Software Construction

RWTH AACHEN UNIVERSITY

The present work was submitted to the Research Group Software Construction

of the Faculty of Mathematics, Computer Science, and Natural Sciences

Bachelor Thesis

A classification approach of information needs for anti-pattern detection in microservice applications

presented by

Tim Voigt

Aachen, October 29, 2022

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Alex Sabau, M.Sc.

Acknowledgment

First and foremost, I want to thank Prof. Dr. rer. nat. Horst Lichter for the opportunity to write this thesis at the chair of the Research Group Software Construction and for reviewing my thesis. Additionally, I want to thank Prof. Dr. rer. nat. Bernhard Rumpe for reviewing this thesis as the second examiner.

Special thanks go to my supervisor Alex Sabau for the continuous and insightful feedback as well as his intriguing input on my concept. The advice and discussions were greatly appreciated.

My closing thanks go to my friends and family for their constant support throughout this thesis.

Tim Voigt

Abstract

Microservice applications are on the rise to become one of the most popular architectural styles. They offer many advantages over monolithic applications, such as being easily scalable and maintainable as well as enabling teams to work autonomously. Nevertheless, it is crucial to properly implement microservice architectures to avoid common drawbacks. Frequently used solutions that impair the quality of an application, often referred to as anti-patterns, and the detection of such in software architectures are a well-known problem in the domain of software engineering. However, in contrast to monolithic architectures, not much work has been done to improve the detection of anti-patterns in microservice architectures.

In this thesis we propose a structure and classification approach of information needs for anti-pattern detection. In this context, the concepts of information requirements and information sources are defined and explained on the basis of an exemplary microservice application. Additionally, a catalog which proposes information requirements of 20 anti-patterns is presented. Building up on this catalog, methods to satisfy these information requirements are proposed. These insights are used to present an overview of the relationship between information requirements and information sources. Overall, the concepts form a basis for further refinement and evaluation of automated detection methods of anti-patterns in microservice applications.

Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

Contents

1.1 Background

Microservices are steadily gaining popularity and are on track to become one of the most popular architectural styles. Especially for large applications, microservice architectures can be beneficial in a multitude of ways. Nevertheless, there still are numerous unsolved problems, particularly in regard of handling the added complexity microservice applications introduce. As opposed to traditional monolithic applications, few methods exist which can detect anti-patterns in microservice applications. These problems are further motivated in section 1.2. To provide a shared foundation, the concepts of *microservice architectures* and *anti-patterns* are described in the following.

1.1.1 Microservice architectures

As an alternative to traditional, monolithic software architectures, *microservice architectures (MSAs)* are rising in popularity [Dra+17]. The basic concept of a MSA is to encapsulate business logic into many small components, instead of one large application, which are easy to deploy and scale individually. Richardson [Ric18] describes this as a form of modularity. According to him, modularity is essential for developing complex applications. It can also be achieved in monolithic applications by using module-like constructs of programming languages. However, according to Richardson, the modularization of monolithic applications tends to degrade over time. Microservices have clearer boundaries, since they are autonomous services, which makes it easier to uphold modularity over time. Additionally, microservice development encourages polyglot programming practices, since the services usually communicate using lightweight protocols, such as *Representational State Transfer (REST)*-APIs and therefore do not need to share a common programming language. This allows developers to work with a programming language that is suited best for implementing the specific functionality of a microservice.

Notably, microservices can be defined by a set of significant characteristics, as highlighted by Nadareishvili et al. [Nad+16]. They are small in size, communicate with other microservices and only implement a small set of functionalities of one domain. Moreover, development happens autonomously. Microservices can be deployed independent of another and through automated processes. Finally, MSAs are decentralized by nature.

Microservices provide a range of benefits, if implemented correctly. Richardson [Ric18] outlines some of them. According to him, microservices enable teams to be autonomous, while allowing easy experimenting and the adoption of new technologies. Richardson also states that microservices have better fault isolation, as well as being easily maintainable.

However, Richardson [Ric18] also defines some drawbacks that come with implementing microservice architectures. First, it is challenging to find the right set of services, since there is no pre-defined way of decomposing a system into microservices. Additionally, the deployment of features which involve multiple microservices requires careful coordination across teams.

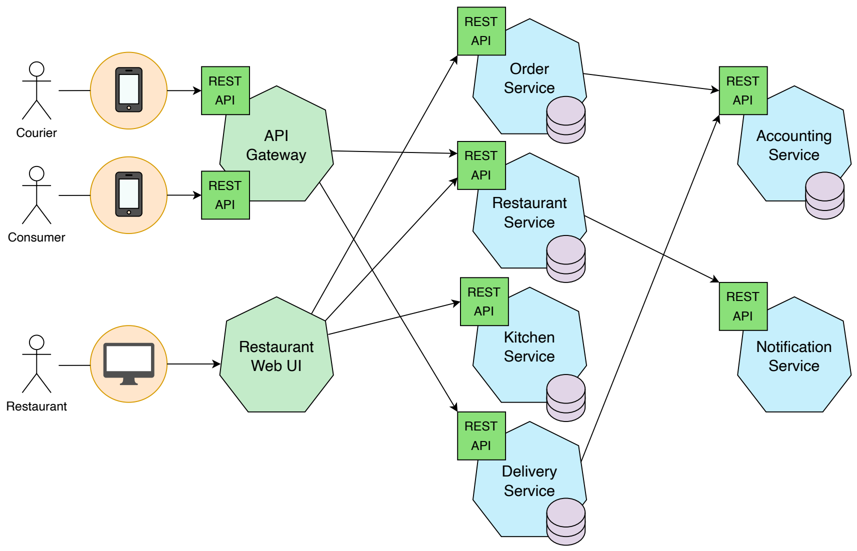

Figure 1.1: Exemplary microservice architecture based on Richardson [Ric18]

An exemplary microservice architecture is shown in figure 1.1. The example is based on a company which provides food delivery services for restaurants. Microservices are displayed as heptagons in two different colors. Those which are colored in green interact directly with users, while those colored in blue only interact with other microservices. Additionally, it is visualized that some microservices offer a REST-API for communication and some have private storage. Couriers and consumers interact with the application using mobile applications, each using their respective REST-APIs to communicate with an API-gateway. The API-gateway routes requests to the relevant microservices – behavior and importance of API-gateways will be discussed in more detail in chapter 4. Restaurants interact with the application using a web-based user interface which in turn uses three microservices to provide the required functionality.

Overall, microservices are becoming a popular architecture for large applications since they have several advantages over monolithic applications. However, it is important that MSAs are implemented properly so that the advantages are properly utilized.

1.1.2 Anti-patterns

The term anti-pattern, which was first popularized by Koenig [Koe95], describes a common solution to a recurring problem which poses risks of being counterproductive. They can impair the quality of an application, if they are not fixed. Moreover, the presence of anti-patterns in a software system has an impact on software evolvability or maintainability [Sal19]. It is important to note that the terms *anti-pattern* and *smell* are often used synonymously in literature [WDC20]. We will use the term *anti-pattern* in this thesis.

Furthermore, it has to be distinguished between architectural anti-patterns and code anti-patterns. Mo et al. [Mo+21] define architectural anti-patterns to be "connections among source files that violate design principles and impact bug-proneness and change-proneness". On the other hand, code anti-patterns, or rather code smells are defined by Fowler [Fow18] as certain structures in code that indicate trouble and can be solved by refactoring. Both types of anti-patterns are important and will be discussed in this thesis. Interestingly, Mo et al. [Mo+21] point out that many existing commercial tools focus on code-related anti-patterns, which leads to a negligence of architectural anti-patterns.

1.2 Motivation

As previously stated, microservices are becoming a popular architectural choice, especially for cloud-based systems, since properly exploit the advantages of being deployed in the cloud [Dra+17]. However, it is important to highlight that designing a MSA is not easy and a migration from a monolithic system is even more challenging [Ric18]. It is crucial to design microservices as optimal as possible, since bad architecture design can negatively impact software quality [WDC20]. Furthermore, microservices add complexity due to the increased amount of inter-service communication, central logging infrastructure and having to handle partial failure of other microservices in the architecture. In addition, it is likely that a microservice architecture will grow over time and therefore further increase the complexity of developing and maintaining microservices.

Because of the aforementioned reasons, it is important to reduce the number of anti-patterns present in a MSA to a minimum. The process of detecting anti-patterns should be straightforward and ideally automated so that it can be executed repeatedly in short intervals, for example before each new deployment of a new microservice version. Consequently, efficient methods for detecting anti-patterns in MSAs would contribute to this goal. Furthermore, automated detection of some anti-patterns, like some tools already realize for code smells, would reduce the time and effort spent on manual inspection. In general, the detection of anti-patterns can help organizations, which recently transitioned to MSAs, to actually profit from the advantages microservices offer.

1.3 Problem Statement

Contrary to anti-patterns of monolithic software architectures, little research has happened on how to quantify and analyze the quality of microservice architectures, specifically on how automated detection of anti-patterns can be realized. For this reason, we will research how anti-patterns can be detected, manually or automatically, in MSAs.

This leads us to the first research question for this thesis. To be able to detect anti-patterns, information from the microservice architecture is necessary. This information is referred to as the *information need* of an anti-pattern. We argue that the term *information need* is a highly abstract term with various facets. Hence, it should be further substantiated to meaningfully work with it. Therefore, the following question has to be answered.

RQ1: *How can the information need for anti-pattern detection in microservice applications be meaningfully structured and classified?*

While answering the previous question gives insight into the required information, the result is not sufficient to actually detect anti-patterns in MSAs. This is because the necessary information has to previously be extracted from a MSA, so that detection can be performed. Consequently, the following question has to be answered.

RQ2: *Which methods can be used to obtain the necessary information for anti-pattern detection in microservice applications?*

Finally, it is of great interest to automate as much of the detection process as possible. However, it is likely that not every anti-pattern can be detected automatically. Finding characteristics of anti-patterns which make them detectable without human interaction can help companies implement detection algorithms and provide an assessment framework for forthcoming anti-patterns. To realize this, the final research question must be answered.

RQ3: *Which anti-patterns in microservice applications can be detected automatically?*

1.4 Structure of this Thesis

Chapter 2 portrays related work for automated anti-pattern detection in monolithic- or microservice-architectures. Chapter 3 defines and explains the concepts of information requirements and information sources using two examples. In chapter 4, the first research question is answered by proposing one or more information requirements for every anti-pattern. Chapter 5 proposes methods to satisfy information requirements using information sources, answering the second research question. Chapter 6 discusses our proposals and whether the research questions were answered successfully. Finally, chapter 7 concludes this thesis and provides an outlook on possible future work.

2 Related Work

Contents

After providing information on the foundations of this thesis and identifying the current problems of detecting anti-patterns in microservice applications, we now introduce related work regarding the detection of anti-patterns. First, studies which are not targeted on microservice architectures are mentioned. They are followed by five papers which were written with MSAs as a focus. As an exception, in this chapter, the terms *smell* and *anti-pattern* are both used in this chapter to preserve the original meaning of the works.

2.1 Non-Microservice Oriented

During our research, we discovered two papers which propose detection methods for a limited amount of architectural smells. These papers do not explicitly focus on microservice architectures, but on software architectures in general – specifically on Java, C and C++ projects.

The first paper, written by Fontana et al. [Fon+17], introduces a tool called Arcan, which is able to detect three architectural smells in an automated fashion. Arcan is designed to work with Java applications. All detectable smells are related to dependency issues – cyclic dependency, unstable dependency and hub-like dependency. A main characteristic of the Arcan tool is the usage of a graph database to build a dependency graph. The authors report the workflow to detect architectural smells as consisting of four steps. First, Java classes and packages are reconstructed from compiled Java files. Next, the dependency graph is created based on the previously extracted classes. In the third step, various metrics, such Fan In or LCOM, are computed using the dependency graph. Finally, the architectural smell engine component of Arcan runs multiple detection algorithms on the dependency graph and stores found architectural smells in the graph. The tool was evaluated on two Java projects. According to the authors, Arcan detected all architectural smells present in the projects. However, five smells were reported, which after manual inspection were found to not be valid architectural smells. In summary, this work introduced a tool for detecting three architectural smells in Java projects using dependency graphs.

The second paper, by Biaggi, Arcelli Fontana, and Roveda [BAR18], extends the Arcan tool introduced in the previous paper [Fon+17] for C and C++ projects. In addition to the three architectural smells Arcan is already able to detect – cyclic dependency, unstable dependency and hub-like dependency – the authors added support for two more, namely Multiple Architectural Smells and Specification-Implementation Violation. The main components making up Arcan stayed the same. Biaggi, Arcelli Fontana, and Roveda extended them to work with C and C++ projects. Furthermore, the fourth component, which is responsible for detecting the architectural smells using a dependency graph, has been extended to enable the detection of the two added architectural smells. Moreover, the new version of the tool was validated on six open-source projects. The authors analyzed two of the six open-source projects manually for architectural smells, which was used to validate the output of Arcan. According to them, all architectural smells found by the tool in those two projects were correct. However, eight architectural smells were not detected by Arcan. To summarize, this work extended the existing Arcan tool to work with C and C++ projects and added detection methods for two additional architectural smells.

2.2 Microservice Oriented

The first work we elaborate, written by Taibi, Lenarduzzi, and Pahl [TLP20], defines a catalog and taxonomy of the most common microservice anti-patterns. The authors extracted 27 anti-patterns based on surveys and interviews of 27 industry practitioners from 27 different organizations (see table 2.1). Taibi, Lenarduzzi, and Pahl define a taxonomy which contains descriptions of 20 of their extracted anti-patterns as well as the caused problems. The seven anti-patterns not included in the taxonomy were proposed by practitioners in technical talks but were never experienced by the interviewees of Taibi, Lenarduzzi, and Pahl. The authors list them in the catalog only for completeness, but not in their taxonomy because of insufficient information. Additionally, the anti-patterns are grouped into five categories: Internal, Communication, Technical, Organizational (Team-Oriented) and Organizational (Technology and Tool Oriented). For 16 anti-patterns, detection approaches are proposed and for 11 of 20, adopted solutions are listed. Furthermore, the catalog also includes the harmfulness of each anti-pattern, as perceived by the interviewees, measured on a 10-point Likert scale. The average harmfulness is 4.81, the median is 5. Finally, the authors conclude their work by summarizing their findings and presenting five lessons learned. They also highlight that correctly splitting a monolith is the most critical issue.

The next work by Pigazzini et al. [Pig+20] is another extension of the previously introduced Arcan tool [Fon+17; BAR18]. The authors focus on detecting three microservice-specific smells, namely Cyclic Dependencies, Hardcoded Endpoints and Shared Persistence. According to them, these three smells were chosen because they are the most common and are technically easy to implement, because they are detectable through static analysis. Since Pigazzini et al. base their work on Arcan, the four main components of the tool are utilized and extended. Consequently, detection of the smells relies heavily on graph-based analysis, similar to the original tool. However, instead of using a

Name	**Harmfulness**	**Category**
Hardcoded Endpoints	8	Internal
Wrong Cuts	8	Technical
Cyclic Dependency	7	Communication
API Versioning	6.05	Internal
Shared Persistence	6.05	Technical
ESB Usage	6	Communication
Legacy Organization	6	Organizational (Team-Oriented)
Local Logging	6	Internal
Megaservice	6	Internal
Inappropriate Service Intimacy	5	Internal
Lack of Monitoring	5	Technical
No API-Gateway	5	Communication
Shared Libraries	4	Communication
Too Many Technologies	4	Organizational (Tech. & Tool Oriented)
Lack of Microservice Skeleton	3.05	Organizational (Tech. & Tool Oriented)
Microservice Greedy	3	Organizational (Team-Oriented)
Focus on Latest Technologies	2.05	Organizational (Tech. & Tool Oriented)
Common Ownership	2	Organizational (Team-Oriented)
No DevOps Tools	2	Organizational (Tech. & Tool Oriented)
Non-homogeneous adoption	2	Organizational (Team-Oriented)
Lack of Service Abstraction	–	–
Magic Pixie Dust	–	–
Microservices as the Goal	–	–
Pride	–	–
Sloth	–	–
Timeout	–	–
Try to Fly Before You Can Walk	–	–

Table 2.1: Anti-pattern catalog by Taibi, Lenarduzzi, and Pahl [TLP20]

dependency graph, the authors use a call graph to enable the support of microservices. Furthermore, detection of the Hardcoded Endpoints and Shared Persistence smells is based on static analysis of source code files. Finally, Pigazzini et al. manually validated their tool using four Java and one JavaScript open-source microservice projects. The tool successfully identified multiple instances of smells in the projects, however the authors did not calculate recall values. In summary, Pigazzini et al. modified the existing Arcan tool to enable the detection of three microservice-specific smells, in applications developed solely in Java or JavaScript, using static analysis.

Every work discussed so far, excluding that of Taibi, Lenarduzzi and Pahl, uses some form of metrics to detect anti-patterns. In their paper, Ntentos et al. [Nte+21] propose a model-based approach to measure conformance to microservice patterns using only generic, platform-independent metrics. The authors define eight metrics in total to measure three main architectural decisions – external API decisions, persistent messaging for inter-service communication decisions and end-to-end tracing decisions. Notably, Ntentos et al. performed intensive manual analysis on 24 microservice-based systems to define a ground truth for later evaluation and to acquire necessary values to calculate the metrics. Moreover, the authors conducted an ordinal regression analysis to derive a predictor variable for an ordinal variable. They report a high level of accuracy. Summarized, Ntentos et al. proposed a set of metrics to judge the conformance of a microservice-based system to microservice-specific patterns. The calculation of these metrics requires a high amount of manual analysis.

The next work by Walker, Das, and Cerny [WDC20] proposes the use of static analysis to detect 11 microservice-specific code smells in MSAs. The 11 code smells are taken from the previously introduced catalog by Taibi, Lenarduzzi, and Pahl [TLP20]. Walker, Das and Cerny introduce an open-source tool called MSANose, which is platform-specific and only works for Java and Java Enterprise applications. According to them, extending the tool to other languages would be trivial. Furthermore, they highlight the importance of analyzing the whole architecture, instead of focusing on individual microservices. Moreover, the authors analyzed state-of-the-art architecture-specific code smell detection tools, including Arcan [Pig+20], to check whether they are able to detect the considered 11 code smells. Only Arcan successfully detected three out of 11 code smells. Walker, Das and Cerny describe in detail each detection method for every microservice code smell. Additionally, the detection methods are manually validated using two microservice-based Java applications. The authors report accurate detection of code smells for both applications. Notably, three detection methods are highlighted by the authors to be validity threats, since the corresponding code smells are almost impossible to reliably detect using static analysis. To conclude, Walker, Das and Cerny developed a new tool called MSANose which is able to detect 11 microservice-specific code smells in Java applications using static analysis.

In their work Hübener et al. [Hüb+22] demonstrate the detection of five microservice-specific anti-patterns using dynamic analysis. The authors rely on distributed tracing, which is a subfield of dynamic analysis, because static analysis is not feasible in their scenario, because they do not have access to source code artifacts. The traces are used as a basis to generate a microservice dependency graph. Hübener et al. then use metrics defined in other works to detect the anti-patterns Megaservice, Nanoservice, Bottleneck-service, Ambiguous-service and Cycle-dependent-service, based on the previously generated dependency graph. Notably, domain knowledge is needed to make sense of the metrics. The authors' approach is demonstrated using a commercial environment, specifically on an architecture containing 426 microservices, which is part of the infrastructure of the Dutch bank ING. In summary, the authors demonstrated how five microservice-specific anti-patterns can be detected by means of analyzing distributed traces.

Up to our knowledge no work has been done on structuring and categorizing the information needs of anti-patterns in MSAs. We will build on the work of Taibi, Lenarduzzi, and Pahl [TLP20] to propose a solution to this problem.

3 Concepts

Contents

3.1 Structuring Information Needs

In order to successfully detect anti-patterns, specific information about various aspects of the observed MSA is necessary. It is not limited to information about application-related aspects, such as code files, but also includes business and domain related aspects [CT22; WDC20]. For the rest of this thesis, the information required to detect an anti-pattern will be referred to as an information requirement, which is defined as follows.

Definition 3.1.1 *An* ***information requirement*** *describes the necessary information required by an algorithm or human to perform anti-pattern detection in microservice architectures.*

Another important concept for anti-pattern detection, which is directly related to the information requirement, is the information source. It is defined as follows.

Definition 3.1.2 *An* ***information source*** *contains information which is able to satisfy one or more information requirements. An information source can be any artifact from a microservice application which is either human- or machine-readable, such as source code or an architecture model. Information can be extracted without human involvement from a machine-readable information source using various methods – like static analysis.*

Figure 3.1 visualizes how anti-patterns, information requirements and information sources interact with each other. An anti-pattern must have one or multiple information requirements associated with it. It is not possible to have a detectable anti-pattern without any associations, since some information from the MSA is always necessary for detection. Similarly, one information requirement can be associated with multiple anti-patterns. In the following, we will refer to this association by saying that an *anti-pattern has an information requirement.* Conversely, an information requirement *belongs* to an anti-pattern, if an association exists between these two. Furthermore, information requirements always require one or multiple information sources in order to be *satisfied.* In case that such an information source does not exist, the information requirement will not be satisfiable and in turn the anti-pattern is not detectable. However, such a case should normally not occur, since information sources can be chosen from MSA artifacts without hard constraints, such as being machine-readable.

Additionally, information sources can be grouped into four main types of information depending on the context the information source is taken out of – domain, business, application and runtime information. These categories will be discussed in more detail later on. As stated in definition 3.1.2, necessary information has to be extracted from the microservice architecture artifacts using manual or automated methods. A manual extraction process is easier, as a human can deal with a plethora information in different formats, in most cases without requiring instructions on how to process them. Compared to an automated process, it is however more costly and requires more effort, since human labor is involved and processing will likely take more time. On the contrary, automated methods can only process machine-readable information sources. In addition, algorithms are necessary to detect anti-patterns based on satisfied information requirements. They utilize the previously extracted information and perform additional computational steps, like unifying multiple information requirements if required and finally judging whether an anti-pattern is present.

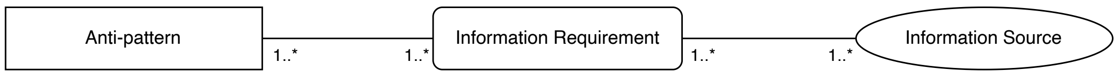

Figure 3.1: Proposed structuring of information needs

In the following, two examples will be given using an open-source demonstration application. The purpose of these examples is the illustration of how a detection of anti-patterns in MSAs can be realized. Additionally, the examples demonstrate our proposed structuring of information needs. The examples are structured as follows. First, the considered anti-pattern is described, followed by a definition of corresponding information requirements and information sources. Finally, methods for extracting the information requirements from the information sources, if possible, as well as methods for detecting the anti-pattern based on the extracted information are proposed.

Demo Application The open-source demonstration application used in the following examples is called "Hot R.O.D." and was created by the Jaeger tracing platform[1]. The application emulates a ride on demand service, where a customer can order a car to a specific location. As seen in figure 3.2, the application consists of four microservices – frontend, customer, driver and route. Due to the fact that this is just a demonstration application, the two storage backends (MySQL and Redis) are mocked. The arrows depict communication between the microservices. All services are written in Go and utilize external libraries, such as Jaeger[2] for tracing or Zap[3] for logging.

[1]`https://github.com/jaegertracing/jaeger/tree/main/examples/hotrod`
[2]`https://github.com/jaegertracing/jaeger`
[3]`https://github.com/uber-go/zap`

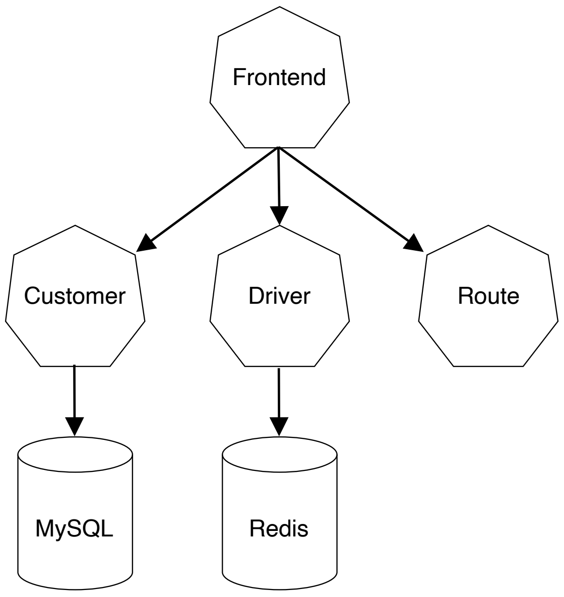

Figure 3.2: Architecture of “Hot R.O.D.” application

Example 1 – Shared Libraries This example is concerned with detecting the *Shared Libraries* anti-pattern. It describes the usage of shared libraries between different microservices, with the focus on in-house (internal) libraries. Sharing libraries between different services is problematic, since this tightly couples the microservices which in turn leads to a loss of independence between them. Additionally, increased coordination is required between teams, if the library is supposed to be modified [TLP20; WDC20].

As mentioned previously, all microservices in the demo application are written in Go. For this reason, we introduce Go specific concepts which are necessary to understand the following definitions for the information requirements and -sources. In Go, a *library* is referred to as a *package*[4]. Such a package is constructed from one or more source files. These source files contain import declarations at the top of each file (see listing 3.1). These import declarations state the dependencies of each file, which are references to other packages. The dependencies can be packages from Go’s standard library, internal packages or packages available from other repositories on the internet. As an example, the first three dependencies in listing 3.1 are packages from the standard library. The following three are external packages and the last two are packages belonging to the “Hot R.O.D.” application (internal packages). The dependency graph of all four microservices of the application is visualized in figure 3.3.

Dependencies on packages in the standard library as well external packages are grouped for the sake of clarity. Because of this simplification, it is impossible to see if shared dependencies on packages in the standard library or external packages are present. However, this is not problematic since the focus lies on internal packages, which are not

[4]`https://go.dev/ref/spec#Packages`

grouped. By looking at the graph, it can easily be seen that shared dependencies on multiple internal packages are present.

We assume that a microservice consists of one or multiple packages, as is the case for the demonstration application. With this knowledge, it is intuitive that the *information requirement* for this anti-pattern are the *dependencies of each microservice.*

```go
// Copyright (c) 2019 The Jaeger Authors.
// Copyright (c) 2017 Uber Technologies, Inc.

package customer

import (
    "context"
    "fmt"
    "net/http"

    "github.com/opentracing-contrib/go-stdlib/nethttp"
    "github.com/opentracing/opentracing-go"
    "go.uber.org/zap"

    "github.com/jaegertracing/jaeger/examples/hotrod/pkg/log"
    "github.com/jaegertracing/jaeger/examples/hotrod/pkg/tracing"
)

//...
```

Source Code 3.1: Dependencies of client.go from customer package

Next, it has to be evaluated which artifacts of the MSA can be used to satisfy the information requirement. The result of this evaluation will be the information source. In order to acquire all dependencies of each microservice, the import declarations in the source files can be utilized. The objective is to have a list of dependencies for each microservice. This can be achieved in two steps. First, all imported packages of each source file have to be extracted. Then, the results have to be merged on a per-microservice basis by looking at which source file belongs to which microservice. Since every necessary information is stored in the source files, they are defined as the information source.

Processing all source files of every microservice can be time-intensive depending on the size of the microservice architecture. In Go, we can make use of a different information source which allows for a more efficient detection. This approach utilizes the Go modules, which are defined in `go.mod` files. Because Go's module system is complex, only the necessary basics are introduced. A module is defined as a collection of packages. In the following, for the sake of simplicity, we assume that a microservice consists of exactly one module. Microservices consisting of multiple modules can be analyzed using this approach as well. However, as will be explained later, it is not possible to analyze microservices that are part of the same module – which applies to the demo application. All external dependencies of a module are specified in the module definition using the

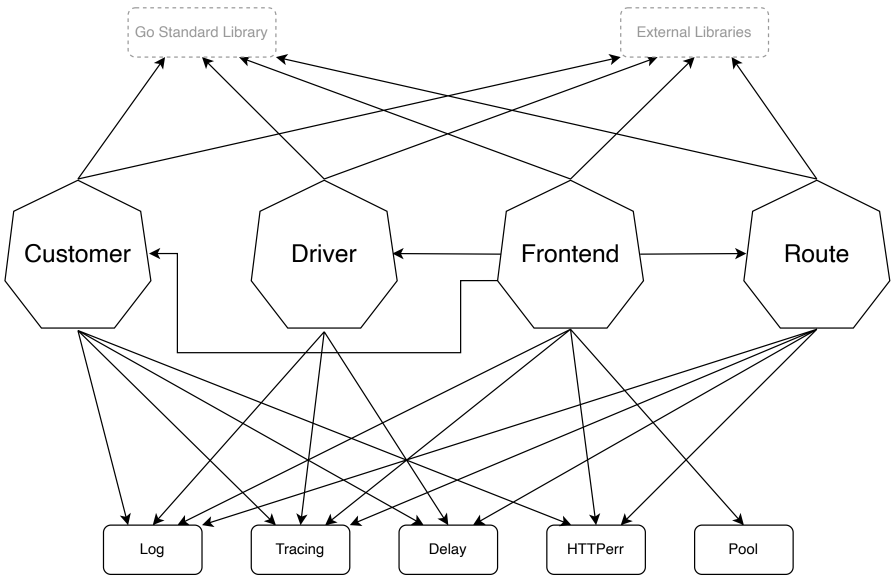

Figure 3.3: Simplified dependency graph of the "Hot R.O.D." application

`require` keyword – as seen in listing 3.2. External dependencies include every package which is not part of the standard library or the module itself. This information is still granular enough to satisfy the information requirement, since shared dependencies on internal packages are not causing the problems the anti-pattern defines. Therefore, the information source of this alternative approach is the `go.mod` file.

Using the Go module definition to extract all dependencies is faster than processing every source file because only a single file has to be parsed. Additionally, no dependency is encountered more than once, as can be the case when using source files as information source. As consequence, post-processing to remove duplicates is not necessary.

It is important to note that relying on the `go.mod` file is not possible for the demonstration application. This is due to the microservices consisting of individual packages – not modules – which are part of the bigger Jaeger tracing system module. As a consequence, the microservices do not have their own module definitions which could be used for analysis. Furthermore, basing the analysis on the `go.mod` file of the Jaeger tracing system module would result in incorrect outcomes, since dependencies unrelated to the demo application are contained in that file as well. In summary, source files of the "Hot R.O.D." application, are defined as the *information source.*

```
module github.com/jaegertracing/jaeger

go 1.19

require (
    //...
    go.opentelemetry.io/collector v0.57.2
```

```
        go.opentelemetry.io/collector/pdata v0.57.2
        go.opentelemetry.io/collector/semconv v0.57.2
        go.opentelemetry.io/otel v1.9.0
        go.uber.org/atomic v1.9.0
        go.uber.org/automaxprocs v1.5.1
        go.uber.org/zap v1.22.0
        //...
    )
    //...
```

Source Code 3.2: Excerpt of Jaeger's go.mod file

After having identified the information source of this anti-pattern, we will now take a look at possible extraction methods for this information source. The dependencies can be extracted manually by simply checking every source file. However, we want to achieve this using automated methods for increased efficiency. One possible approach is to use *static analysis* to detect all import declarations, similar to Go's built-in `go vet` tool. In case of the "Hot R.O.D." application, the analyzer would have to recursively run through every Go source file in the `services` directory, except for those in the `config` directory, yielding 16 lists of dependencies. The `config` directory is ignored because it does not contain a microservice but instead stores configuration parameters for the microservices. As described in the previous paragraph, the resulting lists have to be grouped by corresponding microservice. Every source file belonging to the same package is part of the same microservice. For example, there are four source files belonging to the `customer` package, which constructs the customer microservice. This grouping procedure has to be executed for every analyzed source file. Afterwards, all lists in the same group have to be merged into one combined list. While merging, it is important to remove duplicates, so that only unique values are stored in the combined lists. After acquiring these lists, the information requirement is satisfied.

Figure 3.4 illustrates how the merging process works for all source files belonging to the customer microservice. It is important to note that only internal dependencies are considered in this example, as they are the most relevant. However, the procedure is the same for every type of dependency. As previously described, every source file belonging to the `customer` package is analyzed on imported packages. These imports are then merged into a final list, which describes every dependency the customer microservice has, purged from duplicates. This process would have to be repeated for every remaining microservice. The following paragraph describes how shared libraries can then be detected on the basis of these microservice dependency lists.
To detect the anti-pattern based on the microservice dependency lists, an algorithm has to discover dependencies that occur in more than one of these dependency lists – those are the shared libraries. Ideally, standard packages, such as `fmt` or `net/http` are not considered, since it is very likely that those are shared and no harm is caused by that. The result of this analysis is that multiple shared libraries are present in this demo application. This includes the logging package "`go.uber.org/zap`" or internal packages like "`github.com/jaegertracing/jaeger/examples/hotrod/pkg/log`" which are used

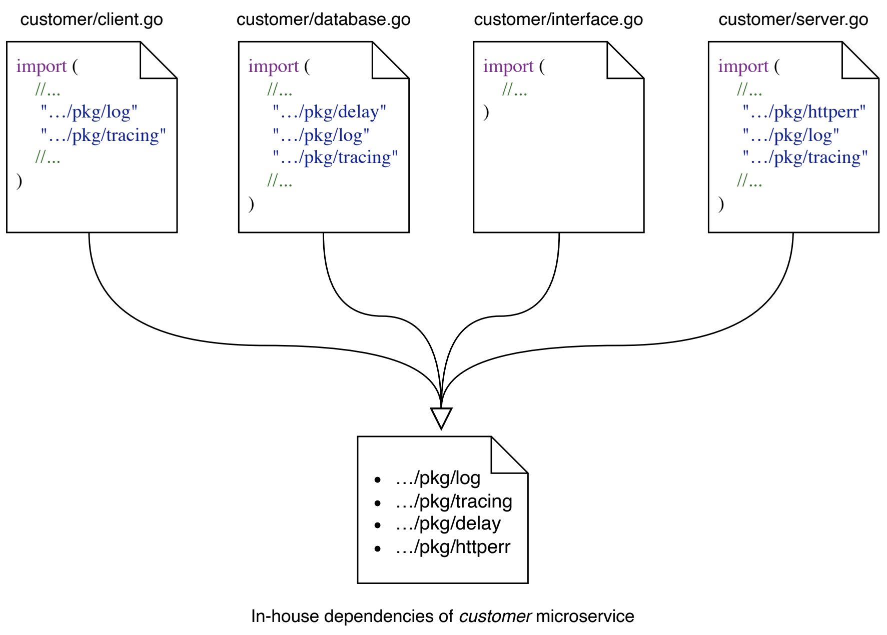

Figure 3.4: Merging dependencies of the *customer* microservice

as abstractions or factories in all microservices. It is expected that these libraries are used in every microservice, since a logging component is necessary if best practices should be adhered to. Nevertheless, the anti-pattern is detected, and human evaluation is needed to determine whether these shared libraries are indeed problematic.

In summary, shared libraries have successfully been detected by first defining the information requirement, followed by the information source and finally methods to satisfy the information requirement and to detect the anti-pattern using this information.

Example 2 – Legacy Organization This example is concerned with detecting the *Legacy Organization* anti-pattern. The anti-pattern describes the problem of a company, which transitioned to microservice architectures, not adapting their processes and policies to MSAs. An example for this are common releases, which are scheduled for every team, although having independent development teams. This causes the problem of developers having to conform to the old processes and policies, instead of benefiting of the introduced microservices [TLP20].

Defining an information requirement for this anti-pattern is similarly straightforward as in the previous example. In order to detect whether a company did not update their processes to take advantage of microservice architectures, we must be able to analyze them. Therefore, we define "*Processes and Policies of the Organization*" as the information requirement.

Next, the information source has to be defined. A noteworthy difference to the information requirement defined previously is how this information requirement can be satisfied. Because the information requirement is inherently focused on the organization instead of the MSA, business knowledge is required to satisfy it. The concept of business knowledge is defined in section 3.2. Using artifacts of the demo application as information source would not yield the information needed to satisfy the information requirement, because processes or policies are not stored in the application. Consequently, *business knowledge* is defined as information source for this anti-pattern.

For the sake of providing this example, we assume that business knowledge is not available in a machine-readable format. Thus, satisfying the information requirement requires manual processing. If the anti-pattern is supposed to be detected automatically, the information has to be processable by an algorithm. To achieve this, a human has to parse relevant business knowledge and store all processes and policies in a machine-readable format. This step is not trivial and can be time-intensive depending on the amount of processes and policies present in the organization as well as the accessibility of required business knowledge. In turn, maintaining an up-to-date version of the machine-readable data requires repeated effort. In any case, having processes and policies accessible in a machine- or exclusively human-readable format satisfies the information requirement.

As the last step, the anti-pattern has to be detected on the basis of the satisfied information requirement. Detection can either happen manually or in an automated fashion. In case it is done manually, a human has to flag processes and policies which have not been adapted to microservice architectures based on own reasoning or previously defined rules. If processes or policies which conform to these criteria, therefore being not adapted, are found, the anti-pattern is present. On the other hand, detecting the anti-pattern using an algorithm, is likely more challenging and requires a higher up-front effort. This is because developing a system which performs automated detection likely requires more work than performing a one-time manual analysis. An algorithm has to be able to evaluate whether a process or policy is adapted to MSAs or not. Currently, it is unclear how this can be achieved.

In summary, it can be said that detecting the legacy organization anti-pattern would increase the quality of a given MSA, since it is classified as harmful by multiple industry practitioners [TLP20]. However, detecting it automatically is very challenging due to the difficulty of converting business knowledge to a machine-readable format, as well as judging it using an algorithm [Hep+05; OLe98].

3.2 Information Sources

Information sources form the base for the detection of anti-patterns – as seen in figure 3.1. They are artifacts from which information for detection is extracted (see definition 3.1.2). This extraction process can be manual or automated. As stated in the definition, automated extraction is only possible for machine-readable information sources. Whether information sources are human- or machine-readable depends on how, and in which format the artifacts are stored.

Furthermore, during our research, we discovered that information sources can be categorized into four classes depending on the contained information: domain-, business-, application-, and runtime-knowledge. Notably, the term *knowledge* is deliberately chosen instead of *information* because the four classes describe a larger collection of information, which is why delineating the terms is useful.

Domain knowledge describes knowledge about a specific *domain*. According to Evans [Eva04], a *domain*, in the context of software engineering, is the subject area to which a program is applied. Moreover, some domains involve the physical world, e.g. an airline-booking service, while some are intangible, e.g. an accounting service [Eva04]. Regarding the extraction of domain knowledge, *domain experts* are crucial, as will be seen later. Vernon [Ver13] describes *domain experts* as "people who know a subject area really well and are likely to have a background in the business domain".

Business knowledge has many varying definitions. The Northern Ireland Business Info [Nor] describes it as "a sum of skills, experiences, capabilities and insight which you collectively create and rely on in your business". However, the following definition will be used as reference throughout this thesis, since it better fits the context of software architectures. Ross [Ros] defines it as "the total set of business concepts, their organizing connections, and the business rules upon which the existence of the business depends".

We define ***application knowledge*** to be knowledge about a microservice application and the artifacts it contains. The knowledge can be extracted from application-related files, such as source code, configuration- or log files. The primary method of accessing this knowledge is through static analysis.

Finally, we define ***runtime knowledge*** to be knowledge which can only be obtained when observing a running system (i.e. microservice architecture). Consequently, knowledge which is covered by this term is strictly related to information only available at application runtime. This primarily includes connections between microservices in the context of this thesis. Runtime knowledge can be accessed using variations of dynamic analysis.

Figure 3.5 displays the seven main information sources used in this thesis, along with their categorization. Each class contains a different type of information and requires different methods of extraction. Each information source is colored to highlight the appropriate extraction method.

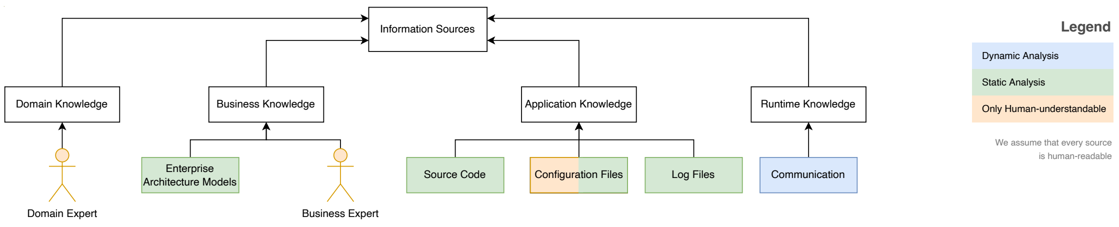

Figure 3.5: Overview of information sources

The knowledge of experts, be it domain- or business-knowledge, is not available in a machine-readable format, which is why a manual extraction process, is necessary. However, if an expert of business knowledge formulates his knowledge in the form of *enterprise architecture models*, which are stored in a machine-readable format, it can be extracted using automated methods – specifically through static analysis. While no standard definition for *enterprise architecture* exists, the consensus is that it, among other aspects, contains strategic information which defines the business along with the information and technology necessary to run it [SML17]. Furthermore, existing approaches by Boer et al. [Boe+05], Naranjo, Sánchez, and Villalobos [NSV15], and Sabau, Hacks, and Steffens [SHS21] show the feasibility of running static analysis methods on enterprise architecture models to extract information.

Due to its nature, application knowledge is stored in a machine-readable format. A plethora of approaches exist which extract information from source code or log files by means of static analysis [Lou06; CT22]. However, automated analysis of configuration files is more challenging. Because of near infinite naming possibilities of configuration variables, it is not easily possible to automatically judge what information is stored in them. Reliable automated analysis of such configuration files would require some form of mapping from variables to type of information – such a mapping would likely require human involvement. However, many configuration files are constructed using a *domain-specific language*, which follow a declared grammar or syntax. Docker's compose files[5] are an example for such a configuration file. Consequently, if an algorithm is designed to analyze Docker compose files – therefore making it platform-specific – it is certainly possible to extract information without human involvement. Hence, it depends on the type of configuration file and the algorithm used to analyze it, whether the configuration file is only human-readable or machine-readable as well. For this reason, we highlight configuration files as a combination of being machine-readable by means of static analysis and being only human-understandable.

Finally, by definition, runtime knowledge focuses on running microservice architectures. Therefore, approaches using dynamic analysis have to be used to extract information about the communication between microservices, since this information is not available in static files.

Detailed extraction methods are discussed in chapter 5. In summary, this section introduced seven information sources, belonging to four main knowledge categories, which will be used throughout this thesis to detect the anti-patterns defined by Taibi, Lenarduzzi, and Pahl [TLP20].

3.3 Information Requirements

As stated in definition 3.1.1, information requirements are a key component for the detection of anti-patterns. This section gives an insight into the information requirements discovered during our research and states how an information requirement can be *satisfied*.

During our research, we were able to propose 20 information requirements by investigating what information is necessary to detect the anti-patterns from the catalog of Taibi, Lenarduzzi, and Pahl [TLP20]. Table 3.1 lists all 20 information requirements

[5]`https://docs.docker.com/compose/compose-file/`

alongside a description of the contained information. The term "contained information" denotes the information necessary to *satisfy* an information requirement. At the same time, contained information is the information humans or algorithms are able to use if an information requirement is *satisfied*.

An information requirement is called *satisfied* if the underlying requirement is fulfilled. For example, the information requirement "Accessed Storage" is satisfied, if a list of every persistent storage medium a microservice accesses is present. It is irrelevant whether it is present in only human- or in machine-readable form. To satisfy information requirements, previously introduced information sources are used. As will be presented in chapter 5, each information requirement has one or more information sources associated with it. Information has to be extracted from the associated information sources in order to satisfy the information requirement. This procedure is exemplified in section 3.1 and follows the same pattern for every pair of information requirement and information source.

While analyzing the discovered information requirements, it became apparent that 10 out of 20 focus on technical information while the other half focuses on business-related information. Technical information denotes information which can be extracted using only application knowledge. On the other hand, business-related information has to be extracted from domain- or business knowledge. Moreover, it was noticed that eight anti-patterns have more than one information requirement. Consequently, the same holds true for the information requirements (see figure 4.1). The anti-patterns that have more than one information requirement are likely to have at least one business-related information requirement. This shows that the detection of some anti-patterns requires more information than others.

Name	**Description**
Accessed Storage	Lists every persistent storage medium (e.g. a database) that a microservice accesses.
Application Dependencies	Lists all dependencies of a microservice on other microservices which are part of the same MSA. A microservice A is dependent on a microservice B, if A sends requests to B.
API Endpoints	Lists all fully qualified API paths (i.e. endpoints) a microservice offers.
External Endpoints in Static Files	Contains information about which endpoints are specified in which static files.
Imported Libraries	Lists all libraries a microservice imports.
Inter-Microservice Communication	Contains detailed information about the communication between microservices of the same MSA. Should include every request and response with corresponding headers.

Table 3.1: Description of information requirements

Name	**Description**
Logging Target	Describes the location where logs of a microservice are stored at. This can, for example, be a log file stored in the local file system of a microservice or a central log aggregator. A microservice can have multiple logging targets.
Microservice Functionalities	Describes business capabilities (i.e. functionalities) implemented by a microservice. Microservices can implement one or more functionalities.
Microservice Owners	Contains the team or individual responsible for a microservice – referred to as the owner.
Owned Storage	Contains a mapping of persistent storage units (e.g. databases) to microservices. The mapping should be injective, meaning every storage is mapped to exactly one microservice. A storage is mapped to a microservice, if it is designed to belong to it.
Presence of Central Monitoring Architecture	Describes whether a central monitoring solution exists, which monitors the metrics of microservices.
Presence of Company-Wide Frameworks	Describes whether a company offers frameworks or boilerplate code which can be used by any team developing microservices.
Presence of Metrics Library or Health Endpoint	Describes whether microservices use a metrics library to log their metrics or if microservices offer a health-endpoint. A health endpoint is an API endpoint which offers health-related information.
Processes and Policies of the Organization	Lists every development-related process and policy of an organization.
Shared Code Fragments	Lists fragments of code which are used in multiple microservices.
Technologies Used	Lists every technology (e.g. programming languages, protocols, etc.) used by a microservice.
Technology Selection Motives	Describes why a technology was chosen to be used in the application.
Transmitted Version Information	Contains API version information transmitted in request or response headers. E.g. the value of `Content-Type` or `Accept` headers.
Usage of CI/CD Pipelines	Contains information about which teams or individuals use continuous integration and continuous deployment pipelines.
Versioning Schema	Describes the versioning schema mandated by the organization, if any.

Table 3.1: Description of information requirements (continued)

4 Anti-pattern Information Requirement Catalog

Contents

After the introduction of anti-patterns in chapter 1 and a proposal to structure their information needs in chapter 3, this chapter deals with the definition of corresponding information requirements. In section 4.1, the results of our analysis are presented in the form of a catalog. Section 4.2 states the main validity threats of the analysis.

4.1 The Catalog

The analysis is based on the 20 anti-patterns listed in the taxonomy by Taibi, Lenarduzzi, and Pahl [TLP20] (see table 2.1). The last seven anti-patterns in table 2.1 were extracted by Taibi et al. from technical talks by practitioners and are not part of the taxonomy. They will not be used in this thesis because these talks do not provide enough information to propose methods for automated detection. Moreover, these anti-patterns were also not explored in other literature, from which more information about them could have been derived.

Determining which information is needed for the detection of anti-patterns in MSAs requires an in-depth analysis of each anti-pattern. The following method was used for our analyses. First, we searched in existing literature for approaches which are able to detect a specific anti-pattern in MSA. Information requirements can then be derived from the necessary input of these approaches. If there was no literature found fulfilling these criteria, the search was expanded to include literature which is not targeted for MSA but traditional, monolithic software architectures. In case the expanded search yields no results as well, information requirements were defined by ourselves. This analysis was conducted for 20 of the 27 anti-patterns defined by Taibi, Lenarduzzi, and Pahl [TLP20] in their paper, which was introduced in chapter 2. Seven anti-patterns were not considered for the reasons mentioned previously. The anti-patterns are ordered by descending perceived harmfulness scores, which is the same order used in the taxonomy by Taibi, Lenarduzzi, and Pahl.

Hardcoded Endpoints

Anti-pattern Description This anti-pattern is also known as *Hardcoded IPs and Ports* and describes the problem of using hardcoded IP addresses or hostnames and ports to connect to other microservices [TLP20]. This practice is problematic because it defeats some advantages of MSA. One such advantage is the ability to have a high-availability deployment of a microservice, which means that multiple replicated instances of the microservice are running simultaneously. Should one of those instances become unavailable, traffic can be redirected to the other instances without impacting the overall availability of the microservice. If such a high-availability deployment of a microservice is accessed by another microservice through hardcoded endpoints, it is impossible to quickly redirect the traffic to replicated instances.

By definition of the term “hardcoded”, endpoints have to be stored in static files, such as source code. Furthermore, they can be stored in different formats, namely IPv4, IPv6 or as a string (hostnames). Each variant can be extended with the specification of a port number. The following examples are all valid endpoints: “`127.0.0.1:42, ::1:42, database`”. However, it is important to point out that the usage of IPv6 for communication in microservices is still rare, but possible. It was first introduced to *Kubernetes* in version 1.9[1].

Detection Methods in Literature Two existing approaches for detection were found in literature focused on MSAs. Pigazzini et al. [Pig+20] try to identify hardcoded endpoints using regular expressions to match IPv4 addresses with ports in source code. This approach can be extended to detect IPv6 addresses as well, however it will not be able to detect hostnames reliably, except if they are known in advance. Walker, Das, and Cerny [WDC20] propose a platform-specific method which utilizes the bytecode of a microservice. To be precise, they observe which parameters are passed into functions used to connect to other microservices. This approach seems promising, since it is able to detect all three formats of hardcoded endpoints. Notably, it will not work with all common programming languages, since some are not compiled and therefore have no bytecode.

Definition of Information Requirements In case it is possible to acquire every hostname of the microservices in the architecture, the approach of Pigazzini et al. should be able to accurately detect the presence of this anti-pattern. Consequently, source code and hostnames of every microservice are required for detection. The corresponding information requirement is titled **“External Endpoints in Static Files”**.

[1] `https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.9.md#ipv6`

Wrong Cuts

Anti-pattern Description One of the hardest problems when migrating from a monolithic architecture to a MSA is splitting the monolith into independent microservices [CBD18; TS20]. If not done properly, the result is a MSA without proper separation of concerns as well as increased data-splitting complexity [TLP20]. This anti-pattern, which is also known as “wrong separation of concerns”, is likely one of the hardest to detect using automated methods, due to the amount of domain knowledge necessary for detection.

Detection Methods in Literature Walker, Das, and Cerny [WDC20] state that detecting wrong cuts is nearly impossible without having a deep understanding of the business domain. However, they still propose a method for detecting this anti-pattern by looking for unbalanced distributions of artifacts within the microservices. According to the authors, unbalanced artifacts are a possible consequence of wrong cuts. Notably, they highlight their detection method as one of the main validity threats. For this reason, and because in the case study conducted by the authors, the anti-pattern was only correctly identified one out of two times, we will not refer to this method any further. In another paper, Pigazzini et al. [Pig+20] further underline the difficulty of detection. It would require analyzing different business processes as well as the architectural structure of the whole architecture along with further information. In addition, the authors state that research is being done to identify decomposition strategies and methods to detect proper decomposition for MSA. They do not propose a detection method themselves because of the aforementioned reasons.

Definition of Information Requirements In conclusion, the detection of this anti-pattern heavily depends on domain knowledge and no automated methods for reliable detection have been found in current literature. Therefore, we propose the following information requirement for this anti-pattern: **“Microservice Functionalities”**.

Cyclic Dependency

Anti-pattern Description This anti-pattern, which is also known as “Circular Dependency”, is one of the most well-known of this catalog as it already has been defined for monolithic architectures. Cyclic dependencies are present if there exists a loop of dependencies – a circular chain of imports – between modules. If we have two modules A and B, a cyclic dependency would look like this $A \rightarrow B \rightarrow A$ (A imports B and B imports A). Furthermore, there exist much more complex cyclic dependency patterns, for the detection of which graph-based algorithms have proven useful [Al-+14]. However, when working in the context of microservice architectures, dependencies between microservices themselves, not individual modules, are decisive for triggering this anti-pattern [TLP20]. Cyclic dependencies are bad practice in such architectures, because it increases the effort needed for maintenance and makes individual microservices harder to reuse in isolation.

Detection Methods in Literature Pigazzini et al. [Pig+20] utilize a call graph between microservices to detect this anti-pattern. They use a platform-specific approach to generate the call graphs, which is targeted at Java projects utilizing the Spring framework. Specifically, the algorithm tries to identify every HTTP request made to other microservices based on the `RestTemplate` class. This approach should be extendable to other programming languages and frameworks as further analysis only depends on the call graph. The authors use their own tool, called "Arcan", to generate the dependency graph based on a call graph. Cyclic dependencies can then be detected either manually, by looking at a graphical representation, or automatically using graph-based algorithms. Walker, Das, and Cerny [WDC20] use a similar approach to the one previously mentioned. They also generate a call graph based on REST API calls, the detection of which is not further documented. Next, they run their cyclic dependency detection algorithm, which is a modified depth-first search, on this graph. In another paper by Hübener et al. [Hüb+22] a detection method using distributed tracing is proposed. This approach was used because, in the scenario described by the authors, acquiring the source code artifacts of all microservices is infeasible. Therefore, Apache Kafka[2] was used to infer dependencies between microservices based on the time-stamped invocations of services. The resulting dependency graph can subsequently be analyzed automatically to detect cyclic dependencies.

Definition of Information Requirements All three papers propose interesting approaches for detecting this anti-pattern. Likely, the two approaches based on static analysis should be preferred since they do not require a running application to perform the detection. Consequently, this allows scanning for cyclic dependencies before the source code is executed in a production environment, which can help to increase maintainability. Therefore, we propose the following information requirement for this anti-pattern: **"Application Dependencies"**.

API Versioning

Anti-pattern Description Most microservices of an architecture communicate with each other using REST APIs – other communication protocols like gRPC also exist. It is common practice to semantically version these endpoints so that consumers of an endpoint are not forced to immediately upgrade if breaking changes are necessary. As the name suggests, REST APIs use API endpoints, which are uniquely identified by URIs. In this case, a version can be defined as part of the URI. Another option, which is often used for protocols that do not define URIs, is including the version in the transmitted headers. There are multiple best-practices for semantically versioning an API, which will

[2] `https://kafka.apache.org/`

be discussed in a later chapter. This anti-pattern, which Richards also refers to as "Static Contract Pitfall" [Ric16], is triggered if endpoints of microservices are not semantically versioned. As already touched upon, if a non-versioned API is updated, the consumers may face connection issues or encounter other problems related to a changed data model of input or return data [TLP20].

Detection Methods in Literature Tighilt et al. [Tig+20] list three hints to the presence of this anti-pattern: "(1) microservices endpoint URLs do not contain version numbers, (2) no custom header information are sent by the client, (3) multiple microservices have similar names". The first two hints are rather intuitive, however one could argue that the third hint should be treated with caution, since microservices can have similar names for multiple reasons and this does not directly indicate a lack of API versioning. Nonetheless, these indicators provide a good base for detection of this anti-pattern. An approach which is based on the first hint can be found in the paper of Walker, Das, and Cerny [WDC20]. They match every fully qualified API path offered by a microservice against a regular expression pattern which detects semantic versioning in the URL path. This is a straightforward approach which has the disadvantage that the pattern has to be customized to the utilized versioning format if it does not conform to standards. The authors only propose this method for fully qualified API paths, however it can be extended to check request and response headers as well. For a completely reliable detection of this anti-pattern, knowledge about how versioning has been implemented is ultimately required. This is because there are almost infinite possibilities of formatting version information in both URIs and headers – assuming best practices are not adhered to. If this information is not present, heuristics can be utilized to detect the lack of API versioning in a subset of possible cases.

Definition of Information Requirements Summarizing the above, we need multiple types of information to reliably detect whether API versioning is not used for specific microservices. First, API endpoints of the microservice act as the base for the analysis. In case that version information is transmitted in the header, this information will be used as well. Finally, if an accurate analysis is indispensable, the API versioning schema should be utilized. Therefore, we propose the following information requirements for this anti-pattern: **"Endpoints"**, **"Transmitted Version Information"** and **"Versioning Schema"**.

Shared Persistence

Anti-pattern Description As previously discussed, one of the main benefits of MSAs, if implemented correctly, is the low coupling between services. It is very common for microservices to have some sort of persistence layer, such as relational databases. This anti-pattern describes the problems that occur when different microservices access the same database or, even worse, the same entities of the same database. If shared persistence is present between multiple microservices, it is certain that these services become tightly coupled and in turn the team- and service-independence is reduced [TLP20]. Therefore, it is important that every microservice has its own persistence layer.

Detection Methods in Literature Tighilt et al. [Tig+20] define three symptoms that characterize this anti-pattern: "(1) multiple microservices share the same configuration files and deployment environments, (2) database tables are prefixed, or (3) databases have a lot of schemas". These symptoms can be good indicators of the presence of shared persistence, however they are formulated rather unspecific and broad. Pigazzini et al. [Pig+20] propose a detection method which operates on a similar idea as the first defined symptom. On a high level, their approach searches for all database references or usages in every microservice. Using this information, the detection of shared persistence is straightforward, since it can quickly be detected if multiple microservices access the same database. However, this only detects access to the same databases and cannot check if the same entities are accessed. Furthermore, the approach is platform-dependent, as the authors rely on specific configuration files of the Java Spring framework to detect connection strings to databases. This method is also used in a paper by Walker, Das, and Cerny [WDC20].

Definition of Information Requirements Multiple other detection methods are thinkable, which do not use configuration files but log files as a source for finding database accesses. These will be discussed in the sections concerned with the corresponding information requirements. Therefore, we propose the following information requirement for this anti-pattern: **"Accessed Storage"**.

ESB Usage

Anti-pattern Description An *Enterprise Service Bus (ESB)* is a central component used in service-oriented architectures and is mainly used to connect enterprise services with each other, independent of the utilized communication protocols. ESBs should however not be used in microservice architectures since it highly increases coupling, because it is a central component. This means that all traffic runs through the bus which creates a single point of failure. Furthermore, it adds complexity for registering and deregistering services [TLP20].

Detection Methods in Literature Because the ESB is designed to be a transparent component, it is rather difficult to detect reliably. Walker, Das, and Cerny [WDC20] propose a detection method using communication information. They define three criteria using which an Enterprise Service Bus can be detected on the basis of incoming and outgoing connection of every module in the application. A module needs to have a high, outlier like, number of total connections, with the number incoming and outgoing connections being balanced and connections to almost every other module in order to be flagged as a potential ESB. It is unclear how the authors managed to obtain the connection information, since the paper is focused on static analysis and information such as incoming or outgoing connections are usually acquired using dynamic analysis.

Definition of Information Requirements Regardless of whether automated or manual analysis of the architecture is conducted, the communication between microservices has to be observed to allow for analysis. Therefore, we propose the following information requirement for this anti-pattern: **"Inter-Microservice Communication"**.

Legacy Organization

Anti-pattern Description This anti-pattern, which is also known as the "Red Flag Law" [Ric] has already been analyzed in detail in section 3.1. For the sake of completeness it is mentioned here nonetheless. When a company migrates from a monolithic architecture to a microservice architecture, it must transition its processes and policies related to development to fit to MSA as well. If this does not happen and developers are still required to follow the unadjusted processes, they cannot benefit of the newly introduced architecture change [TLP20]. According to Taibi et al., an example for a legacy organization is that teams still have to schedule common releases although having independent development teams.

Detection Methods in Literature No detection approaches have been found in existing literature.

Definition of Information Requirements Detection of this anti-pattern relies solely on business knowledge. Therefore, we propose the following information requirement for this anti-pattern: **"Processes and Policies of the Organization"**.

Local Logging

Anti-pattern Description Due to the distributed nature of MSA, getting a complete overview of all logs becomes increasingly difficult with the number of microservices present. For this reason, some type of central log aggregation is typically used to unify all logs in one location. Local logging describes the problem of logs only being stored locally in each microservice and not being aggregated. This can result in errors staying hidden inside of microservice containers because these logs are not being analyzed frequently enough. Furthermore, if, for example, a user-facing error has to be investigated, a developer has to access each involved microservice individually, access and merge those logs to reenact why the error occurred. This comes at the expense of time and requires the developer to have access privileges on every required machine.

Detection Methods in Literature We have not found any literature which proposes detection methods for this anti-pattern. However, Tighilt et al. [Tig+20] define symptoms that occur if no log aggregation is being performed: "(1) the presence of log files inside microservices, (2) files being written by the microservice, (3) the usage of time aware databases, and (4) logging frameworks and tools" They provide good initial pointers to artifacts that should be searched to check for this anti-pattern. This is especially helpful for manual analysis, although an automated approach can be based on these indicators but likely needs more detailed and machine-readable information.

Definition of Information Requirements Local logging is probably one of the anti-patterns most difficult to detect because a variety of different information is necessary to reliably determine if logs are aggregated. This is because aggregation can happen in multiple ways, such as letting microservices directly log to a central location using a dedicated library or writing logs to a local file which is periodically scraped by another service which assumes the responsibility of uploading them. Therefore, we propose the following information requirement for this anti-pattern: **"Logging Target"**.

Megaservice

Anti-pattern Description One important aspect of developing MSAs is correctly designing the scope of microservices. If one service ends up having multiple non-trivial functionalities and resembles a monolith, it is referred to as a megaservice. Such services should certainly be avoided as they can result in maintenance issues, reduced performance and increased testing difficulty [Tig+20]. In addition, the advantages of microservice architectures are not utilized properly [TLP20].

Detection Methods in Literature We have found multiple papers which deal with the detection of such megaservices. Taibi, Lenarduzzi, and Pahl [TLP20] suggest that services which implement multiple business processes, are composed of multiple modules and are developed by multiple teams or developers are characteristic for megaservices. These indicators require mainly organizational information. Tighilt et al. [Tig+20] focus solely on technical information. They propose a high number of modules, files and lines of code along with an above average number of incoming requests as possible symptoms of megaservices. Detection of this anti-pattern is also feasible using distributed tracing according to Hübener et al. [Hüb+22]. In their paper, they mention megaservice as one of the anti-patterns detectable using metrics on a dependency graph. However, they neither define these metrics in the paper itself nor the referenced ones. Presumably the authors relied on the total number of dependencies in their generated dependency graph for detection. A service on which an unusual high number of other services depend could be classified as a megaservice. This approach is extendable to static analysis methods, as long as dependency graphs can be generated. Esas [Esa22] utilized a simpler approach by manually counting the number of endpoints of each microservice. Classification as megaservice would occur if an outlier-like number of endpoints were counted.

Definition of Information Requirements Judging from the previously quoted literature, detection can be based on either technical- or organizational-information. The usage of organizational-information is likely to provide the best results, while also being the most difficult to process automatically. Therefore, we propose the following information requirements for this anti-pattern: **"Microservice Owners"**, **"Microservice Functionalities"** and **"Endpoints"**.

Inappropriate Service Intimacy

Anti-pattern Description As discussed previously, in the *Shared Persistency* paragraph, it is important that every microservice only accesses its own private persistence layer. This anti-pattern is related to shared persistence, as it describes microservices which connect to private data from other services in addition to its own data. This behavior increases coupling between microservices and indicates a possible mistake made during modelling [TLP20].

Detection Methods in Literature Taibi et al. propose a detection method which is based on requests from one microservice to the private data of another. This primarily includes requests to private databases. If a microservice performs such requests, it can be argued that it violates service intimacy. Walker, Das, and Cerny [WDC20] utilize a variant of their shared persistence detection method. Compared to the original method, an instance of inappropriate service intimacy is only detected if a microservice accesses a shared database while having its own private persistence layer.

Definition of Information Requirements In contrast to the shared persistence anti-pattern, further information is required for detection. In order to judge whether microservices have their own private persistence layer, storage accessed and owned by the microservice is considered. Therefore, we propose the following information requirements for this anti-pattern: **"Accessed Storage"** and **"Owned Storage"**.

Lack of Monitoring

Anti-pattern Description A lack of monitoring, which is also referred to as insufficient monitoring, causes multiple problems and leads to a decrease in quality of the MSA. If a service becomes unavailable, and it is not monitored, developers will likely not notice this in a reasonable timeframe without periodically checking each microservice manually [TLP20]. Furthermore, individual service performance and failures are not tracked if monitoring is not used [Tig+20].

Detection Methods in Literature No literature was found that proposes concrete methods for detecting this anti-pattern. However, Tighilt et al. [Tig+20] provide indications for a lack of monitoring: the use of *local logging* for some microservices and the absence of health check endpoints. A health check endpoint[3] is a dedicated endpoint which exposes health related information. They can be one of the main sources of information for monitoring systems or orchestrators like Kubernetes to check if a service is alive.

Definition of Information Requirements From the information mentioned above, it can be deduced that several pieces of information are required for detection. First, it should be checked whether the local logging anti-pattern is present as suggested by Tighilt et al. [Tig+20]. Next, knowledge of whether a metrics library or health endpoint is used improves detection quality. Finally, it is necessary to know whether a central monitoring architecture is present to judge whether there exists a monitoring solution which can be used by microservices. Therefore, we propose the following information requirements for this anti-pattern: **"Presence of Metrics Library or Health Endpoint"** and **"Presence of Central Monitoring Architecture"**.

No API Gateway

Anti-pattern Description API gateways[4] are an important component of complex microservice architectures. They offer multiple functions, such as abstracting the internal structure for consumers or providing authorization. Furthermore, if an API gateway is present, microservices do not have to talk to each other directly – which can cause problems if a service becomes unavailable – but can communicate over the gateway, which can act as a load balancer if required. Taibi et al. discovered that their interviewees reported communication and maintenance issues if the MSA consisted of over 50 interconnected services and no API gateway was present [TLP20].

Detection Methods in Literature Tighilt et al. [Tig+20] define two symptoms of this anti-pattern. According to them, a consumer application sending multiple requests to different URLs, as well as systems with multiple frontends are indications that no API-Gateway is used. The term "system" likely refers to the whole microservice application in this case. These symptoms are extended by Taibi, Lenarduzzi, and Pahl [TLP20], who propose direct communication between microservices as an indicator. Walker, Das, and Cerny [WDC20] bring up the important point that detecting this anti-pattern is not necessarily possible using only code analysis methods. The authors refer to the increased usage of cloud-based routing frameworks, like AWS API Gateway[5], which are not detectable using static code analysis, to support this claim. Because their paper is concerned only with static analysis methods, they propose no detection methods.

Definition of Information Requirements Every mentioned symptom requires some form of information about the communication in a MSA. Therefore, we propose the

[3]https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring
[4]https://microservices.io/patterns/apigateway.html
[5]https://aws.amazon.com/de/api-gateway/

following information requirement for this anti-pattern: **“Communication between Services (Incoming and Outgoing)”**.

Shared Libraries

Anti-pattern Description This anti-pattern has already been analyzed in detail in section 3.1. For the sake of completeness it is mentioned here nonetheless. As previously mentioned, shared libraries between different microservices are problematic, because the coupling increases and independence decreases. This can reduce the advantages of microservice architecture. Additionally, coordination between teams is required, if the shared library should be modified [TLP20].

Detection Methods in Literature Walker, Das, and Cerny [WDC20] propose a detection method which utilizes dependency management files. Their approach is to scan those files for each application module to locate shared libraries. An important aspect the authors highlight is that in-house libraries should be the focus of this analysis, as some overlapping use of libraries is almost inevitable.

Definition of Information Requirements Every possible approach to detect this anti-pattern has to use some form of information about each service’s dependencies. Therefore, we propose the following information requirement for this anti-pattern: **“Imported Libraries”**.

Too Many Technologies

Anti-pattern Description One of the main advantages microservice architectures offer is the ability to develop using different languages for one microservice application. Polyglot programming allows to use the strengths of each programming language where they are necessary. However, if too many technologies are adopted for an application, this can have negative effects, especially if a new developer joins the team [TLP20]. To prevent this, this anti-pattern defines the problem of using too many technologies, which includes programming languages, protocols, frameworks and more [TLP20].

Detection Methods in Literature Walker, Das, and Cerny [WDC20] describe important challenges of detecting too many technologies. First, there is no common definition of the amount of technologies regarded as too many. Such a definition would not be sensible, since different companies and applications have different requirements. Therefore, a threshold should be defined manually [WDC20]. The detection method defined by the authors counted the standards used for every application layer (presentation, business and data), with the option to configure a threshold for every layer. An important note is that Walker et al. highlight this method as one of the main validity threats, because the correctness of this method is up to interpretation.

Definition of Information Requirements This anti-pattern is challenging to detect for multiple reasons. First, no precise definition of what the term "technology" entails in this context was found in current literature. As a consequence, it is not obvious which technologies should be considered for detection. Furthermore, the developer's reasons for choosing a technology should also be considered, since a technology selected only because of recent popularity is more likely to cause problems long-term than an established technology at the company. Therefore, we propose the following information requirements for this anti-pattern: **"Technologies Used"** and **"Technology Selection Motives"**.

Lack of Microservice Skeleton

Anti-pattern Description A microservice skeleton is a form of boilerplate code, which can help developer teams to focus on the business logics of their microservices. The skeleton, as the name suggests, contains the basic source code required to set up a microservice using the company's infrastructure, for example the connection to an API gateway [TLP20]. In case such a boilerplate does not exist, developers have to program new microservices from scratch which increases development time as well as the likelihood of errors occurring [TLP20].

Detection Methods in Literature No detection approaches were found in existing literature.

Definition of Information Requirements One possible approach to detect a microservice skeleton is to look for similarities between different microservices. If every service has, for example, the same directory structure with some identical files or code fragments, it is likely that shared boilerplate code was used. Nonetheless, it is still possible that the microservices were developed from scratch every time. Therefore, it would be helpful to use business knowledge in order to figure out whether a company-wide skeleton or framework for microservices exists. Therefore, we propose the following information requirements for this anti-pattern: **"Presence of Company-wide Frameworks"** and **"Shared Code Fragments"**.

Microservice Greedy

Anti-pattern Description This anti-pattern describes the behavior of creating new microservices for small features, which do not require a microservice of their own, like serving static HTML pages [TLP20]. Taibi, Lenarduzzi, and Pahl also state that having too many microservices in an architecture can have a negative impact on the overall maintainability. Therefore, it should be carefully considered whether new microservices are needed.

Detection Methods in Literature The only method we found in recent literature is proposed by Walker, Das, and Cerny [WDC20]. This approach is similar to those defined by the same authors for the wrong cut anti-pattern, and relies on counting different microservice artifacts. These artifacts include source code files, service objects and entity objects [WDC20]. Outliers, meaning microservices with too few counted artifacts, are regarded as fulfilling the anti-pattern. It is important to highlight that the authors highlight this method as a validity threat – analogous to the detection method for the wrong cuts anti-pattern.

Definition of Information Requirements From the previously mentioned approach, as well as the description given by Taibi, Lenarduzzi, and Pahl [TLP20], it can be deduced that the functionalities of each microservice are decisive when detecting the microservice greedy anti-pattern. Microservice functionalities can be extracted from the business capabilities a microservice implements. Although implemented business capabilities do not include basic microservice functionalities, like connecting to other microservices, the extracted functionalities suffice to detect this anti-pattern. This is because the basic microservice functionalities are included in every microservice, even if they offer only a small "greedy" feature. Therefore, we propose the following information requirement for this anti-pattern: **"Microservice Functionalities"**.

Focus on Latest Technologies

Anti-pattern Description According to Taibi, Lenarduzzi, and Pahl [TLP20], this anti-pattern describes the problem of focusing on the adoption of new and fashionable technologies when migrating to microservice architectures, instead of focusing on required technologies. As a result, the migration to microservice architectures will not solve existing problems and likely contain infrastructure which is not actually necessary [Ric; TLP20].

Detection Methods in Literature No detection approaches were found in existing literature.

Definition of Information Requirements Defining an information requirement for this anti-pattern is challenging, since it is hard to detect if technologies are only chosen because they are fashionable and new, or because they are necessary. Because of this, the motives for selecting a technology should be regarded when analyzing whether the anti-pattern is present. If no motive, besides the technology being fashionable or new, exists, it is likely that the focus of the migration to MSA lies on using the latest technologies. Therefore, we propose the following information requirements for this anti-pattern: **"Technology Selection Motives"** and **"Technologies Used"**.

Common Ownership

Anti-pattern Description One advantage microservice architectures offer, is allowing multiple development teams to independently develop and deploy new features and microservices without coordinating with other teams. The anti-pattern "Common Ownership" describes a MSA in which every microservice is owned by one development team [TLP20]. According to Taibi, Lenarduzzi, and Pahl [TLP20], the company will thus not benefit from the development independency MSAs enable.

Detection Methods in Literature No detection approaches were found in existing literature.

Definition of Information Requirements If the microservice owners are known, detecting if one development team or individual developer owns all microservices is straightforward. Therefore, we propose the following information requirement for this anti-pattern: **"Microservice Owners"**.

No DevOps Tools

Anti-pattern Description This anti-pattern describes the lack of continuous delivery and continuous deployment (CI/CD) tools [TLP20]. It is also referred to as "No Continuous Integration / Continuous Delivery" [Tig+20]. Because no CI/CD pipelines are present, developers have to manually perform tests and manually have to deploy the microservices. This can result in decreased productivity and may increase errors made during deployment due to the lack of automation, according to Taibi, Lenarduzzi, and Pahl [TLP20]. Furthermore, the lack of DevOps tools prevents utilization of a key advantage of microservice architectures. CI/CD pipelines allow independent teams to test and deploy their microservices in small and frequent iterations. This is not possible for monolithic applications which involve multiple teams.

Detection Methods in Literature No detection approaches were found in existing literature. However, Tighilt et al. [Tig+20] define four symptoms of this anti-pattern: "(1) no version control repositories on microservices, (2) no unit/integration/functional tests, (3) no automated delivery tools, or (4) no staging environments". These symptoms cannot guarantee a successful detection but can be used to develop a method that utilizes them. Furthermore, information requirements can be defined based on the symptoms.

Definition of Information Requirements If we know whether CI/CD pipelines are used by development teams, we can judge whether this anti-pattern is present. Therefore, we propose the following information requirement for this anti-pattern: **"Usage of CI/CD Pipelines"**.

Non-homogeneous Adoption

Anti-pattern Description This anti-pattern is also known as “Scattershot adoption” [Ric] and describes development teams within an organization transitioning to microservice architectures without central coordination and infrastructure. This is a problem because the effort of, for example, building infrastructure or deployment pipelines is duplicated across teams. Furthermore, some development teams may not have the skills necessary to build the infrastructure required for microservices [TLP20; Ric].

Detection Methods in Literature No detection methods were found in existing literature.

Definition of Information Requirements This anti-pattern could potentially be detected by comparing the amount of teams already using microservices to those who do not. If there is a discrepancy between the two amounts, it is likely that non-homogeneous adoption is happening in the organization. Moreover, the existence of multiple deployment pipelines or unshared microservice-related infrastructure could also indicate the presence of this anti-pattern. Therefore, we propose the following information requirements for this anti-pattern: **“Microservice Owners”** and **“Usage of CI/CD pipelines”**.

4.2 Threats to Validity

Limitations of this analysis have to be stated. All information requirements have been proposed on the basis of related works and our own concepts.

First, it is important to highlight that the proposed information requirements have not been evaluated yet. In regard to the related works we used, it has to be noted that the threat of biases exists, since some works utilize or refer to the research of other literature we used as well. In addition, only a limited number of related works was used.

Moreover, no unique solution regarding the proposal of information requirements exists, since the definition of them is a fundamentally qualitative process.

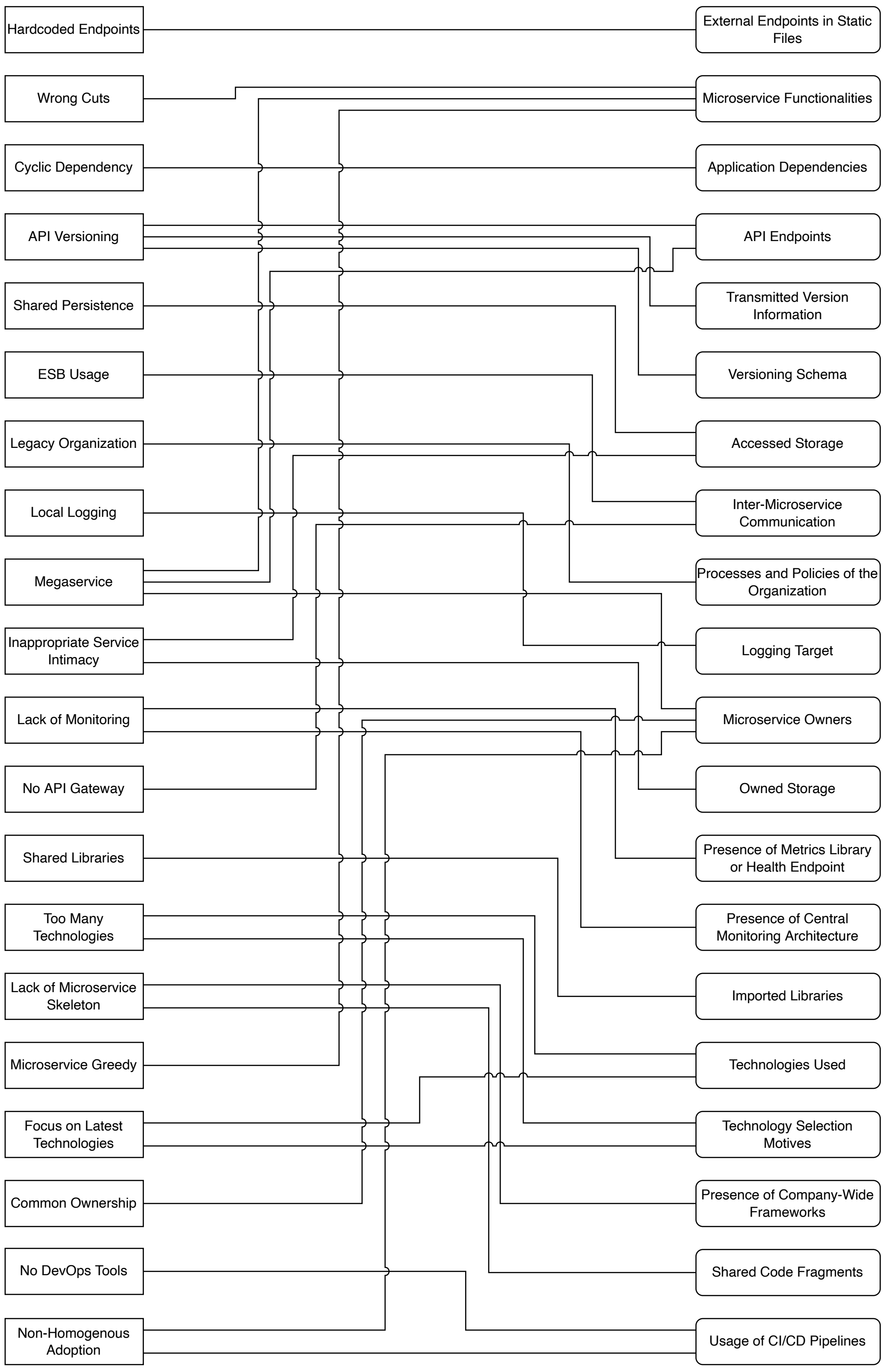

Figure 4.1: Relationship between anti-patterns and information requirements

5 Extracting Information from Microservice Architectures

Contents

After defining information requirements for every anti-pattern, this chapter proposes extraction methods to satisfy these information requirements. Furthermore, the relationship between information requirements and information sources is visualized and discussed.

5.1 Proposal of Information Extraction Methods

In chapter 3, information requirements and information sources have been defined and explained. Moreover, in chapter 4, we have assigned information requirements to each anti-pattern. In this section, we propose methods to satisfy these information requirements based on the previously defined information sources.

Such methods are highly dependent on the microservice architecture and organization in the context of which they are utilized. For this reason, the methods proposed below are not guaranteed to extract the wanted information under every possible circumstance. In addition, it can be challenging to satisfy business-related information requirements, since business- or domain-knowledge has to be present in machine-readable information sources. One such machine-readable information source containing business-knowledge has been introduced in section 3.2 – enterprise architecture models. Notably, before utilizing them, the user should evaluate whether enterprise architecture models actually represent the current state, since that is not guaranteed – for example due to architecture erosion [PB21; Li+22; RW11]. As consequence of using an eroded enterprise architecture model, the extracted information will not reflect the actual state of the enterprise architecture and could therefore cause inaccurate detection results.

The research method we used to discover the extraction methods is quite similar to that used in chapter 4. First, we determined whether extraction methods for the specific type of required information already exist in other works. If only broadly-related methods were found, a transfer to the specific context of microservices was performed. If no existing methods were found, we proposed our own, based on existing literature and considering where the information requirement is present in microservice architectures.

In the following, information extraction methods for every information requirement are proposed. They are grouped by the respective information requirement and sorted by alphabetical order.

Accessed Storage If this information requirement is satisfied, a list of every persistent storage medium that each microservice accesses is present. *Persistent storage mediums* can, for example, be databases or key-value stores.

Multiple approaches are thinkable to extract this information. First, static analysis methods can be used to detect calls to well-known libraries used to interface with persistent storage mediums. Such libraries could be popular third-party libraries, such as SQLAlchemy[1], or official libraries provided by the persistent storage medium vendor, for example the Redis library for Python[2]. This approach presupposes information about the utilized libraries and persistent storage mediums. Furthermore, the final list would include every possible access to persistent storage mediums, independent of whether it is used during runtime, due to the use of static analysis. The information source in this case is **source code**.

Another approach is to analyze access log files from persistent storage mediums. This approach has two main requirements: the persistent storage medium must log every access, which includes the accessed resource as well as the request origin, and the observed microservices are either currently active or were active before analysis. These ensure that the necessary information is present. For example, PostgreSQL databases can be configured to log the executed SQL query, as well as the remote host name or IP address[3]. Analyzing these log files yields the information necessary to satisfy the information requirement, as it can be traced which microservice accessed which persistent storage medium. The utilized information source are **log files** of the persistent storage mediums.

Finally, as an alternative to the two approaches already mentioned, the communication between microservices and persistent storage mediums can be used as information source. The idea is that if a microservice accesses data from a persistent storage medium, it executes a TCP request – all mainstream persistent storage mediums currently communicate using a proprietary protocol on top of TCP. If such a request is detected, the targeted persistent storage medium is added to the list of accessed storage for that microservice. This idea implies that an executed request always stands in context of accessing data. This approach is likely the most elaborate and, additionally, does not yield detailed information about the accessed resource. The utilized information source is the **communication** between microservices and persistent storage mediums.

Application Dependencies This information requirement is satisfied, if every dependency of a microservice is known. Dependencies, in this case, refer to dependencies on

[1]`https://www.sqlalchemy.org/`

[2]`https://pypi.org/project/redis/`

[3]`https://www.postgresql.org/docs/9.0/runtime-config-logging.html#GUC-LOG-LINE-PREFIX`

other microservices, not dependencies of the source code on libraries.

Hübener et al. [Hüb+22] demonstrate how to extract microservice dependencies in the form of a dependency graph using distributed tracing. Distributed tracing belongs to the group of dynamic analysis methods. The authors used Apache Kafka as their source for the traces, however other sources, like OpenTelemetry[4] are possible as well. The information contained in a dependency graph satisfies the information requirement, since dependencies of every microservice belonging to the application are visible. Distributed traces can be seen as a subclass of overall communication in a MSA. Therefore, the utilized information source is the **communication** between microservices of the application.

API Endpoints This information requirement is satisfied if all API endpoints that a microservice offers are known. According to Ibsen and Anstey [IA18], an API endpoint is “an abstraction that models the end of a message channel through which a system can send or receive messages”.

If the source code of a microservice can be accessed, static analysis can be used to detect the specification of endpoints in it. Oftentimes, developers rely on external or in-house libraries to specify the endpoints of the API their microservice exposes. These specifications likely rely on multiple functions the libraries offer. Consequently, the idea is to detect calls to those functions in the source code using static analysis. As a result, all endpoints defined by the microservice are known, since they have to be defined in the source code. However, this approach assumes that information about the utilized libraries is present, specifically how endpoints are defined using it. **Source code** is the information source used in this approach. Notably, there already exist commercial applications which are able to automatically detect API endpoints, such as AppDynamics[5].

In other cases, for example if the source code is inaccessible or no information about the utilized libraries is available, a running MSA can be examined. The incoming traffic of a microservice can be observed to deduce the exposed API endpoints. This idea is designed to work with REST-based APIs. However, it can be adapted to other types of APIs, such as gRPC, since both use HTTP as transport protocol. Independent of the protocol, only endpoints which are triggered in the observed time frame can be detected. Therefore, it is essential that all available endpoints are triggered during observation in order to satisfy the information requirement. If a dedicated system integration testing environment, which is identical to the production environment, is present and if all available endpoints can be triggered, the information requirement can be satisfied. However, it can be challenging to trigger all API endpoints without knowing them in advance. The information source for this approach is the incoming **communication** of microservices.

[4]`https://opentelemetry.io/`

[5]`https://docs.appdynamics.com/appd/21.x/21.1/en/application-monitoring/configure-instrumentation/service-endpoint-detection`

External Endpoints in Static Files This information requirement is satisfied if a list of hardcoded external endpoints for every static file of a microservice is present. An external endpoint is an endpoint which belongs to an entity of the MSA different from the considered microservice. The term is used to emphasize that it is not an API endpoint offered by the considered microservice. Furthermore, only *hardcoded* endpoints are regarded, meaning they can only be changed by modifying the source code or configuration file. Specifically, in accordance to the definition used by Taibi, Lenarduzzi, and Pahl [TLP20], only IP-addresses are considered as hardcoded endpoints.

Extracting the necessary information can be realized through static analysis. In the simplest form, source code and configuration files are searched for occurrences of IP-addresses. Since this does not guarantee that the IP-address actually belongs to a used external endpoint, a more sophisticated approach can be used. This involves the analysis of the target destination of HTTP calls, by looking at function calls of known HTTP libraries. Should an IP-address be detected as destination, it will be added to the list for that source file. However, analyzing function calls of HTTP libraries requires programming language and library specific information so that an algorithm knows which function calls to analyze and how necessary parameters are defined. This information can for example be stored in some form of knowledge base, which has to be updated regularly in accordance to updates to the utilized HTTP libraries. The utilized information source for this extraction is **source code**.

Imported Libraries This information requirement is satisfied if a list of all libraries a microservice depends on is acquired. A microservice depends on a library if the library is imported by one or more source files, which are part of the microservice.

This information source, as well as methods for information extraction, are discussed in detail in the first example of section 3.1. Essentially, import statements in source files are aggregated using static code analysis methods. Finally, post-processing is necessary to remove duplicates and group dependencies by microservice.

As an alternative, methods which are tailored to specific programming languages can be utilized. In the previously referenced example, the module definition of Go is used to acquire a list of imported libraries. These methods are likely to be more efficient, but need to be adapted to the programming languages used in the microservice application. In summary, both approaches utilize **source code** as information source.

Inter-Microservice Communication This information requirement needs detailed information about the communication between microservices in order to be satisfied. Notably, this information requirement is broadly formulated, because a broad range of information is required for the detection of some anti-patterns. We suggest two approaches to extract communication information from microservice architectures.

The first approach relies on logs generated by the microservices which contain information about incoming and outgoing requests. Ideally, these logs contain protocol-specific information, like HTTP headers, as well as basic information about the request, for example the source host. Obviously, logging of such information has to be enabled for all microservices. If the necessary logs are available, they have to be collected and analyzed using static analysis methods. The resulting data satisfies the information requirement. This approach uses **log files** as information source. In addition to requiring the permanent logging of connections, the extracted information is not real-time.

Should real-time connection information be necessary, or if no log files are available, another dynamic approach is possible as well. However, this requires one or more proxy-like components which are able to monitor the traffic for every microservice. Such components could for example be a sidecar proxy, placed in front of every microservice. This would enable real-time communication analysis, which can also be stored for longer periods and satisfies the information requirement. The utilized information source is the **communication** between microservices.

Logging Target This information requirement is satisfied if a list containing the logging locations for each microservice is present. A logging location can, for example, be the local file system or a central log aggregator.

To get an accurate view of the logging targets, two steps are required. This approach requires a machine-readable knowledge base which stores information about where and how various logging libraries store their logging target configuration. First, microservice dependencies are checked for known logging libraries. If a known logging library is found, it is possible to extract the logging target using static analysis methods from either source code files or configuration files. Consequently, the utilized information sources are **source code** and **configuration files**.

Microservice Functionalities Information about all functionalities (i.e. business capabilities) which a microservice implements is necessary to satisfy this information requirement. Each microservice can implement one or more functionalities.

This information belongs to the class of domain knowledge and therefore is not available in machine-readable form without additional processing. As a result, automated extraction is challenging in contrast to manual extraction methods. Information can be extracted manually by a domain expert who has knowledge about the domain in which the microservice operates. The domain expert can judge which functionalities the microservice implements and consequently satisfy the information requirement.

Notably, this process can be adapted to automated extraction. One possibility is to interview a domain expert, for example through a computer-guided survey, about the implemented functionalities. If this interview is designed properly, a list of functionalities is the outcome of it. This list would satisfy the information requirement. The utilized information source is a **domain expert**.

Microservice Owners This information requirement is satisfied if the teams or individuals responsible for a microservice are known. This group of people is referred to as microservice owners. We came up with two approaches on how this information requirement can be satisfied.

The first one requires the codebase of the microservice to be tracked by a version control system, like Git. In Git, each individual contributing to the codebase is identified by a username and email address. Furthermore, it is straightforward to get a list of these contributors. Executing the command `git shortlog -ens` yields a list of every contributor identified by username and email, along with the total number of commits made by each. This list already satisfies the information requirement. However, a more sophisticated analysis is necessary to get information about the team each contributor belongs to, if any. It may be possible to deduce this information through common patterns observed in contributing behavior. Otherwise, the previously acquired list could be enriched with business knowledge to the individual teams. In any case, the **source code** is utilized as information source in this approach.

An alternative method uses enterprise architecture models to acquire a list of microservice owners. Notably, this approach requires the models to be designed in a way that the necessary information is included. Additionally, the enterprise architecture models have to be up-to-date to ensure that the correct information is used. If the owners of each microservice are listed in an enterprise architecture model, they can simply be extracted by means of static analysis. In addition, if the authors are associated to a team in the enterprise architecture model, this information can be extracted as well and can be used to enrich the resulting list of microservice owners. The information source utilized in this approach are **enterprise architecture models**.

Owned Storage In order to satisfy this information requirement, it has to be known which persistent storage medium belongs to which microservice. A persistent storage medium *belongs* to a microservice, if it is designed to contain private data of the microservice. In a well constructed MSA, no persistent storage medium is owned by more than one microservice [Ric18].

It may be possible to extract this information from *infrastructure-as-code (IaC)* definition files. IaC enables developers to define infrastructure in machine-readable files using declarative approaches. Examples for IaC include Docker-compose files[6], Kubernetes manifest files[7] or more complex solutions like Ansible[8]. By analyzing these files, it can be deduced which persistent storage mediums belong to which microservices. As an example, if a Docker-compose file contains exactly two images, one for an arbitrary microservice and one for a key-value store, it is very likely that the key-value store belongs to the specified microservice. However, if multiple microservice share the same database and their private data is stored in tables, instead of a dedicated database, the proposed extraction method becomes more challenging. In addition, business knowledge can be utilized to get accurate results for cases the method based on IaC files does not work with. Possibly, each persistent storage medium could be assigned one microservice in enterprise architecture models, which can then be parsed using static analysis. Therefore, the utilized information source is mainly **source code** with the addition of

[6]https://docs.docker.com/compose/compose-file/
[7]https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/
[8]https://www.ansible.com

enterprise architecture models to improve extracted information quality.

Presence of Central Monitoring Architecture This information requirement is satisfied if the presence or absence of a central monitoring solution is detected. Central means that the solution is shared within an organization and used by multiple independent teams or departments.

The method we propose uses business knowledge in the form of enterprise architecture models to detect this. We did not find a reliable alternative method which relies on application- or runtime-knowledge. However, it is not ruled out that an approach using static or dynamic analysis is feasible. Therefore, further analysis of alternative approaches is an interesting topic for forthcoming research. As already mentioned, using enterprise architecture models requires the necessary information to be modeled in them. It is possible to model a monitoring solution as an object in enterprise architecture, which can be detected using static analysis methods. If no such object is found, it implies that no central monitoring solution exists – provided that the enterprise architecture model is correct and up-to-date. The information source utilized in this approach are **enterprise architecture models**.

Presence of Company-Wide Frameworks This information requirement is satisfied if the presence or absence of programming frameworks or boilerplate code, which are used throughout an organization and dedicated to microservices, is detected. Boilerplate code in this case refers to code snippets which can be used as a skeleton for the development of new microservices.

Similarly to the previous method, this extraction method also utilizes enterprise architecture models. If one or more programming frameworks or microservice skeletons (i.e. boilerplate code) are modeled in enterprise architecture, they can be detected using static analysis methods. Should no such model be found, the absence of company-wide frameworks or boilerplate code is assumed. The information source utilized in this approach are **enterprise architecture models**.

Presence of Metrics Library or Health Endpoint This information requirement is satisfied, if it is known whether a microservice uses a metrics library or a health endpoint to report health-related metrics or if it does not use them. A *health-endpoint*, sometimes also called health check endpoint, is a pattern in microservice development [Ric18]. It reports the health of the microservice after performing corresponding checks. Similarly, a metrics library is used by the microservice to send metrics to an aggregator, such as Prometheus[9] [Ric18].

By convention, a health-endpoint is always called `/health`. Using this information, we can use the methods introduced in the paragraph on the *API Endpoints* information requirement to validate whether such an API endpoint is implemented. Notably, the health-endpoint could be called differently, which makes it more challenging to detect

[9]`https://prometheus.io`

it automatically. A thinkable approach to detect non-standard named health-endpoints is through the usage of heuristics which take the returned payload of an endpoint and endpoints with similar names into account. Hence, manual validation of the result could prove helpful. The utilized information source is the microservices' **source code**.

Moreover, a metrics tracking library can be detected using the methods presented in the paragraph on the *Imported Libraries* information requirement. Having a list of libraries a microservice imports makes it possible to check it against a list of known metrics libraries, to see if it is used in the microservice. The result of this check satisfies the information requirement. The utilized information source of this approach is also the **source code**.

Processes and Policies of the Organization This information requirement is satisfied if all development-related processes and policies of an organization are known.

Automatically extracting this information is feasible, if the underlying business knowledge is stored in machine-readable formats – like enterprise architecture models. The modeled processes or policies can be extracted using static analysis methods. If this information is not modeled in a machine-readable information source, it may be possible to interview a business expert using a structured questionnaire. However, it is challenging to store information about processes and policies in a format that makes it possible for algorithms to later process and interpret them. Therefore, the format should be sufficiently evaluated beforehand. In summary, the utilized information source is either an **enterprise architecture model**, **business expert**, or a combination of both.

Shared Code Fragments This information requirement is satisfied if a list of shared code fragments and shared directory structures of an entire microservice application has been extracted. Shared code fragments are parts of the source code which are duplicated across two or more microservices. The minimum length of these fragments should be customizable and use a sensible default, such as 10 lines of code or more. Customization is important in order to adapt to different circumstances and to avoid generating false positives or false negatives. A directory structure is called shared, if it occurs in more than one microservice's source code. It may be useful to restrict the similarity analysis to a specific depth, such as two hierarchy levels, depending on the desired sensitivity.

The method we propose utilizes static code analysis to extract shared code fragments and directory structures. Consequently, the source files of every microservice have to be analyzed. Some *integrated development environments (IDEs)*, for example most JetBrains IDEs[10] show that detecting duplicate code fragments is feasible. Therefore, the utilized information source is the **source code** of all microservices.

Technologies Used To satisfy this information requirement, an ideally complete list of all technologies that are used in the microservice application and organization is necessary. Since *technologies* is a vague term, we define it to include programming languages,

[10]`https://www.jetbrains.com/help/idea/analyzing-duplicates.html`

protocols, frameworks and databases in accordance with Taibi, Lenarduzzi, and Pahl [TLP20]. Notably, no proper definition was found in current literature. Furthermore, the definition may differ between organizations. Multiple methods are needed for every type of technology that should be detected. In the following, we propose one method for every previously mentioned technology.

Programming languages can be detected by analyzing the source files, specifically analyzing the file extension should be sufficient.

Next, used protocols can be detected using static code analysis. It can be checked whether known libraries, which are associated to a specific protocol, are imported and used in the source code. To further illustrate how this method could work, we use the example of a Python microservice which communicates using the gRPC protocol. In order to use gRPC, the official gRPC library is installed using the command `pip install grpcio`. Moreover, each Python module that handles gRPC communication must import the previously installed library using `import grpc`. These imports can be then detected using static code analysis. This method works analogously for other libraries or protocols.

Furthermore, it is possible to detect utilized frameworks with a similar method, by checking if framework-specific files, classes or similar artifacts are present. In this context, a *framework* is a tool or a set of tools which provides support and guidance for the development of the basic microservice structure. Additionally, frameworks offer ready-made components which help developers to focus on the business capabilities of a microservice. As an alternative to checking for framework-specific files, frameworks could also be detected using business knowledge and a similar method proposed in the paragraph on the *presence of company-wide frameworks*.

Finally, utilized databases can be discovered by analyzing the communication of a microservice application. It is possible to distinguish individual databases, such as PostgreSQL or MySQL by their network traffic and their use of different ports, assuming that standard ports are not changed.

In summary, the information sources used to satisfy this information requirement are, depending on the utilized methods, a combination of **source code**, **business knowledge** and **communication** in the MSA.

Technology Selection Motives This information requirement is satisfied if the motives for selecting a particular technology are known.

The main requirement for satisfying this information requirements is the documentation of all selection motives. In theory, it is possible to store technology selection motives in enterprise architecture models, which would make them machine-readable by means of static analysis. However, a format suitable for subsequent automated analysis should be chosen. Consequently, the utilized information sources are **enterprise architecture models**.

Transmitted Version Information This information requirement is satisfied by knowing if and how a microservice transmits version information about its API. We consider the two most common API versioning patterns. The first pattern includes version information in the API's *uniform resource identifier (URI)*. For example, the URI `https://example.org/api/v1/hello` describes that version 1 of the API is accessed. Alternatively, version information can be included in HTTP headers, which is particularly suited for REST-based APIs. It is common to use the `Content-Type` header for this purpose[11], which could look like this: `application/vnd.example.test.v1+json`. In the following we propose an extraction method for each of the two patterns.

First, the URIs that are accessed in a running microservice application can be analyzed using heuristics to check whether they contain a versioning schema. The versioning is not guaranteed to adhere to best practices. As a result, it can be stored at any location in the URI – `https://v1.api.example.org/hello` is a valid versioning schema. Therefore, multiple common possibilities should be checked. Notably, using heuristics does not guarantee that transmitted version information is found.

The same approach using heuristics can be used for version information being transmitted in the HTTP headers. First, the `Content-Type` header should be checked to see whether it defines a MIME-type like exemplified before. Should no custom versioning be found, other non-standard headers could be checked as well. Like before, this does not guarantee that transmitted version information is found if it is present.

Both approaches utilize **communication** in the microservice application as information source.

Usage of CI/CD Pipelines This information requirement is satisfied by knowing if the considered microservice is being built, tested or deployed using a CI/CD pipeline that is part of the organization's infrastructure. Some Git hosting services, such as GitHub[12] or GitLab[13] offer managed CI/CD pipelines, which are defined using source files in the microservice's repository. Furthermore, an organization can also provide a central CI which is hosted on-premise or use another cloud provider.

One approach to extract the necessary information is to look for pipeline definition files in the microservice's repository. If, for example, a non-empty `.gitlab-ci.yml` file is present, it is very likely that GitLab's CI/CD pipelines are utilized. Ideally, a catalog of standard file names should be known. However, if no such file is found, business knowledge is likely needed to extract the required information. Enterprise architecture models can be analyzed using static analysis methods to check whether CI/CD-related components are modeled. In summary, the utilized information sources are **source code** files, as well as **enterprise architecture models**.

Versioning Schema This information requirement is satisfied if it is known whether an organization prescribes a specific versioning schema for APIs.

It may be possible to apply heuristics to all known URIs of API endpoints and custom MIME-types to detect a shared schema. Notably, it is not guaranteed that a detected

[11]`https://opensource.zalando.com/restful-api-guidelines/#114`
[12]`https://github.com/features/actions`
[13]`https://docs.gitlab.com/ee/ci/`

schema is actually prescribed by the organization. Instead, it is possible that multiple teams of developers mutually agreed on some schema. In this case, the schema is not considered to be prescribed by an organization and should therefore not be reported as such. It is currently unclear how a distinction between these cases can be made, making the development of a solution to this problem an interesting topic for further research.

In cases where the application of heuristics is not possible, an approach using business knowledge by analyzing enterprise architecture models or interviewing business experts may be feasible. To summarize, the utilized information source is either the **communication** in a microservice application or business knowledge in the form of **enterprise architecture models** or a **business expert**.

5.2 Relationship between Information Requirements and Information Sources

In the previous section, each information requirement was linked to one or more information sources necessary to satisfy the information requirement. Figure 5.1 visualizes these relationships.

At the top, all information sources are listed, grouped by the contained information, as seen in section 3.2 before. Furthermore, the individual information sources are colored to indicate which methods are required to extract the contained information. Orange shading indicates that information can not be extracted using automated methods and therefore manual methods have to be used. Green shading indicates that static analysis methods can be used to extract information, which means that automated analysis is feasible. Finally, blue shading indicates that a running MSA has to be analyzed using dynamic analysis methods to extract information automatically.

At the bottom, all information requirements are listed. They are colored as well to indicate which methods are necessary to satisfy the respective information requirement. The coloring is directly related to the associated information sources. For example, the information requirement *Transmitted Version Information* is colored green and blue, since the information sources *Source Code* (green) and *Communication* (blue) are required to satisfy it. Dotted arrows indicate that the targeted information source is necessary to satisfy the information requirement. Additionally, nine information requirements have been grouped, because all of them are business-focused. In this case, a dotted arrow *with an empty arrow head* is used to indicate that one or more of the three associated information sources can be used to satisfy a contained information requirement, depending on the solution proposed in the previous section. In addition, this figure enables seeing which information sources are used most often – "Source Code" and "Microservice Communication" – as well as which information requirements need more information sources than average – "Technologies Used" requires three information sources.

5.3 Threats to Validity

It is important to state the limitations of this research. With regard to *internal validity*, it has to be pointed out that the extraction methods suggested in this chapter have not yet been evaluated, due to the limited scope of this thesis. Although the methods were constructed with careful research, it can not be guaranteed that they function as envisioned, without conducting a proper evaluation.

Regarding *external validity*, it is important to highlight that the proposed extraction methods may need to be adapted depending on the environment they are utilized in. Microservice architecture can vastly differ depending on the organization and the technologies used. Albeit we tried to propose methods which are as generalized as possible, the need for modification may still arise.

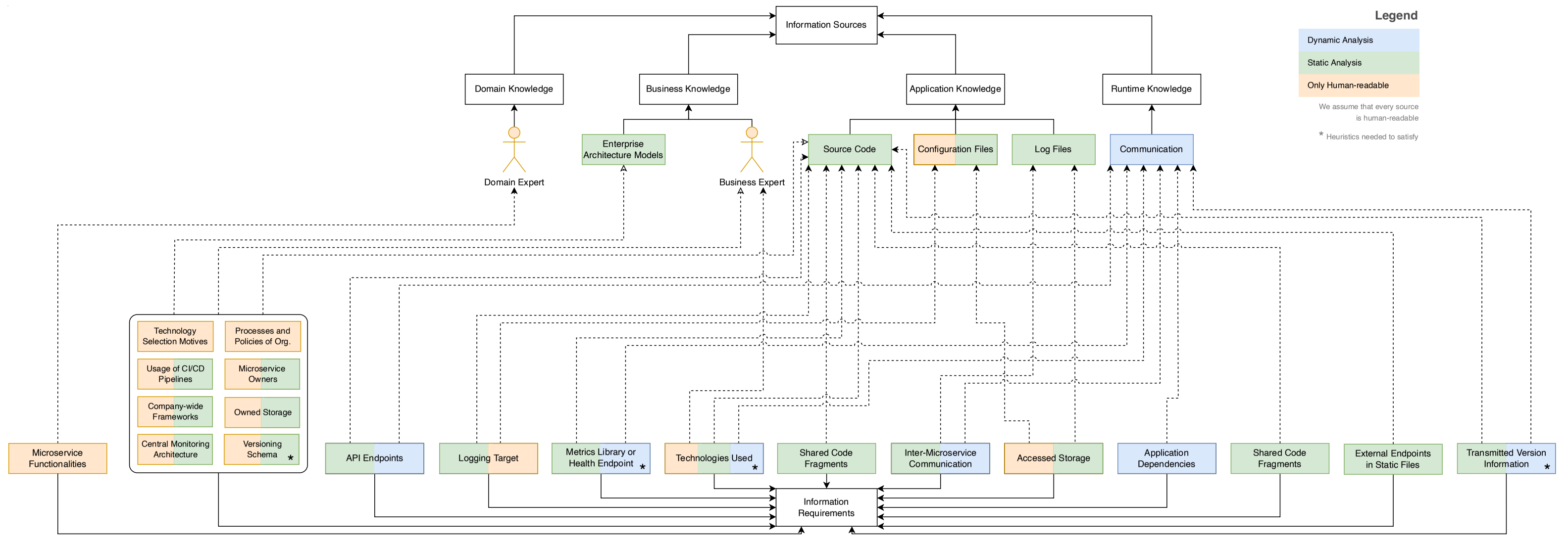

Figure 5.1: Overview of relationships between information requirements and information sources

6 Discussion

After the proposal of information requirements and extracting methods, this section is used to discuss our findings and validate whether they answer the previously defined research questions.

The goal of the first research question was to discover how information needs can be structured and categorized. To answer this question, the concepts of *information requirements* and *information sources* were introduced in chapter 3. Furthermore, we proposed four categories of information sources – domain-, business-, application- and runtime-knowledge. Additionally, we were able to propose one or more information requirements for every anti-pattern in the catalog of Taibi, Lenarduzzi, and Pahl [TLP20] in chapter 4. An overview of the findings can be seen in figure 4.1. Consequently, it can be concluded that the first research question has been successfully answered.

The second research question is about how information requirements can be satisfied using information from MSAs. As a first step, information sources were linked to the previously proposed information requirements. Moreover, methods to extract the required information were proposed in chapter 5. It is important to note that these methods have not been implemented or validated yet, due to the limited scope of this thesis. A validation of the extraction methods is certainly an interesting objective for future research. Furthermore, the extraction methods are designed to be as generalizable as possible, so they can be adapted to a variety of microservice applications. Arguably, the extraction methods proposed in chapter 5 answer the second research question, since they make the information required for detection obtainable.

The third and final research question is concerned with discovering anti-patterns that have information requirements which can be satisfied without human involvement. To answer this question, analyzing figure 5.1 is helpful. Thanks to the coloring of information sources and information requirements, it becomes clear which information requirements can be satisfied without human interaction. Every information requirement which uses information sources from which information can be extracted using static- or dynamic analysis, is satisfiable automatically. Now, it can be deduced which anti-patterns are detectable using automated methods by finding those which only need automatically satisfiable information requirements. Consequently, four anti-patterns are automatically detectable. In addition, six more anti-patterns belong to this group, if the necessary information is modelled in up-to-date enterprise architecture models (EAM). Assuming that a knowledge base which allows automated parsing of necessary configuration files is present, the following three anti-patterns are detectable without human interaction as well. Table 6.1 lists the results of this analysis.

Name	**Automatically Detectable**	**Additional Requirements**
Hardcoded Endpoints	Yes	None
Wrong Cuts	No	–
Cyclic Dependency	Yes	None
API Versioning	Yes	EAM up-to-date
Shared Persistence	Yes	Configuration file parsable
ESB Usage	Yes	None
Legacy Organization	No	–
Local Logging	Yes	Configuration file parsable
Megaservice	No	–
Inappropriate Service Intimacy	Yes	Configuration file parsable
Lack of Monitoring	Yes	EAM up-to-date
No API–Gateway	Yes	None
Shared Libraries	No	–
Too Many Technologies	No	–
Lack of Microservice Skeleton	Yes	EAM up-to-date
Microservice Greedy	No	–
Focus on Latest Technologies	No	–
Common Ownership	Yes	EAM up-to-date
No DevOps Tools	Yes	EAM up-to-date
Non-homogeneous adoption	Yes	EAM up-to-date

Table 6.1: Automatically detectable anti-patterns (ordered by decreasing harmfulness)

The third column of the table refers to the previously mentioned requirements, meaning updated enterprise architecture models and the presence of a knowledge base which allows automated parsing of relevant configuration files. As a result of the information given above, the third research question can be considered answered.

7 Conclusion

Contents

As conclusion of this thesis, the most important findings are summarized in section 7.1 and an outlook on possible future work is given in section 7.2.

7.1 Summary

This thesis is a first step towards the detection of microservice-specific anti-patterns in microservice architectures using automated methods. The ability to automatically detect anti-patterns has the potential to increase the quality of microservice applications and can help organizations transition from monolithic architectures with fewer problems. Additionally, this thesis provides groundwork for further research on this topic with the proposal of a fundamental structuring of anti-pattern information needs.

First, necessary concepts were introduced. Specifically, the terms *information requirement* and *information source* were defined along with an explanation of the relationship between anti-patterns, information requirements and information sources. Additionally, the complete detection processes of two anti-patterns were demonstrated on the basis of a demo application.

Next, concrete information sources were defined along with the four main knowledge classes which were discovered during our research. It was stated which types of methods can be utilized to extract information from the main knowledge classes. This was followed by an elaboration of information requirements and which information they contain specifically.

After all required concepts were introduced, all anti-patterns from the catalog of Taibi, Lenarduzzi, and Pahl [TLP20] were investigated to deduce the information requirements that have to be satisfied in order to detect an anti-pattern.

Building up on this insight, methods to satisfy the previously discovered information requirements using the information sources were proposed. In addition, a visual overview of the relationship between information requirements and information sources was introduced.

Finally, the results were analyzed to validate that they answer the research questions defined in the beginning.

7.2 Future Work

We now want to give an outlook on interesting research that could be done in the future in the context of this work.

Evaluation
Most importantly, a proper evaluation of our proposed information requirements and methods for information extraction should be conducted. Due to the limited scope of this thesis, we were unable to perform such an evaluation. An evaluation is needed to determine whether the results of this thesis can be build upon and used in forthcoming research without significant modifications. A thinkable evaluation concept can include interviews of a set of experts to determine whether the information requirements were chosen correctly, or the implementation of proposed extraction methods to validate whether they yield the desired information. A case study would also be feasible.

Extension of information requirements and extraction methods
Additionally, proposed information requirements and extraction methods can be extended on the basis of another microservice-specific anti-pattern catalog. An extension would enable the detection of a broader range of anti-patterns.

Automated processing of human-readable information
Furthermore, research on how only human-readable classes of information sources, like business- and domain-knowledge, can be processed reliably by algorithms would be very interesting. Automated processing of such information has the potential to enable automatic detection of even more microservice-specific anti-patterns – especially those which focus on non-technical aspects. It is thinkable that machine learning methods can enable the extraction and processing of such information sources, for example by training models to extract unstructured information.

Automatically resolving anti-patterns
Finally, one of the most exciting future research topics is the question of how detected anti-patterns can be automatically resolved. This is likely a complex problem to solve, since a holistic view of the microservice application is indispensable, but it would be a great contribution to the improvement of microservice architecture quality.

Bibliography

[Al-+14] H. A. Al-Mutawa et al. "On the Shape of Circular Dependencies in Java Programs." In: *2014 23rd Australian Software Engineering Conference*. ISSN: 2377-5408. Apr. 2014, pp. 48–57. DOI: `10.1109/ASWEC.2014.15` (cit. on p. 25).

[BAR18] A. Biaggi, F. Arcelli Fontana, and R. Roveda. "An Architectural Smells Detection Tool for C and C++ Projects." In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Aug. 2018, pp. 417–420. DOI: `10.1109/SEAA.2018.00074` (cit. on pp. 5, 6).

[Boe+05] F. de Boer et al. "Enterprise Architecture Analysis with XML." In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. ISSN: 1530-1605. Jan. 2005, 222b–222b. DOI: `10.1109/HICSS.2005.242` (cit. on p. 20).

[CBD18] A. Carrasco, B. v. Bladel, and S. Demeyer. "Migrating towards microservices: migration and architecture smells." In: *Proceedings of the 2nd International Workshop on Refactoring*. IWoR 2018. New York, NY, USA: Association for Computing Machinery, Sept. 2018, pp. 1–6. ISBN: 978-1-4503-5974-0. DOI: `10.1145/3242163.3242164`. URL: `https://doi.org/10.1145/3242163.3242164` (visited on 06/08/2022) (cit. on p. 25).

[CT22] T. Cerny and D. Taibi. *Static analysis tools in the era of cloud-native systems*. Number: arXiv:2205.08527 arXiv:2205.08527 [cs]. May 2022. URL: `http://arxiv.org/abs/2205.08527` (visited on 06/27/2022) (cit. on pp. 11, 20).

[Dra+17] N. Dragoni et al. "Microservices: Yesterday, Today, and Tomorrow." en. In: *Present and Ulterior Software Engineering*. Ed. by M. Mazzara and B. Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: `10.1007/978-3-319-67425-4_12`. URL: `https://doi.org/10.1007/978-3-319-67425-4_12` (visited on 10/21/2022) (cit. on pp. 1, 3).

[Esa22] Ö. Esas. "Design Patterns and Anti-Patterns in Microservices Architecture: A Classification Proposal and Study on Open Source Projects." English. MA thesis. Milano: Politecnico Milano, Apr. 2022. URL: `https://www.politesi.polimi.it/handle/10589/186745?mode=complete` (cit. on p. 30).

[Eva04] E. J. Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. en. Addison-Wesley Professional, 2004. ISBN: 978-0-321-12521-7 (cit. on p. 19).

[Fon+17] F. A. Fontana et al. “Arcan: A Tool for Architectural Smells Detection.” In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* Apr. 2017, pp. 282–285. DOI: `10.1109/ICSAW.2017.16` (cit. on pp. 5, 6).

[Fow18] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Englisch. 2nd ed. Boston: Addison-Wesley Professional, Nov. 2018. ISBN: 978-0-13-475759-9 (cit. on p. 3).

[Hep+05] M. Hepp et al. “Semantic business process management: a vision towards using semantic Web services for business process management.” In: *IEEE International Conference on e-Business Engineering (ICEBE’05).* Oct. 2005, pp. 535–540. DOI: `10.1109/ICEBE.2005.110` (cit. on p. 18).

[Hüb+22] T. Hübener et al. “Automatic Anti-Pattern Detection in Microservice Architectures Based on Distributed Tracing.” In: *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* May 2022, pp. 75–76. DOI: `10.1109/ICSE-SEIP55303.2022.9794000` (cit. on pp. 8, 26, 30, 41).

[IA18] C. Ibsen and J. Anstey. *Camel in Action.* en. Simon and Schuster, Feb. 2018. ISBN: 978-1-63835-280-8 (cit. on p. 41).

[Koe95] A. Koenig. “Patterns and antipatterns.” In: *Journal of Object-Oriented Programming* 8.1 (1995), pp. 46–48 (cit. on p. 3).

[Li+22] R. Li et al. “Understanding software architecture erosion: A systematic mapping study.” en. In: *Journal of Software: Evolution and Process* 34.3 (Mar. 2022). ISSN: 2047-7473, 2047-7481. DOI: `10.1002/smr.2423`. URL: `https://onlinelibrary.wiley.com/doi/10.1002/smr.2423` (visited on 06/17/2022) (cit. on p. 39).

[Lou06] P. Louridas. “Static code analysis.” In: *IEEE Software* 23.4 (July 2006). Conference Name: IEEE Software. ISSN: 1937-4194. DOI: `10.1109/MS.2006.114` (cit. on p. 20).

[Mo+21] R. Mo et al. “Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles.” In: *IEEE Transactions on Software Engineering* 47.5 (May 2021). Conference Name: IEEE Transactions on Software Engineering, pp. 1008–1028. ISSN: 1939-3520. DOI: `10.1109/TSE.2019.2910856` (cit. on p. 3).

[Nad+16] I. Nadareishvili et al. *Microservice Architecture: Aligning Principles, Practices, and Culture.* en. ”O’Reilly Media, Inc.”, July 2016. ISBN: 978-1-4919-5634-2 (cit. on p. 2).

[Nor] Northern Ireland Business Info. *What is knowledge in business?* English. URL: `https://www.nibusinessinfo.co.uk/content/what-knowledge-business` (visited on 10/03/2022) (cit. on p. 19).

[NSV15] D. Naranjo, M. Sánchez, and J. Villalobos. "PRIMROSe: A Graph-Based Approach for Enterprise Architecture Analysis." en. In: *Enterprise Information Systems*. Ed. by J. Cordeiro et al. Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2015, pp. 434–452. ISBN: 978-3-319-22348-3. DOI: `10.1007/978-3-319-22348-3_24` (cit. on p. 20).

[Nte+21] E. Ntentos et al. "Detector-based component model abstraction for microservice-based systems." en. In: *Computing* 103.11 (Nov. 2021), pp. 2521–2551. ISSN: 0010-485X, 1436-5057. DOI: `10.1007/s00607-021-01002-z`. URL: `https://link.springer.com/10.1007/s00607-021-01002-z` (visited on 06/27/2022) (cit. on p. 8).

[OLe98] D. O'Leary. "Enterprise knowledge management." In: *Computer* 31.3 (Mar. 1998). Conference Name: Computer, pp. 54–61. ISSN: 1558-0814. DOI: `10.1109/2.660190` (cit. on p. 18).

[PB21] T. Portugal and J. Barata. "Enterprise Architecture Erosion: A Definition and Research Framework." en. In: *AMCIS 2021 Proceedings. 7.* 2021, p. 5 (cit. on p. 39).

[Pig+20] I. Pigazzini et al. "Towards microservice smells detection." en. In: *Proceedings of the 3rd International Conference on Technical Debt*. Seoul Republic of Korea: ACM, June 2020, pp. 92–97. ISBN: 978-1-4503-7960-1. DOI: `10.1145/3387906.3388625`. URL: `https://dl.acm.org/doi/10.1145/3387906.3388625` (visited on 05/31/2022) (cit. on pp. 6, 8, 24–26, 28).

[Ric] C. Richardson. *Potholes in the road from monolithic hell - Microservices adoption anti-patterns*. URL: `https://microservices.io/microservices/general/2018/11/04/potholes-in-road-from-monolithic-hell.html` (visited on 07/08/2022) (cit. on pp. 29, 35, 37).

[Ric16] M. Richards. *Microservices antipatterns and pitfalls*. en-US. Oct. 2016. URL: `https://www.oreilly.com/content/microservices-antipatterns-and-pitfalls/` (visited on 06/08/2022) (cit. on p. 27).

[Ric18] C. Richardson. *Microservices Patterns: With examples in Java*. English. 1st edition. Shelter Island, New York: Manning, Nov. 2018. ISBN: 978-1-61729-454-9 (cit. on pp. 1–3, 44, 45).

[Ros] R. Ross. *What Business Knowledge Is : Commentary*. en. URL: `https://www.brcommunity.com/articles.php?id=b959#fn1` (visited on 10/03/2022) (cit. on p. 19).

[RW11] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. en. Addison-Wesley, Oct. 2011, p. 25. ISBN: 978-0-321-71833-4 (cit. on p. 39).

[Sal19] J. Salentin. "Towards a Catalogue of Enterprise Architecture Smells and their Detection." en. Bachelor Thesis. Aachen: RWTH, July 2019 (cit. on p. 3).

[SHS21] A. R. Sabau, S. Hacks, and A. Steffens. "Implementation of a continuous delivery pipeline for enterprise architecture model evolution." en. In: *Software and Systems Modeling* 20.1 (Feb. 2021), pp. 117–145. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-020-00828-z. URL: https://link.springer.com/10.1007/s10270-020-00828-z (visited on 10/27/2022) (cit. on p. 20).

[SML17] P. Saint-Louis, M. C. Morency, and J. Lapalme. "Defining Enterprise Architecture: A Systematic Literature Review." In: *2017 IEEE 21st International Enterprise Distributed Object Computing Workshop (EDOCW)*. ISSN: 2325-6605. Oct. 2017, pp. 41–49. DOI: 10.1109/EDOCW.2017.16 (cit. on p. 20).

[Tig+20] R. Tighilt et al. "On the Study of Microservices Antipatterns: a Catalog Proposal." en. In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. Virtual Event Germany: ACM, July 2020, pp. 1–13. ISBN: 978-1-4503-7769-0. DOI: 10.1145/3424771.3424812. URL: https://dl.acm.org/doi/10.1145/3424771.3424812 (visited on 06/08/2022) (cit. on pp. 27, 28, 30–32, 36).

[TLP20] D. Taibi, V. Lenarduzzi, and C. Pahl. "Microservices Anti-patterns: A Taxonomy." en. In: *Microservices*. Ed. by A. Bucchiarone et al. Cham: Springer International Publishing, 2020, pp. 111–128. ISBN: 978-3-030-31645-7 978-3-030-31646-4. DOI: 10.1007/978-3-030-31646-4_5. URL: http://link.springer.com/10.1007/978-3-030-31646-4_5 (visited on 05/23/2022) (cit. on pp. 6–9, 13, 17, 18, 20, 23–25, 27–37, 42, 47, 53, 55).

[TS20] D. Taibi and K. Systä. "A Decomposition and Metric-Based Evaluation Framework for Microservices." en. In: *Cloud Computing and Services Science*. Ed. by D. Ferguson et al. Vol. 1218. Series Title: Communications in Computer and Information Science. Cham: Springer International Publishing, 2020, pp. 133–149. ISBN: 978-3-030-49431-5 978-3-030-49432-2. DOI: 10.1007/978-3-030-49432-2_7. URL: http://link.springer.com/10.1007/978-3-030-49432-2_7 (visited on 05/31/2022) (cit. on p. 25).

[Ver13] V. Vernon. *Implementing Domain-Driven Design*. en. Addison-Wesley, Feb. 2013. ISBN: 978-0-13-303988-7 (cit. on p. 19).

[WDC20] A. Walker, D. Das, and T. Cerny. "Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study." en. In: *Applied Sciences* 10.21 (Jan. 2020). Number: 21 Publisher: Multidisciplinary Digital Publishing Institute, p. 7800. ISSN: 2076-3417. DOI: 10.3390/app10217800. URL: https://www.mdpi.com/2076-3417/10/21/7800 (visited on 06/29/2022) (cit. on pp. 3, 8, 11, 13, 24–28, 31–33, 35).