Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Evaluating the Effectiveness of Proposed Service-based Maintainability Metrics for Microservices

Marcel Szidlovszky

Course of Study:	Softwareengineering
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Justus Bogner
Commenced:	May 8, 2019
Completed:	November 8, 2019

Abstract

The maintenance of software systems is a critical process in development. Although the cost of maintenance is estimated to be 60-80% of the software development life cycle, it is still common in development to neglect the maintainability and focus more on other components. This can be felt by the developers, who have to maintain the software for preventive, corrective, adaptive or perfective reasons.

In order to be able to correctly plan complex maintenance, i.e. to estimate the cost and time involved, one must first determine how maintainable a software is. For this, static code and runtime data analyses can be used, which refer to software metrics. For conventional monolithic architectures, there is already a lot of literature that evaluates and proves the effectiveness of maintainability metrics. But since there has been a change from monolithic architectures to microservice architectures in recent years, especially for large enterprise systems and **S**oftware **A**s **A** **S**ervice (SAAS), we have to adapt. World-renowned web services like Netflix and Amazon show us how successful this relatively new type of architecture can be. These systems follow the SAAS model and therefore have a constant maintenance process, where an effective maintainability measurement would be very advantageous.

With this paper we want to fill the gap in literature and evaluate whether proposed service-based maintainability metrics can effectively be used for microservices. We will specifically investigate service-based metrics that have not yet been evaluated. In order to achieve this goal, we first created an up-to-date list of service-based maintainability metrics and compiled a set of example systems to evaluate the measurability and effectiveness of metrics. The effectiveness was determined by the measurability and the significance of the examined metrics. The significance was evaluated using the following evaluation methods:

On the one hand, a correlation between metrics and experts was evaluated, based on the characteristics: size, coupling and cohesion. These were based on the example systems and examined for significant agreements. The second part of the evaluation includes a scenario-based analysis in which the relationship between metrics and theoretical reactions of architectures was qualitatively evaluated. It was analyzed whether they could predict the estimated effort and the differences, of the reactions, for the different architectures.

The result of the evaluations shows that both the coupling metrics and the size metrics can be very effective in determining maintainability. However, no statement could be made about the cohesion metric examined, as it turned out that the example systems used for the evaluation were possibly unsuitable.

Kurzfassung

Die Wartung von Software systemen ist ein kritischer Prozess in der Entwicklung. Obwohl die Kosten der Wartung auf 60-80% des Softwareentwicklungslebenszyklus geschätzt wird, kommt es trotzdem häufig vor, dass in der Entwicklung die Wartbarkeit an letzter stelle gesetzt und man den Fokus mehr auf andere Komponenten legt. Dies bekommen, dann die Entwickler, die aus präventiven, korrektiven, adaptiven oder perfektiven Gründen die Software warten müssen, zu spüren.

Um eine aufwändige Wartung richtig planen zu können, das heißt den Kosten- und Zeitaufwand zu schätzen, muss man als erstes bestimmen wie wartbar eine Software ist. Dafür werden vorallem Statische-Code und Runtime-Daten Analysen verwendet, welche sich unter anderem auch auf Software Metriken beziehen. Für die herkömmlichen Monolithische Architekturen gibt es bereits eine Menge Literatur, die die Wirksamkeit von Wartbarkeitsmetriken bewertet und nachweist. Da es aber in den letzten jahren, vorallem bei großen Enterprise Systemen und **S**oftware **A**s **A** **S**ervice (SAAS) einen Wechsel von Monolithischen Architekturen zu Microservice Architekturen gab, müssen wir uns anpassen. Weltbekannte Webservices wie Netflix und Amazon zeigen uns dabei wie erfolgreich dieser relativ neuer Architekturtyp sein kann. Diese Systeme folgen dem SAAS Modell und haben somit einen konstanten Wartungsprozess, wobei eine effektive Wartbarkeitsmessung, sehr von Vorteil wäre.

Mit diesem Paper wollen wir die Lücke in der Literatur füllen und evaluieren ob Service-basierte Wartbarkeitsmetriken effektiv für Microservices benutzt werden können. Spezifisch sollen bisher noch nicht evaluierte Service-basierten Metriken untersucht werden. Um dieses Ziel zu erreichen wurde als erstes eine aktuelle Liste von vorgestellten Service-basierten Wartbarkeitsmetriken erstellt und eine Menge von Beispielsystemen zusammengestellt, anhand deren wir die Messbarkeit und Effektivität evaluieren können. Die Effektivität wurde dabei über die Messbarkeit und der Aussagekraft der untersuchten Metriken bestimmt. Die Aussagekraft wurde mithilfe folgender Evaluationsmethoden evaluiert:

Zum einen wurde eine Korrelation zwischen Metriken und Experten bewertungen, basierend auf den Wartbarkeitseigenschaften: Größe, Kopplung und Kohäsion, für Beispielsysteme aufgesellt und nach signifikanten Übereinstimmungen untersucht. Der zweite Teil der Evaluation beinhaltet eine Szenario-basierte Analyse, in der qualitativ Evaluiert wurde, wie Metriken im Verhältnis zu theoretischen Reaktionen von Architekturen stehen. Dabei wurde analysiert ob sie den geschätzten Aufwand und die Verschiedenheiten, der Reaktionen von den verschiedenen Architekturen, vorhersagen konnten.

Das Ergebnis der Evaluationen zeigt, dass sowohl die untersuchten Kopplungsmetriken, als auch Größenmetriken, sehr effektiv sein können um die Wartbarkeit zu bestimmen. Allerdings konnte keine Aussage über die untersuchte Kohäsionsmetrik getroffen werden, da sich herausgestellt hat, dass die, für die Evaluation benutzten Beispielsysteme, möglicherweiße ungeeignet waren.

Contents

List of Figures

List of Tables

1 Introduction

This chapter introduces the motivation, goals and structure of this thesis. It helps to understand why this evaluation is needed and how it will be carried out.

1.1 Motivation

Maintenance is one of the most expensive and complex software activity there is (see Section 2.1). Since in practice large software products and systems are reused again and again and more software is offered as a service, it becomes more and more important to be able to plan maintenance in detail. In order to plan maintainability, we have to be able to make statements about the quality and maintainability of a software architecture.

The most common possibilities to evaluate the maintainability of a system are expert ratings, static code and runtime data analyses. Expert ratings however, have several disadvantages:
On the one hand, a larger group of experts must be interviewed in order to find a meaningful mean value. It takes a lot of time to assemble and survey such a group and on the other hand it is not unusual that strong contradictions and uneven scores arise.

On the flip side, static code and runtime data analyses are more homogeneous and usually more efficient to measure, as the measurement can mostly be carried out automatically. These two methods are largely based on **metrics**. Metrics can usually be measured automatically and depending on how precisely the experts evaluate the systems, metrics are also more fine-grained and more vulnerability-oriented then expert assessments. But the question is whether they are also effective to evaluate maintainability.

For monolithic architecture there is already a lot of literature that evaluates maintainability metrics and prove their effectiveness [DJ03; KNR15; MKPS00]. For microservice architectures, however, there are not so many detailed evaluations yet. In literature, it is generally said that the complexity of single services and the overall maintainability for microservices is better and based on other properties than for monolithic architectures [DGL+17; New15]. Because of these changes other metrics are needed for this kind of architecture. With literatures like [QX09] and [PRFT07] and reviews like [BWZ17],we already have a number of metrics proposed. But literature is still missing evaluations of these metrics.

1.2 Goals

The main goal of this thesis is to investigate whether service-based metrics are effective for determining the maintainability of microservices. In order to achieve this goal, different evaluation methods will be used and analysed. However, before we reach the main goal, we have to solve the following sub goals and intermediate problems:

Up-to-date service-based maintainability metrics list To evaluate different proposed metrics, first we have to create an up-to-date list of the majority of proposed service-based maintainability metrics. For this a detailed literature search is performed, based on the literature review from Bogner et al. [BWZ17], that already contains a detailed list of maintainability metrics for microservices, which will be extended and updated if necessary. In addition, we can narrow down the list, since we only examine service-based metrics.

Measurement techniques Since a wide variety of papers, only introduce the metrics and illustrate calculations, but do not show any way to measure we have to investigate into different possibilities to measure metrics. As we only measure microservices this becomes a bigger challenge, because this architecture scheme is polyglot and most automatically measurement methods are language based [New15]. The possibility that we need different measurement technologies for the same metric is very large.

Selection of example systems To evaluate the metrics on different systems we need to make a selection of example systems. It is important that these systems do not differentiate too much so that we can find scenarios that are applicable for a number of systems. The selection of the systems is also limited by the expert evaluations, as these are already available and were not created in the process of this paper.

Correlation to expert assessments Since expert ratings are a meaningful assessment of the properties and maintainability of software architectures, the first thing we want to find out is whether there is a correlation between the ratings and measured metrics in the same design properties of maintainability. With this correlation we find out whether metrics can correctly reflect the maintainability of an architecture.

Evaluating the maintainability prediction of metrics in a scenario-based analysis The second part of this evaluation consists of a qualitative evaluation to determine whether service-based metrics can successfully predict the maintainability of an architecture. For this evaluation we use realistic maintainability scenarios, which we simulate on different architectures and compare the results with metrics.

1.3 Structure

The paper starts with a foundation explanation, which should help to understand the topics, that are basis for this evaluation. This is followed by a brief description of similar works dealing among other things with the evaluation of metrics.

Afterwards the design and preparations for the evaluation are discussed. This includes gathering proposed metrics, selecting metrics to evaluate and setup expert ratings. Also a list of systems, on which we evaluate the metrics, was gathered. To evaluate the effectiveness of service-based metrics following methods are used:

Firstly a quantitative evaluation is conducted, in which the correlation between metrics and expert ratings are analysed. I will use expert ratings from existing studies. However, these studies had to be revised with a second iteration to make them more homogeneous. To find correlations, graphical and static analysis techniques, such as Pearson's correlation efficiency, will be used.

The second part of the evaluation includes a qualitative analysis that will be conducted with the help of scenario-based analysis. This is carried out by creating different realistic maintainability scenarios and theoretically carrying them out on different architectures. From this an estimation is made, which is compared with the metrics.

At the end we discuss the results and implications of these two evaluations and draw a conclusion.

2 Foundations

In the following, the basics are explained on which this evaluation is based. They should help to understand why maintainability is such an important process in software development and what microservices are and how they stand out from other architectures.

2.1 Maintainability

With the development of more and more new systems and an ever-increasing amount of source code, it is important to keep the maintainability on a high level. Since the focus during development is rarely on good maintainability, but on other quality attributes and the rapid completion of the project, a large part of the development costs is required for maintenance [BR00].

In ISO/IEC 14764 software maintenance, is defined as changes (activities) made to the software to 'provide cost-effective' support [Sta06]. These changes include pre and post delivery stage, but it should be noted that changes made before delivery are only part of the maintenance if they specifically include planning for post-delivery operations, supportability, and logistics determination. As a result, maintainability is defined as the capability of software to be changed for maintenance modifications [Sta06].

In the following we will discuss the reasons for software maintenance, cost estimation and evaluation of maintainability.

2.1.1 Maintainability Reasons

ISO/IEC 14764 and other maintainability literature, usually give the same four main reasons for software maintenance[Sta06]:

Preventive Reasons This category normally does not add any functionality. Instead, possible errors and failures are prevented, which are found by analysing usage data. This category also includes changes that are made to improve the resilience of a software. Note, that as soon as an error or failure occurs, maintenance falls under the corrective category.

Corrective Reasons Bugs can occur for various reasons. There may be a change by hardware or software, or they simply haven't been found yet. Nevertheless, these bugs have to be fixed in any case. Corrective maintenance is mostly preventable and the costs for it can usually be kept to a minimum.

Adaptive Reasons Software must constantly evolve and since it is becoming more and more common to offer software as a service (service here is not meant as a service from a microservice architecture, but as a delivery model which gets centrally hosted and in most cases is licensed on a subscription basis), it is important to keep it up to date and adapt it to the requirements. Adaptive reasons come from changes in requirements, hardware and other technologies used.

Perfective Reasons This category includes all fine-tunings performed on software. This can be changes to the user interface as well as improvements to the user experience or performance.

Legal and Business Reasons These reasons are not one of the four most important and biggest ones, but they should not be overlooked. Legal and Business reasons can include changes to the license or terms of use.

2.1.2 Cost of Maintenance

Often forgotten and underestimated maintenance in the development process makes it the most expensive process in software engineering. Various studies and well-known software engineers indicate that maintenance accounts for approximately *50-70%* of the **S**oftware **D**evelopment **L**ife **C**ycle (SDLC) phases and total effort of a software life cycle. Figure 2.1 shows Boehm's interpretation of the maintenance cost, compared to the other *SDLC* phases [Pig96; Sch87].

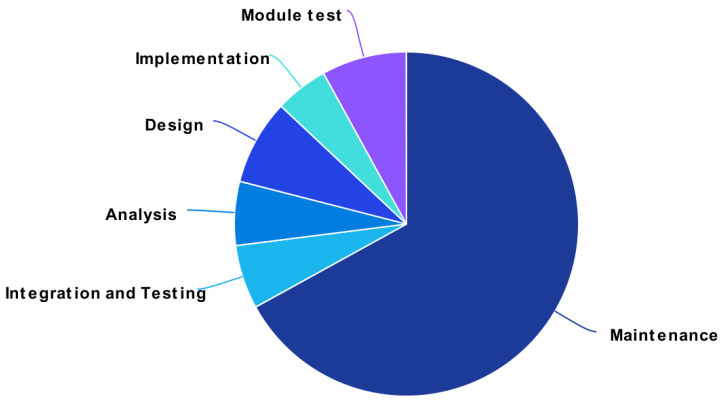

Figure 2.1: Cost allocation by Boehm

The high costs and the big effort are the reasons why we have to plan software maintenance properly. To plan it correctly we have to evaluate the maintainability of a system so that the time and cost can be estimated correctly:

2.1.3 Evaluating Maintainability

One method to evaluate the maintainability that has already been mentioned are expert assessments. For these assessments, different developers who are experts in this area are asked about the System Under Test (SUT). However, this procedure has several disadvantages:

On the one hand, it can be very time-consuming to find a set of experts who investigate the SUT objectively and effectively. Additionally, the respective interviews and investigations of the SUT also increase the time expenditure.

Experts	Granularity	Complexity	Coupling	Cohesion	Overall Maintainability
1	4*	5*	4*	5*	5*
2	8	10	7	8	8
3	6*	10	5	8	9
4	10	7	1*	8	5*
5	3*	3*	5	3*	4*
6	9	10	8	10	9
7	9	9	10*	8	10
8	10	8	10*	8	10
9	10	10	10*	10	7
10	4*	6*	2*	6	5*
11	10	10	10*	6	10

Table 2.1: Expert ratings for *Piggy Metrics* from the first iteration

Example assessments from table 2.1 show another disadvantage of expert ratings: a potential heterogeneity of ratings. All values, marked with *, have a difference of at least 3 to the respective median, which are almost 50% of the assessments.
So expert assessments can be very time consuming and partly not significant.

Other methods to evaluate maintainability could be static code and runtime data analysis. Static code analysis is a white-box process to evaluate the source code. Runtime data analysis measures various data sets that are generated during the runtime of a software. Both analyses work largely on the basis of metrics.

Maintainability Metrics

The earliest approach of metrics has been used to evaluate the software quality of procedural programming paradigm. As architectures evolved from procedural to object-oriented, to service-based paradigms, metrics had to evolve as well. Where in the beginning McCabe's Cyclomatic Complexity (CC)(represents the number of independent control flows through a program) [McC76], could be used for the complexity of procedural paradigms, it was generally claimed that it was no longer applicable for object-orientation [OW14]. This led to the new and further development of metrics for object-oriented paradigm. We can actually see the pioneers of metrics for service-oriented architectures in a number of object-oriented metrics. For example *'Weighted Service Interface Count'* originates in the object-oriented metric *'Weighted Methods Per Class'* and *'Total Response for Service'* is based on the idea of *'Response for Class'* [HCA08; PRFT07].

Since maintainability is a system internal attribute that cannot be measured directly, metrics must be collected for other attributes that affect maintainability. In contrast to experts, metrics are often more efficient to collect, because they are largely measured automatically, in addition their measurement is constant for each system, so the result is more homogeneous. Most proposed maintainability metrics are categorized in the following 4 main design properties:

Coupling Coupling describes the connections between software modules. Intuitively, high coupling clearly leads to a worse overview and thus to increased complexity. It follows that poor (high) coupling also leads to poor maintainability. With microservices, coupling describes the connections between services.

Cohesion Metrics that deal with the grouping of the code fall under the category *cohesion*. This category is the only one that has a positive influence on maintainability, i.e. the greater the cohesion, the better for maintainability. This is also why the principle of loose coupling and high cohesion is often referred to.[New15]

Complexity Complexity is not a distinct category in itself. In some papers, it is seen as the combination of coupling, cohesion and size [New15]. In general, complexity can also be subjective as people can have different interpretation of complex systems. For this reason and because there are very few complexity metrics (see 4.1), this paper only examines metrics for cohesion, coupling and size.

Size Probably one of the most well-known basic software metrics is *Lines of Code*, which is a simple, but effective metric to describe the size of a software. The size of a software system has a negative influence on maintainability, since it most likely increases the complexity of the overall system. For service-based metrics and microservices, granularity and size are often described together, but to better explain this, we first need to characterize microservices:

2.2 Microservices

For microservices the maintainability rules change slightly. In general, it is claimed that microservices are easier to maintain than monolithic architectures.[DGL+17; New15] This is intuitively due to the smaller blocks of code, instead of having one big block of code like a monolithic architecture, which reduces complexity of the whole architecture. But before we get into the maintainability of microservices, we will look at what microservices are and what the differences to other architectures are.

The standard monolithic approach has been refined over the years. Through object-oriented programming we have begun to divide the code into methods and classes. After that, we arranged them in packages. These packages were then, mostly on an abstract level, divided into components. The service-oriented architectures, which also include microservices, continue this idea. The goal is to divide the large monolithic block into small distributed services so that different software attributes, such as complexity or maintainability, become easier. [New15]

As Martin Fowler and James Lewis once described:

> 'There is no precise definition of this architectural style, there are certain common characteristics [...]'

there is no proper definition, so we describe microservices by their most important characteristic and advantages: [FL14]

2.2.1 Important Microservice Characteristics

Distributed Features In a monolithic architecture, the components are often divided technologically. This influences the team structure in such a way that the development team is divided into technological specialists (UI, middleware, backend). That results into the disadvantage that a maintenance which affects all technical layers directly becomes a cross-team project. Microservices, on the other hand, likes to follow the approach of dividing services into business capabilities. Ideally, a microservice would correspond to one feature, but in most cases this is not practicable because it could lead to a service overload and thus to a complexity overload. The advantage of this division is that cross-functional teams can be used, which can be distributed to the individual services. This has the advantage that maintenance over several technical layers does not require any cross-team effort. These phenomena are an example of conway's law [New15]:

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
>
> *Melvyn Conway, 1967*

Technology diversity The separation of the monolith in small distributed microservices allows the ability for using different technologies for every service, whereby it is always possible to choose the perfect fitted technology. For a service, which for example is responsible for a Web UI, we can use languages, which are advantageous for it. In general, you can choose the perfect technology (language, database or other) for each service to take benefit of its advantages [FL14].

These possibilities lead to the fact that microservices are often described as polyglot. This means that they can include multiple languages, as every service can be programmed with a different language. This is important to note as it is a disadvantage for measuring metrics because it requires different measurement techniques.

Individual Deployment In microservices, the principle is that each service can be seen as its own system and should work independent. This also includes the deployment. In service-based architectures, each service is deployed individually, which leads to the advantages that one has a smaller chance of cascading failures and can distribute a better resource management to the services. However, this also makes communication and coupling more complex, since remote calls are more expensive and more complex to manage, which includes creating and maintaining [New15].

Smart endpoints and lightweight communication With *'smart endpoints and dumb pipes'* being a famous concept in the microservice community, microservice applications tend to be as decoupled and as cohesive as possible. They mostly communicate via RESTful HTTP APIs or lightweight messaging [FL14]. They most likely have no Enterprise Service Bus opposing to some Service-Oriented Architectures (SOA). This characteristic makes the metric measurement harder, as communication and dependencies are distributed over the complete architecture and is not easy to follow.

Design for Failure With this procedure, the aim is not only to prevent errors, but also to improve recovery and resilience in the event of an error. Since with microservices errors can be isolated in a service by well-defined service boundaries, it is important to pay attention to error handling when communicating between services in order to prevent the chance of cascading failures. It is important to understand that this concept does not say that you want to have failures, but that you are prepared for the case of failure.

2.2.2 Changes in Maintainability

If you look through the properties of microservices, you will notice the increased complexity of coupling: The change to the network layer, the attempted lightweight communication and as loose a coupling as possible.

Although it is generally claimed that microservices are easier to maintain due to their split architecture, we definitely have a more complex evaluation of maintainability. Complexity is largely based on coupling (in microservices), which is not a simple property to measure, especially when talking about microservices. Technology diversity increases the number of tools needed to measure and the expertise required. Individual deployment, distributed features and development, are characteristics that have a negative impact on the consistency of services, which further increases the complexity of measurement. The lightweight communication makes the coupling more confusing.

The last point that should be noted is the relation of size to granularity. The granularity of microservices describes the size of the individual services. Therefore, microservices can also have good and bad granularity, since the services can be selected in different sizes. With monoliths, the size is usually determined by the functionality and can only be influenced slightly by other factors. For this reason we correlate size metrics with granularity ratings. What is a good size for microservices, however, is mostly individual interpretation and there are no fixed defaults [FL14].

2.3 Scenario-Based Analysis

Scenario-based analysis was developed to determine the quality of software. Scenarios are specifically simulated on architectures and how they react to different situations and changes [BG04]. The system under test can be in the development phase as well as fully developed. Scenario is vaguely defined on purpose, so that this analysis method can be adaptable to many different software qualities.
Tekinerdogan defines scenarios as:

> 'A scenario is considered to be a brief description of some anticipated or desired use of the system. Scenarios that can be directly supported by the architecture(s) are called direct scenarios. Scenarios that require the redesign of the architecture are called indirect scenarios.'

[Tek04] This means that we use indirect scenarios for our use case because we want to have maintainability scenarios. These are possible implementation and maintenance processes that can be performed on the architecture.

In order to make scientific and objective statements in a qualitative evaluation, it is important to strictly follow a fixed plan so that the results are not falsified. In scenario-based analyses, therefore, various methods have been developed for different goals, which can be followed to perform such an analysis:

- Software Architecture Analysis Method (SAAM) focuses on the evaluation of software architecture based on a required quality attribute, but is also introduced to compare different architectures [BG04].
- Architecture Level Modifiability Analysis (ALMA) focuses on a goal-oriented assessments. The procedure includes the determination of a goal and the analysis of the architecture towards the goal [BG04].
- Performance Assessment of Software Architecture (PASA) is used to determine the performance reaction of an architecture to scenarios. It is specifically to analyse the performance goals of a system [BG04].
- Architecture Trade-Off Analysis Method (ATAM) is claimed to be most effective when applied on the final stage of a software architecture. The main goal is analysing the capability of systems for different quality attributes [BG04].

For the evaluation in chapter 6 we decided to base our process on the SAAM model, as it seems to be most effective for analysing the maintainability for different architectures.

3 Related Work

Rud et al., Perepletchikov and Qingqing et al. already proposed many different service-based metrics in their papers [PRF07; PRFT07; QX09; RSD06]. Most of them are already grouped into the different properties of maintainability: coupling, cohesion, size and complexity. However, all of these papers, that propose metrics, have a common problem: They present and explain the calculation and hypothesis of proposed metrics but do not evaluate them.

In general, only two works could be found that evaluate service-based metrics:

In paper [PRT10] the metric *Total Interface Cohesion of a Service* was examined with the help of a controlled experiment, in order to determine the impact of service cohesion on the analysability of service-oriented software. In this experiment 10 participants were asked about the cohesiveness and the analysability of 5 different services and a correlation to the measured metric was calculated. We used a very similar method in chapter 5 with all evaluated metrics. A very strong correlation was measured, however, Perepletchikov et al. describes the result as 'unlikely in a real-world setting', since the structure of the services examined was very deterministic.

A second controlled experiment by Perepletchikov et al. examines service-oriented coupling metrics using an empirical study [PR10]. The aim of this experiment is to investigate the interrelationship between investigated coupling metrics and the specific sub-characteristics of maintainability (i.e., analysability, stability and changeability) they purport to predict. Specifically, they wanted to achieve two main goals: Evaluate the predictive capability and relative impact (strength) of selected metrics. Their selected metric list consists of:

- Weighted extraservice incoming coupling of an element
- Weighted extraservice outgoing coupling of an element
- Element to extraservice interface outgoing coupling
- Extraservice incoming coupling of service interface
- Weighted intraservice incoming coupling of an element

For the study (experiment) 10 software engineers were interviewed. They solved maintenance tasks, from which data such as duration and other dependent variable were extracted and compared with the evaluated metrics. Perepletchikov et al. found evidence for correlations between evaluated metrics and the dependent variables.

Altogether only a small portion of proposed service-based metrics are evaluated. However, examples of different evaluation methodologies can also be extracted from other architectures and metrics:

Dagpinar et al. [DJ03] and Muthanna et al. [MKPS00] use rather conventional methods to evaluate metrics. Both papers analyse measurement methods and investigate correlations between metrics and maintenance data. For maintenance data they both used real-life maintenance logs to extract

for example the maintenance reason. They used this data to compare the effect of metrics on the different maintainability reason probabilities. However Dagpinar et al. deals more with general metrics, where Muthanna et al. specifies on object-oriented metrics. Both papers provide evidence that 'metrics can effectively be used to predict maintainability of software systems' [DJ03].

Kumar et al. offer a rather modern approach to evaluate the effectiveness of object-oriented metrics [KNR15]. They use a software maintainability estimation model designed by application of artificial intelligence (AI) techniques to analyse estimated maintainability. The model is described by using performance evaluation parameters like:

- Mean Relative Error
- Mean Absolute Relative Error
- Standard Error
- Mean True error
- Estimate of true error

The Algorithm uses metrics as input data to train the network and build the model, which is compared to maintenance data (number of lines changed per class from maintenance changes, from real maintenance logs). This approach will be even more interesting for the future as we have not yet reached the full potential of machine learning, however the authors already found higher accuracy of the maintainability model, when using metrics.

Finally, Bouwers et al. performed a detailed industrial experience report, to evaluate the usefulness of the software metrics *Component Balance* and *Profile Dependency* [BDV13]. They not only give a detailed conclusion on the usefulness of these two metrics, but also give detailed reflections on the evaluation used. The evaluation was processed using two different methodologies to gather data. First, they observed real-word experiences of using the metrics and recorded it in form of memos. The second method are interviews, were experts gave ratings from 1 to 5, on the usefulness of metrics evaluated. Afterwards they analysed the results using a quantitative part based on the scores and a qualitative part in which the reports of the interviews are analysed. Their conclusion was that this method of evaluation can be an important part of future software metrics evaluation. We use a similar approach as we also combine a quantitative and a qualitative analysis to determine the effectiveness of software metrics.

In summary, we see that there are generally many ways to evaluate the effectiveness of metrics. However, so far only controlled experiment exists for the evaluation of service-based metrics. In both cases, these are performed with systems created especially for the evaluation, which can strongly influence the final result of the experiment, as the authors stated. A more 'real-life' experiment where real systems are used to evaluate the metrics, does not exist yet. This gap in literature should be filled by this paper, as we use real systems to evaluate the metrics.

4 Research Design

This chapter discusses the preparations that had to be made in order to carry out the evaluation approaches. This includes the creation of the metric list and the processing of the expert ratings.

4.1 Metrics

In order to evaluate service-based metrics, an up-to-date list must first be compiled. For the evaluation, we only use metrics that have already been proposed. The literature review [BWZ17] by Bogner et al. is used as a basis for the metric research, as it provides an up-to-date list for usable maintainability metrics for microservices. For this paper we shortened the list a little, since we search specifically for metrics that can be measured for single services, to be able to evaluate the metrics on the API specifications presented in section 4.2. Additionally, we searched for new proposed service-based metrics.

4.1.1 Proposed Metrics

Table 4.1 shows all usable **service-based** maintainability metrics for microservices proposed by several literature.

4.1.2 Criteria

As the table 4.1 contains far too many metrics to discuss and evaluate, we need to define criteria on which we select the metrics to examine. With these criteria we should be able to make a short and useful selection of metrics.

Efficient Measurement

As already described, the measurement of metrics in microservices can become very complex. Therefore, a few metrics are dropped because we would have to measure them on many different systems and that would be too elaborately. Since we have a rather unusual use case here and have to measure the metrics for many different architectures, we only use metrics that can be measured with static methods, since runtime data analysis is very time-consuming to set up. *SIUC* is one such example which requires runtime data for measurement. There is already a tool for measurement, but it needs tracing and since almost none of the architectures investigated supports tracing, the effort, implementing tracing in every architecture would be too great, to evaluate this metric.

Metric name	Property	Authors	Source	Abbreviation
Services Interdependence in the System	Coupling	Rud et al.	[RSD06]	SIS
Absolute Importance of the Service	Coupling	Rud et al.	[RSD06]	AIS
Absolute Dependence of the Service	Coupling	Rud et al.	[RSD06]	ADS
Absolute Criticality of the Service	Coupling	Rud et al.	[RSD06]	ACS
Number of Services Involved in the Compound Service	Complexity	Rud et al.	[RSD06]	$NSIC$
Service Interface to Intra Element Coupling	Coupling	Perepletchikov et al.	[PRFT07]	$SIIEC$
Extra-Service Incoming Coupling of Service Interface	Coupling	Perepletchikov et al.	[PRFT07]	$ESICSI$
Total Response for Service	Coupling	Perepletchikov et al.	[PRFT07]	TRS
Service Interface Data Cohesion	Cohesion	Perepletchikov et al.	[PRF07]	$SIDC$
Service Interface Usage Cohesion	Cohesion	Perepletchikov et al.	[PRF07]	$SIUC$
Service Sequential Usage Cohesion	Cohesion	Perepletchikov et al.	[PRF07]	$SSUC$
Strict Service Implementation Cohesion	Cohesion	Perepletchikov et al.	[PRF07]	$SSIC$
Loose Service Implementation Cohesion	Cohesion	Perepletchikov et al.	[PRF07]	$LSIC$
Total Interface Cohesion of a Service	Cohesion	Perepletchikov et al.	[PRF07]	$TICS$
Service Granularity	Complexity	Qingqing et al.	[QX09]	SG
Relative Coupling of Service	Coupling	Qingqing et al.	[QX09]	RCS
Relative Importance of Service	Coupling	Qingqing et al.	[QX09]	RIS
Weighted Service Interface Count	Size	Hirzalla et al.	[HCA08]	$WSIC$
InterAction Number	Size	Jin et al.	[JLZ+18]	IRN
Dynamic Relative Dependence of Service	Coupling	Schlinger	[Sch19]	$DRDS$
Dynamic Relative Importance of Service	Coupling	Schlinger	[Sch19]	$DRIS$
Dynamic Relative Dependence of Service in the System	Coupling	Schlinger	[Sch19]	$DRDSS$
Dynamic Relative Importance of Service in the System	Coupling	Schlinger	[Sch19]	$DRISS$

Table 4.1: Service-based maintainability metrics for microservices

Other metrics may require an extended documentation for measuring, which means it depends heavily on the example architecture, on which its gets measured on.

Even Coverage

To get impressions for each design property of maintainability, we want to examine all categories. The complexity is excluded (see 2.1.3 for the reasons).

Diversity

In the investigation of metrics in literature, the same metrics are sometimes found under different names. Most of them are already excluded from the table 4.1. However, the metrics from Schlinger [Sch19] are also excluded from evaluation, as they are runtime copies from other metrics.

Service-Based

Because of the lack of evaluations so far, we exclusively examine service-based metrics, we filtered every non-service-based metric in the literature search. Service-based mean that metrics are strictly for service-oriented architectures and microservices and not applicable to other architectures.

So for a quick overview, the criteria for choosing metrics to measure are:

- Efficient measurement
- Even topic coverage
- Diversity
- Service-based

4.1.3 Metric Selection

The following metrics got selected from the proposed metrics list in 4.1.1:

- Weighted Service Interface Count
- Services Interdependence in the System
- Absolute Importance of the Service
- Absolute Dependence of the Service
- Absolute Criticality of the Service
- Service Interface Data Cohesion

Weighted Service Interface Count ($WSIC$)

Definition $WSIC(S)$ is defined as the number of weighted exposed interface operations for a service S. The default weighting per operation is one, alternatively, M. Hirzalla proposes weights by the complexity of the data types of parameters or the number of parameters. By default, and in this paper used case $WSIC$ describes the number of exposed operations per service.

Hypothesis The more operations a service has, the bigger and more complex it is. This means that the $WSIC$ makes statements about size and thus indirectly about the complexity of services. It follows that the larger the value per service, the worse the size and granularity of a microservice architecture is.

Measurement In order to get a system wide value, the average is calculated with:

$$WSIC_{AVG} = \frac{\sum_{S \in Y} WSIC(S)}{Y}$$

With Y being the set of all services. The $WSIC$ can be measured with static code analysis. Another measurement option are small scripts that extract the $WSIC$ from the source code via method signatures. For API specifications the $WSIC$ can be calculated with the existing 'Open API Maintainability Evaluation Tool'[1].

In this work, the $WSIC$ was calculated using the tool or small scripts in combination with manuel extraction. The more polyglot a microservice architecture is, the more complex the calculation becomes, since usually different technologies have to be used for different languages.

Service Interdependence per Service(SIS)

Definition SIS is derived from Services Interdependence in the System(SIY). $SIS(S)$ are dependencies for a service S to other services that are bi-directional. This means that the services are interdependent. According to Rud et al. such connections should not exist at all, therefore the SIY should be zero. The best practice would be to resolve the connections by merging the two services into one. [RSD06]

Hypothesis If a service has many dependencies, the system-wide coupling quickly becomes complex and overloaded. Bidirectional connections should therefore be prevented not only for services, but also for programming in general, using merges. Any value above zero has a negative influence on the coupling.

[1] https://gilbert.informatik.uni-stuttgart.de/evolvability-assurance/openapi-evaluation/cli

Measurement The SIS and SIY can both be measured with architecture design diagramms and descriptions. If there is no such information they have to be extracted from code, which can be complex and expensive. For a system wide value we use the SIY which can be calculated from SIS for all services with:

$$SIY = \frac{\sum_{S \in Y} SIS(S)}{2}$$

In this work, the $SIS(S)$ for each complete system could be manually read from the architecture diagram.

Absolute Importance of the Service(AIS)

Definition $AIS(S)$ is defined as the count of clients which depends on a service S. It is important to note that clients or services that are on the same node are not counted as they do not cause any extra network traffic and it is unlikely that only one service will fail on the same node [RSD06].

Hypothesis The more dependencies in a microservice architecture exist the more complex the coupling, which has a negative impact on the maintainability for the architecture.

Measurement The $AIS(S)$ for each service S can be measured via architecture design diagramms or descriptions. If no such information is provided, these metric must be extracted from source code, which can be expensive. The AIS average can be calculated with:

$$AIS_{AVG} = \frac{\sum_{S \in Y} AIS(S)}{Y}$$

With Y being the set of all services in the architecture. In this work, the $AIS(S)$ for each complete system could be manually read from the architecture diagram.

Absolute Dependence of the Service (ADS)

Definition $ADS(S)$ is the inverse to $AIS(S)$. With $ADS(S)$, all clients on which a service S depends are counted. Just like $AIS(S)$, when measuring $ADS(S)$ you have to pay attention to count only clients that are not on the same node as S, for the same reasons.

The hypothesis and measurement methods for ADS is the absolute same as for AIS 4.1.3.

Absolute Criticality of the Service (ACS)

Definition The ACS is a combination from ADS and AIS. $ACS(S)$ is the product of the absolute importance and absolute dependence for a service S.

Hypothesis If a service S only depends on many other services, or only many other services depend on S, it does not necessarily mean that the coupling is very complex. A combination however, could lead to a 'master service', which becomes a hazard point in the architecture for failures and cascading failures.

Measurement To calculate ACS for a service S we simply solve the following calculation:

$$ACS(S) = AIS(S) \times ADS(S)$$

To calculate the average we used following calculation:

$$ACS_{AVG} = \frac{\sum_{S \in Y} ACS(S)}{Y}$$

Service Interface Data Cohesion ($SIDC$)

Definition $SIDC(S)$ calculates the cohesion of a given service S with the cohesiveness of their exposed operations. Initially it was proposed that the cohesiveness of operations is calculated using the similarity of data types of their parameters. However there is also a new method introduced, which calculates the cohesiveness of operations via the return data type and a combination of the return and parameter data type.

Hypothesis A service is highly cohesive when all exposed operations have the same input parameter (return) data types. The resulting value is between 0 and 1, where 0 is not at all cohesive and 1 is strongly cohesive. The cohesiveness of a service and its operations has a positive influence on the maintainability of given service.

Measurement To calculate the original $SIDC(S)$ for a service S, the service operations (so) with common parameters are written to a set. The cardinality of this set is then divided by the number of different parameter types (totalParamTypes):

$$SIDC(S) = \frac{|Common(Param(so \in S))|}{totalParamTypes}$$

The hypothesis and the approach to calculate the $SIDC(S)$ via return data types is the same. However the approach to calculate the $SIDC$ with a combination from parameters and returns is a bit more complicated:

$$SIDC(S) = \frac{Common(Param(so \in S)) + Common(returnType(so \in S))}{Total(so \in S) * 2}$$

, where:

- $Common(Param(so \in S))$ calculates the number of service operation pairs that have at least one parameter type in common.
- $Common(returnType(so \in S))$ calculates the number of service operation pairs that have the same return type.
- $Total(so \in S)$ are all possible combinations of operation pairs for a service S.

Note, that operations with no input parameters and no return types will be excluded from *SIDC* calculations [PRT10].

Extracting *SIDC* from source code can be very expensive. This is partly due to the polyglot property of microservices, because different technologies need different measurement methods. The measurement can be done with static code analyses, tracing tools (runtime data) and scripts. In this paper, the *SIDC* was calculated using scripts that extract parameter types and return values from source code, using method signatures. Additionally for the APIs and complete systems that provided APIs the *SIDC* was calculated using a static code analysis tool[2].

4.2 Systems Used for the Evaluation

For the evaluation 7 complete architectures and 8 APIs were used. Table 4.2 shows all complete microservice architectures, with their respective *GitHub*, used for the quantitative and qualitative evaluation.

Architecture	Github
Sock Shop	https://github.com/microservices-demo/microservices-demo
Tea Shop	https://github.com/DescartesResearch/TeaStore
Service-based Web Shop v1	https://github.com/xJREB/service-based-web-shop-v1
Service-based Web Shop v2	https://github.com/xJREB/service-based-web-shop-v2
Piggy Metrics	https://github.com/sqshq/PiggyMetrics
Photo Sharing App	https://github.com/nginxinc/mra-ingenious
Robot Shop	https://github.com/instana/robot-shop

Table 4.2: Microservice architectures used in the evaluation

A quick overview of these systems is provided in a *GitHub* repository [3]. These systems are all example architectures that are not actively deployed. They have been developed to demonstrate microservices or demonstrate specific technologies and processes for microservices. These architectures have a total of 52 services that use the following languages:

- Go

[2]https://gilbert.informatik.uni-stuttgart.de/evolvability-assurance/openapi-evaluation/cli
[3]https://github.com/xJREB/microservices-maintainability-benchmark/tree/master/system-one-pagers

- Java
- JavaScript
- Ruby
- Python
- PHP
- Shell

Additionally the following APIs were used for the quantitative evaluation in 5:

- Amazon Transcribe Service
- BBC iPlayer
- BBC Nitro
- Google Apps Customer Management
- Government of British Columbia: Job Postings
- TomTom Maps
- TomTom Routing
- TomTom Search

A *GitHub* where the APIs are collected has already been created[4].

These architectures and APIs were chosen because the studies for the expert ratings were already available and some of these architectures are also perfectly applicable to our scenarios in section 6, due to their similar functionality.

4.3 Expert Ratings

The original expert ratings which are used for the quantitative evaluation in chapter 5 were collected in a previous study conducted at the *Institute of Software Technology of the University of Stuttgart*.

In the first iteration, ten experts were interviewed about the eight APIs and eleven experts were interviewed about the seven example microservice architectures which are presented in 4.2. The API ratings were conducted in the categories: size, complexity, cohesion and overall maintainability, for the microservice architectures the categories: granularity, complexity, coupling, cohesion and overall maintainability were used. The respective ratings range from one to ten, with one being the worst and ten the best for each category.

[4] https://github.com/xJREB/openapi-maintainability-benchmark/tree/master/openapi-files

Since the expert ratings in the first iteration were partly inhomogeneous, we performed a second iteration using the *Delphi* method in the process of this paper, to reduce deviations [DH63]. In the second iteration, every expert who had ratings that differed by four or more from the median, was interviewed again to improve their rating. They had insight into the median and comments of other experts who had justified their ratings.

Table 4.3 and 4.4 shows the respective improvements of the average standard deviation (SD) in every category for the expert assessments, achieved with the second iteration.

Category	Average SD (1. Iteration)	Average SD (2. Iteration)
Granularity	1.75	1.34
Coupling	1.71	1.51
Cohesion	1.64	1.4
Complexity	1.58	1.37
Overall Maintainability	1.71	1.61

Table 4.3: Improvement of the second iteration for complete architectures

Category	Average SD (1. Iteration)	Average SD (2. Iteration)
Size	1.51	1.43
Cohesion	2.24	1.72
Complexity	1.85	1.61
Overall Maintainability	2.10	1.69

Table 4.4: Improvement of the second iteration for APIs

It is important to note that the studies were conducted independently of each other, i.e. the assessments for the APIs and the complete architectures are not related.

4.4 Research Process

Figure 4.1 shows the complete research process performed in this paper.

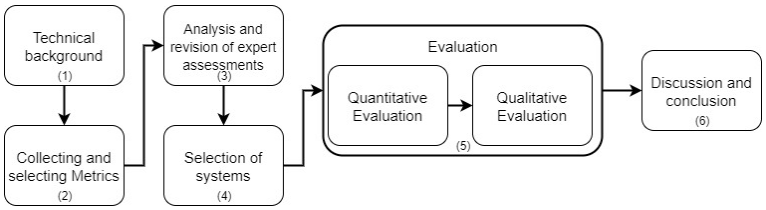

Figure 4.1: Research process

(1) The first step was to understand the technical backgrounds, i.e. microservices, maintainability and scenario-based analysis, of this work.

(2) Afterwards proposed metrics had to be searched and a list had to be compiled.

(3) Then we analysed the expert ratings and decided on a second iteration using the *Delphi* method.

(4) The fourth step was to decide that all systems used for the expert ratings will also be used for the evaluation.

(5) The evaluation includes both the correlation to expert ratings and the scenario-based analysis.

(6) The last step was to analyse and discuss results for both evaluations from step (5) to form a conclusion.

5 Quantitative Evaluation - Comparison with Expert Ratings

This part evaluates the metrics with a quantitative analysis. This is done with the help of a study, described in 4.3. These ratings will be compared with the measured metrics and correlations are investigated.

For measuring the correlation we use the Pearson product-moment correlation coefficient, which consists of correlation coefficient R and a p-value. We use this correlation method to find out if there is a linear correlation between the metric values and expert assessments. The p-value evaluates the significance of our correlation, it is generally assumed that a p-value below 0.05 indicates a significant correlation. To interpret the coefficient R we use the generally used methodology from Cohen(1988), which defines:

- (+/-) 0.10 weak agreement
- (+/-) 0.30 fair agreement
- (+/-) 0.50 strong agreement

This methodology also applies to negative values, since we want to have negative correlations for some metrics and assessments so that they agree correctly.

5.1 Size/Granularity

The following metrics address the categories size and granularity of services. Important to note is that the experts got questioned about the granularity of the architectures. Since high values of the metrics mean poor size and granularity and thus bad maintainability, but high values of the experts ratings mean good size and granularity, we want a negative correlation, in order to have an agreement between metrics and expert assessments. As an additional reminder, the experts rated the size of APIs, but the granularity for the complete Systems

5.1.1 Weighted Service Interface Count

Table 5.1 shows the measured $WSIC_{AVG}$ for complete systems and $WSIC$ for APIs and their corresponding **granularity** ratings for complete systems and **size** ratings for APIs. The $WSIC$ was measured using various tools. For all APIs and for complete systems that offered APIs, a static

Complete system	*WSIC* Average	Expert Rating Median (granularity)
Sock Shop	3.4	8
Service-based Web Shop v1	4.57	8
Service-based Web Shop v2	5	7
Piggy Metrics	3.6	9
Tea Store	10	5
Photo Sharing App	6.23	8
Robot Shop	4.25	7

API	*WSIC*	Expert Rating Median (size)
Amazon Transcribe Service	9	8
BBC iPlayer	30	5
BBC Nitro	25	3.5
Google Apps Customer Management	17	7.5
Government of British Columbia: Job Postings	5	10
TomTom Maps	10	8
TomTom Routing	4	9
TomTom Search	19	6

Table 5.1: Measured *WSIC* and granularity/size expert ratings

code analysis tool was used, which automatically measures the *WSIC*[1]. For complete systems that do not offer APIs, the *WSIC* was calculated using scripts and manual work. Note that the values are rounded to 2 digits after the decimal point.

As we see in figure 5.1, the $WSIC_{AVG}$ has a strong agreement with the granularity assessments. This result is also supported by a good p value of 0.02 . With the p-value being below the 0.05 threshold we can conclude that the **granularity** expert ratings and the $WSIC_{AVG}$ show an agreement.

In fact, the *WSIC* correlates not only with the expert ratings for the complete systems, but also with the size ratings of APIs, as figure 5.2 shows.

[1] https://gilbert.informatik.uni-stuttgart.de/evolvability-assurance/openapi-evaluation/cli

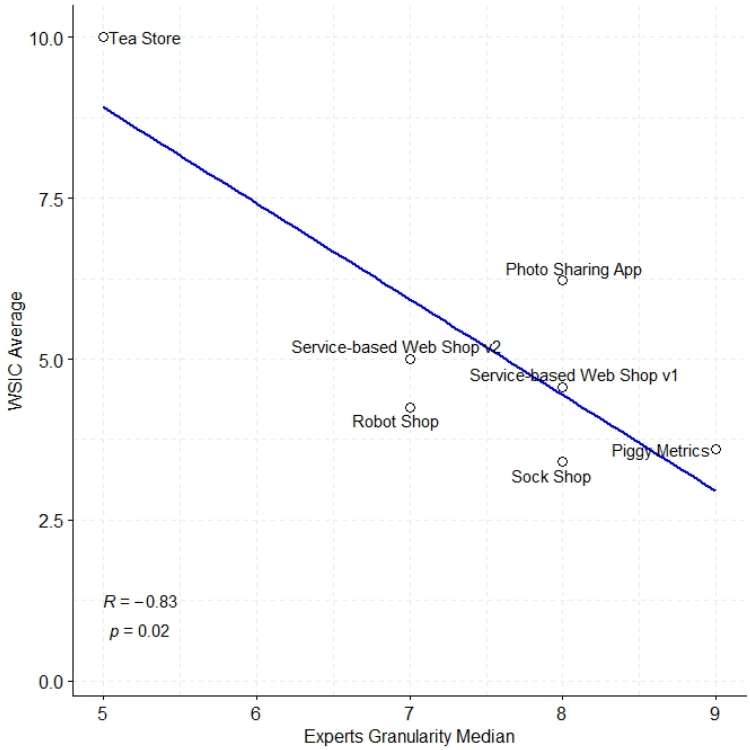

Figure 5.1: $WSIC_{AVG}$, complete system correlation

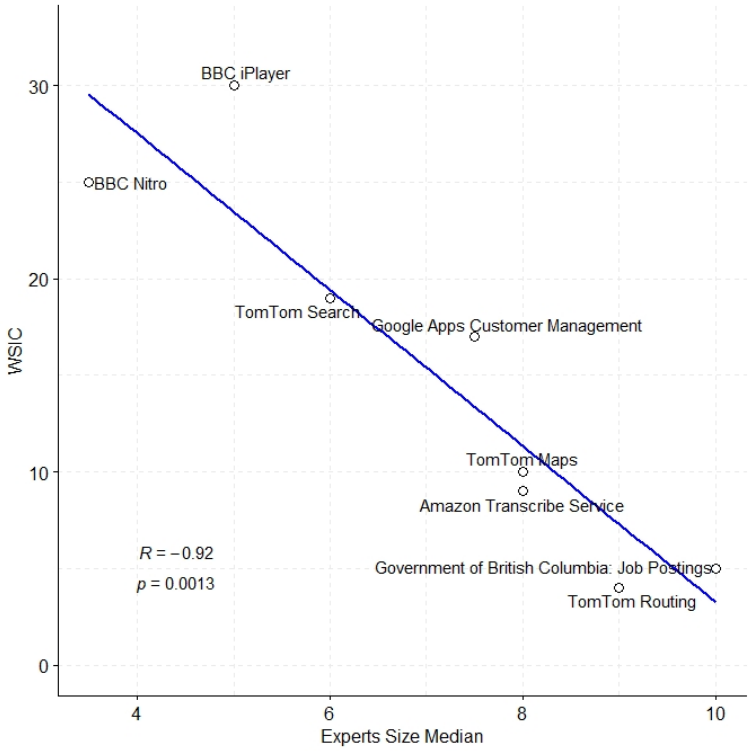

Figure 5.2: $WSIC$, API correlation

With $WSIC = -0.92$ and $p = 0.0013$ the $WSIC$ correlates even stronger with the API ratings. This is probably mainly due to the fact that the ratings for an API is always based on **one** API which is comparable to one of many service for the complete architectures. This is also the reason, for why we had to use $WSIC_{AVG}$ for the complete architectures, but just the $WSIC$ for the APIs.

5.2 Coupling

The following metrics address the coupling of services.

Since high values of the evaluated metrics mean poor coupling and thus bad maintainability, but high values of the experts ratings mean good coupling, a negative correlation concludes to an agreement between these two values. For the coupling metrics, we only investigate the complete systems, since we do not have coupling in single APIs. All metrics could be measured manually from the design diagrams. It should be noted that no database dependencies were taken into account, as these can either be deployed independently or on the same node, whereby they are excluded by definition from the calculation.

5.2.1 Absolute Dependence of the Service and Absolute Importance of the Service

Since ADS_{AVG} and AIS_{AVG} are equivalent in an architecture (dependencies always consist of a source and a target), we only need to evaluate these metrics once.

System	Metric Average	Expert Rating Median
Sock Shop	0.9	10
Service-based Web Shop v1	1.15	6
Service-based Web Shop v2	1.22	8
Piggy Metrics	2.33	7
Tea Store	1.83	6
Photo Sharing App	1.14	7
Robot Shop	1.25	6

Table 5.2: Measured ADS_{AVG},AIS_{AVG} and coupling expert ratings

Table 5.2 shows the measured ADS_{AVG} and AIS_{AVG} values and the corresponding **coupling** expert ratings. In figure 5.3 we see $R = -0.39$, which means we have an fair agreement. In fact, there is also the consensus that *Sock Shop* has the best value for AIS and ADS and also the best expert rating.

Also note that with *Piggy Metrics*, which has by far the worst ADS/AIS value but a mediocre expert rating median, the experts were very divided. With a standard deviation of 2.03, this rating was one of the most controversial. This is mostly, due to interdependence's in the architecture, which were not considered by all experts. More about interdependence's see subsection 5.2.3. In

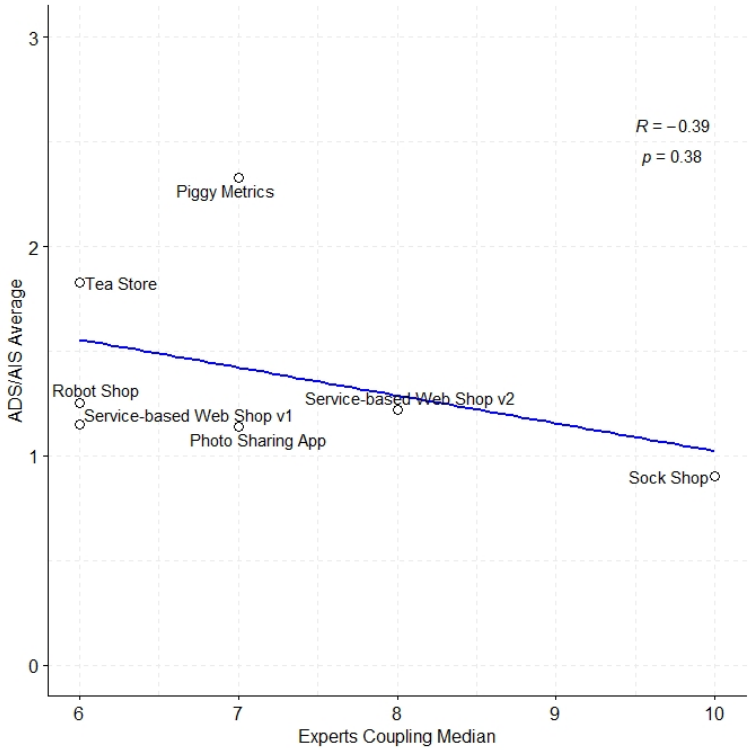

Figure 5.3: ADS_{AVG} and AIS_{AVG} correlation

figure 5.4 we removed *Piggy Metrics* from our data set, and we get $R = -0.63$ wich is a strong agreement. However figure 5.4 shows that there are still some outliers like *Tea Store*, which results into a p-value still being over the 0.05 threshold.

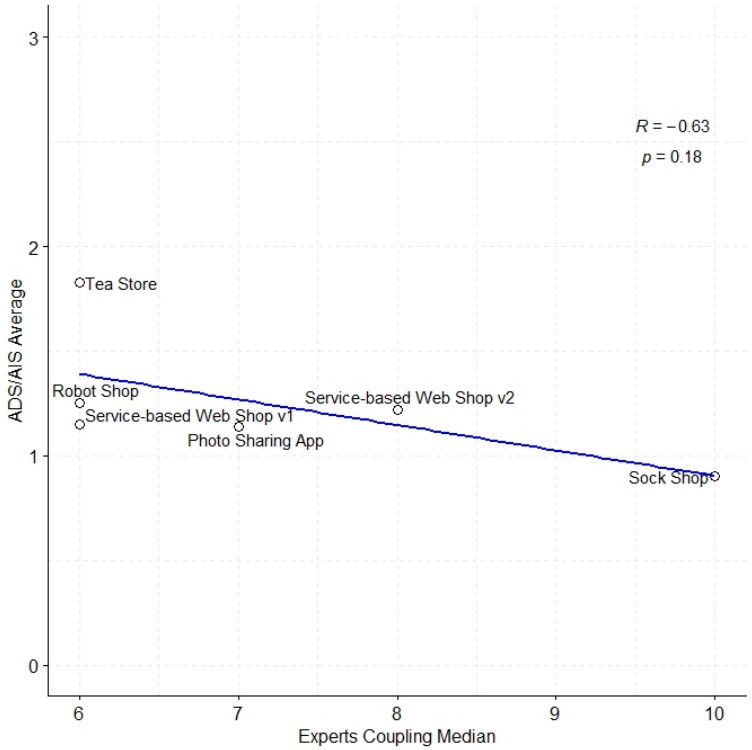

Figure 5.4: ADS_{AVG} and AIS_{AVG} correlation, without using *Piggy Metrics*

5.2.2 Absolute Criticality of the Service

System	Metric Average	Expert Rating Median
Sock Shop	0.9	10
Service-based Web Shop v1	1	6
Service-based Web Shop v2	0.78	8
Piggy Metrics	3.4	7
Tea Store	1.33	6
Photo Sharing App	1	7
Robot Shop	0.5	6

Table 5.3: Measured ACS_{AVG} and coupling expert ratings

The ACS correlates very poorly with the metrics. This is partly for the same reasons as *ADS* and *AIS*. The interdependencies of *Piggy Metrics* are not considered by all experts and therefore the ratings are very inhomogeneous. However even if we remove *Piggy Metrics* from the calculation, the correlation does not get better ($R = -0.12$, $p = 0.82$).

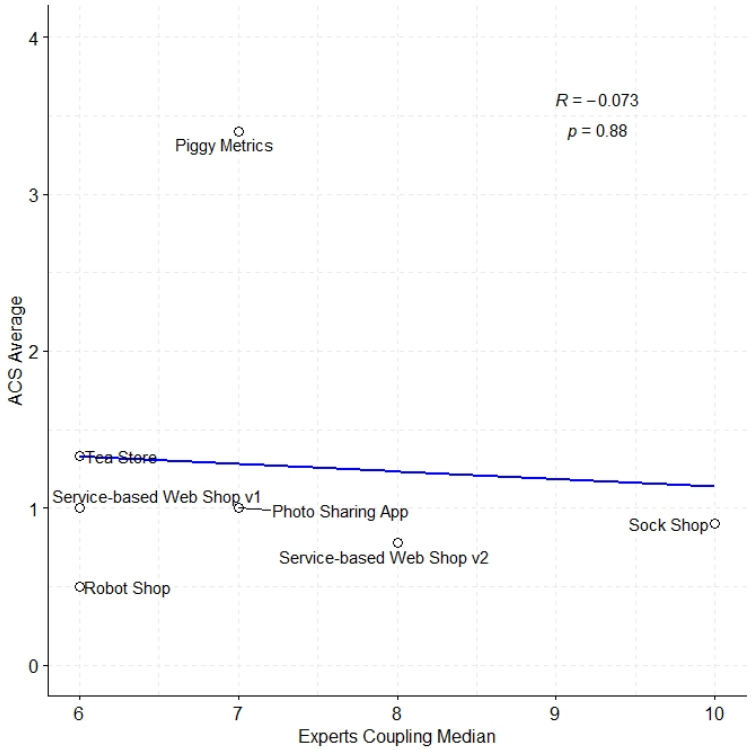

Figure 5.5: ACS_{AVG} correlation

The other main reason is the special calculation of the ACS (reminder: $ACS(S) = AIS(S) \times ADS(S)$). Services that either only have a high AIS and an $ADS = 0$, or only high ADS and $AIS = 0$, are set to 0 by the calculation. In contrast, experts are of the opinion that services, where one of this cases occurs, will deteriorate the general coupling.

5.2.3 Absolute Interdependence per Service

The authors Rud et al. described an interdependence between services as an absolute no-go and that they should be prevented in any case. This is reflected in the architectures examined, since such interdependence's occur only in the architecture of *Piggy Metrics*[2]. This was also recognized by a number experts, as some have noted these dependencies in both directions and therefore gave a rather bad ranking. However, it was not taken up by all experts and therefore the ranking is not the worst of all architectures.

As 5 out of 7 experts who commented on their rating mentioned these interdependences, we can conclude that there are correlations between this metric and expert ratings. Experts who have picked up these connections are of the opinion that they worsen the coupling, which approves the intention of the author of this metric. Rating correlations can not be measure here, since *Piggy Metrics* is the only architecture, where $AIS > 0$.

[2]https://github.com/sqshq/PiggyMetrics

5.3 Cohesion

The following metric address the cohesion of services.

Since high values of the metrics mean high cohesion and thus better maintainability and high values of the experts ratings mean good cohesion, we want a positive correlation to have an agreement.

Service Interface Data Cohesion

The $SIDC$ was calculated using the combined method from return and parameter types. Reminder:
$SIDC(S) = \frac{Common(Param(so \in S)) + Common(returnType(so \in S))}{Total(so \in S) * 2}$

Complete system	$SIDC$ average	Expert rating median
Sock Shop	0.04	9
Service-based Web Shop v1	0.37	6
Service-based Web Shop v2	0.37	7
Piggy Metrics	0.12	8
Tea Store	0.35	5
Photo Sharing App	0.16	8
Robot Shop	0.23	8

API	$SIDC$	Expert rating median
Amazon Transcribe Service	0.51	8
BBC iPlayer	0.79	7.5
BBC Nitro	0.8	5
Google Apps Customer Management	0.43	8
Government of British Columbia: Job Postings	0.5	8
TomTom Maps	0.32	8
TomTom Routing	0.5	8
TomTom Search	0.5	8

Table 5.4: Measured $SIDC$ and cohesion expert ratings

Table 5.4 shows the measured values for the complete Systems and the APIs. Note that for the APIs we did not have to calculate an average, since just single APIs were measured and rated. The $SIDC$ was measured using various tools. For all APIs and for complete systems that offered APIs, a static code analysis tool was used, which automatically calculates the $SIDC$[3]. For complete systems that do not offer APIs, the $SIDC$ was calculated using scripts and manual work. Note that the values are rounded to 2 digits after the decimal point.

[3] https://gilbert.informatik.uni-stuttgart.de/evolvability-assurance/openapi-evaluation/cli

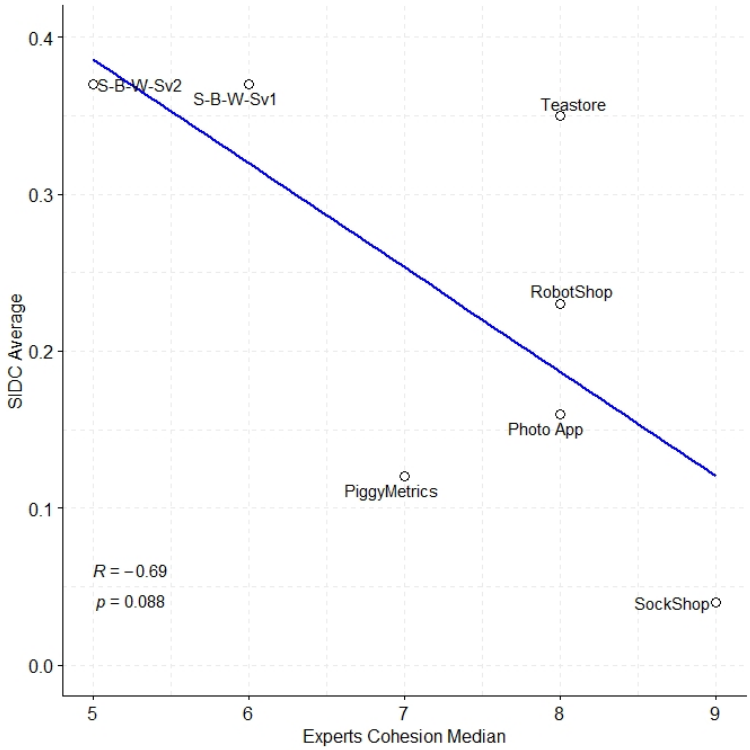

Figure 5.6: $SIDC_{AVG}$ systems correlation

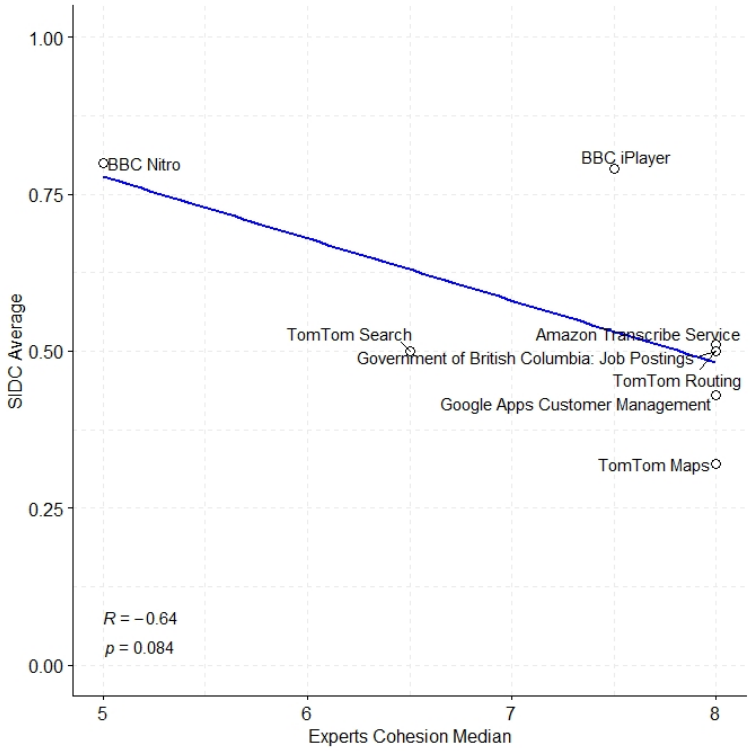

Figure 5.7: *SIDC* APIs correlation

As shown in figures 5.6 and 5.7, it is immediately apparent that the $SIDC$ actually correlates negatively with the cohesion ratings. However, neither of the two correlations is significant, as shown by their p-values. This can partly be due to the minimal variance of the expert assessments. For the APIs we only have 2 of 10 ratings that deviate from 8. The complete systems also show a high similarity in their ratings.

5.4 Correlation Summary

Table 5.5 shows that the size metrics have a strong agreement with the expert assessments. The coupling metrics show signs of a correlation, however there are special cases and outliers, where metrics and experts do not agree. For the cohesion there was no correlation found, and actually signs of a negative agreement are being reported.

Metric	correlation coefficient R	significance p
Systems $WSIC_{AVG}$	-0.83	0.02
APIs $WSIC$	-0.92	0.0013
ADS_{AVG}/AIS_{AVG}	-0.63	0.18
ACS_{AVG}	-0.073	0.88
Systems $SIDC_{AVG}$	-0.69	0.088
APIs $SIDC$	-0.64	0.084

Table 5.5: Summary of the correlations between expert assessments and metrics

Additionally, in the progress of this evaluation, every metric was tested for a correlation to the overall maintainability assessments by the experts, however there was no significant correlation found.

In chapter 7, the results are further analysed and discussed in more detail.

6 Qualitative Evaluation - Scenario-based Analysis

In this chapter the metrics are evaluated with a qualitative evaluation. We theoretically execute different realistic scenarios on different architectures and compare them with the metrics. The scenarios were developed and executed based on the *SAAM* model. Initially *SAAM* was introduced for modifiability, but also described as usable for any quality attribute. Since modifiability is similar to maintainability and the model is introduced for comparing different architectures, it is very practical for our case. However, we modified the process model a little, to make it more effective for our use case.

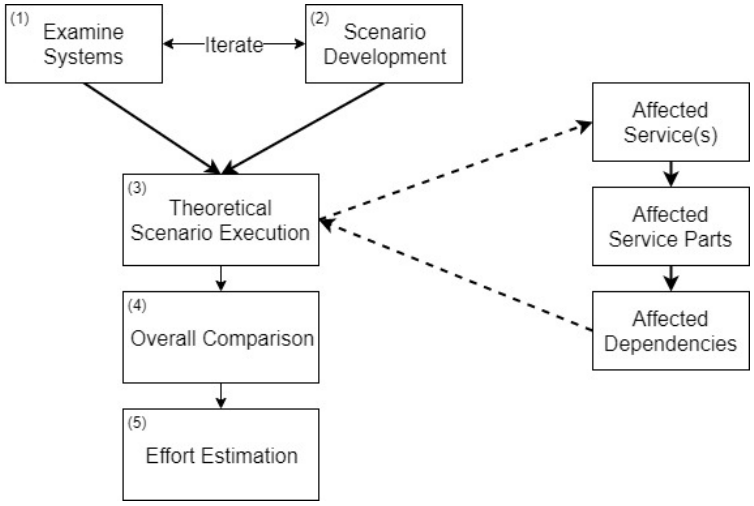

Figure 6.1: Scenario process model

Figure 6.1 shows the process model which we will follow for every scenario. This includes following steps:

(1) + (2) In these two steps, we iterate over the functionalities of the examined systems and the scenario development. The goal is to find scenarios that are equally applicable to all systems investigated. The difference to the original SAAM model is that we iterate over the features of the systems and not over the architecture, because we do not want to have architecture specific scenarios.

(3) This step contains the theoretical scenario execution. It evaluates how many services the scenario affects, which parts of a service and how many dependencies **could** be affected by the change.

(4) Now the information from step 3 is compared. Any main differences are noted. Normally this step is inspected last, but due to the fact, that we specifically want to compare scenarios on the different systems, we compare the systems before we give our final effort estimation.

(5) From the information we gather in step (3) and (4), we now estimate the effort for a scenario. The effort is estimated on a scale from 1 to 10, where 1 is low effort and 10 high effort.

After this process we compare the gathered information, which includes the implementation differences and the estimated effort with the measured metrics for every architecture.

The following architectures will be used in this evaluation:

- Sock Shop
- Tea Store
- Robot Shop

These were chosen because we already measured the metrics for the quantitative evaluation and because of their similar functionality, as they are all web shops with about the same set of functionality. The appendix A.1 contains design diagrams describing their different architectures.

6.1 Scenario Descriptions

Table 6.1 shows developed maintenance scenarios during the iterations over the different systems (step (1) and (2) of our process model).

Scenario name	Scenario description
Add payment method	Add PayPal as an additional payment method to provide the user multiple payment options.
Add item category	Add a new item category to increase the assortment of the shop.
Changeable user data	Add the feature for users to change their data (password, email).
Additional currency	Add an additional currency, to be able to present your products in different currencies to appeal to a larger number of customers.

Table 6.1: Developed evaluation scenarios

6.2 Scenario Evaluations

In the following we will carry out the scenarios to gather information about the theoretical reaction to our changes. They will be theoretically evaluated individually on the 3 example architectures. Specialities, differences and estimations will be noted. This includes step (3) to (5) of our process model. Afterwards the results will be compared with the metrics.

6.2.1 Scenario 'Add payment method'

With the first scenario we have found a scenario that is both very realistic and well applicable to all example architectures, as all 3 architectures so far only support payment via credit card.

Evaluation step	*Sock Shop*	*Tea Store*	*Robot Shop*
Affected services	Payment, Front-End	Persistence, WebUI	Payment, Web UI
Possible affected dependencies	Order, (Edge-Router, Load Tester: These are dependencies, which could be affected, however as they both include just short shell and python scripts for testing and deployment. It is very clear that they are not affected by any changes.)	Recommender, Image	none
Differences	Minimal higher expenditure in contrast to the *Robot Shop*. Because of the dependency to 'Payment' service, minimal changes have to be made to the 'Order' service as he passes payment request from the UI to the 'Payment' Service.	No 'Payment' service, which increases the search, where to implement these changes, drastically. This also changes where to implement it, as we have the option to implement it either in 'Persistence' or 'WebUi' service, or introduce a new payment service.	Minimal less effort than *Sock Shop*
Effort estimation	3	4	3

Table 6.2: Theoretical reaction on 'Add payment method' scenario

The biggest difference between the architectures is that the size of the services for the *Tea Store* makes the search, where the changes would be implemented, more expensive. Especially the 'Persistence' service is a large service, where a lot of data is managed. This affected the search, where the payment is handled negatively and resulted into higher effort. Depending on where the changes get implemented, possible changes to endpoints of the 'Persistence' could ripple to the 'Image' and 'Registry' services. This is also reflected in our size metric $WSIC$, as the *Tea Store* and especially the 'Persistence' service has the highest $WSIC$ value of all investigated complete systems. Also the higher coupling is represented as *Tea Store* has higher ADS/AIS and ACS values comparing to *Sock Shop* and *Robot Shop*. The metric $SIDC$ does not reflect the cohesiveness well, because the *Tea Store* has the best value but in generally has lower cohesion due to the longer search and possible implementation locations.

The effort for the changes for *Sock Shop* and *Robot Shop* in their respective 'Payment' service is very similar. This is also reflected by the metric for size, as they have both about the same $WSIC$ values and as the $WSIC$ is relatively small for both service ($WSIC = 2$ for *Sock Shops* 'Payment' service and $WSIC = 3$ for *Robot Shops* 'Payment' service), the cohesion has no impact on this scenario. The minimal increased effort, by the additional coupled 'Order' service is reflected by the coupling metrics as the *Sock Shop* has an overall slightly higher coupling metrics value.

6.2.2 Scenario 'Add item category'

Our second scenario is also very applicable, as an assortment expansion is very realistic and all architectures have already products sorted into different categories.

Evaluation step	Sock Shop	Tea Store	Robot Shop
Affected Services	Catalogue	Persistence	Catalogue
Possible affected dependencies	Front-End	Recommender, Image, WebUI	Web UI
Differences	It quickly becomes clear that only changes in the database are necessary. In addition, it must be checked whether the UI is affected by additional categories.	The size of the 'Persistence'-service makes it harder to find out if additional changes need to be made besides the database. For the Front-End the same applies as for the *Sock Shop*.	Same as *Sock Shop*
Effort estimation	1	2	1

Table 6.3: Theoretical reaction on 'Add item category' scenario

In our second scenario we find again only small differences between the architectures. Since *Sock Shop* and *Robot Shop* show a very similar architecture, there is no difference between these two.

The main difference is again caused by *Tea Stores* 'Persistence' service, which is very critical as reflected by the metrics (combination of high $WSIC$ and coupling as shown in the first scenario).

6.2.3 Scenario 'Changeable user data'

Since each of the systems provides a registration and login feature, but no possibility to change the password or email, we want to add this feature.

Evaluation step	Sock Shop	Tea Store	Robot Shop
Affected Services	Front-End, User	Persistence, Web UI	User, WebUI
Possible affected dependencies	none	none	none
Differences	It is straightforward to determine where to add the new code and the new endpoints.	No differences, also straightforward. In fact the functionality is already implemented, but we do not reduce the estimation for this, since we do not compare the implementation itself	Also straightforward and no differences
Effort estimation	4	4	4

Table 6.4: Theoretical reaction on 'Changeable user data' scenario

This scenario is straightforward. All implemented changes are code additions, which means existing dependencies are not affected by any change. In other words, we cannot make any statement about the coupling with this scenario. The size and cohesiveness of the services have also no effect on the implementation of this scenario, so we can't make a statement with this scenario. The only thing we can note from this scenario, that again changes in *Tea Store* go in the critical 'Persistence' service.

6.2.4 Scenario 'Additional currency'

This scenario is intended to be a bit more extensive maintenance, as changes must be made to the database, endpoints and the user interface.

Evaluation step	Sock Shop	Tea Store	Robot Shop
Affected Services	Catalogue, Front-End	Persistence, WebUI	Catalogue, Web UI
Possible affected dependencies	None	Recommender, Image	None
Differences	Depending on the implementation method more changes have to be made to the Front-End or to the Catalogue service. In any case, the database in the catalogue must be changed to store the additional currency or relational conversion rate.	Same as *Sock Shop*, but a little more expensive, since there are dependencies that have to be checked, to see if the changed endpoints have an effect.	Same effort as *Sock Shop*
Effort estimation	5	6	5

Table 6.5: Theoretical reaction on 'Additional currency' scenario

The additional effort in *Tea Store* again comes from the critical service 'Persistence'. The higher coupling of this service gives us extra dependencies, which we have to check. Additionally, the big size of this service increases our effort needed. Both again correlates to the comparable high values of coupling and size metrics.

Sock Shop and *Robot Shop* again are very similar. Both systems need the same effort for this scenario which is reflected by the similarity of their measured metrics.

6.3 Scenarios Summary

Scenario	Sock Shop	Tea Store	Robot Shop
Add payment method	3	4	3
Add item category	1	2	1
Changeable user data	4	4	4
Additional currency	5	6	5

Table 6.6: Effort estimation summary, from all scenarios

Table 6.6 shows a quick reminder of all scenarios and effort estimations. Particularly noteworthy, when comparing the metrics to the scenario is the strong relation between the size of the services and the maintainability effort. This is also very well reflected by the $WSIC$ metric, as it always predicted about the same effort for *Sock Shop* and *Robot Shop*.
The coupling metrics had a rather small effect. From these, AIS is especially important, as it says for how many dependencies and services, changes to endpoints can affect something.
We found no evidence, that the $SIDC$ gave correct maintainability predictions, as the best value

was for the *Tea Store*, but *Tea Store* had the most effort. However it also important to note, that with the overall small services, it is hard to say that cohesion had any impact on the effort of the different scenarios.

Summarized the measured metrics, besides $SIDC$, actually give a very good impression of the 3 architectures. As all 4 scenarios have shown, the architectures *Robot Shop* and *Sock Shop* react very similarly to maintenance scenarios. This is due to the fact that both architectures have a very similar structure, which can also be seen from their design diagram (see appendix A.1). Size or granularity and coupling metrics reflect this very well as table 6.7 shows that they are fairly similar for both of these systems.

Mean	Sock Shop	Robot Shop	Tea Store
$WSIC$	3.4	4.25	10
ADS/AIS	0.9	1.25	1.83
ACS	0.9	0.5	1.33
$SIDC$	0.04	0.23	0.35
Effort estimation	3.25	3.25	4

Table 6.7: Mean metric and effort comparison of *Robot Shop*, *Sock Shop* and *Tea Store*

In addition, the metrics, except $SIDC$ give an authentic impression of the *Tea Store* architecture compared to the other two. Maintaining is a bit more complicated as the 'Persistence' service is a critical point with a number of dependencies and its size. In fact, it was not possible to find a scenario which is applicable to all three architectures and had to be mainly implemented in another service then the 'Persistence' service.

7 Discussion

In this chapter we further discuss and analyse the results and implications from chapter 5 and chapter 6. In addition, the threats to validity of the evaluation and the conclusion of this paper will be reviewed.

7.1 Discussion and Implications

Following we shortly discuss the solving and results of the intermediate goals and further discuss the outcome of the evaluations.

7.1.1 Solving the Intermediate Goals

When starting this evaluation we defined intermediate goals that had to be achieved, to perform this evaluation (see section 1.2).

- Up-to-date service-based maintainability metrics list.

  For the up-to-date metric list we performed a literature search and created a table for several service-based metrics. While these are not all metrics proposed, they give an impression for what metrics are available.

- Measurement techniques

  For measuring the different metrics we used small scripts, a tool available for Open API specification files and manual extraction for source code, design diagrams and developer documentations (see section 4). For different metrics, there are different measurement problems, which will be discussed.

- Selection of example systems

  For evaluating the metrics, we had to create a set of systems on which we measured the metrics, theoretically execute the scenarios and gathered expert assessments. Since the expert assessments were gathered from existing studies, we adopted the set of systems from these studies. They are mentioned in section 4.2. For the scenario-based analysis this set was reduced to three architectures, which have similar functionalities to find scenarios, applicable to these three architectures.

- Correlation to expert assessments

  We used the Pearson product-moment correlation coefficient to calculate correlations between metrics and expert assessments related to their maintainability property. We found a strong correlation for the size metric, some evidence of correlations with outliers for the coupling metrics and no evidence of correlation for the cohesion metric.

- Evaluating the maintainability prediction of metrics in a scenario-based analysis

  In chapter 6 we developed and theoretically executed maintainability scenarios on three architectures. The conclusion was found that the evaluated size and coupling metrics helped predict critical services in a complete system and relate to the estimated efforts.

7.1.2 Effectiveness of Evaluated Metrics

For metrics to be effective, they must not only be meaningful, but also easy to measure and easy to understand. Especially the ease of measurement suffers in microservice architecture, as the polyglot property of service-oriented architectures has a negative impact on the measurement difficulty.

Size

The $WSIC$ has shown a strong relationship with the experts rating both the complete systems, as well as the APIs. Also in the qualitative evaluation, it was able to predict the extra maintenance effort for the tea store and the approximate same effort between the robot shop and the sock shop. Another positive feature is the focus on weak points, as it can highlight critical sizes of services, as the qualitative evaluation has shown. For API specifications, we have tool support and for other languages the measurement can be easily automated via scripts. The development of tools for additional languages should also not be too complex and would be advantageous as the $WSIC$ has proven to be an **effective** metric for measuring the size and giving intake on the maintainability.

However, an important point should be mentioned, which was noticed during the evaluation of the correlation to expert assessments. Experts gave worse ratings not only for services that are too large, but also if the distribution of the complete systems seemed too fine-grained, i.e. there were to many services and the services were particularly tiny. The design philosophy of microservices says the more finely granular the better, but there should also be no overload of services. So even if the $WSIC$ correlates with the ratings, it could still be important for the future to pay attention not only to just reduce the $WSIC$, but to check also on the number of services. In our case the small $WSIC$ did not affect the maintainability as no complete systems exceeded a service number of ten, which is a manageable number.

Coupling

In the quantitative evaluation we have seen some evidence of correlations between experts and coupling metrics. Both the metrics and the expert ratings identify interdependences in systems which deteriorate the coupling. Other correlations showed evidence on agreements between metrics and expert assessments, however no correlation was significant enough to conclude to a full agreement.

In the scenarios in chapter 6 we can discover the usefulness of coupling metrics, as they can help to identify critical services in an architecture. For example, one could take the relatively high values for the 'Persistence' service in the *Tea Store* architecture as an indicator that this service is a critical point and should perhaps be split. However, the evaluated metrics have one disadvantage: Measurement. Although in this evaluation we had the advantage that all the complete systems we examined, provided design diagrams that included dependencies between services, we can predict that measurements, for systems that do not provide this information, will be expensive and little automated, with the current state of tool support. Overall we can come to the conclusion that the evaluated coupling metrics definitely can be effective to find critical services in a system and following predict maintainability, depending on the measurement.

Cohesion

For this paper, the *SIDC* had to be measured in 8 API specification files and 7 complete systems with a total of 52 services and 7 different languages. Since the *SIDC* was measured by method signatures with the help of scripts, a separate script had to be written for each language. Besides, depending on the language, scripting was not enough and manuel effort was also needed. Take the following examples:

```
1    private String getString (int param){
2        logic;
3        return value;
4    }
```

Listing 7.1: Java example of type signatures

```
1    function getString (param){
2        logic
3        return value
4    }
```

Listing 7.2: JavaScript example of type signatures

In *Java* it is easy to determine what the parameter type is and what the return type is. For *JavaScript*, one must hope that there is a high usage of descriptive comments, an API specification or other documentation. If not you either need high manual effort or runtime data to extract the *SIDC*. However, our use case of this metric was a special case and in 'real-world' use cases, you normally want to calculate the *SIDC* on one system, which reduces the effort. We have also seen some tool support for the calculation for APIs, which can be further developed to support several techniques and languages. So depending on the measured system, the measurement can definitely affect the effectiveness as we do not want an extensive amount of effort going in to the measurement.

The cohesion metric *SIDC* did very poorly in the correlation to the expert evaluations. In fact, it shows more evidence on negative agreement with the expert ratings. For the complete systems it is partly due to the very small services. With a smaller number of service operations it is harder to have many common parameters and return types than with a larger number. For this reason, no statement can be made about the cohesion and the *SIDC* from the scenarios, since the size has a stronger influence on the estimated effort than the cohesion to a certain point. With *Sock Shop* and *Robot Shop* having very small services, a bad cohesion is not noticeable and does not influence the maintainability that much. In fact, when building a correlation we have a strong agreement

($R = 0.9$ $p = 0.037$, without the two impactful outliers *Service-based Web Shop v1 and v2*) between the $WSIC$ and $SIDC$ for the complete systems. This correlation can be also found for the APIs ($R = 0.73$ $p = 0.04$), when correlating the $WSIC$ and the $SIDC$. These correlations suggest that the $SIDC$ maybe effective only from a certain size on. In summary, we can say that the cohesion metric $SIDC$ seems not effective for small services, however after proving the correlation to the size, the $SIDC$ may be a good indicator for the cohesion of bigger services. Further work is needed to evaluate this correlation and identify possible size thresholds for using the $SIDC$.

Metric	Positive evidence
$WSIC$	Strong evidence
AIS	Evidence
ADS	Circumstantial Evidence
ACS	Evidence
SIS	Evidence
$SIDC$	No possible statement.

Table 7.1: Evidence summary on the meaningfulness of evaluated metrics

Table 7.1 shows a summary of the evidence found for meaningfulness of all metrics examined. In general, we have proven with outliers, that service-based metrics can be effectively used to predict maintainability and give a good impression of the respective property of maintainability. This result is consistent with other literatures in this topic as Perepletchikov et al. also came to the conclusion that metrics are suitable means to determine maintainability, in their evaluations. Since they have evaluated different metrics, one should now be able to compile a considerable list of evaluated metrics.

7.2 Threats to Validity

In the following we will discuss factors that could be a threat to the validity of this evaluation.

Since the measurements of some metrics were partly carried out manually, it can come to deviations and errors, by the human factor. However, these errors should be completely eliminated or reduced to a minimum, as the measurements were repeated and the results were double-checked. Measurement errors may also occur due to the use of external tools which have not been validated. The use of external tools has been limited to one tool.

Since the expert ratings in 5 contain only a relatively small number of experts (10 for APIs and 11 for complete systems), each expert can have an effect on the final result. This is especially important if there are experts who have little to none experience in the field of maintainability and microservices. The experts all stated that they were at least moderate in their familiarity with service-oriented architectures, but how familiar a participant is, is valued by himself. A participant may not be as familiar as he thinks and therefore he could judge the maintainability properties of an architecture wrong, which would affect the output of the correlations.

With the quantitative evaluation in 5 we also have the problem that we have a rather small set to build correlations. Outliers, wrong measurement, wrong ratings have a bigger impact on such correlations. This threatens the validity of the found correlations and conclusions, since interpreted relations and correlations could not exist or not interpreted correlations could exist.

In the qualitative evaluation the effort of different maintainability scenarios is estimated. These estimates are carried out without using metrics, which means that the estimates could be subjective and may be incorrect. However, the incorrectness should be kept to a minimum by a consistent execution of a fixed procedure for all scenarios and architectures.

The results of this work cannot generally be mapped to the external validity, which means that not every service-based metric is necessarily effective, based on this evaluation.

8 Conclusion

The goal of this work was to find out if proposed service-based metrics can be used effectively to determine the maintainability of microservices. This chapter summarizes what steps have been taken to achieve this goal and what we have discovered. Additional some ideas for future work are presented.

8.1 Summary

The main objective of this paper was to make a statement on the effectiveness of proposed service-based metrics. For this purpose a current list of presented metrics was compiled, from which a subset was selected to represent the different properties of maintainability, i.e. coupling, size and cohesion.

In the first part of the evaluation, the metrics were compared with expert ratings based on example architectures. These expert ratings were taken from existing studies and improved with a second iteration via the *Delphi method*. In the comparison we came to the conclusion that the metrics partly correlate with the expert ratings. However, there are some outliers and special cases in which the metrics and the expert ratings differ.

In the second part of the evaluation, realistic maintenance scenarios were created on the basis of three example architectures. With these scenarios, we monitored the theoretical reaction to changes in every architecture, which were then compared with each other and an effort estimation was carried out. The observed peculiarities, differences and effort estimations were then compared with the metrics and successful or unsuccessful predictions were highlighted.

The result of these two evaluations shows that the evaluated metrics for size and coupling, can be effective to determine the maintainability of architectures, however we could not make a statement for the evaluated cohesion metric. However, additional work should be put into the measurement of the metrics, as these are partially very time-consuming and currently almost no tool support exists.

8.2 Future Work

In the course of the comparison with expert evaluations, we have come across various points in which the metrics and experts differ. A possible extension of this evaluation is the addition of more experts and systems, so that one can form even more meaningful correlations. The correlation can also be extended and performed with more metrics, as it definitely can confirm the effectiveness of metrics.

Another point is possible improvements that could be made for different metrics. For example, in chapter discussion, we talked about the problem of metrics for size and how a smaller size is only better to a certain point. Specially we mentioned the WSIC and how that metric could be improved with additional looking at component balance or the number of services in a complete system. Also there is still the common problem, that no official statement exists on the size and number of services in microservice architectures.

The last and probably most important point for the future is the measurement and collection of service-based metrics. We need to adapt to the polyglot properties of microservices and create tools and techniques that are polyglot as well. Especially for the coupling metrics, progress has to be made as for this property the metrics are mostly difficult to measure. An additional point is the runtime data collection. Since we have examined many different systems here and existing runtime tools require tracing or other different architecture additions to collect metric data, only metrics that can be calculated by static methods could be examined. An additional development of tools for the support of runtime data metrics would certainly be advantageous.

A Appendix

A.1 Architectures

A.1.1 Sock Shop

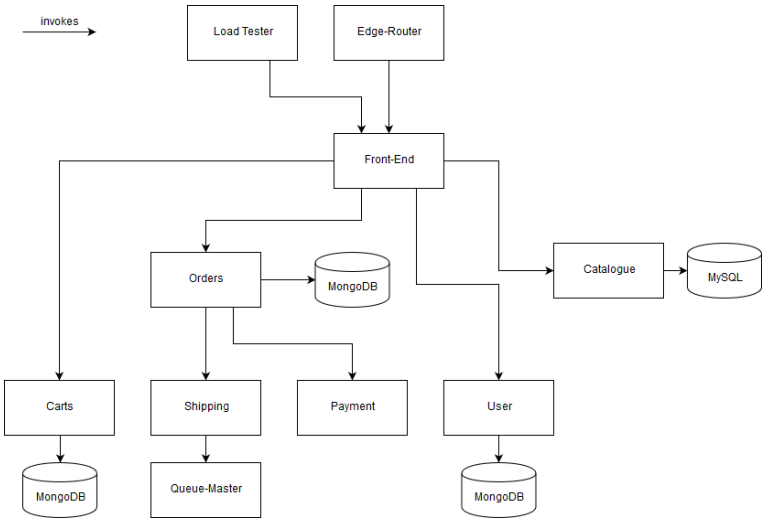

Figure A.1: *Sock Shop* architecture

A.1.2 Tea Store

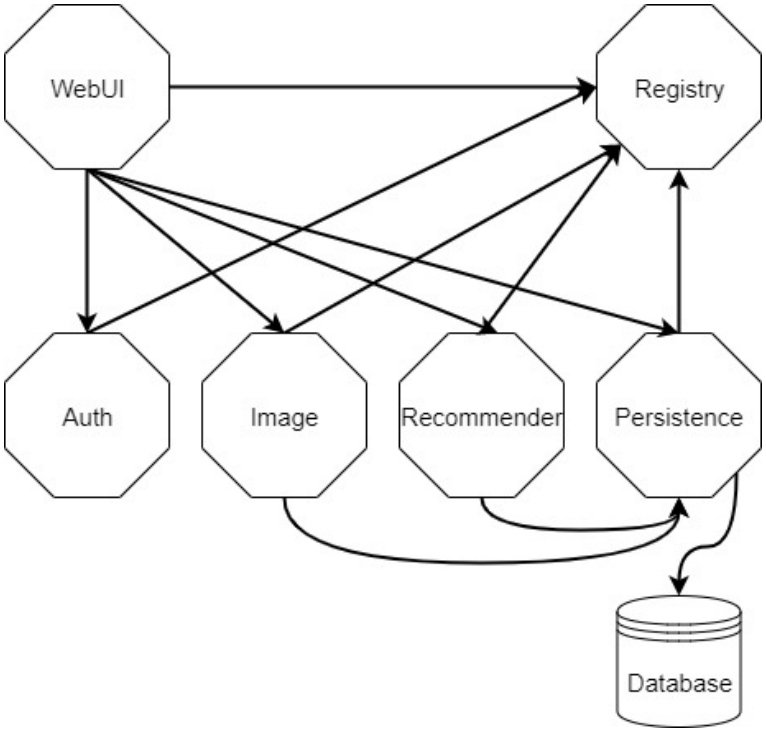

Figure A.2: *Tea Store* architecture

A.1.3 Robot Shop

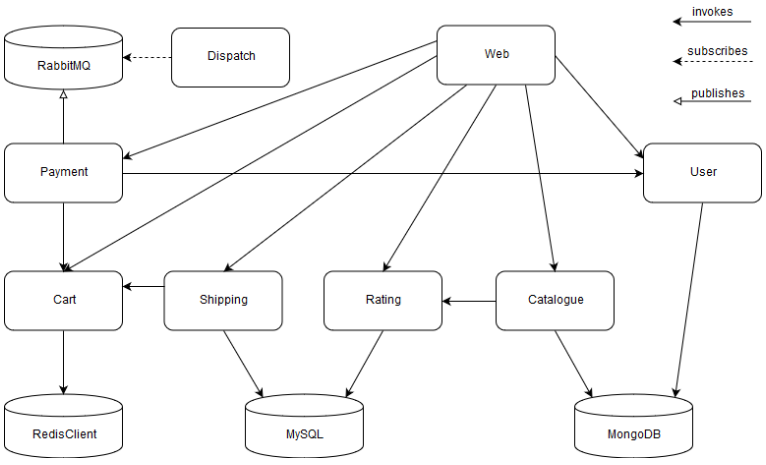

Figure A.3: *Robot Shop* architecture

Bibliography

[BDV13] E. Bouwers, A. van Deursen, J. Visser. “Evaluating usefulness of software metrics: an industrial experience report”. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 921–930 (cit. on p. 24).

[BG04] M. A. Babar, I. Gorton. “Comparison of scenario-based software architecture evaluation methods”. In: *11th Asia-Pacific Software Engineering Conference*. IEEE. 2004, pp. 600–607 (cit. on pp. 20, 21).

[BR00] K. H. Bennett, V. T. Rajlich. “Software maintenance and evolution: a roadmap”. In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 73–87 (cit. on p. 15).

[BWZ17] J. Bogner, S. Wagner, 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*. ACM. 2017, pp. 107–115 (cit. on pp. 11, 12, 25).

[DGL+17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina. “Microservices: yesterday, today, and tomorrow”. In: *Present and ulterior software engineering*. Springer, 2017, pp. 195–216 (cit. on pp. 11, 18).

[DH63] N. Dalkey, O. Helmer. “An experimental application of the Delphi method to the use of experts”. In: *Management science* 9.3 (1963), pp. 458–467 (cit. on p. 33).

[DJ03] M. Dagpinar, J. H. Jahnke. “Predicting maintainability with object-oriented metrics-an empirical comparison”. In: *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* IEEE. 2003, pp. 155–164 (cit. on pp. 11, 23, 24).

[FL14] M. Fowler, J. Lewis. *Microservices*. 2014 (cit. on pp. 19, 20).

[HCA08] M. Hirzalla, J. Cleland-Huang, A. Arsanjani. “A metrics suite for evaluating flexibility and complexity in service oriented architectures”. In: *International Conference on Service-Oriented Computing*. Springer. 2008, pp. 41–52 (cit. on pp. 17, 26).

[JLZ+18] W. Jin, T. Liu, Q. Zheng, D. Cui, Y. Cai. “Functionality-oriented microservice extraction based on execution trace clustering”. In: *2018 IEEE International Conference on Web Services (ICWS)*. IEEE. 2018, pp. 211–218 (cit. on p. 26).

[KABC96] R. Kazman, G. Abowd, L. Bass, P. Clements. “Scenario-based analysis of software architecture”. In: *IEEE software* 13.6 (1996), pp. 47–55.

[KNR15] L. Kumar, D. K. Naik, S. K. Rath. “Validating the effectiveness of object-oriented metrics for predicting maintainability”. In: *Procedia Computer Science* 57 (2015), pp. 798–806 (cit. on pp. 11, 24).

[McC76] T. J. McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320 (cit. on p. 17).

[MKPS00] S. Muthanna, K. Kontogiannis, K. Ponnambalam, B. Stacey. “A maintainability model for industrial software systems using design level metrics”. In: *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE. 2000, pp. 248–256 (cit. on pp. 11, 23).

[New15] S. Newman. *Building microservices: designing fine-grained systems*. Ö’Reilly Media, Inc.", 2015 (cit. on pp. 11, 12, 18, 19).

[OW14] J.-P. Ostberg, S. Wagner. “On automatically collectable metrics for software maintainability evaluation”. In: *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*. IEEE. 2014, pp. 32–37 (cit. on p. 17).

[Pig96] T. M. Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996 (cit. on p. 16).

[PR10] M. Perepletchikov, C. Ryan. “A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software”. In: *IEEE Transactions on software engineering* 37.4 (2010), pp. 449–465 (cit. on p. 23).

[PRF07] M. Perepletchikov, C. Ryan, K. Frampton. “Cohesion metrics for predicting maintainability of service-oriented software”. In: *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE. 2007, pp. 328–335 (cit. on pp. 23, 26).

[PRFT07] M. Perepletchikov, C. Ryan, K. Frampton, Z. Tari. “Coupling metrics for predicting maintainability in service-oriented designs”. In: *2007 Australian Software Engineering Conference (ASWEC’07)*. IEEE. 2007, pp. 329–340 (cit. on pp. 11, 17, 23, 26).

[PRT10] M. Perepletchikov, C. Ryan, Z. Tari. “The impact of service cohesion on the analyzability of service-oriented software”. In: *IEEE Transactions on Services Computing* 3.2 (2010), pp. 89–103 (cit. on pp. 23, 31).

[QX09] Z. Qingqing, L. Xinke. “Complexity metrics for service-oriented systems”. In: *2009 second international symposium on knowledge acquisition and modeling*. Vol. 3. IEEE. 2009, pp. 375–378 (cit. on pp. 11, 23, 26).

[RSD06] D. Rud, A. Schmietendorf, R. Dumke. “Product metrics for service-oriented infrastructures”. In: *IWSM/MetriKon* (2006), pp. 161–174 (cit. on pp. 23, 26, 28, 29).

[Sch19] S. Schlinger. *Benutzung von Laufzeitdaten für die Wartbarkeitsanalyse von Serviceund Microservice-basierten Systemen*. 2019 (cit. on pp. 26, 27).

[Sch87] N. F. Schneidewind. “The state of software maintenance”. In: *IEEE Transactions on Software Engineering* 3 (1987), pp. 303–310 (cit. on p. 16).

[SCKP08] B. Shim, S. Choue, S. Kim, S. Park. “A design quality model for service-oriented architecture”. In: *2008 15th Asia-Pacific Software Engineering Conference*. IEEE. 2008, pp. 403–410.

[Sta06] I. Standard. “Software engineering–software life cycle processes–maintenance”. In: *ISO Standard* 14764 (2006), p. 2006 (cit. on p. 15).

[Tek04] B. Tekinerdogan. “ASAAM: Aspectual software architecture analysis method”. In: *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*. IEEE. 2004, pp. 5–14 (cit. on p. 21).

All links were last followed on November 04, 2019.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature