoint \in c_b.providedEndpoints))$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[223]

Coupling of services based times that they occur in the same request trace — UNSUPPORTED

Formula:

$$\frac{|\{ rt \mid rt \in RT \wedge c_x, c_y \in rt.nodes \}|}{|RT|}$$

Applicable Entities:	Associated Factor:
Component	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[223]

Total number of links in a system — IMPLEMENTED

Formula:

$$|L|$$

Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[41], [127], [280], [16], [320]

Table B.19.: Additional architectural measures - 19

Number of services which have incoming links	IMPLEMENTED
Formula: $\lvert \{ s \mid s \in S \wedge \exists l \in L(l.targetEndpoint \in s.providedEndpoints) \} \rvert$	
Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[259], [115]
Number of services which have outgoing links	IMPLEMENTED
Formula: $\lvert \{ s \mid s \in S \wedge \exists l \in L(l.sourceComponent = s) \} \rvert$	
Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[259], [115]
Number of services which have both incoming and outgoing links	IMPLEMENTED
Formula: $\lvert \{ s \mid s \in S \wedge \exists l \in L(l.sourceComponent = s) \exists l \in L(l.targetEndpoint \in s.providedEndpoints) \} \rvert$	
Applicable Entities:	Associated Factor:
System	**Low coupling**
Associated Quality Aspects:	Literature Sources:
Modifiability	[259], [115]
Ratio of state dependency of endpoints	IMPLEMENTED
Formula: $\frac{\lvert \{ e \mid e \in E \wedge e.RDA_E \neq \emptyset \} \rvert}{\lvert E \rvert}$	
Applicable Entities:	Associated Factor:
Component, **System**, **Request Trace**	**Mostly stateless services**
Associated Quality Aspects:	Literature Sources:
Testability, **Modularity**, **Replaceability**, **Elasticity**	[133]
Ratio of stateful components	IMPLEMENTED
Formula: $\frac{\lvert \{ c \mid c \in C \wedge c.stateless = false \} \rvert}{\lvert C \rvert}$	
Applicable Entities:	Associated Factor:
System, **Request Trace**	**Mostly stateless services**
Associated Quality Aspects:	Literature Sources:
Testability, **Modularity**, **Replaceability**, **Elasticity**	[232]

Table B.20.: Additional architectural measures - 20

Externally available endpoints	IMPLEMENTED
Formula: $\lvert \{ ee \mid ee \in EE \} \rvert$	
Applicable Entities:	Associated Factor:
System	**Separation by gateways**
Associated Quality Aspects:	Literature Sources:
Modularity, **Availability**	[320]

Centralization of externally available endpoints	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
System	**Separation by gateways**
Associated Quality Aspects:	Literature Sources:
Modularity, **Availability**	[204], [206]

API Composition utilization metric	UNSUPPORTED
Formula: n/a	
Applicable Entities:	Associated Factor:
System	**Separation by gateways**
Associated Quality Aspects:	Literature Sources:
Modularity, **Availability**	[204], [206]

Ratio of request traces through a gateway	IMPLEMENTED
Formula: $\frac{\lvert \{ rt \mid rt \in RT \wedge proxiedByGateway(rt) \} \rvert}{\lvert RT \rvert}$	
Functions: $proxiedByGateway{:}rt \rightarrow ((rt.referencedEndpoint \in c.providedEndpoints \wedge c.externalIngressProxiedBy = p \wedge p \in PBS \wedge p.kind = \text{"API Gateway"}) \vee (rt.referencedEndpoint \in g.providedEndpoints \wedge g \in PBS \wedge g.kind = \text{"API Gateway"}))$	
Applicable Entities:	Associated Factor:
System	**Separation by gateways**
Associated Quality Aspects:	Literature Sources:
Modularity, **Availability**	[316]

Average number of endpoints per service	IMPLEMENTED
Formula: $\frac{\lvert E \rvert}{\lvert C \rvert}$	
Applicable Entities:	Associated Factor:
System	**Sparsity**
Associated Quality Aspects:	Literature Sources:
Simplicity	[36], [37], [114], [41], [128], [245], [137], [180], [65]

Table B.21.: Additional architectural measures - 21

Number of dependencies	UNSUPPORTED
Formula: n/a	
Applicable Entities: **Component**	Associated Factor: **Sparsity**
Associated Quality Aspects: **Simplicity**	Literature Sources: [12]
Number of versions per service	UNSUPPORTED
Formula: n/a	
Applicable Entities: **Component**	Associated Factor: **Sparsity**
Associated Quality Aspects: **Simplicity**	Literature Sources: [36], [114]
Concurrently available versions complexity	UNSUPPORTED
Formula: n/a	
Applicable Entities: **Component**	Associated Factor: **Sparsity**
Associated Quality Aspects: **Simplicity**	Literature Sources: [133]
Service support for transactions	UNSUPPORTED
Formula: n/a	
Applicable Entities: **Component**	Associated Factor: **Sparsity**
Associated Quality Aspects: **Simplicity**	Literature Sources: [36], [114]
Number of services	IMPLEMENTED
Formula: $\|S\|$	
Applicable Entities: **System**	Associated Factor: **Sparsity**
Associated Quality Aspects: **Simplicity**	Literature Sources: [259], [237], [114], [115], [317], [248]
Number of backing services	IMPLEMENTED
Formula: $\|BS\|$	
Applicable Entities: **System**	Associated Factor: **Sparsity**
Associated Quality Aspects: **Simplicity**	Literature Sources: [259]

Table B.22.: Additional architectural measures - 22

Lines of code (LOC) for deployment configuration UNSUPPORTED

Formula:

$$n/a$$

Applicable Entities:	Associated Factor:
Component	**Use infrastructure as code**
Associated Quality Aspects:	Literature Sources:
Modifiability, **Adaptability**, **Reusability**, **Recoverability**	[160], [279]

Average broker backend sharing IMPLEMENTED

Formula:

$$\frac{\sum_{i=1}^{|BBS|} \frac{\sum_{j=1}^{|S|} \begin{cases} 1 & \text{if } \exists l \in L(l.sourceComponent = s_j \wedge l.targetEndpoint \in bbs_i.providedEndpoints) \\ 0 & \text{else} \end{cases}}{|S|}}{|BBS|}$$

Applicable Entities:	Associated Factor:
System	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Average storage backend sharing IMPLEMENTED

Formula:

$$\frac{\sum_{i=1}^{|SBS|} \frac{\sum_{j=1}^{|S|} \begin{cases} 1 & \text{if } \exists l \in L(l.sourceComponent = s_j \wedge l.targetEndpoint \in sbs_i.providedEndpoints) \\ 0 & \text{else} \end{cases}}{|S|}}{|SBS|}$$

Applicable Entities:	Associated Factor:
System	**Backing service decentralization**
Associated Quality Aspects:	Literature Sources:
Modifiability	new

Median service replication level IMPLEMENTED

Formula:

$$median(\left\{ (repl_1, ..., repl_n) \;\middle|\; repl_i = \sum_{j=1}^{|\{\, dm \mid dm \in DM \wedge dm.deployed = s_i \wedge s \in S \,\}|} dm_j.replicas \right\})$$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Service replication**
Associated Quality Aspects:	Literature Sources:
Availability, **Time-behavior**	new

Smallest service replication value IMPLEMENTED

Formula:

$$min(\left\{ (repl_1, ..., repl_n) \;\middle|\; repl_i = \sum_{j=1}^{|\{\, dm \mid dm \in DM \wedge dm.deployed = s_i \wedge s \in S \,\}|} dm_j.replicas \right\})$$

Applicable Entities:	Associated Factor:
System, **Request Trace**	**Service replication**
Associated Quality Aspects:	Literature Sources:
Availability, **Time-behavior**	new

Bibliography

[1] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke, "Decomposition of monolith applications into microservices architectures: A systematic review," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4213–4242, 2023. (Cited on page 23.)

[2] A. P. Achilleos, K. Kritikos, A. Rossini, G. M. Kapitsaki, J. Domaschka, M. Orzechowski, D. Seybold, F. Griesinger, N. Nikolov, D. Romero, and G. A. Papadopoulos, "The cloud application modelling and execution language," *Journal of Cloud Computing*, vol. 8, no. 1, 2019. [Online]. Available: https://doi.org/10.1186/s13677-019-0138-7 (Cited on pages 54, 221, 277, 278, and 279.)

[3] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield, *Building Secure and Reliable Systems*. O'Reilly Media, Inc., 2020. [Online]. Available: https://learning.oreilly.com/library/view/building-secure-and/9781492083115/ (Cited on pages 61, 108, 110, 112, 118, 121, 138, 139, 141, and 148.)

[4] G. Adzic and R. Chatley, "Serverless Computing: Economic and Architectural Impact," in *Proceedings of 11th Joint Meeting on Foundations of Software Engineering*. Paderborn, Germany: ACM, 2017. [Online]. Available: https://doi.org/10.1145/3106237.3117767 (Cited on page 88.)

[5] A. B. AL-Badareen, J.-M. Desharnais, and A. Abran, "A suite of rules for developing and evaluating software quality models," in *Software Measurement*. Springer International Publishing, 2015, pp. 1–13. [Online]. Available: https://doi.org/10.1007/978-3-319-24285-9_1 (Cited on pages 34 and 43.)

[6] O. Al-Debagy and P. Martinek, "A metrics framework for evaluating microservices architecture designs," *Journal of Web Engineering*, vol. 19, no. 3-4, pp. 341–370, 2020. [Online]. Available: https://doi.org/10.13052/jwe1540-9589.19341 (Cited on page 297.)

[7] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013. [Online]. Available: https://doi.org/10.1109/tse.2012.64 (Cited on pages 9, 48, 276, 277, 278, and 280.)

[8] J. Alonso, L. Orue-Echevarria, V. Casola, A. I. Torre, M. Huarte, E. Osaba, and J. L. Lobo, "Understanding the challenges and novel architectural

models of multi-cloud native applications - a systematic literature review," *Journal of Cloud Computing*, vol. 12, no. 1, 2023. [Online]. Available: https://doi.org/10.1186/s13677-022-00367-6 (Cited on pages 9, 27, and 47.)

[9] N. Alshuqayran, N. Ali, and R. Evans, "A Systematic Mapping Study in Microservice Architecture," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2016. [Online]. Available: https://doi.org/10.1109/soca.2016.15 (Cited on pages 21 and 53.)

[10] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010. [Online]. Available: https://doi.org/10.1109/icsm.2010.5609747 (Cited on page 41.)

[11] A. Amiri, U. Zdun, and A. V. Hoorn, "Modeling and empirical validation of reliability and performance trade-offs of dynamic routing in service- and cloud-based architectures," *IEEE Transactions on Services Computing*, pp. 1–1, 2022. [Online]. Available: https://doi.org/10.1109/tsc.2021.3098178 (Cited on page 280.)

[12] S. Apel, F. Hertrampf, and S. Späthe, "Towards a metrics-based software quality rating for a microservice architecture," in *Innovations for Community Services*. Springer International Publishing, 2019, pp. 205–220. [Online]. Available: https://doi.org/10.1007/978-3-030-22482-0_15 (Cited on pages 165, 168, 169, 170, 280, 288, 289, 292, 295, 298, 299, and 308.)

[13] J. Arundel and J. Domingus, *Cloud Native DevOps with Kubernetes*. O'Reilly Media, Inc., 2019. [Online]. Available: https://learning.oreilly.com/library/view/cloud-native-devops/9781492040750/ (Cited on pages 61, 108, 109, 121, 122, 123, 127, 129, 133, 134, 138, 140, and 143.)

[14] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, M. Galster, and P. Avgeriou, "A mapping study on design-time quality attributes and metrics," *Journal of Systems and Software*, vol. 127, pp. 52–77, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2017.01.026 (Cited on pages 40 and 41.)

[15] T. Asik and Y. E. Selcuk, "Policy enforcement upon software based on microservice architecture," in *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, 2017, pp. 283–287. [Online]. Available: https://doi.org/10.1109/SERA.2017.7965739 (Cited on pages 295, 296, 297, and 302.)

[16] W. K. G. Assuncao, T. E. Colanzi, L. Carvalho, J. A. Pereira, A. Garcia, M. J. de Lima, and C. Lucena, "A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021. [Online]. Available: https://doi.org/10.1109/saner50967.2021.00042 (Cited on page 305.)

[17] D. Athanasopoulos and A. V. Zarras, "Fine-grained metrics of cohesion lack for service interfaces," in *2011 IEEE International Conference on Web Services*.

IEEE, 2011. [Online]. Available: https://doi.org/10.1109/icws.2011.27 (Cited on page 295.)

[18] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, "Cohesion-driven decomposition of service interfaces without access to source code," *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 550–562, 2015. [Online]. Available: https://doi.org/10.1109/tsc.2014.2310195 (Cited on page 295.)

[19] O. Baddreddin and K. Rahad, "The impact of design and uml modeling on codebase quality and sustainability," in *28th CASCON*, ser. CASCON '18. Markham, Ontario, Canada: IBM, 2018, pp. 236–244. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/3291291.3291315 (Cited on page 48.)

[20] A. Balalaie, A. Heydarnoori, and P. Jamshidi, *Migrating to Cloud-Native Architectures Using Microservices: An Experience Report*. Springer International Publishing, 2016, pp. 201–215. [Online]. Available: https://doi.org/10.1007/978-3-319-33313-7_15 (Cited on page 23.)

[21] A. Bambhore Tukaram, S. Schneider, N. E. Díaz Ferreyra, G. Simhandl, U. Zdun, and R. Scandariato, "Towards a security benchmark for the architectural design of microservice applications," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ser. ARES 2022, , Ed. ACM, 2022. [Online]. Available: https://doi.org/10.1145/3538969.3543807 (Cited on page 280.)

[22] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002. [Online]. Available: https://doi.org/10.1109/32.979986 (Cited on page 31.)

[23] G. Baranwal and D. P. Vidyarthi, "A framework for selection of best cloud service provider using ranked voting method," in *International advance computing conference (IACC)*. IEEE, 2014. [Online]. Available: https://doi.org/10.1109/iadcc.2014.6779430 (Cited on page 167.)

[24] L. Baresi and M. Garriga, "Microservices: The evolution and extinction of web services?" in *Microservices*. Springer International Publishing, 2019, pp. 3–28. [Online]. Available: https://doi.org/10.1007/978-3-030-31646-4_1 (Cited on pages 21, 22, 23, and 26.)

[25] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, pp. 528–532, 1994. (Cited on page 39.)

[26] K. Bastani and J. Long, *Cloud Native Java*. O'Reilly Media, Inc., 2017. [Online]. Available: https://learning.oreilly.com/library/view/cloud-native-java/9781449374631/ (Cited on pages 61, 113, 114, 118, 121, 122, 123, 129, 132, 135, 136, 138, 139, 146, 147, 148, and 149.)

[27] M. Becker, S. Lehrig, and S. Becker, "Systematically deriving quality metrics for cloud computing systems," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New

York, NY, USA: Association for Computing Machinery, 2015, pp. 169–174. [Online]. Available: https://doi.org/10.1145/2668930.2688043 (Cited on pages 39 and 172.)

[28] J. Bellendorf and Z. Á. Mann, "Specification of cloud topologies and orchestration using TOSCA: a survey," *Computing*, vol. 102, no. 8, pp. 1793–1815, 2019. [Online]. Available: https://doi.org/10.1007/s00607-019-00750-3 (Cited on page 55.)

[29] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann, "A Systematic Review of Cloud Modeling Languages," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–38, 2018. [Online]. Available: https://doi.org/10.1145/3150227 (Cited on pages 52 and 53.)

[30] D. Bermbach, E. Wittern, and S. Tai, *Cloud Service Benchmarking*. Springer International Publishing, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-55483-9 (Cited on pages 73 and 187.)

[31] A. Bernal, M. E. Cambronero, V. Valero, A. Nunez, and P. C. Canizares, "A Framework for Modeling Cloud Infrastructures and User Interactions," *IEEE Access*, vol. 7, pp. 43 269–43 285, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2907180 (Cited on page 50.)

[32] A. Bertolino, G. D. Angelis, M. Gallego, B. García, F. Gortázar, F. Lonetti, and E. Marchetti, "A systematic review on cloud testing," *ACM Computing Surveys*, vol. 52, no. 5, pp. 1–42, 2019. [Online]. Available: https://doi.org/10.1145/3331447 (Cited on page 87.)

[33] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: Portable automated deployment and management of cloud applications," in *Advanced Web Services*. Springer New York, 2013, pp. 527–549. [Online]. Available: https://doi.org/10.1007/978-1-4614-7535-4_22 (Cited on page 55.)

[34] G. Blair, N. Bencomo, and R. B. France, "Models@Run.Time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009. [Online]. Available: https://doi.org/10.1109/MC.2009.326 (Cited on pages 48 and 49.)

[35] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd International Conference on Software Engineering*, ser. ICSE '76. Washington, DC, USA: IEEE Computer Society Press, 1976, pp. 592–605. [Online]. Available: https://doi.org/10.5555/800253.807736 (Cited on pages 28 and 31.)

[36] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service- and microservice-based systems: A literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, ser. IWSM Mensura '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 107–115. [Online]. Available: https://doi.org/10.1145/3143434.3143443 (Cited on pages 162, 168, 294, 295, 296, 301, 302, 307, and 308.)

[37] ——, "Collecting service-based maintainability metrics from RESTful API descriptions: Static analysis and threshold derivation," in *Communications in Computer and Information Science*. Springer International Publishing, 2020, pp. 215–227. [Online]. Available: https://doi.org/10.1007/978-3-030-59155-7_16 (Cited on pages 295 and 307.)

[38] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified modeling language user guide,(the 2nd edition)*, ser. Addison-Wesley Object Technology Series. Addison-Wesley, 2005. [Online]. Available: https://www.oreilly.com/library/view/the-unified-modeling/0321267974/ (Cited on pages 47, 50, 51, and 54.)

[39] A. Bouguettaya, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, M. Singh, Y. Wang, B. Blake, S. Dustdar, F. Leymann, M. Papazoglou, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, and B. Benatallah, "A service computing manifesto," *Communications of the ACM*, vol. 60, no. 4, pp. 64–72, 2017. [Online]. Available: https://doi.org/10.1145/2983528 (Cited on page 19.)

[40] J. Braeuer, R. Ploesch, and M. Saft, "Measuring maintainability of OO-software - validating the IT-CISQ quality model," in *Advances in Intelligent Systems and Computing*. Springer International Publishing, 2016, pp. 283–301. [Online]. Available: https://doi.org/10.1007/978-3-319-46535-7_22 (Cited on page 44.)

[41] M. Brito, J. Cunha, and J. a. Saraiva, "Identification of microservices from monolithic applications through topic modelling," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ser. SAC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1409–1418. [Online]. Available: https://doi.org/10.1145/3412841.3442016 (Cited on pages 292, 295, 301, 305, and 307.)

[42] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From Monolithic to Microservices: An Experience Report from the Banking Domain," *IEEE Software*, vol. 35, no. 3, pp. 50–55, 2018. [Online]. Available: https://doi.org/10.1109/ms.2018.2141026 (Cited on page 23.)

[43] V. Bushong, A. S. Abdelfattah, A. A. Maruf, D. Das, A. Lehman, E. Jaroszewski, M. Coffey, T. Cerny, K. Frajtak, P. Tisnovsky, and M. Bures, "On microservice analysis and architecture evolution: A systematic mapping study," *Applied Sciences*, vol. 11, no. 17, p. 7856, 2021. [Online]. Available: https://doi.org/10.3390/app11177856 (Cited on pages 21 and 22.)

[44] S. Böhm and G. Wirtz, "Pulceo - a novel architecture for universal and lightweight cloud-edge orchestration," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2023, pp. 37–47. [Online]. Available: https://doi.org/10.1109/sose58276.2023.00011 (Cited on page 17.)

[45] E. Börger, "Approaches to modeling business processes: a critical analysis of BPMN, workflow patterns and YAWL," *Software & Systems*

Modeling, vol. 11, no. 3, pp. 305–318, 2011. [Online]. Available: https://doi.org/10.1007/s10270-011-0214-z (Cited on page 51.)

[46] C. Calero, M. Piattini, and M. Genero, "Empirical validation of referential integrity metrics," *Information and Software Technology*, vol. 43, no. 15, pp. 949–957, 2001. [Online]. Available: https://doi.org/10.1016/s0950-5849(01)00202-6 (Cited on page 44.)

[47] M. Cardarelli, L. Iovino, P. D. Francesco, A. D. Salle, I. Malavolta, and P. Lago, "An extensible data-driven approach for evaluating the quality of microservice architectures," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing - SAC '19*. ACM Press, 2019. [Online]. Available: https://doi.org/10.1145/3297280.3297400 (Cited on page 280.)

[48] A. Cater-Steel, M. Toleman, and T. Rout, "Addressing the challenges of replications of surveys in software engineering research," in *International Symposium on Empirical Software Engineering*. IEEE, 2005. [Online]. Available: https://doi.org/10.1109/isese.2005.1541829 (Cited on page 65.)

[49] T. Cerny, M. J. Donahoo, and J. Pechanec, "Disambiguation and comparison of SOA, microservices and self-contained systems," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems - RACS '17*. ACM Press, 2017. [Online]. Available: https://doi.org/10.1145/3129676.3129682 (Cited on pages 21 and 22.)

[50] T. Cerny, J. Svacina, D. Das, V. Bushong, M. Bures, P. Tisnovsky, K. Frajtak, D. Shin, and J. Huang, "On Code Analysis Opportunities and Challenges for Enterprise Systems and Microservices," *IEEE Access*, vol. 8, pp. 159 449–159 470, 2020. [Online]. Available: https://doi.org/10.1109/access.2020.3019985 (Cited on pages 4 and 9.)

[51] T. Cerny, A. S. Abdelfattah, V. Bushong, A. A. Maruf, and D. Taibi, "Microservice architecture reconstruction and visualization techniques: A review," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022. [Online]. Available: https://doi.org/10.1109/sose55356.2022.00011 (Cited on page 48.)

[52] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM TODS*, vol. 1, no. 1, pp. 9–36, 1976. [Online]. Available: https://doi.org/10.1145/320434.320440 (Cited on pages 51 and 54.)

[53] T. Chen, R. Bahsoon, and X. Yao, "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–40, 2018. [Online]. Available: https://doi.org/10.1145/3190507 (Cited on page 17.)

[54] M. Ciavotta, G. P. Gibilisco, D. Ardagna, E. D. Nitto, M. Lattuada, and M. A. A. Da Silva, "Architectural Design of Cloud Applications: A Performance-Aware Cost Minimization Approach," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1571–1591, 2022. [Online]. Available: https://doi.org/10.1109/TCC.2020.3015703 (Cited on pages 277, 278, and 279.)

[55] CNCF, "CNCF Cloud Native Definition v1.0," Online, 2018. [Online]. Available: https://github.com/cncf/toc/blob/master/DEFINITION.md (Cited on page 60.)

[56] ——, "Cncf cloud native definition v1.1," Online, 2024. [Online]. Available: https://github.com/cncf/toc/blob/main/DEFINITION.md (Cited on pages 6, 7, and 26.)

[57] V. Cortellessa, D. Di Pompeo, and M. Tucci, "Exploring Sustainable Alternatives for the Deployment of Microservices Architectures in the Cloud," in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024, pp. 34–45. [Online]. Available: https://doi.org/10.1109/ICSA59870.2024.00012 (Cited on pages 277, 278, and 279.)

[58] J. Daniel, E. Guerra, T. Rosa, and A. Goldman, "Towards the detection of microservice patterns based on metrics," in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2023, pp. 132–139. [Online]. Available: https://doi.org/10.1109/SEAA60479.2023.00029 (Cited on pages 32, 280, 288, 289, 290, 291, 292, and 295.)

[59] C. Davis, "Realizing Software Reliability in the Face of Infrastructure Instability," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 34–40, 2017. [Online]. Available: https://doi.org/10.1109/mcc.2017.4250927 (Cited on page 90.)

[60] ——, *Cloud Native Patterns*. Manning Publications Co., 2019. [Online]. Available: https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/ (Cited on pages 3, 25, 32, 61, 113, 114, 115, 116, 118, 120, 121, 122, 131, 137, 138, 143, 146, 149, 150, and 239.)

[61] R. H. de Souza, P. A. Flores, M. A. R. Dantas, and F. Siqueira, "Architectural recovering model for distributed databases: A reliability, availability and serviceability approach," in *Symposium on Computers and Communication (ISCC)*. IEEE, 2016. [Online]. Available: https://doi.org/10.1109/iscc.2016.7543799 (Cited on pages 168 and 171.)

[62] A. DeCastellarnau, "A classification of response scale characteristics that affect data quality: a literature review," *Quality & Quantity*, vol. 52, no. 4, pp. 1523–1559, 2017. [Online]. Available: https://doi.org/10.1007/s11135-017-0533-4 (Cited on page 66.)

[63] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard, "An activity-based quality model for maintainability," in *2007 IEEE International Conference on Software Maintenance*, , Ed. IEEE, 2007, pp. 184–193. [Online]. Available: https://doi.org/10.1109/icsm.2007.4362631 (Cited on page 30.)

[64] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-native computing: A survey from the perspective of services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, 2024. [Online]. Available: https://doi.org/10.1109/jproc.2024.3353855 (Cited on page 87.)

[65] U. Desai, S. Bandyopadhyay, and S. Tamilselvam, "Graph neural network to dilute outliers for refactoring monolith application," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, pp. 72–80, 2021. [Online]. Available: https://doi.org/10.1609/aaai.v35i1.16079 (Cited on page 307.)

[66] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 638–653, 2002. [Online]. Available: https://doi.org/10.1109/tse.2002.1019479 (Cited on page 48.)

[67] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*. Springer International Publishing, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12 (Cited on page 21.)

[68] R. Dromey, "A model for software product quality," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 146–162, 1995. [Online]. Available: https://doi.org/10.1109/32.345830 (Cited on pages 28, 31, and 34.)

[69] ——, "Cornering the chimera [software quality]," *IEEE Software*, vol. 13, no. 1, pp. 33–43, 1996. [Online]. Available: https://doi.org/10.1109/52.476284 (Cited on pages 28, 29, 30, 33, and 34.)

[70] K. Dürr, "Providing Tooling Support for Modeling Cloud-Native Application Architectures," mathesis, Distributed Systems Group - University of Bamberg, 2022, unpublished. (Cited on pages 70, 78, 79, 221, and 256.)

[71] K. Dürr and R. Lichtenthäler, "An evaluation of modeling options for cloud-native application architectures to enable quality investigations," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2022. [Online]. Available: https://doi.org/10.1109/ucc56403.2022.00053 (Cited on pages 12, 54, 71, 78, 221, 223, 224, and 349.)

[72] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A performance tester's dream or nightmare?" in *In Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE 20)*, 2020. [Online]. Available: https://doi.org/10.1145/3358960.3379124 (Cited on pages 74 and 75.)

[73] M. Elaasar, *Definition of Modeling vs. Programming Languages*. Springer International Publishing, 2018, pp. 35–51. [Online]. Available: https://doi.org/10.1007/978-3-030-03418-4_3 (Cited on page 47.)

[74] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann, "Evaluation of microservice architectures: A metric and tool-based approach," in *Lecture Notes in Business Information Processing*. Springer International Publishing, 2018, pp. 74–89. [Online]. Available: https://doi.org/10.1007/978-3-319-92901-9_8 (Cited on pages 293, 295, 296, 298, and 302.)

[75] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005. [Online]. Available: https://www.thomaserl.com/book/service-oriented-architecture-

concepts-technology-and-design/overview/index.html (Cited on pages 18 and 20.)

[76] ——, *SOA: Principles of Service Design*. Pearson Education, 2007. [Online]. Available: https://www.oreilly.com/library/view/soa-principles-of/9780132344821/ (Cited on pages 18, 19, and 20.)

[77] ——, *SOA Design Patterns*, 2008, Ed. Prentice Hall, 2008. [Online]. Available: https://www.oreilly.com/library/view/soa-design-patterns/9780136135166/ (Cited on pages 18, 19, 20, 21, and 50.)

[78] ——, *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. Prentice Hall, 2016. [Online]. Available: https://learning.oreilly.com/library/view/service-oriented-architecture-analysis/9780133858709/ (Cited on pages 18, 20, 21, 23, and 50.)

[79] T. Erl and E. B. Monroy, *Cloud Computing: Concepts, Technology, Security, and Architecture*, 2nd ed. Prentice Hall, 2023. [Online]. Available: https://learning.oreilly.com/library/view/cloud-computing-concepts/9780138052287/ (Cited on pages 15, 16, and 17.)

[80] T. Erl, R. Puttini, and Z. Mahmood, *Cloud Computing: Concepts, Technology & Architecture*, 1st ed. Pearson, 2013. [Online]. Available: https://www.oreilly.com/library/view/cloud-computing-concepts/9780133387568/ (Cited on pages 39 and 172.)

[81] Eurostat, "Cloud computing - statistics on the use by enterprises," online, 2023. [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises (Cited on page 3.)

[82] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Pearson International, 2003. [Online]. Available: https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/ (Cited on page 22.)

[83] S. Farshidi, I. B. Kwantes, and S. Jansen, "Business process modeling language selection for research modelers," *Software and Systems Modeling*, vol. 23, no. 1, pp. 137–162, 2023. [Online]. Available: https://doi.org/10.1007/s10270-023-01110-8 (Cited on page 51.)

[84] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns*. Springer Vienna, 2014. [Online]. Available: https://doi.org/10.1007/978-3-7091-1568-8 (Cited on pages 17, 22, 25, 32, and 60.)

[85] R. Ferenc, P. Hegedűs, and T. Gyimóthy, "Software product quality models," in *Evolving Software Systems*. Springer Berlin Heidelberg, 2013, pp. 65–100. [Online]. Available: https://doi.org/10.1007/978-3-642-45398-4_3 (Cited on pages 8, 28, 30, and 83.)

[86] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012. [Online]. Available: https://doi.org/10.1016/j.jss.2011.05.044 (Cited on page 41.)

[87] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "CloudMF: Model-Driven Management of Multi-Cloud Applications," *ACM TOIT*, vol. 18, no. 2, pp. 16:1–16:24, 2018. [Online]. Available: https://doi.org/10.1145/3125621 (Cited on pages 54 and 221.)

[88] G. Filippone, N. Q. Mehmood, M. Autili, F. Rossi, and M. Tivoli, "From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. IEEE, 2023. [Online]. Available: https://doi.org/10.1109/icsa56044.2023.00013 (Cited on page 304.)

[89] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st ed. Pearson International, 2002. [Online]. Available: https://www.pearson.de/patterns-of-enterprise-application-architecture-9780321127426 (Cited on page 4.)

[90] J. M. S. França and M. S. Soares, "SOAQM: Quality model for SOA applications based on ISO 25010," in *Proceedings of the 17th International Conference on Enterprise Information Systems*. SCITEPRESS - Science and and Technology Publications, 2015. [Online]. Available: https://doi.org/10.5220/0005369100600070 (Cited on page 21.)

[91] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *FOSE*. IEEE, 2007. [Online]. Available: https://doi.org/10.1109/fose.2007.14 (Cited on page 49.)

[92] P. D. Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *Journal of Systems and Software*, vol. 150, pp. 77–97, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.01.001 (Cited on pages 21, 22, and 53.)

[93] A. Frisch and R. Lichtenthäler, "A Review of Software Architecture Optimization Approaches for Cloud Applications," in *17th Central European Workshop on Services and their Composition (ZEUS)*, vol. 4030. CEUR-WS, 2025, pp. 16–25. [Online]. Available: https://ceur-ws.org/Vol-4030/paper4.pdf (Cited on pages 277 and 278.)

[94] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices Migration in Industry: Intentions, Strategies, and Challenges," in *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00081 (Cited on page 23.)

[95] T. Galli, F. Chiclana, and F. Siewe, "Software product quality models, developments, trends, and evaluation," *SN Computer Science*, vol. 1, no. 3, 2020. [Online]. Available: https://doi.org/10.1007/s42979-020-00140-z (Cited on pages 30, 34, 35, and 45.)

[96] M. Galster and P. Avgeriou, "Qualitative analysis of the impact of SOA patterns on quality attributes," in *2012 12th International Conference on Quality Software*. IEEE, 2012. [Online]. Available: https://doi.org/10.1109/qsic.2012.35 (Cited on page 21.)

[97] I. U. P. Gamage and I. Perera, "Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach," in *2021 Moratuwa Engineering Research Conference (MERCon)*, 2021, pp. 699–704. [Online]. Available: https://doi.org/10.1109/MERCon52712.2021.9525743 (Cited on pages 298, 299, and 302.)

[98] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. [Online]. Available: https://www.oreilly.com/library/view/design-patterns-elements/0201633612/ (Cited on page 32.)

[99] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-Native Applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017. [Online]. Available: https://doi.org/10.1109/mcc.2017.4250939 (Cited on pages 7, 8, 16, 26, 27, 60, and 85.)

[100] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009. [Online]. Available: https://doi.org/10.1109/csmr.2009.59 (Cited on page 33.)

[101] J. Garrison and K. Nova, *Cloud Native Infrastructure*. O'Reilly Media, Inc., 2017. [Online]. Available: https://learning.oreilly.com/library/view/cloud-native-infrastructure/9781491984291/ (Cited on pages 61, 117, 118, 121, 122, 123, 134, and 139.)

[102] M. Geiger, S. Harrer, J. Lenhard, and G. Wirtz, "Bpmn 2.0: The state of support and implementation," *Future Generation Computer Systems*, vol. 80, pp. 250–262, 2018. [Online]. Available: https://doi.org/10.1016/j.future.2017.01.006 (Cited on page 49.)

[103] P. Genfer and U. Zdun, "Identifying domain-based cyclic dependencies in microservice APIs using source code detectors," in *Software Architecture*. Springer International Publishing, 2021, pp. 207–222. [Online]. Available: https://doi.org/10.1007/978-3-030-86044-8_15 (Cited on page 294.)

[104] C. M. Gerpheide, R. R. H. Schiffelers, and A. Serebrenik, "Assessing and improving quality of QVTo model transformations," *Software Quality Journal*, vol. 24, no. 3, pp. 797–834, 2015. [Online]. Available: https://doi.org/10.1007/s11219-015-9280-8 (Cited on pages 43 and 44.)

[105] D. Gesvindr and B. Buhnova, "Performance challenges, current bad practices, and hints in paas cloud application design," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 4, pp. 3–12, 2016. [Online]. Available: https://doi.org/10.1145/2897356.2897358 (Cited on page 32.)

[106] F. D. Giraldo, S. España, Ó. Pastor, and W. J. Giraldo, "Considerations about quality in model-driven engineering," *Software Quality Journal*, vol. 26, no. 2,

pp. 685–750, 2016. [Online]. Available: https://doi.org/10.1007/s11219-016-9350-6 (Cited on page 47.)

[107] A. Goeb and K. Lochmann, "A software quality model for SOA," in *Proceedings of the 8th international workshop on Software quality - WoSQ '11*. ACM Press, 2011. [Online]. Available: https://doi.org/10.1145/2024587.2024593 (Cited on pages 21 and 31.)

[108] S. R. Goniwada, *Cloud Native Architecture and Design*. Apress, 2021. [Online]. Available: https://doi.org/10.1007/978-1-4842-7226-8 (Cited on pages 61, 109, 110, 111, 112, 113, 114, 115, 116, 118, 120, 121, 122, 123, 124, 125, 128, 131, 133, 134, 135, 136, 139, 140, 141, 144, 145, 146, 147, 148, and 267.)

[109] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle, "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017. [Online]. Available: https://doi.org/10.1109/icsaw.2017.9 (Cited on page 55.)

[110] X. Guerron, S. Abrahão, E. Insfran, M. Fernández-Diego, and F. González-Ladrón-De-Guevara, "A taxonomy of quality metrics for cloud services," *IEEE Access*, vol. 8, pp. 131 461–131 498, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3009079 (Cited on pages 167, 168, 171, 291, and 292.)

[111] L. Guzman, A. M. Vollmer, M. Ciolkowski, and M. Gillmann, "Formative evaluation of a tool for managing software quality," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, cited Wagner2015. [Online]. Available: https://doi.org/10.1109/esem.2017.43 (Cited on page 45.)

[112] U. Habiba, R. Masood, M. A. Shibli, and M. A. Niazi, "Cloud identity management security issues & solutions: a taxonomy," *Complex Adaptive Systems Modeling*, vol. 2, no. 1, 2014. [Online]. Available: https://doi.org/10.1186/s40294-014-0005-9 (Cited on page 86.)

[113] N. Herbst, C. L. Abad, A. Iosup, A. Bauer, S. Kounev, G. Oikonomou, E. V. Eyk, G. Kousiouris, A. Evangelinou, R. Krebs, and T. Brecht, "Quantifying Cloud Performance and Dependability," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no. 4, pp. 1–36, 2018. [Online]. Available: https://doi.org/10.1145/3236332 (Cited on pages 38 and 89.)

[114] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani, "A metrics suite for evaluating flexibility and complexity in service oriented architectures," in *Service-Oriented Computing – ICSOC 2008 Workshops*. Springer Berlin Heidelberg, 2009, pp. 41–52. [Online]. Available: https://doi.org/10.1007/978-3-642-01247-1_5 (Cited on pages 169, 307, and 308.)

[115] H. Hofmeister and G. Wirtz, "Supporting service-oriented design with metrics," in *2008 12th International IEEE Enterprise Distributed Object*

Computing Conference. IEEE, 2008. [Online]. Available: https://doi.org/10.1109/edoc.2008.13 (Cited on pages 169, 293, 294, 302, 306, and 308.)

[116] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003. [Online]. Available: https://www.oreilly.com/library/view/enterprise-integration-patterns/0321200683/ (Cited on pages 32 and 239.)

[117] C. N. Höfer and G. Karagiannis, "Cloud computing services: taxonomy and comparison," *Journal of Internet Services and Applications*, vol. 2, no. 2, pp. 81–94, 2011. [Online]. Available: https://doi.org/10.1007/s13174-011-0027-x (Cited on page 17.)

[118] B. Ibryam and R. Huß, *Kubernetes Patterns*. O'Reilly Media, Inc., 2020. [Online]. Available: https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ (Cited on pages 32, 61, 115, 118, 123, 133, 134, 135, 137, 138, 143, and 267.)

[119] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. Bauer, A. V. Papadopoulos, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscalers for complex workflows," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, 2018. [Online]. Available: https://doi.org/10.1145/3164537 (Cited on page 268.)

[120] K. Indrasiri and S. Suhothayan, *Design Patterns for Cloud Native Applications*. O'Reilly Media, Inc., 2021. [Online]. Available: https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ (Cited on pages 32, 61, 107, 113, 114, 116, 118, 123, 127, 128, 132, 133, 134, 135, 137, 138, 139, 140, 142, 143, 144, 145, 146, 147, 148, and 150.)

[121] ISO/IEC, "ISO/IEC 9126 software engineering— product quality," Online, 2001. [Online]. Available: https://www.iso.org/standard/22749.html (Cited on pages 35 and 45.)

[122] ——, "ISO/IEC 25000 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)," Online, 2014. [Online]. Available: https://www.iso.org/standard/64764.html (Cited on pages 4, 5, 8, 11, 28, 29, 30, 35, 36, 61, 84, 85, 86, 87, 88, 89, 90, 91, and 279.)

[123] ISO/IEC/IEEE, "ISO/IEC/IEEE 42010:2022 - Software, systems and enterprise — Architecture description," 2022. [Online]. Available: https://www.iso.org/standard/74393.html (Cited on page 46.)

[124] Y. Izrailevsky and C. Bell, "Cloud reliability," *IEEE Cloud Computing*, vol. 5, no. 3, pp. 39–44, 2018. [Online]. Available: https://doi.org/10.1109/mcc.2018.032591615 (Cited on page 90.)

[125] P. Jamshidi, C. Pahl, S. Chinenyeze, and X. Liu, "Cloud migration patterns: a multi-cloud service architecture perspective," in *Service-Oriented Computing-ICSOC 2014 Workshops*. Springer, 2015, pp. 6–19. [Online]. Available: https://doi.org/10.1007/978-3-319-22885-3_2 (Cited on page 32.)

[126] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software*, vol. 35, no. 3, pp.

24–35, 2018. [Online]. Available: https://doi.org/10.1109/ms.2018.2141039 (Cited on pages 21, 22, and 23.)

[127] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai, "Functionality-oriented microservice extraction based on execution trace clustering," in *International Conference on Web Services (ICWS)*. IEEE, 2018. [Online]. Available: https://doi.org/10.1109/icws.2018.00034 (Cited on pages 295 and 305.)

[128] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, 2021. [Online]. Available: https://doi.org/10.1109/tse.2019.2910531 (Cited on pages 292, 295, 301, and 307.)

[129] C. T. Joseph and K. Chandrasekaran, "Straddling the crevasse: A review of microservice software architecture foundations and recent advancements," *Software: Practice and Experience*, vol. 49, no. 10, pp. 1448–1484, 2019. [Online]. Available: https://doi.org/10.1002/spe.2729 (Cited on page 21.)

[130] X. jun Wang, "Metrics for evaluating coupling and service granularity in service oriented architecture," in *2009 International Conference on Information Engineering and Computer Science*. IEEE, 2009. [Online]. Available: https://doi.org/10.1109/iciecs.2009.5362767 (Cited on pages 300 and 301.)

[131] H.-W. Jung, "Validating the external quality subcharacteristics of software products according to ISO/IEC 9126," *Computer Standards & Interfaces*, vol. 29, no. 6, pp. 653–661, 2007. [Online]. Available: https://doi.org/10.1016/j.csi.2007.03.004 (Cited on page 44.)

[132] S. Kapferer and O. Zimmermann, "Domain-Driven Architecture Modeling and Rapid Prototyping with Context Mapper," in *8th MODELSWARD*, vol. 1361. Springer, 2020, pp. 250–272. [Online]. Available: https://doi.org/10.1007/978-3-030-67445-8_11 (Cited on pages 54 and 221.)

[133] T. Karhikeyan and J. Geetha, "A metrics suite and fuzzy model for measuring coupling in service oriented architecture," in *2012 International Conference on Recent Advances in Computing and Software Systems*. IEEE, 2012. [Online]. Available: https://doi.org/10.1109/racss.2012.6212677 (Cited on pages 165, 268, 289, 300, 301, 306, and 308.)

[134] M. Kasunic, "Designing an Effective Survey," Carnegie-Mellon University, Software Engineering Institute, Tech. Rep. ADA441817, 2005. [Online]. Available: https://www.sei.cmu.edu/documents/1611/2005_002_001_14435.pdf (Cited on page 65.)

[135] V. Kaushik, P. Bhardwaj, and K. Lohani, *Game of Definitions—Do the NIST Definitions of Cloud Service Models Need an Update? A Remark*. Springer Nature Singapore, 2022, pp. 653–666. [Online]. Available: https://doi.org/10.1007/978-981-19-5037-7_47 (Cited on page 17.)

[136] A. Kazemi, A. Rostampour, A. Zamiri, P. Jamshidi, H. Haghighi, and F. Shams, "An information retrieval based approach for measuring service

conceptual cohesion," in *2011 11th International Conference on Quality Software*. IEEE, 2011. [Online]. Available: https://doi.org/10.1109/qsic.2011.24 (Cited on pages 168 and 295.)

[137] A. Kazemi, H. Haghighi, and F. Shams, "ABSIM: an automated business service identification method," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 09, pp. 1303–1342, 2013. [Online]. Available: https://doi.org/10.1142/s0218194013500411 (Cited on pages 294, 302, and 307.)

[138] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017. [Online]. Available: https://doi.org/10.1109/mcc.2017.4250933 (Cited on page 26.)

[139] F. Khomh and Y.-G. Guéhéneuc, "DEQUALITE: building design-based software quality models," in *Proceedings of the 15th Conference on Pattern Languages of Programs*. ACM Press, 2008. [Online]. Available: https://doi.org/10.1145/1753196.1753199 (Cited on pages 43 and 44.)

[140] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Software Engineering Group, School of Computer Science and Mathematics, Keele University, techreport 2.3, 2007. [Online]. Available: https://legacyfileshare.elsevier.com/promis_misc/525444systematicreviewsguide.pdf (Cited on page 62.)

[141] B. Kitchenham and S. Pfleeger, "Software quality: the elusive target [special issues section]," *IEEE Software*, vol. 13, no. 1, pp. 12–21, 1996. [Online]. Available: https://doi.org/10.1109/52.476281 (Cited on pages 28, 30, and 43.)

[142] B. Kitchenham, S. Linkman, A. Pasquini, and V. Nanni, "The squid approach to defining a quality model," *Software Quality Control*, vol. 6, no. 3, pp. 211–233, 1997. [Online]. Available: https://doi.org/10.1023/a:1018516103435 (Cited on page 29.)

[143] M. Kläs, K. Lochmann, and L. Heinemann, "Evaluating a quality model for software product assessments – a case study," in *Proc. of SQMB*, 2011, cited Deissenboeck2007. [Online]. Available: https://publica.fraunhofer.de/entities/publication/f00e0896-ff4f-45bd-a869-52c6bc9b1ed1 (Cited on page 45.)

[144] F. Knoch, "Evaluating Cost Implications of Cloud-Native Application Characteristics with a Focus on Reliability Aspects," Master's thesis, Distributed Systems Group - University of Bamberg, 2023, unpublished. (Cited on page 71.)

[145] F. Knoch, R. Lichtenthäler, and G. Wirtz, *Evaluating Cloud-Native Deployment Options with a Focus on Reliability Aspects*. Springer Nature Switzerland, 2024, pp. 63–82. [Online]. Available: https://doi.org/10.1007/978-3-031-72578-4_4 (Cited on pages 12, 71, 224, and 349.)

[146] N. Knoll and R. Lichtenthäler, "An experimental evaluation of relations between architectural and runtime metrics in microservices systems," in *Proceedings of the 13th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2023. [Online]. Available: https://doi.org/10.5220/0011728600003488 (Cited on page 44.)

[147] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, *Winery - A Modeling Tool for TOSCA-Based Cloud Applications*. Springer Berlin Heidelberg, 2013, pp. 700–704. [Online]. Available: https://doi.org/10.1007/978-3-642-45005-1_64 (Cited on pages 55 and 251.)

[148] S. Kounev, K.-D. Lange, and J. von Kistowski, *Systems Benchmarking*. Springer International Publishing, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-41705-5 (Cited on page 187.)

[149] N. Kratzke, "Cloud-native observability: The many-faceted benefits of structured and unified logging—a multi-case study," *Future Internet*, vol. 14, no. 10, p. 274, 2022. [Online]. Available: https://doi.org/10.3390/fi14100274 (Cited on pages 87, 123, and 219.)

[150] N. Kratzke and R. Peinl, "ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects," in *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*. IEEE, 2016. [Online]. Available: https://doi.org/10.1109/edocw.2016.7584353 (Cited on pages 26 and 46.)

[151] N. Kratzke and P.-C. Quint, "Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2017.01.001 (Cited on pages 6, 7, 15, 23, 24, 26, and 60.)

[152] P. Kruchten, H. Obbink, and J. Stafford, "The past, present, and future for software architecture," *IEEE Software*, vol. 23, no. 2, pp. 22–30, 2006. [Online]. Available: https://doi.org/10.1109/ms.2006.59 (Cited on page 45.)

[153] P. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995. [Online]. Available: https://doi.org/10.1109/52.469759 (Cited on page 50.)

[154] T. Kuroda and A. Gokhale, "Model-based automation for hardware provisioning in IT infrastructure," in *SysCon*. IEEE, 2014. [Online]. Available: https://doi.org/10.1109/syscon.2014.6819272 (Cited on page 50.)

[155] K. Kvam, R. Lie, and D. Bakkelund, "Legacy system exorcism by pareto's principle," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '05*. ACM Press, 2005, found through Yan2017. [Online]. Available: https://doi.org/10.1145/1094855.1094959 (Cited on page 44.)

[156] H. J. La, J. S. Her, and S. D. Kim, "Framework for evaluating reusability of component-as-a-service (CaaS)," in *2013 5th International Workshop on*

Principles of Engineering Service-Oriented Systems (PESOS). IEEE, 2013. [Online]. Available: https://doi.org/10.1109/pesos.2013.6635976 (Cited on pages 297 and 304.)

[157] P. Lago, I. Malavolta, H. Muccini, P. Pelliccione, and A. Tang, "The road ahead for architectural languages," *IEEE Software*, vol. 32, no. 1, pp. 98–105, 2015. [Online]. Available: https://doi.org/10.1109/ms.2014.28 (Cited on page 47.)

[158] C. Lampasona, A. Mayr, and M. Saft, "Early validation of software quality models with respect to minimality and completeness: An empirical analysis," in *Software Metrik Kongress*, 2013. [Online]. Available: https://publica.fraunhofer.de/entities/publication/69654c01-d2ab-4797-9e89-7126e3cbf170 (Cited on page 44.)

[159] M. Lankhorst, H. Proper, and H. Jonkers, "The anatomy of the ArchiMate language," *IJISMD*, vol. 1, no. 1, pp. 1–32, 2010. [Online]. Available: https://doi.org/10.4018/jismd.2010092301 (Cited on pages 51 and 54.)

[160] M. Lehmann and F. E. Sandnes, "A framework for evaluating continuous microservice delivery strategies," in *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ser. ICC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3018896.3018961 (Cited on page 309.)

[161] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of DevOps concepts and challenges," *ACM Computing Surveys*, vol. 52, no. 6, pp. 1–35, 2020. [Online]. Available: https://doi.org/10.1145/3359981 (Cited on pages 7 and 22.)

[162] L. Lelovic, M. Mathews, A. Elsayed, T. Cerny, K. Frajtak, P. Tisnovsky, and D. Taibi, "Architectural languages in the microservice era: a systematic mapping study," in *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, ser. RACS '22. ACM, 2022, pp. 39–46. [Online]. Available: https://doi.org/10.1145/3538641.3561486 (Cited on page 53.)

[163] L. Lelovic, M. Mathews, A. Abdelfattah, and T. Cerny, "Microservices architecture language for describing service view," in *Proceedings of the 13th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2023, pp. 220–227. [Online]. Available: https://doi.org/10.5220/0011850200003488 (Cited on page 47.)

[164] J. Lenhard, M. Blom, and S. Herold, "Exploring the suitability of source code metrics for indicating architectural inconsistencies," *Software Quality Journal*, vol. 27, no. 1, pp. 241–274, 2018. [Online]. Available: https://doi.org/10.1007/s11219-018-9404-z (Cited on page 50.)

[165] J.-L. Letouzey and T. Coq, "The SQALE analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code," in *2010 Second International Conference on Advances in System Testing and Validation Lifecycle*, , Ed. IEEE, 2010. [Online].

Available: https://doi.org/10.1109/valid.2010.31 (Cited on page 30.)

[166] J. Lewis and M. Fowler, "Microservices - a definition of this new architectural term," , 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html (Cited on pages 21 and 22.)

[167] F. Leymann, J. Wettinger, S. Wagner, and C. Fehling, "Native cloud applications - why virtual machines, images and containers miss the point!" in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2016. [Online]. Available: https://doi.org/10.5220/0006811300010001 (Cited on page 25.)

[168] R. Lichtenthaeler, "Clounaq - cloud-native architectural quality," in *16th Central European Workshop on Services and their Composition (ZEUS)*. CEUR-WS, 2024, pp. 22–26. [Online]. Available: https://ceur-ws.org/Vol-3673/paper4.pdf (Cited on pages 12, 249, 256, and 349.)

[169] R. Lichtenthäler and G. Wirtz, "Towards a Quality Model for Cloud-native Applications," in *Service-Oriented and Cloud Computing*. Springer International Publishing, 2022, pp. 109–117. [Online]. Available: https://doi.org/10.1007/978-3-031-04718-3_7 (Cited on pages 12, 60, 62, 70, 83, 91, 104, and 221.)

[170] ——, "A Review of Approaches for Quality Model Validations in the Context of Cloud-native Applications," in *14th Central European Workshop on Services and their Composition (ZEUS)*, 2022, pp. 30–41. [Online]. Available: http://ceur-ws.org/Vol-3113/paper6.pdf (Cited on pages 42, 43, 44, and 269.)

[171] ——, "Formulating a quality model for cloud-native software architectures: conceptual and methodological considerations," *Cluster Computing*, 2024. [Online]. Available: https://doi.org/10.1007/s10586-024-04343-4 (Cited on pages 12, 33, 42, 59, 62, 64, 70, 83, 277, and 278.)

[172] ——, "An Experimental Validation of Architectural Measures for Cloud-Native Quality Evaluations," in *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, 2025, pp. 374–384. [Online]. Available: https://doi.org/10.1109/CLOUD67622.2025.00045 (Cited on pages 12, 72, 74, 186, and 270.)

[173] ——, "Evaluating Cloud-native Software Architectures with Clounaq," in *Service-Oriented Computing*, M. Aiello, J. Barzen, S. Dustdar, and F. Leymann, Eds. Springer Nature Switzerland, 2025, pp. 132–141. [Online]. Available: https://doi.org/10.1007/978-3-032-07313-6_8 (Cited on pages 12, 76, 206, 224, 232, 260, and 267.)

[174] R. Lichtenthäler, J. Fritzsch, and G. Wirtz, "Cloud-Native Architectural Characteristics and their Impacts on Software Quality: A Validation Survey," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 9–18. [Online]. Available: https://doi.org/10.1109/SOSE58276.2023.00008 (Cited

on pages 12, 64, 83, 104, 151, 152, 154, 155, 156, 157, and 349.)

[175] R. W. Lissitz and S. B. Green, "Effect of the number of scale points on reliability: A monte carlo approach." *Journal of applied psychology*, vol. 60, no. 1, p. 10, 1975. [Online]. Available: https://doi.org/10.1037/h0076268 (Cited on page 66.)

[176] K. Lochmann and A. Goeb, "A unifying model for software quality," in *Proceedings of the 8th international workshop on Software quality - WoSQ '11*. ACM Press, 2011. [Online]. Available: https://doi.org/10.1145/2024587.2024591 (Cited on pages 28, 29, 30, 31, and 38.)

[177] K. Lochmann, J. Ramadani, and S. Wagner, "Are comprehensive quality models necessary for evaluating software quality?" in *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*. ACM, 2013. [Online]. Available: https://doi.org/10.1145/2499393.2499404 (Cited on page 39.)

[178] M. Loukides, *The Cloud in 2021: Adoption Continues*. O'Reilly Media, Inc., 2021. [Online]. Available: https://www.oreilly.com/radar/the-cloud-in-2021-adoption-continues/ (Cited on page 3.)

[179] ——, "Technology trends for 2024," online, 2024. [Online]. Available: https://www.oreilly.com/radar/technology-trends-for-2024/ (Cited on pages 3 and 6.)

[180] Q. Ma, N. Zhou, Y. Zhu, and H. Wang, "Evaluating service identification with design metrics on business process decomposition," in *International Conference on Services Computing*. IEEE, 2009. [Online]. Available: ttps://doi.org/10.1109/scc.2009.44 (Cited on pages 294, 302, and 307.)

[181] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869–891, 2013. [Online]. Available: https://doi.org/10.1109/tse.2012.74 (Cited on pages 47 and 77.)

[182] J. Manner, "A structured literature review approach to define serverless computing and function as a service," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 2023, pp. 516–522. [Online]. Available: https://doi.org/10.1109/cloud60044.2023.00068 (Cited on page 17.)

[183] B. Martens, M. Walterbusch, and F. Teuteberg, "Costing of cloud computing services: A total cost of ownership approach," in *2012 45th Hawaii International Conference on System Sciences*. IEEE, 2012. [Online]. Available: https://doi.org/10.1109/hicss.2012.186 (Cited on pages 270 and 271.)

[184] A. Mayr, R. Plosch, M. Klas, C. Lampasona, and M. Saft, "A comprehensive code-based quality model for embedded systems: Systematic development and validation by industrial projects," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012. [Online]. Available: https://doi.org/10.1109/issre.2012.4 (Cited on pages 31, 43, 44, 277, 278, and 279.)

[185] A. Mayr, R. Plosch, and M. Saft, "Objective measurement of safety in the context of IEC 61508-3," in *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2013. [Online]. Available: https://doi.org/10.1109/seaa.2013.32 (Cited on page 43.)

[186] J. A. McCall and G. F. W. Paul K. Richards, "Factors in software quality. volume i. concepts and definitions of software quality," General Electric Co, Sunnyvale, CA, techreport ADA049014, 1977. [Online]. Available: https://apps.dtic.mil/sti/tr/pdf/ADA049014.pdf (Cited on page 31.)

[187] J. H. McDonald, *Handbook of Biological Statistics*, 3rd ed. Sparky House Publishing, Baltimore, Maryland, 2014. [Online]. Available: https://www.biostathandbook.com/HandbookBioStatThird.pdf (Cited on page 68.)

[188] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000. [Online]. Available: https://doi.org/10.1109/32.825767 (Cited on page 47.)

[189] K. Mehmood, S. Si-Said Cherfi, and I. Comyn-Wattiau, "Data Quality through Conceptual Model Quality - Reconciling Researchers and Practitioners through a Customizable Quality Model," in *ICIQ (International Conference on Information Quality), Hasso-Plattner-Institute Potsdam, Germany*, X, France, 2009, pp. 61–74. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01125702 (Cited on pages 43 and 44.)

[190] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Gaithersburg, Gaithersburg, Tech. Rep., 2011. [Online]. Available: https://doi.org/10.6028/nist.sp.800-145 (Cited on pages 4, 15, 16, and 17.)

[191] N. C. Mendonça, D. Garlan, B. Schmerl, and J. Cámara, "Generality vs. reusability in architecture-based self-adaptation," in *12th ECSA Companion*. ACM, 2018. [Online]. Available: https://doi.org/10.1145/3241403.3241423 (Cited on page 49.)

[192] S. Misra, I. Akman, and R. Colomo-Palacios, "Framework for evaluation and validation of software complexity measures," *IET Software*, vol. 6, no. 4, p. 323, 2012. [Online]. Available: https://doi.org/10.1049/iet-sen.2011.0206 (Cited on page 45.)

[193] S. K. Moghaddam, R. Buyya, and K. Ramamohanarao, "Performance-aware management of cloud resources: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 52, no. 4, 2019. [Online]. Available: https://doi.org/10.1145/3337956 (Cited on pages 17 and 88.)

[194] D. L. Moody, "Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions," *Data & Knowledge Engineering*, vol. 55, no. 3, pp. 243–276, 2005. [Online]. Available: https://doi.org/10.1016/j.datak.2004.12.005 (Cited on pages 35 and 43.)

[195] ——, "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Transactions*

on Software Engineering, vol. 35, no. 6, pp. 756–779, 2009. [Online]. Available: https://doi.org/10.1109/TSE.2009.67 (Cited on pages 51, 52, 78, and 239.)

[196] S. Morasca, *Fundamental Aspects of Software Measurement*. Springer Berlin Heidelberg, 2013, pp. 1–45. [Online]. Available: https://doi.org/10.1007/978-3-642-36054-1_1 (Cited on pages 38 and 39.)

[197] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Software quality metrics aggregation in industry," *Journal of Software: Evolution and Process*, vol. 25, no. 10, pp. 1117–1135, 2012. [Online]. Available: https://doi.org/10.1002/smr.1558 (Cited on pages 40 and 42.)

[198] K. Mordal-Manet, J. Laval, S. Ducasse, N. Anquetil, F. Balmas, F. Bellingard, L. Bouhier, P. Vaillergues, and T. J. McCabe, "An empirical model for continuous and weighted metric aggregation," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011. [Online]. Available: https://doi.org/10.1109/csmr.2011.20 (Cited on pages 41 and 274.)

[199] H. Mumtaz, P. Singh, and K. Blincoe, "A systematic mapping study on architectural smells detection," *Journal of Systems and Software*, vol. 173, p. 110885, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110885 (Cited on page 33.)

[200] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," *Journal of Network and Computer Applications*, vol. 60, pp. 54–67, 2016. [Online]. Available: https://doi.org/10.1016/j.jnca.2015.11.014 (Cited on pages 90 and 198.)

[201] F. Nawaz, A. Mohsin, and N. K. Janjua, "Service Description Languages in Cloud Computing: State-Of-The-Art and Research Issues," *Service Oriented Computing and Applications*, vol. 13, no. 2, pp. 109–125, 2019. [Online]. Available: https://doi.org/10.1007/s11761-019-00263-z (Cited on pages 50 and 53.)

[202] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly and Associates, 2015. [Online]. Available: https://www.oreilly.com/library/view/building-microservices/9781491950340/ (Cited on page 22.)

[203] P. Nistala, K. V. Nori, and R. Reddy, "Software quality models: A systematic mapping study," in *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/icssp.2019.00025 (Cited on pages 30, 34, and 35.)

[204] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger, "Assessing architecture conformance to coupling-related patterns and practices in microservices," in *Software Architecture*. Springer International Publishing, 2020, pp. 3–20. [Online]. Available: https://doi.org/10.1007/978-3-030-58923-3_1 (Cited on pages 32, 166, 167, 170, 290, and 307.)

[205] ——, "Metrics for assessing architecture conformance to microservice architecture patterns and practices," in *International Conference on Service-Oriented Computing*. Springer, 2020, pp. 580–596. [Online]. Available:

https://doi.org/10.1007/978-3-030-65310-1_42 (Cited on pages 32, 162, 165, 170, 290, 292, and 303.)

[206] E. Ntentos, U. Zdun, K. Plakidas, and S. Geiger, "Semi-automatic feedback for improving architecture conformance to microservice patterns and practices," in *18th ICSA*. IEEE, 2021. [Online]. Available: https://doi.org/10.1109/icsa51549.2021.00012 (Cited on pages 48, 50, 52, 166, 167, 170, 277, 278, 279, 280, 290, and 307.)

[207] E. Ntentos, U. Zdun, G. Falazi, U. Breitenbucher, and F. Leymann, "Assessing architecture conformance to security-related practices in infrastructure as code based deployments," in *2022 IEEE International Conference on Services Computing (SCC)*. IEEE, 2022. [Online]. Available: https://doi.org/10.1109/scc55611.2022.00029 (Cited on pages 163, 166, 167, 275, 280, 288, 289, 290, 291, and 292.)

[208] OASIS, "The Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 2.0," OASIS, 2025. [Online]. Available: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html (Cited on pages 12, 54, 56, 71, and 221.)

[209] Object Management Group (OMG), "BPMN - Business Process Model and Notation 2.0.2," online, 2014. [Online]. Available: https://www.omg.org/spec/BPMN (Cited on page 54.)

[210] S. H. Oh, H. J. La, and S. D. Kim, "A reusability evaluation suite for cloud services," in *2011 IEEE 8th International Conference on e-Business Engineering*. IEEE, 2011. [Online]. Available: https://doi.org/10.1109/icebe.2011.27 (Cited on page 304.)

[211] J. A. Oliveira, M. Vargas, and R. Rodrigues, "Soa reuse: Systematic literature review updating and research directions," in *Proceedings of the XIV Brazilian Symposium on Information Systems*, ser. SBSI'18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3229345.3229419 (Cited on pages 297 and 304.)

[212] P. Oliveira, F. P. Lima, M. T. Valente, and A. Serebrenik, "Rttool: A tool for extracting relative thresholds for source code metrics," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014. [Online]. Available: https://doi.org/10.1109/icsme.2014.112 (Cited on page 41.)

[213] M. Oriol, J. Marco, and X. Franch, "Quality models for web services: A systematic mapping," *Information and Software Technology*, vol. 56, no. 10, pp. 1167–1182, 2014. [Online]. Available: https://doi.org/10.1016/j.infsof.2014.03.012 (Cited on page 31.)

[214] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015. [Online]. Available: https://doi.org/10.1109/mcc.2015.51 (Cited on pages 16 and 22.)

[215] C. Pahl, L. Zhang, and F. Fowley, "A look at cloud architecture interoperability through standards," in *Proceedings of the Fourth International*

Conference on Cloud Computing, Grids, and Virtualization. IARIA, 2013. [Online]. Available: https://hdl.handle.net/10863/32836 (Cited on pages 89 and 91.)

[216] C. Pahl, P. Jamshidi, and O. Zimmermann, "Architectural Principles for Cloud Software," *ACM Transactions on Internet Technology*, vol. 18, no. 2, pp. 1–23, 2018. [Online]. Available: https://doi.org/10.1145/3104028 (Cited on pages 25, 46, and 60.)

[217] A. V. Papadopoulos, A. Iosup, L. Versluis, A. Bauer, N. Herbst, J. V. Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, and P. Tuma, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. [Online]. Available: https://doi.org/10.1109/tse.2019.2927908 (Cited on page 73.)

[218] M. P. Papazoglou and D. Georgakopoulos, "Introduction: Service-oriented computing," *Communications of the ACM*, vol. 46, no. 10, pp. 24–28, 2003. [Online]. Available: https://doi.org/10.1145/944217.944233 (Cited on pages 18 and 22.)

[219] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," *International Journal of Cooperative Information Systems*, vol. 17, no. 2, pp. 223–255, 2008. [Online]. Available: https://doi.org/10.1142/s0218843008001816 (Cited on pages 18 and 19.)

[220] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big" web services," in *Proceeding of the 17th international conference on World Wide Web - WWW '08*. ACM Press, 2008. [Online]. Available: https://doi.org/10.1145/1367497.1367606 (Cited on page 19.)

[221] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1: Reality check and service design," *IEEE Software*, vol. 34, no. 1, pp. 91–98, 2017. [Online]. Available: https://doi.org/10.1109/ms.2017.24 (Cited on page 21.)

[222] ——, "Microservices in practice, part 2: Service integration and sustainability," *IEEE Software*, vol. 34, no. 2, pp. 97–104, 2017. [Online]. Available: https://doi.org/10.1109/ms.2017.56 (Cited on page 21.)

[223] X. Peng, C. Zhang, Z. Zhao, A. Isami, X. Guo, and Y. Cui, "Trace analysis based microservice architecture measurement," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1589–1599. [Online]. Available: https://doi.org/10.1145/3540250.3558951 (Cited on pages 168, 294, 296, 297, 298, 304, and 305.)

[224] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE, 2007. [Online]. Available: https://doi.org/10.1109/qsic.2007.4385516 (Cited on

pages 168, 269, 295, and 296.)

[225] K. J. P. G. Perera and I. Perera, "Thearchitect: A serverless-microservices based high-level architecture generation tool," in *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, 2018, pp. 204–210. [Online]. Available: https://doi.org/10.1109/ICIS.2018.8466390 (Cited on pages 299 and 302.)

[226] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992. [Online]. Available: https://doi.org/10.1145/141874.141884 (Cited on pages 45 and 47.)

[227] Pham Thi Quynh and H. Q. Thang, "Dynamic coupling metrics for service - oriented software," *International Journal of Computer Science and Engineering*, 2009. [Online]. Available: https://doi.org/10.5281/ZENODO.1058263 (Cited on pages 299, 300, 302, and 304.)

[228] P. Plebani, S. Schulte, D. A. Tamburri, and S. Dustdar, "Service-oriented computing: A trajectory for research to 2030," *IEEE Internet Computing*, vol. 28, no. 3, pp. 59–63, 2024. [Online]. Available: https://doi.org/10.1109/mic.2023.3338908 (Cited on pages 20 and 21.)

[229] R. Plösch, H. Gruber, A. Hentschel, C. Körner, G. Pomberger, S. Schiffer, M. Saft, and S. Storck, "The EMISQ method and its tool support-expert-based evaluation of internal software quality," *Innovations Syst Softw Eng*, vol. 4, no. 1, pp. 3–15, 2008. [Online]. Available: https://doi.org/10.1007/s11334-007-0039-7 (Cited on page 85.)

[230] F. Ponce, J. Soldani, H. Astudillo, and A. Brogi, "Smells and refactorings for microservices security: A multivocal literature review," *Journal of Systems and Software*, vol. 192, p. 111393, 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111393 (Cited on pages 33, 85, and 280.)

[231] S. Pulnil and T. Senivongse, "A microservices quality model based on microservices anti-patterns," in *2022 19th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 2022, pp. 1–6. [Online]. Available: https://doi.org/10.1109/jcsse54890.2022.9836297 (Cited on page 31.)

[232] K. Qian, J. Liu, and F. Tsui, "Decoupling metrics for services composition," in *5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse (ICIS-COMSAR'06)*. IEEE, 2006. [Online]. Available: https://doi.org/10.1109/icis-comsar.2006.30 (Cited on pages 165, 170, 303, and 306.)

[233] F. Rademacher, J. Sorgalla, P. Wizenty, S. Sachweh, and A. Zündorf, "Graphical and Textual Model-Driven Microservice Development," in *Microservices, Science and Engineering*. Springer, 2020, pp. 147–179. [Online]. Available: https://doi.org/10.1007/978-3-030-31646-4_7 (Cited on pages 49 and 52.)

[234] F. Rademacher, J. Sorgalla, P. Wizenty, and S. Trebbau, "Towards Holistic Modeling of Microservice Architectures Using LEMMA," in *2nd MDE4SA Workshop*, vol. 2978. CEUR-WS.org, 2021. [Online]. Available: https://ceur-ws.org/Vol-2978/mde4sa-paper2.pdf (Cited on pages 54 and 221.)

[235] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.12.004 (Cited on page 17.)

[236] A. Rainer and S. Beecham, "Supplementary guidelines and assessment scheme for the use of evidence-based software engineering," School of Computer Science, University of Hertfordshire, School of Computer Science, University of Hertfordshire„ techreport CS-TR-469, 2008. [Online]. Available: https://ebse.webspace.durham.ac.uk/wp-content/uploads/sites/49/2022/08/EBSE_supplementary_guidelines_v2.0_DRAFT.pdf (Cited on page 62.)

[237] V. Raj and S. Ravichandra, "Microservices: A perfect soa based solution for enterprise applications compared to web services," in *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, 2018, pp. 1531–1536. [Online]. Available: https://doi.org/10.1109/RTEICT42901.2018.9012140 (Cited on pages 169, 302, and 308.)

[238] V. Raj and R. Sadam, "Evaluation of SOA-based web services and microservices architecture using complexity metrics," *SN Computer Science*, vol. 2, no. 5, 2021. [Online]. Available: https://doi.org/10.1007/s42979-021-00767-6 (Cited on pages 169 and 302.)

[239] M. Ramachandran and V. Chang, "Recommendations and best practices for cloud enterprise security," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE, 2014, pp. 983–988. [Online]. Available: https://doi.org/10.1109/cloudcom.2014.105 (Cited on page 32.)

[240] RedHat, "Understanding cloud-native applications," Online, 2018. [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps (Cited on page 60.)

[241] P. Reznik, J. Dobson, and M. Gienow, *Cloud Native Transformation*. O'Reilly Media, Inc., 2019. [Online]. Available: https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ (Cited on pages 61, 112, 118, 119, 120, 121, 122, 124, 134, 139, 143, 144, 148, 156, and 267.)

[242] C. Richardson, *Microservices Patterns*, 1st ed. Manning Publications Co., 2019. [Online]. Available: https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/ (Cited on pages 32, 61, 113, 114, 116, 117, 118, 120, 121, 122, 123, 138, 145, 146, 147, 150, and 239.)

[243] S. Rizvi, H. Roddy, J. Gualdoni, and I. Myzyri, "Three-step approach to QoS maintenance in cloud computing using a third-party auditor,"

Procedia computer science, vol. 114, pp. 83–92, 2017. [Online]. Available: https://doi.org/10.1016/j.procs.2017.09.014 (Cited on page 292.)

[244] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "REST APIs: A large-scale analysis of compliance with principles and best practices," in *2016 International Conference on Web Engineering*. Springer International Publishing, 2016, pp. 21–39. [Online]. Available: https://doi.org/10.1007/978-3-319-38791-8_2 (Cited on page 32.)

[245] T. d. O. Rosa, A. Goldman, and E. M. Guerra, "How 'micro' are your services?" in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 75–78. [Online]. Available: https://doi.org/10.1109/ICSA-C50368.2020.00023 (Cited on pages 290, 298, 299, and 307.)

[246] L. H. Rosenberg, "Applying and interpreting object oriented metrics," in *Software Technology Conference*, 1998. [Online]. Available: http://www.literateprogramming.com/ooapply.pdf (Cited on page 41.)

[247] A. Rotem-Gal-Oz, *SOA Patterns*. Manning, 2012. [Online]. Available: https://www.oreilly.com/library/view/soa-patterns/9781933988269/ (Cited on page 21.)

[248] D. Rud, A. Schmietendorf, and R. R. Dumke, "Product Metrics for Service-Oriented Infrastructures," in *IWSM/MetriKon*, 2006. (Cited on pages 294, 301, 302, and 308.)

[249] B. Ruecker, *Practical Process Automation*. O'Reilly Media, Inc., 2021. [Online]. Available: https://learning.oreilly.com/library/view/practical-process-automation/9781492061441/ (Cited on pages 61, 116, 117, 121, 122, 128, and 146.)

[250] J. Ryoo, "Architecture: Description languages," in *Encyclopedia of Software Engineering*. CRC Press, 2010, pp. 74–80. [Online]. Available: https://doi.org/10.1081/e-ese-120044218 (Cited on page 47.)

[251] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns," *SICS Software-Intensive Cyber-Physical Systems*, 2019. [Online]. Available: https://doi.org/10.1007/s00450-019-00397-7 (Cited on page 280.)

[252] J. Sandobalin, E. Insfran, and S. Abrahao, "Argon: A model-driven infrastructure provisioning tool," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/models-c.2019.00114 (Cited on pages 54 and 55.)

[253] ——, "On the effectiveness of tools to support infrastructure as code: Model-driven versus code-centric," *IEEE Access*, vol. 8, pp. 17 734–17 761, 2020. [Online]. Available: https://doi.org/10.1109/access.2020.2966597 (Cited on pages 17, 49, 87, and 89.)

[254] G. Schermann, J. Cito, and P. Leitner, "All the services large and micro: Revisiting industrial practice in services computing," in *Service-Oriented Computing – ICSOC 2015 Workshops*. Springer Berlin Heidelberg, 2016, pp. 36–47. [Online]. Available: https://doi.org/10.1007/978-3-662-50539-7_4 (Cited on page 19.)

[255] B. Scholl, T. Swanson, and P. Jausovec, *Cloud Native*. O'Reilly Media, Inc., 2019. [Online]. Available: https://learning.oreilly.com/library/view/cloud-native/9781492053811/ (Cited on pages 61, 107, 108, 109, 110, 111, 114, 115, 116, 120, 121, 122, 123, 125, 127, 129, 130, 131, 132, 133, 134, 135, 138, 140, 141, 142, 143, 145, 146, and 267.)

[256] S. Sehestedt, C.-H. Cheng, and E. Bouwers, "Towards quantitative metrics for architecture models," in *Proceedings of the WICSA 2014 Companion Volume*, ser. WICSA '14. ACM, 2014, pp. 1–4. [Online]. Available: https://doi.org/10.1145/2578128.2578226 (Cited on page 47.)

[257] M. Shaw and P. Clements, "The golden age of software architecture," *IEEE Software*, vol. 23, no. 2, pp. 31–39, 2006. [Online]. Available: https://doi.org/10.1109/ms.2006.58 (Cited on page 51.)

[258] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218–238, 2014. [Online]. Available: https://doi.org/10.1016/j.ins.2014.04.054 (Cited on page 18.)

[259] B. Shim, S. Choue, S. Kim, and S. Park, "A design quality model for service-oriented architecture," in *2008 15th Asia-Pacific Software Engineering Conference*. IEEE, 2008. [Online]. Available: https://doi.org/10.1109/apsec.2008.32 (Cited on pages 171, 288, 289, 295, 296, 301, 302, 306, and 308.)

[260] J. Siebert, L. Joeckel, J. Heidrich, A. Trendowicz, K. Nakamichi, K. Ohashi, I. Namba, R. Yamamoto, and M. Aoyama, "Construction of a quality model for machine learning systems," *Software Quality Journal*, vol. 30, no. 2, pp. 307–335, 2021. [Online]. Available: https://doi.org/10.1007/s11219-021-09557-y (Cited on pages 31, 34, and 43.)

[261] A. Sill, "Cloud native standards and call for community participation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 56–61, 2017. [Online]. Available: https://doi.org/10.1109/mcc.2017.4250925 (Cited on page 119.)

[262] S. Silva, A. Tuyishime, T. Santilli, P. Pelliccione, and L. Iovino, "Quality metrics in software architecture," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. IEEE, 2023. [Online]. Available: https://doi.org/10.1109/icsa56044.2023.00014 (Cited on page 171.)

[263] S. Singh and I. Chana, "Q-aware: Quality of service based cloud resource provisioning," *Computers & Electrical Engineering*, vol. 47, pp. 138–160, 2015. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2015.02.003 (Cited on page 291.)

[264] Y. Singh, A. Kaur, and R. Malhotra, "Empirical validation of object-oriented metrics for predicting fault proneness models," *Software Quality*

Journal, vol. 18, no. 1, pp. 3–35, 2009. [Online]. Available: https://doi.org/10.1007/s11219-009-9079-6 (Cited on pages 44 and 194.)

[265] S. Slimani, T. Hamrouni, and F. B. Charrada, "Service-oriented replication strategies for improving quality-of-service in cloud computing: a survey," *Cluster Computing*, vol. 24, no. 1, pp. 361–392, 2020. [Online]. Available: https://doi.org/10.1007/s10586-020-03108-z (Cited on page 193.)

[266] R. C. Soares, R. Capilla, V. dos Santos, and E. Y. Nakagawa, "Trends in continuous evaluation of software architectures," *Computing*, vol. 105, no. 9, pp. 1957–1980, 2023. [Online]. Available: https://doi.org/10.1007/s00607-023-01161-1 (Cited on page 47.)

[267] J. Soldani, G. Muntoni, D. Neri, and A. Brogi, "The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures," *Software: Practice and Experience*, 2021. [Online]. Available: https://doi.org/10.1002/spe.2974 (Cited on pages 277, 278, and 279.)

[268] I. Sommerville, *Software Engineering*, 9th ed. Addison-Wesley, 2011. [Online]. Available: https://engineering.futureuniversity.com/BOOKSFORIT/Software-Engineering-9th-Edition-by-Ian-Sommerville.pdf (Cited on pages 45, 46, 47, 48, 49, and 51.)

[269] T. Sousa, H. S. Ferreira, and F. F. Correia, "A survey on the adoption of patterns for engineering software for the cloud," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021. [Online]. Available: https://doi.org/10.1109/tse.2021.3052177 (Cited on page 280.)

[270] M. Stocker, O. Zimmermann, D. Lübke, U. Zdun, and C. Pautasso, "Interface Quality Patterns – Communicating and Improving theQuality of Microservices APIs," in *Proceedings of the 23rd European Conference on Pattern Languages of Programs - EuroPLoP '18*. ACM Press, 2018. [Online]. Available: https://doi.org/10.1145/3282308.3282319 (Cited on pages 206, 207, and 219.)

[271] M. Straesser, J. Mathiasch, A. Bauer, and S. Kounev, "A systematic approach for benchmarking of container orchestration frameworks," in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 187–198. [Online]. Available: https://doi.org/10.1145/3578244.3583726 (Cited on page 171.)

[272] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019. [Online]. Available: https://doi.org/10.1109/access.2019.2911732 (Cited on page 86.)

[273] S. Sundareswaran, A. Squicciarini, and D. Lin, "Ensuring distributed accountability for data sharing in the cloud," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 556–568, 2012. [Online]. Available: https://doi.org/10.1109/tdsc.2012.26 (Cited on page 86.)

[274] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Software & Systems Modeling*, vol. 15, no. 1, pp. 31–69, 2013. [Online]. Available: https://doi.org/10.1007/s10270-013-0394-9 (Cited on page 49.)

[275] A. Taherkordi, F. Zahid, Y. Verginadis, and G. Horn, "Future cloud systems design: Challenges and research directions," *IEEE Access*, vol. 6, pp. 74 120–74 150, 2018. [Online]. Available: https://doi.org/10.1109/access.2018.2883149 (Cited on page 17.)

[276] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018. [Online]. Available: https://doi.org/10.1109/ms.2018.2141031 (Cited on page 33.)

[277] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SciTePress, 2018, pp. 221–232. [Online]. Available: https://doi.org/10.5220/0006798302210232 (Cited on page 32.)

[278] ——, "Microservices anti-patterns: A taxonomy," in *Microservices*, , Ed. Springer International Publishing, 2019, pp. 111–128. [Online]. Available: https://doi.org/10.1007/978-3-030-31646-4_5 (Cited on page 280.)

[279] V. Talwar, Q. Wu, C. Pu, W. Yan, G. Jung, and D. Milojicic, "Comparison of approaches to service deployment," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE, 2005. [Online]. Available: https://doi.org/10.1109/icdcs.2005.18 (Cited on page 309.)

[280] U. Tiwari and S. Kumar, "In-out interaction complexity metrics for component-based software," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 5, pp. 1–4, 2014. [Online]. Available: https://doi.org/10.1145/2659118.2659135 (Cited on pages 299, 300, and 305.)

[281] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert, "Self-managing cloud-native applications: Design, implementation, and experience," *Future Generation Computer Systems*, vol. 72, pp. 165–179, 2017. [Online]. Available: https://doi.org/10.1016/j.future.2016.09.002 (Cited on pages 25 and 60.)

[282] K. A. Torkura, M. I. Sukmana, and C. Meinel, "Integrating continuous security assessments in microservices and cloud native applications," in *10th UCC*. ACM, 2017. [Online]. Available: https://doi.org/10.1145/3147213.3147229 (Cited on page 77.)

[283] M. Ulan, W. Löwe, M. Ericsson, and A. Wingkvist, "Towards meaningful software metrics aggregation," in *Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop*, 2019. [Online]. Available: https://ceur-ws.org/Vol-2605/12.pdf (Cited on page 40.)

[284] ——, "Copula-based software metrics aggregation," *Software Quality Journal*, vol. 29, no. 4, pp. 863–899, 2021. [Online]. Available: https://doi.org/10.1007/s11219-021-09568-9 (Cited on page 41.)

[285] J. A. Valdivia, X. Limon, and K. Cortes-Verdin, "Quality attributes in patterns related to microservice architecture: a systematic literature review," in *2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, 2019, pp. 181–190. [Online]. Available: https://doi.org/10.1109/CONISOFT.2019.00034 (Cited on page 32.)

[286] G. Vale, F. F. Correia, E. M. Guerra, T. d. O. Rosa, J. Fritzsch, and J. Bogner, "Designing microservice systems using patterns: An empirical study on quality trade-offs," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE Computer Society, 2022. [Online]. Available: https://doi.org/10.1109/ICSA-C54293.2022.00020 (Cited on pages 274 and 280.)

[287] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003. [Online]. Available: https://doi.org/10.1023/a:1022883727209 (Cited on page 32.)

[288] E. van der Bent, J. Hage, J. Visser, and G. Gousios, "How good is your puppet? an empirically defined and validated quality model for puppet," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018. [Online]. Available: https://doi.org/10.1109/saner.2018.8330206 (Cited on pages 40 and 41.)

[289] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing*, ser. WoSC '17. New York, NY, USA: ACM, 2017, pp. 1–4. [Online]. Available: https://doi.org/10.1145/3154847.3154848 (Cited on page 17.)

[290] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Generation Computer Systems*, vol. 79, pp. 849–861, 2018. [Online]. Available: https://doi.org/10.1016/j.future.2017.09.020 (Cited on page 17.)

[291] H. Venkitachalam, K. A. Powale, C. Granrath, and J. Richenhagen, "Automated continuous evaluation of autosar software architecture for complex powertrain systems," in *INFORMATIK*. Gesellschaft für Informatik, Bonn, 2017. [Online]. Available: https://doi.org/10.18420/IN2017_156 (Cited on pages 85 and 171.)

[292] VMwareTanzu(Pivotal), "Cloud-Native Applications: Ship Faster, Reduce Risk, Grow Your Business," Online, 2020. [Online]. Available: https://tanzu.vmware.com/de/cloud-native (Cited on page 60.)

[293] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 223–236. [Online]. Available: https://doi.org/10.1109/mascots.2018.00030

(Cited on pages 72, 74, 76, 206, 218, and 239.)

[294] S. Wagner, K. Lochmann, S. Winter, F. Deissenboeck, E. Juergens, M. Herrmannsdoerfer, and L. Heinemann, "The quamoco quality meta-model," Technische Universität München, Institut für Informatik, techreport TUM-I128, 2012. [Online]. Available: https://mediatum.ub.tum.de/doc/1110600/file.pdf (Cited on pages 11, 34, 35, 36, 37, 61, 151, 195, 269, and 273.)

[295] S. Wagner, A. Goeb, L. Heinemann, M. Kläs, C. Lampasona, K. Lochmann, A. Mayr, R. Plösch, A. Seidl, J. Streit *et al.*, "Operationalised product quality models and assessment: The Quamoco approach," *Information and Software Technology*, vol. 62, pp. 101–123, 2015. [Online]. Available: https://doi.org/10.1016/j.infsof.2015.02.009 (Cited on pages 8, 34, 35, and 36.)

[296] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and challenges of microservices: an exploratory study," *Empirical Software Engineering*, vol. 26, no. 4, 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09910-y (Cited on pages 21 and 23.)

[297] A. Wasserman, "Toward a discipline of software engineering," *IEEE Software*, vol. 13, no. 6, pp. 23–31, 1996. [Online]. Available: https://doi.org/10.1109/52.542291 (Cited on pages 46 and 221.)

[298] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Pearson, 2005. [Online]. Available: https://www.oreilly.com/library/view/web-services-platform/0131488740/ (Cited on pages 19 and 20.)

[299] P. Wegner, "Research Paradigms in Computer Science," in *Proceedings of the 2Nd International Conference on Software Engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 322–330. [Online]. Available: https://dl.acm.org/doi/10.5555/800253.807694 (Cited on page 59.)

[300] H. Wei, N. Madhavji, and J. Steinbacher, "A map of cloud-native practices and tools toachieve desirable system qualities," in *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. IEEE, 2025. [Online]. Available: https://doi.org/10.1109/ICSA65012.2025.00030 (Cited on page 27.)

[301] Y. Wei and M. B. Blake, "Service-oriented computing and cloud computing: Challenges and opportunities," *IEEE Internet Computing*, vol. 14, no. 6, pp. 72–75, 2010. [Online]. Available: https://doi.org/10.1109/mic.2010.147 (Cited on pages 15 and 20.)

[302] J. Weinman, "The Economics of Computing Workload Aggregation: Capacity, Utilization, and Cost Implications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 6–11, 2017. [Online]. Available: https://doi.org/10.1109/mcc.2017.4250936 (Cited on page 16.)

[303] D. Weyns, I. Gerostathopoulos, N. Abbas, J. Andersson, S. Biffl, P. Brada, T. Bures, A. Di Salle, M. Galster, P. Lago, G. Lewis, M. Litoiu, A. Musil, J. Musil, P. Patros, and P. Pelliccione, "Self-adaptation in industry: A survey," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 18, no. 2, pp. 1–44, 2023. [Online]. Available: https://doi.org/10.1145/3589227 (Cited on page 49.)

[304] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014. [Online]. Available: https://doi.org/10.1109/ms.2013.65 (Cited on page 49.)

[305] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov, "The EDMM modeling and transformation system," in *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 294–298. [Online]. Available: https://doi.org/10.1007/978-3-030-45989-5_26 (Cited on page 54.)

[306] M. Wurster, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, and J. Soldani, "Technology-agnostic declarative deployment automation of cloud applications," in *Service-Oriented and Cloud Computing*. Springer, 2020, pp. 97–112. [Online]. Available: https://doi.org/10.1007/978-3-030-44769-4_8 (Cited on page 50.)

[307] M. Wurster, U. Breitenbücher, A. Brogi, F. Leymann, and J. Soldani, "Cloud-native Deploy-ability: An Analysis of Required Features of Deployment Technologies to Deploy Arbitrary Cloud-native Applications," in *10th CLOSER*. Scitepress, 2020. [Online]. Available: https://doi.org/10.5220/0009571001710180 (Cited on page 60.)

[308] A. Yamashita, "Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015. [Online]. Available: https://doi.org/10.1109/icsm.2015.7332493 (Cited on page 41.)

[309] M. Yan, X. Xia, X. Zhang, L. Xu, and D. Yang, "A systematic mapping study of quality assessment models for software products," in *International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2017. [Online]. Available: https://doi.org/10.1109/sate.2017.16 (Cited on pages 30, 35, 42, and 43.)

[310] B. Yang, A. Sailer, and A. Mohindra, "Survey and evaluation of blue-green deployment techniques in cloud native environments," in *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 69–81. [Online]. Available: https://doi.org/10.1007/978-3-030-45989-5_6 (Cited on page 17.)

[311] W. D. Yu, R. B. Radhakrishna, S. Pingali, and V. Kolluri, "Modeling the measurements of QoS requirements in web service systems," *SIMULATION*, vol. 83, no. 1, pp. 75–91, 2007. [Online]. Available: https://doi.org/10.1177/0037549707079228 (Cited on page 44.)

[312] V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller, "Facing the unplanned migration of serverless applications," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing - UCC'19*. ACM Press, 2019. [Online]. Available: https://doi.org/10.1145/3344341.3368813 (Cited on page 89.)

[313] V. Yussupov, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, and J. Soldani, "Serverless or serverful? a pattern-based approach for exploring hosting alternatives," in *Service-Oriented Computing*. Springer International Publishing, 2022, pp. 45–67. [Online]. Available: https://doi.org/10.1007/978-3-031-18304-1_3 (Cited on pages 32, 170, and 280.)

[314] U. Zdun, E. Navarro, and F. Leymann, "Ensuring and assessing architecture conformance to microservice decomposition patterns," in *Service-Oriented Computing*. Springer International Publishing, 2017, pp. 411–429. [Online]. Available: https://doi.org/10.1007/978-3-319-69035-3_29 (Cited on pages 32, 303, and 304.)

[315] U. Zdun, P.-J. Queval, G. Simhandl, R. Scandariato, S. Chakravarty, M. Jelic, and A. Jovanovic, "Microservice security metrics for secure communication, identity management, and observability," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–34, 2023. [Online]. Available: https://doi.org/10.1145/3532183 (Cited on pages 62, 167, 280, 288, 289, and 290.)

[316] U. Zdun, P.-J. Queval, G. Simhandl, R. Scandariato, S. Chakravarty, M. Jelić, and A. Jovanović, "Detection strategies for microservice security tactics," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–17, 2023. [Online]. Available: https://doi.org/10.1109/TDSC.2023.3276487 (Cited on pages 167, 288, 298, and 307.)

[317] Q. Zhang and L. Xinke, "Complexity metrics for service-oriented systems," in *2009 Second International Symposium on Knowledge Acquisition and Modeling*. IEEE, 2009. [Online]. Available: https://doi.org/10.1109/kam.2009.90 (Cited on pages 169, 293, 302, and 308.)

[318] X. Zheng, P. Martin, K. Brohman, and L. D. Xu, "CLOUDQUAL: A quality model for cloud services," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1527–1536, 2014. [Online]. Available: https://doi.org/10.1109/tii.2014.2306329 (Cited on page 85.)

[319] Z. Zhou, Q. Zhi, S. Morisaki, and S. Yamamoto, "A systematic literature review on enterprise architecture visualization methodologies," *IEEE Access*, vol. 8, pp. 96 404–96 427, 2020. [Online]. Available: https://doi.org/10.1109/access.2020.2995850 (Cited on page 51.)

[320] O. Zimmermann, "Metrics for architectural synthesis and evaluation – requirements and compilation by viewpoint. an industrial experience report," in *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*. IEEE, 2015. [Online]. Available: https://doi.org/10.1109/sam.2015.9 (Cited on pages 10, 40, 169, 171, 295, 298, 299, 305,

and 307.)

[321] ——, "Microservices tenets," *Computer Science - Research and Development*, vol. 32, no. 3-4, pp. 301–310, 2016. [Online]. Available: https://doi.org/10.1007/s00450-016-0337-0 (Cited on page 22.)

[322] ——, "Architectural refactoring for the cloud: a decision-centric view on cloud migration," *Computing*, vol. 99, no. 2, pp. 129–145, 2017. [Online]. Available: https://doi.org/10.1007/s00607-016-0520-y (Cited on page 33.)

[323] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*, ser. Addison-Wesley Signature Series (Vernon). Addison-Wesley Professional, 2022. [Online]. Available: https://www.oreilly.com/library/view/patterns-for-api/9780137670093/ (Cited on pages 32 and 206.)

Glossary of Product Factors

Separation by gateways see 114. 113, 154, 183, 199, 211, 217, 272, 307
Service independence see 125. 125, 128, 153, 203, 204, 206, 207, 210, 212
Service replication see 131. 130–132, 152, 157, 171, 186, 188–190, 198, 201, 206, 213, 214, 217, 262, 270, 272, 309
Service-orientation see 111. 111, 203, 204, 206–208, 211
Sharded data store replication see 133. 132, 142, 154, 183, 201, 214, 219, 262, 272
Sparsity see 130. 130, 171, 200, 213, 262, 272, 307, 308
Specialized stateful services see 115. 115, 155, 183, 199, 210, 211, 214, 215, 260, 262, 272
Standardization see 119. 119, 184, 199, 211, 272
Standardized self-contained deployment unit see 139. 139, 184, 201, 206–208, 215, 216, 231, 272

Usage of existing solutions for non-core capabilities see 118. 119, 155, 157, 184, 199, 207, 213, 262, 272, 286
Use infrastructure as code see 125. 124, 152, 157, 158, 185, 200, 206, 210–212, 215, 218, 272, 309

Vertical data replication see 132. 132, 152, 163, 185, 186, 191, 192, 201, 206, 212, 214, 217, 262, 272, 286

University of Bamberg Press

Cloud-native is used as a term in industry and academia for describing how software should be built in order to run in cloud environments properly. The usage and popularity of the term leads to the desire to build web-based enterprise applications in a cloud-native way to benefit from the advantages of modern cloud environments. However, due to the complexity of modern cloud environments, cloud-native spans a broad range of aspects from the design of an application's structure and communication to the selected deployment technologies and infrastructure. Thus, understanding thoroughly what a cloud-native application design means and deciding how a specific application can benefit from cloud-native patterns and characteristics, is challenging for software architects and developers.

This work presents the formulation and validation of a hierarchical quality model for cloud-native software architectures focusing on design time aspects. Cloud-native characteristics are formulated as factors for the quality model and linked to quality aspects through impact relationships. An impact describes how a quality aspect is influenced, if a factor is present in a system. Together with the quality model, a modeling approach for cloud-native software architectures and the Clounaq tool was developed. The Clounaq tool offers software architects a model-based method for designing and evaluating software architectures based on the quality model. For the evaluation approach, architectural measures quantify the factors of the quality model. Software architects and developers can thus better understand consequences of cloud-native characteristics and evaluate for new or existing applications, how the application architecture design can benefit from cloud-native characteristics.

The resulting quality model captures the topic of cloud-native software architectures from a quality-focused point of view. It explains how applications can be built in a cloud-native way to achieve certain benefits. Through the characterization in the form of a quality model, software architects can evaluate and prioritize which cloud-native characteristics are relevant and applicable to a certain application at hand. The presented modeling approach demonstrates how core aspects of cloud-native software architectures can be represented in a formal textual and a graphical notation. The Clounaq tool includes implemented calculations for the catalog of architectural measures aligned with the modeling approach and can thus be used by practitioners and researchers for future work on cloud-native software architectures.

ISBN: 978-3-98989-117-3

9 783989 891173

www.uni-bamberg.de/ubp