Masterthesis

Automatic Detection of Architectural Anti Patterns in Microservices

Bernhard Speiser, BSc

Supervisor: DI Dr. Gottfried Bauer

University of Applied Sciences Wiener Neustadt

28.04.2020

Abstract

It is a common trend for organizations to migrate their software systems from a monolithic architecture to a microservice architecture, which usually starts without any experience of the microservice architecture. As with any other technology, microservice architectures have various patterns and anti patterns. Furthermore, it is hard to keep track of the regularly occurring decay of a software architecture. Existing tools often require an immense configuration effort in advance and are not specialized in detecting a taxonomy of microservice specific anti patterns. Therefore a prototype is implemented to automatically detect a specified set of microservice related anti patterns in C# projects. In addition, a list of open source projects, implementing the microservice architecture, is considered and compiled. Furthermore, within the context of a case study, this list of projects is then analyzed by the built prototype. Based on the outcome of the analysis one open source project is proposed as an advantageous guideline for the implementation of a microservice project. Additionally, as a key result the *API Versioning*, *Hardcoded Endpoints*, *Lack of Monitoring* and *Shared Libraries* anti patterns were identified as most frequently occurring pitfalls in microservice projects in particular.

Contents

List of Abbreviations

CNV Calculated Number of Violations

ESB Enterprise Service Bus

LOC Lines of Code

RNV Relative Number of Violations

RQ1 First Research Question

RQ2 Second Research Question

SOA Service Oriented Architecture

TNV Total Number of Violations

List of Tables

1 Introduction

1.1 Microservices

Over the last decade, a new trend in software architecture emerged: *Microservices*. With microservices the increasing complexity of software systems can be managed by decomposing large systems into a set of independent (and isolated) services [1]. *Dragoni et al.* provides a definition: *"A microservice is a cohesive, independent process interacting via messages"* [1]. Microservices provide benefits in terms of maintainability, scalability, reusability and independence with regards to development and deployment [1, 2, 3]. The key characteristics of microservices are flexibility, modularity and evolution [1]. Flexibility is the possibility to keep up with a changing business environment while supporting modifications that are necessary to stay competitive. A microservice architecture is composed of multiple isolated components (modularity) instead of having a single component (commonly known as monolith). A microservice architecture should stay maintainable while evolving (evolution) [1]. Each microservice is dedicated to a single business process/capability [4]. Apart from this, microservices can be developed utilizing different technology stacks, since they communicate by using messages. Microservices scale independently from other services and can be deployed on a platform and hardware that best suits their needs [5].

1.2 Architectural Smells and Anti Patterns

According to *Garcia et al.* architectural anti patterns are negatively affecting any system quality, while architectural smells will affect lifecycle properties, such as understandability, testability, extensibility, and reusability [6], to a higher extend. Thus, the general definition of anti patterns allows architectural smells to be classified as anti patterns. Architectural anti patterns (and therefore smells) are a commonly (although not always intentionally) used set of architectural design decisions that negatively impact system properties [6]. Furthermore, they represent common "solutions" that are not necessarily faulty or errant, but still negatively impact software quality [6, 7], maintenance and evolution costs [8]. This thesis will use the term *anti patterns* to refer to anti patterns and smells. Architectural anti patterns, as outlined by *Ernst et al.*, are one of the greatest sources of technical debt [9, 10], also regarded to as *architectural debt* [11]. Thus, it is important to identify, investigate and remove them through different refactoring steps [10].

1.3 Motivation and Problem Statement

It is a common trend for organizations to migrate their systems from a monolithic architecture to a microservice architecture [12, 13, 14]. This process has various motivations, such as, being an enabler for DevOps [15, 16, 17] and reduced development times [4] resulting in an increase in productivity.

As with any other (trending) technology, microservice architectures show various patterns [5] and anti patterns. Common microservice related anti patterns have been identified by *Taibi et al.* [18, 19] and *Neri et al.* [20]. A recent analysis [21] of microservice related discussions on StackOverflow, a Q&A website, has shown, that *best practices* are a frequently discussed topic (technical: 14.10%, conceptual: 26.19%) among (the majority of) developers. It is therefore concluded that there are misconceptions among developers regarding best practices (and as a consequence, anti patterns) of microservices. One factor to explain this, could be, that the migration (or even the initial implementation) of projects, starts without any or insufficient experience with microservices [12, 22], respectively.

Besides that, it is common knowledge, that it is hard to keep track of the evolution of an architecture. Furthermore, *the study of software architecture has recognized architectural decay as a regularly occurring phenomenon in long-lived systems* [23]. It would be a benefit, for academics and organisations, if there is a way of continuously verifying the architecture, for example, based on a taxonomy of anti patterns. Such a tool could explicitly tell the developer, when he or she makes an (obvious) mistake or violates a best practice. Tools, such as *Sonargraph Architect*[1], are available for architecture verification, but they (usually) have an immense configuration effort upfront. Furthermore, it is quite a challenge, to introduce them late into a project, since, in the worst case, the architecture has to be reverse engineered. These efforts are a potential reason not to use these tools all together.

1.4 Objectives and Research Questions

The first objective of this work is to conceptualize and implement a prototype of a *microservice architectural anti pattern analyzer* tool. The second objective is to compile a list of open source microservice projects. This list will then be analyzed by an own prototype specifically designed and built. The built prototype[2] and the list of open source microservice projects can then also be used for further research by the research community.

Based on these objectives, the following research questions arise:

- ***RQ1**: Based on the results provided by the prototype, which open source project can be identified as an advantageous guideline for the implementation of a microservice project?*
- ***RQ2**: Which of the revealed and investigated anti patterns occur most frequently with respect to the list of specifically considered open source projects and therefore represent frequently occurring pitfalls?*

[1] https://www.hello2morrow.com/products/sonargraph/architect9 Accessed 17.12.2019
[2] Publicly available at: https://github.com/speiser/masterthesis-computerscience

2 Background and Theory

2.1 Microservice Anti Patterns

This subsection lists commonly occurring architectural anti patterns regarding microservices which are identified by research literature. The listed anti patterns were identified by surveys of experienced practitioners, by [18] and its subsequent survey [19]. The list is further supported by systematical literature reviews, such as [20] and [24].

2.1.1 (Lack of) API Versioning

Description: Semantically versioning of APIs is necessary to ensure the independent deployment of microservices. It guarantees that not yet adapted microservices can still communicate with the newly deployed service, even after a breaking change was made. The lack of semantically versioning could result in bad requests (e.g.: in the case that the request contract was changed) or unknown response data is returned to the invoking microservice.
Sources: [18, 19, 24, 25, 26]

2.1.2 Cyclic Dependency

Description: Cyclic dependency is a common architectural anti pattern, even outside of the microservice landscape. A cyclic dependency, in the context of microservices, is a cyclic chain of calls between microservices, which makes it hard (to impossible) to reuse or maintain these services. It can be detected by following each "call chain". It can either be a direct call between services, e.g.: *A calls B, B calls A* or transitive e.g.: *A calls B, B calls C and C calls A.*
Sources: [18, 19, 24]

2.1.3 ESB Usage

Description: Enterprise services buses (ESB) *were a dominant communication strategy* [21] in Service Oriented Architectures (SOA). But in microservices ESBs *may become a bottleneck both architecturally and organizationally* [27] and can lead to centralization of business logic. Instead a lightweight message bus should be adopted.
Sources: [18, 19, 20, 21]

2.1.4 Hardcoded Endpoints

Description: Hardcoded Endpoints (such as an IP address or port) is an anti pattern since it does not allow to scale the invoked microservice (A), since the invoking service (B) only "knows" this (hardcoded) microservice location. Thus, no load balancer for microservices can be utilized. Furthermore, if the location of service A is changed, service B has to be reconfigured (or in worst case, rebuilt and redeployed). A common solution for this anti pattern is to

implement a service discovery pattern, as described in [5].
Sources: [18, 19, 20, 24]

2.1.5 Lack of Monitoring

Description: A microservice should contain an endpoint for a "health check". The bare minimum would be to continuously check whether the service is still running or needs to be restarted. An extended health check could respond with the current load of the microservice, so that a load balancer could decide, whether it is necessary to start another instance of the service or not. Without a health check the microservice could be offline (or malfunctioning) and the developers would not notice it until someone checks it manually.
Sources: [19]

2.1.6 Megaservice

Description: A megaservice is a service that does a lot of things. From a structural point of view it can be regarded as a monolith. Thus, it comes with the same disadvantages/problems as a monolith. It is difficult to say at what point a microservice becomes a megaservice, as there is no "rule" that defines how big a microservice should be. There are few size metrics, such as *Lines of Code (LOC)*, which *is sometimes seen as controversial. Nonetheless, size metrics on their own are never sufficiently accurate. Moreover, it is often hard to define "acceptable" value ranges for these metrics* [28]. A different approach to detect a megaservice could be to check whether it implements multiple business processes or not.
Sources: [19, 24]

2.1.7 No API-Gateway

Description: According to [5], an API-Gateway is the entry point of the system that routes requests to the appropriate microservices. An API-Gateway usually implements shared logic like authentication and ratelimiters. With an API-Gateway a client does never directly communicate with a microservice.
Sources: [18, 19, 20]

2.1.8 Shared Libraries

Description: Libraries shared across multiple microservices come with the benefit of less code duplication, but tightly couples the services together. This leads to a loss of independence, one of the key benefits of microservices. Code changes in shared libraries require coordination across multiple development teams, which hinders autonomy and thus reduces productivity.
Sources: [18, 19]

2.1.9 Shared Persistence

Description: The shared persistence anti pattern occurs when different microservices access or manage the same database. Worst case, different microservices access the same entities. This anti pattern couples microservices, leading to a loss of independence and result in scaling problems.
Sources: [18, 19, 20, 24]

2.1.10 Wrong Cuts

Description: There are multiple reasons why wrong cuts of microservices happen, such as, not knowing the domain (see *domain driven design*) or cutting based on technical layers (presentation, business, data) instead of business domains.
Sources: [18, 19, 24]

2.2 Automatic Anti Pattern Detection Approaches

As *Azadi et al.* [10], *Fontana et al.* [29], *Fernandes et al.* [30] and multiple other studies illustrate, there are already existing tools for detecting (general[3]) architectural anti patterns. These tools and research literature show, that there are different approaches to detect anti patterns automatically. The intent of this subsection is to give a brief overview of these approaches.

For example, *Arcan* [31, 32, 33], builds a dependency graph on class and package level, which is stored in a graph database. This graph can then be used to, for example, detect cyclic dependencies on class and package level.

Another approach is to parse (at least some elements of) the source code into an abstract representation, such as an abstract syntax tree. This approach is partly used by [34]. Other tools extract metrics [30], such as LOC. The abstract syntax tree or the metrics are then used to attempt to draw conclusions about existing anti patterns.

Other approaches, such as [35], try to detect predefined search patterns (in the case of [35] regular expression patterns). There are also approaches, such as ArchUnit[4], where a developer (or architect) can write code (in this case unit tests) to assert/ensure certain architecture rules.

According to [36], [37] and its successfully replicated study [38] are automatic detection approaches utilizing machine learning techniques achieving promising results. *A machine learning classifier needs first to be trained using a set of code smell examples to generate a model. The generated models are then used to identify or detect code smells in unseen or new instances. The power of the*

[3]General in the sense of not focusing on microservice architectural anti patterns.
[4]https://www.archunit.org/ Accessed 25.03.2020

generated model relies on various criteria related to the dataset, the machine learning classifiers, the parameters of the classifier itself, etc. [36]

3 Related Work

This section discusses recent[5] published works by *Rahman et al.* [34] and *Borges and Khan* [35]. *Rahman et al.* focused on creating a dataset of open source microservice projects. *Borges and Khan* built two scripts, that attempt to detect anti patterns in a single (pre-selected) project.

3.1 *Rahman et al.* - A curated Dataset of Microservices-Based Systems

This work [34] provides a dataset of open source microservice projects. The work is split into 3 different parts: *project selection*, *data collection* and *dataset production*. The *dataset production*, is the result of combining the results of *project selection* and *data collection*.

Project Selection
To find open source projects for their dataset they searched on GitHub, for microservice projects which use Docker and are developed in Java, with the query:

```
"micro-service" OR microservice OR "micro-service"
filename:Dockerfile language:Java
```

which resulted in 18,369 repositories, from which they manually analyzed the first 1000 repositories. They then opened questions on different forums (StackOverflow, ResearchGate and Quora) to ask whether practitioners were aware of other relevant open source microservice projects. The resulting list contains the top 20 repositories fulfilling their requirements. Apparently not all projects in the list were developed in Java (as initially stated), but a few in C#, which are used for the dataset of this thesis.

Data Collection
They decorated the list of projects with information such as LOC and analyzed the dependencies between services. For the LOC count the tool *SLOCcount*[6] was utilized. For the analysis of dependencies a tool was developed. The strategy of the tool is to analyze the `docker-compose` files[7], since it contains the definition of services (and sometimes the information on which other services a service depends on). On top of that, the tool parses the Java source code to find internal API calls between services. Based on this information, the tool can create a directed dependency graph.

[5] Both works were published in September 2019.
[6] https://dwheeler.com/sloccount/ Accessed 25.03.2020
[7] This approach is also utilized for the `CyclicDependencyAnalyzer` of this thesis!

3.2 *Borges and Khan* - Algorithm for Detecting Anti Patterns in Microservices Projects

This work [35] provides two python scripts used to analyze a single pre-selected open source microservice project.

The first script uses regular expressions to extract data from the source code of the microservice project. From a microservice perspective the following extracted values are relevant: count of hardcoded IP addresses, version numbers and project imports. They also detect inappropriate method names, such as "run", and too long (threshold: 30 characters) class or method names. Furthermore, the script counts the number of parameters (threshold: 5) of each method for detecting too big interfaces. Finally it detects methods and classes instantiated from too many files (threshold: 30). These anti patterns can be regarded as *general* anti patterns, which are already covered by multiple static analysis tools.

The second script builds a network graph out of classes, where each class represents a node in the network. Based on this network the closeness and betweenness values of nodes are calculated. High closeness values indicate that a resource is close to others. High betweenness values indicate that a resource is used by many files. For their exact (mathematical) definition of closeness and betweenness see [35]. They use the calculated values as result and did not further use them, to, for example, detect the *Shared Libraries* anti pattern. Furthermore, the work plots a visualization of the imports between nodes.

4 Methodology

To fulfill the main objectives and to yield in-depth results with respect to practical usage, it is necessary to implement a prototype (anti pattern analyzer), to identify open source projects, implementing the microservice architecture and to conduct a case study. The results of the case study are supposed to allow substantial answers to the stated research questions. To facilitate the comprehension of the complete methodology section, the prototype is described in more detail than perhaps expected in a methodology section.

It is not the goal (or intent) to analyze any anti patterns related to domain driven design, and thus, analyzing wrong cuts of microservices, as there are already multiple approaches available for service decomposition [14, 39, 40].

4.1 Prototype

The prototype is constructed for and will be able to detect a selected subset of anti patterns identified in the *2.1 Microservice Anti Patterns* section, based on the work of [18, 19, 20, 24]. From a *high-level* point of view regarding main

components, the prototype consists of several analyzers, each of which tests one specific anti pattern. The prototype in particular contains analyzers for:

- API Versioning
- Cyclic Dependency
- Hardcoded Endpoints
- Megaservice
- Lack of Monitoring
- Shared Libraries
- Shared Persistence

Each analyzer has meta information available about all microservices of the project that is analyzed. For retrieving this meta information, configuration files and source code will be parsed and transformed to an internal representation. Due to the complexity of these transformation tasks, the implemented prototype will have some restrictions, such as:

- `docker-compose.yml` must be present in the project,
- source code language needs to be C# and
- project type is ASP.NET Core.

On a *"lower level"*, the prototype consists of 5 sub-projects (main services), *ConfigurationService*, *ServiceExtractorProvider*, *SourceCodeService*, *AnalyzerService* and the *Prototype "Frontend"*.

ConfigurationService
The `ConfigurationService` is responsible for loading and parsing the prototype configuration file. The configuration file contains information, such as which projects should be analyzed and which analyzers should be used. Furthermore, it also allows to configure some analyzer settings.

ServiceExtractorProvider
The `ServiceExtractorProvider` provides the correct `ServiceExtractor` based on the detected directory structure of a given project. A `ServiceExtractor` is responsible for creating a directed graph of the detected services. A certain node in the graph corresponds to a service and an edge to a dependency. Since only projects with a `docker-compose.yml` file can be used in this prototype, correspondingly only a `DockerComposeServiceExtractor` was implemented[8]. This `ServiceExtractor` reads the `docker-compose.yml` file into the memory, similar to *Rahman et al.* [34] as stated in the *Related Work 3.1* section, and creates a

[8]The implementation is abstracted by an interface, so it can be replaced by any other approach.

graph based on it. Each node contains the path to the source code location as meta information.

SourceCodeService
The `SourceCodeService` is responsible for parsing source code files (`".cs"`), project files (`".csproj"`) and configuration files (`".json"`) to an internal representation. For parsing of source code files, Roslyn[9] is utilized. The parsed syntax tree is then transformed into a "slimmed down" internal syntax tree, which contains the syntax information necessary for the analyzers, such as class hierarchies, method invocations and attributes. For each project file all referenced libraries are extracted and collected. Configuration files are read into memory and the entire content of the file is stored as a string mapped to its corresponding project.

AnalyzerService
The `AnalyzerService` initializes all analyzers (types implementing an `IAnalyzer` interface) which are found in the assembly. It invokes each analyzer for each project and collects their results. Each analyzer is invoked with the service graph (which contains all services of that project as nodes) and the configuration as invocation arguments. The analyzer can utilize each of the services described above. When the analyzer is finished, it returns a result which contains:

- the name of the analyzer
- the **Total Number of Violations (TNV)**
- the **Relative Number of Violations (RNV)**
- (an array of error messages for the visualization in the frontend)

The calculation of the RNV will be further discussed in the *5.1 Prototype Implementation* section, for each analyzer respectively.

Frontend
The frontend calls the `AnalyzerService` and waits for its resulting report. It then scales all relative values so that a meaningful comparison between all projects will be possible. This result is then formatted and printed for direct representation to the console.

4.2 Project Selection

A list of open source projects will be compiled. Since there are multiple restrictions, as stated above in the *4.1 Prototype* section, the goal is to have a dataset of at least 5-10 projects available for investigation. *Rahman et al.* [34] provides a dataset of 20 microservice based projects with links to their respective repositories on GitHub. To extend this list of repositories, other microservice open source projects will be selected by an extensive (manual) search on GitHub. First, a search will be performed with the query:

[9] https://github.com/dotnet/roslyn Accessed 27.03.2020

```
filename:docker-compose language:C# topic:microservice
```

The query should, in theory, only provide repositories, which contain a `docker -compose` file, are written (mostly) in C# and contain the string "microservice" either in the title, description, as tag or in the *readme*. The resulting repositories[10] are then manually analyzed to ensure that only projects that fit to the constraints of the prototype are selected finally. To avoid imbalances and incomparability, a mixture of demo projects, created for educational purposes, and industrial projects must be excluded. Therefore only demo projects are finally included in the case study.

4.3 Case Study

For the case study the prototype will be set up and used to analyze all of the projects from the compiled dataset. Furthermore, since for example the *Shared Libraries* anti pattern is not considered as harmful from the perspective of practitioners as the *Hardcoded Endpoints* anti pattern [18, 19], a weighting is calculated for all anti patterns for which a corresponding analyzer is implemented at all. This weighting is based on the surveys conducted by *Taibi et al.* [18, 19]. This weighting will then be factored into the report yielded by the prototype to have the required data to answer both research questions. An anti pattern weighting is calculated as:

$$\frac{p1 * h1 + p2 * h2}{p1 + p2} \tag{1}$$

where $p1$ and $p2$ are the amount of participants and $h1$ and $h2$ are the resulting *perceived harmfulness*[11] values of the two surveys.

To answer **RQ1**, the projects are ranked based on each projects sum of each analyzer result set, which is the product of the scaled[12] RNV value and the weighting of the analyzed anti pattern. The *best* project is considered to be the one where the resulting value is the lowest:

$$min\left(\sum_{i=1}^{c} scaledRNV_i \times analyzerWeight_i\right) \tag{2}$$

where $c = count(analyzers)$. This *best practice project* is then suggested as an *advantageous guideline for the implementation of a microservice project.*

To answer **RQ2**, the resulting sets of findings (from the report) are aggregated and then proposed to be part of a helpful list of *frequently occurring pitfalls.*

[10] Repositories from *Rahman et al.* and GitHub.

[11] *Harmfulness was measured on a 10-point Likert scale, 0 means "the anti pattern is not harmful" and 10 means "the anti pattern is extremely harmful"* [19].

[12] Scaling will be explained in the *5.3 Case Study* section.

5 Results

5.1 Prototype Implementation

This subsection will provide an insight into the implementation of each analyzer. For reference, the "high-level" architecture is described in the *4.1 Prototype* section. For each analyzer, a general overview and pseudocode is provided, which explains the general technique, in which way the respective anti pattern is detected. This basic approach and description can be used as a guide for any future implementation in other programming languages. The entire implementation is available as a reference in an open source GitHub repository[13]. As the pseudocode will show, each analyzer follows the same "*working pattern*". It iterates over the `services` array provided by the `ServiceExtractorProvider`[14]. Then it checks, based on implemented rules, whether the current service *implements* the anti pattern. Based on the results of all services, a **Total Number of Violations (TNV)** and a **Relative Number of Violations (RNV)**, represented by `total` and a `relative` respectively, is calculated. The RNV is calculated in addition, to allow reasonable comparison of results of projects of different sizes.

5.1.1 API Versioning

For each service in a project, the `ApiVersioningAnalyzer` retrieves the internal source code representation.

```
controllerCount = 0
numberOfViolations = 0

for (var service in services) {
  source = GetParsedSource(service.ProjectPath)
```

Then all controllers are extracted from the internal source code representation. A controller in ASP.NET handles incoming request, verifies the incoming model and calls the correct endpoint, based on the request type (e.g. `GET, POST, ..`) and the model. The analyzer then loops over all controllers found in the service, to see whether it has an `ApiVersion` attribute. If the controller contains this attribute, the controller is regarded as versioned and thus, can be skipped.

```
  controllers = source.GetControllers()
  controllerCount += count(controllers)

  for (controller in controllers) {
    if (controller.ApiVersionAttribute != null) {
      continue
    }
```

If no `ApiVersion` attribute is present, it is attempted to get the `Route` attribute of the controller, which is the `URI` that could contain a version string, such as

[13] https://github.com/speiser/masterthesis-computerscience
[14] As described in the *4.1 Prototype* section.

`"api/v1/my_endpoint"`. Again if the attribute is present and contains a version string the controller is regarded as versioned.

```
    if (controller.RouteAttribute.HasVersionString()) {
      continue
    }
```

If the previous checks failed, all HTTP endpoints of the controller are retrieved. Subsequently, similar to the previous test, every endpoint is checked, whether it has a `Route` attribute containing a version string. If **all** endpoints are versioned, the controller is regarded as versioned.

```
    endpoints = controller.GetEndpoints()

    allEndpointsHaveVersionString =
      endpoints.All(ep => ep.RouteAttribute.HasVersionString())

    if (allEndpointsHaveVersionString) {
      continue
    }
```

In the case that all of the above checks fail, the controller is regarded as not versioned and therefore it is considered a violation.

```
    numberOfViolations++
  }
}
```

After each service was analyzed, the number of `total` (TNV) and `relative` (RNV) violations are calculated and reported.

```
total = numberOfViolations
relative = total / controllerCount
```

5.1.2 Cyclic Dependency

For each service in a project, the `CyclicDependencyAnalyzer` checks whether the service depends on itself over a chain of several dependencies. If a cyclic dependency is detected it is counted as violation. The check, aiming for simplicity, is outlined by the `HasCyclicDependency` method, as presented in an abstracted way below. Basically, it follows the dependency chain recursively and stores all previously visited services. If one of the services in the chain has a dependency on the checked service, it is identified and noticed as a cyclic dependency.

```
numberOfViolations = 0

for (service in services) {
  if (!service.HasCyclicDependency()) {
    continue
  }

  numberOfViolations++
}
```

After each considered and investigated service was analyzed, the number of `total` (TNV) and `relative` (RNV) violations are calculated and reported.

```
total = numberOfViolations
relative = total / count(services)
```

5.1.3 Hardcoded Endpoints

For each service in a project, the `HardcodedEndpointsAnalyzer` retrieves all configuration files of the service. Auto-generated and irrelevant files, such as, `package-lock.json` and `launchSettings.json`[15] are excluded.

```
numberOfViolations = 0

for (service in services) {
  files = GetConfigFiles(service.ProjectPath)
```

A `HashSet` is utilized to ensure that no duplicate endpoints are included when storing all extracted endpoints.

```
  endpoints = new HashSet<string>()
```

Then two regular expressions are applied to each configuration file, to extract all IP addresses and URLs. The regular expression `urlRegex`[16]

```
(https?):\/\/(www\.)?[a-z0-9\.:].*?(?=\s)
```

to extract all URLs and `ipRegex`[17]

```
\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b.
```

to extract IP addresses.

```
  for (file in files) {
    urls = urlRegex.Match(file)
    ips = ipRegex.Match(file)

    for (url in urls) {
      endpoints.Add(url)
    }

    for (ip in ips) {
      endpoints.Add(ip)
    }
  }
```

All found endpoints in each service count for the number of violations.

```
  numberOfViolations += count(endpoints)
}
```

[15] A full list of all excluded files and directories can be found in the section **A** of the *Appendix*.

[16] Based on https://superuser.com/questions/623168/regex-to-parse-urls-from-text Accessed 22.03.2020

[17] Based on https://www.regular-expressions.info/ip.html Accessed 22.03.2020

After each service was analyzed, the number of `total` (TNV) and `relative` (RNV) violations are calculated and reported to follow the procedure of evaluation.

```
total = numberOfViolations
relative = total / count(services)
```

5.1.4 Megaservice

The `MegaserviceAnalyzer` gets all controllers from each service.

```
numberOfViolations = 0

for (var service in services) {
  source = GetParsedSource(service.ProjectPath)

  controllers = source.GetControllers()
```

If the amount of controllers surpasses a certain size threshold it is considered as a megaservice. For the case study in this thesis the `SizeThreshold` was set to 3. As stated in the *2.1.6 Megaservice* section, there is no "rule" that defines how big a microservice should be and that size metrics are never sufficiently accurate. The limit of maximum 3 controllers is *quite generous* to detect if a service implements multiple business processes. Since this size metric is not based on a scientific source, it was deliberately set higher than 1 (as in 1 business case).

```
  if (count(controllers) > SizeThreshold) {
    numberOfViolations++
  }
}
```

After each service was analyzed, the number of `total` (TNV) and `relative` (RNV) violations are calculated and reported again with respect to the specific analyzer.

```
total = numberOfViolations
relative = total / count(services)
```

5.1.5 Lack of Monitoring

The `MonitoringAnalyzer` retrieves the parsed source code of each service.

```
numberOfViolations = 0

for (service in services) {
  source = GetParsedSource(service.ProjectPath)
```

It then fetches the `Startup` class, which is used to, for example, set up dependency injection, load configuration files and to configure services.

```
  startupClass = source.GetClass("Startup")
```

The `Startup` class contains a `ConfigureServices` method. In this method, a developer can invoke the `AddHealthCheck` method. This either adds a default health check implementation or an optional custom implementation. The service can be monitored if the `AddHealthCheck` invocation exists.

```
  configureServicesMethod
    = startupClass.GetMethod("ConfigureServices")

  if (configureServicesMethod.InvokesAddHealthCheck()) {
    continue
  }
```

If the previous check failed, there is another potential option to add a health check. This is the `MapHealthCheck` method. This is basically the same behaviour as the previously explained method, but is located in the `Configure` method which is also found in the `Startup` class. If this method is not called, it is considered as a violation.

```
  configureMethod = startupClass.GetMethod("Configure")
  if (configureMethod.InvokesMapHealthCheck()) {
    continue
  }

  numberOfViolations++
}
```

After each service was analyzed, the number of `total` (TNV) and `relative` (RNV) violations are calculated and reported.

```
total = numberOfViolations
relative = total / count(services)
```

5.1.6 Shared Libraries

After creating a map (`Dictionary`), which will store how often (`int`) a particular library (`string`) is used, the `SharedLibraryAnalyzer` retrieves the parsed project file for each service.

```
usedLibraries = new Dictionary<string, int>()

for (service in services) {
  projectFile = GetParsedProjectFile(service.ProjectPath)
```

Then it detects all libraries used by the service, excluding a few default/standard libraries[18], such as those provided by Microsoft.

```
  projectLibraries = projectFile.Libraries
  projectLibraries = projectLibraries.FilterIgnoredLibraries()
```

After that the counter for each used library is increased by 1 in the previously created map.

[18] A full list of excluded libraries can be found in the section **B** of the *Appendix*.

```
  for (library in projectLibraries) {
    usedLibraries[library]++
  }
}
```

In the last step all libraries that are used exactly only once are left out, so that a list of shared libraries remains.

```
sharedLibraries = usedLibraries.Where(item => item.Value > 1)
```

After each service was analyzed, the number of `total` (TNV) and `relative` (RNV) violations are calculated (and reported).

```
total = count(sharedLibraries)
relative = total / count(services)
```

5.1.7 Shared Persistence

At first a `HashSet` is created, which will contain service name and connection string pairs. To ensure that no duplicate connection strings of the same service are included a `HashSet` is utilized. The `SharedPersistenceAnalyzer` then retrieves the configuration file of each service.

```
projectConnections = new HashSet<(string, string)>()

for (service in services) {
  files = GetConfigFiles(service.ProjectPath)
```

The analyzer only considers *appsettings* files, since connection strings are usually stored in a `appsettings(.ENV).json` file.

```
  for (file in files) {
    if (!file.IsAppsettingsFile()) {
      continue
    }
```

Subsequently the connection strings are extracted from the *appsettings* file and stored as pair of service name and connection string.

```
    connectionStrings = ExtractConnectionStrings(file)
    for (connectionString in connectionStrings) {
      projectConnections.Add((service.Name, connectionString))
    }
  }
}
```

Then all the connection strings are extracted to a list and duplicates are filtered, so that only shared connection strings are counted.

```
allConnectionStrings = projectConnections.Select(x => x.Item2)
sharedConnectionStrings = GetDuplicates(allConnectionStrings)
```

After each service was analyzed, the number of `total` (TNV) and `relative` (RNV) violations are calculated and reported.

```
total = count(sharedConnectionStrings)
relative = total / count(services)
```

5.2 Project Dataset

4 projects of interest, out of the 20 projects provided by *Rahman et al.* [34], were selected. The search query `filename:docker-compose language:C# topic: microservice` resulted in a result set of 176 repositories on GitHub. After each repository was manually investigated, 3 further repositories were added to the resulting dataset. The 4 repositories selected from the dataset of [34] were also a result of the specified search query. This means, that a total amount of 192 repositories were manually investigated. Only *non forked* repositories with more than 40 commits were taken into consideration for selection. In total 7 repositories were selected, which meets the objective of 5-10 repositories as defined in the *4.2 Project Selection* section. The selected repositories fit into the set restrictions of the prototype without the need to adapt anything. Thus, a full automated analysis will be possible. *Table 1* contains the resulting list of repositories. To allow replication of the results of the case study, both a link[19] to the repository and the hash of the checked-out commit are provided.

Disclaimer: GitHub Terms of Service[20] permits the extraction and usage of public information (such as source code in a public repository), so no licensing problems arise in the context of this thesis.

Name	GitHub Repository	Commit-Hash
crm	`https://git.io/JvdAZ`	`5e6e8de996a2aed4cf8b5a5d184f110a5e829ab6`
EnterprisePlanner	`https://git.io/JvdAW`	`5b458a0ea3ea1d37a1135e3d92afcdeffc100662`
eShopOnContainers	`https://git.io/JvdAl`	`d227823da496991d64f52bc72cafb6405f5d029b`
microservices-dotnetcore-docker-sf-k8s	`https://git.io/JvdA8`	`69e46d1f3bb2f9819309e92a3d84530150e7e2ed`
pitstop	`https://git.io/JvdA4`	`e5a35cc2df87c6c2d69792ad82b7302b6555b1e9`
VehicleTracker	`https://git.io/JvdAB`	`b4c1159feb81d12c9883ca16329ecb978131849b`
vehicle-tracking-microservices	`https://git.io/JvdAR`	`60adfde6231196b0b66ecd8646710c0f2141a73f`

Table 1: The resulting list of repositories.

5.3 Case Study Results

Before analyzing the projects of the compiled dataset, the weighting of each anti pattern was calculated with the equation (1) provided in the *4.3 Case Study* section. The survey [18] was conducted with 72 participants and [19] with 27, accordingly, $p1 = 72$ and $p2 = 27$. *Table 2* contains the resulting weight for each anti pattern. The missing of entries in *Table 2* means that these anti patterns were not part of the survey.

[19] Non-shortend URLs are available in the section **C** of the *Appendix*.

[20] https://help.github.com/en/github/site-policy/github-terms-of-service Accessed 08.04.2020

Anti Pattern	Weight [18]	Weight [19]	Calculated Weight
API Versioning	6.500	6.050	6.377
Cyclic Dependency	7.000	7.000	7.000
Hardcoded Endpoints	8.000	8.000	8.000
Megaservice		6.000	6.000
Lack of Monitoring		5.000	5.000
Shared Libraries	4.000	4.000	4.000
Shared Persistence	6.500	6.050	6.377

Table 2: Perceived harmfulness values of the respective anti patterns, taken from the surveys conducted by *Taibi et al.* where *harmfulness was measured on a 10-point Likert scale, 0 means "the anti pattern is not harmful" and 10 means "the anti pattern is extremely harmful"*. The resulting calculated weight is calculated using equation (1) of the *4.3 Case Study* section.

After that, the prototype analyzed the 7 projects from the dataset. Before the final results of the projects are calculated, all results of the individual analyzers must be scaled across all projects. The scaling factor for each anti pattern is calculated[21] as follows:

$$scalingFactor_i = \frac{1}{maximum_i} \tag{3}$$

where $maximum_i$ is the highest **Relative Number of Violations (RNV)** reported from the analyzer i across all analyzed projects. The $scalingFactor_i$ is then applied to all RNV values in each project reported by the analyzer i:

$$scaledRNV_i = scalingFactor_i \times RNV_i \tag{4}$$

Thereafter the resulting value for each project was calculated with:

$$CNV_i = \sum_{i=1}^{c} scaledRNV_i \times analyzerWeight_i \tag{5}$$

where $c = count(analyzers) = 7$ as described in the *4.3 Case Study* section. The resulting **Calculated Number of Violations (CNV)** of each project are presented in *Table 3*. Each CNV value was rounded to 4 decimal places. The projects are ranked from the lowest to the highest value, where lower will be regarded as being *better*, where better means the count and severity/harmfulness of the implemented anti patterns is considered to be lower.

Since the size threshold used in the `MegaserviceAnalyzer` as described in *5.1.4*, is not based on any sources, a separate analysis of the projects was carried out, without this analyzer. This is due the fact that one could argue that

[21] As a side note: All of the calculations are automatically done by the prototype. The calculations are listed and described here for the sake of reproducibility.

Project	CNV	Rank
vehicle-tracking-microservices	8.4286	1
VehicleTracker	13.5201	2
microservices-dotnetcore-docker-sf-k8s	14.6178	3
pitstop	18.0370	4
eShopOnContainers	21.2526	5
EnterprisePlanner	21.3260	6
crm	24.7582	7

Table 3: Prototype results, illustrated as **Calculated Number of Violations (CNV)**, after analyzing the projects from the compiled dataset, ranked accordingly from lowest to highest result value. Resulting values are rounded to 4 decimal places.

the results are not meaningful since they are not based on a scientific oriented, proven metric. *Table 4* presents the prototype's results while not considering the *Megaservice* anti pattern. The rank of both ways of analysis is given to explicitly show differences in results.

Project	CNV	Rank	Previous Rank
vehicle-tracking-microservices	8.4286	1	1
pitstop	11.8147	2	4
VehicleTracker	13.5201	3	2
eShopOnContainers	14.2526	4	5
microservices-dotnetcore-docker-sf-k8s	14.6178	5	3
crm	17.7582	6	7
EnterprisePlanner	21.3260	7	6

Table 4: Prototype results, illustrated as **Calculated Number of Violations (CNV)**, after analyzing the projects from the compiled dataset without the megaservice analyzer, ranked accordingly from lowest to highest result value, in comparison to the previous results. Resulting values are rounded to 4 decimal places.

As *Table 4* illustrates is the `vehicle-tracking-microservices` project remaining the exclusive one with the lowest CNV value. Thus, as an answer to **RQ1**, the open source project `vehicle-tracking-microservices` is suggested as *an advantageous guideline for the implementation of a microservice project.*

Table 5 provides an overview of how often each anti pattern has been implemented by each project. As one can see, the response of **RQ1** is further supported by the fact that the `vehicle-tracking-microservices` project is the only project where the implementation contains only 2 of the 7 anti patterns and provides versioning on each controller in every service. However, one can see that the `pitstop` project is the only project where each service has its own health check.

	API Versioning	*Cyclic Dependency*	*Hardcoded Endpoints*	*Megaservice*	*Monitoring*	*Shared Libraries*	*Shared Persistence*
crm	7	0	25	1	1	5	0
EnterprisePlanner	6	0	0	0	4	5	1
eShopOnContainers	23	0	45	2	1	6	1
microservices-dotnetcore-docker-sf-k8s	5	0	11	0	1	7	0
pitstop	8	0	18	1	0	1	0
VehicleTracker	3	0	0	0	4	3	0
vehicle-tracking-microservices	0	0	0	0	5	6	0

Table 5: Overview of how often each anti pattern has been implemented by each individual project. The values given are the **Total Number of Violations (TNV)**.

For the sake of completeness, *Table 6* again shows an overview of how often each anti pattern was implemented by the respective projects, but this time the corresponding RNV values are given. The weighting is explicitly not considered in *Table 6* as it shows how often an anti pattern was detected in relation to the project's size.

	API Versioning	*Cyclic Dependency*	*Hardcoded Endpoints*	*Megaservice*	*Monitoring*	*Shared Libraries*	*Shared Persistence*
crm	1.0000	0.0000	1.0000	1.0000	0.2000	0.5952	0.0000
EnterprisePlanner	1.0000	0.0000	0.0000	0.0000	1.0000	0.8929	1.0000
eShopOnContainers	0.6389	0.0000	0.9000	1.0000	0.0625	0.2679	0.2500
microservices-dotnetcore-docker-sf-k8s	0.6250	0.0000	0.7040	0.0000	0.2000	1.0000	0.0000
pitstop	1.0000	0.0000	0.6400	0.8889	0.0000	0.0794	0.0000
VehicleTracker	1.0000	0.0000	0.0000	0.0000	1.0000	0.5357	0.0000
vehicle-tracking-microservices	0.0000	0.0000	0.0000	0.0000	1.0000	0.8571	0.0000

Table 6: Overview of how often each anti pattern has been implemented by each individual project. The values given are the **Relative Number of Violations (RNV)**.

Table 7 provides the sum over all projects of the RNV values for each anti pattern respectively. This is the aggregated representation of *Table 6*. The results are ranked from highest to lowest value and, again, rounded to 4 decimal places. The higher the calculated value, the more often a violation was detected by the prototype (relative to the project's size).

Anti Pattern	sum(RNV)
API Versioning	5.2639
Shared Libraries	4.2282
Lack of Monitoring	3.4625
Hardcoded Endpoints	3.2440
Megaservice	2.8889
Shared Persistence	1.2500
Cyclic Dependency	0.0000

Table 7: Sum of **Relative Number of Violations (RNV)** over all projects for each anti pattern. Ranked from highest to lowest by result value. Resulting values are rounded to 4 decimal places.

Table 8 illustrates the sum of the **Total Number of Violations (TNV)** for each anti pattern over all projects. This is the aggregated representation of *Table 5*. In total 99 hardcoded endpoints were found and 52 controllers are not versioned across all 7 projects. 33 libraries were shared between services. Furthermore, *Table 9* shows how many projects implemented the corresponding anti pattern. This means that every project is sharing libraries between at least two services. Every project, except one, is missing *API Versioning* on at least one controller and at least one service does not have a health check (*Lack of Monitoring*).

Anti Pattern	sum(TNV)
Hardcoded Endpoints	99
API Versioning	52
Shared Libraries	33
Lack of Monitoring	16
Megaservice	4
Shared Persistence	2
Cyclic Dependency	0

Table 8: Sum of the **Total Number of Violations (TNV)** over all projects for each anti pattern. Ranked from highest to lowest by result value.

Anti Pattern	#Projects
Shared Libraries	7
API Versioning	6
Lack of Monitoring	6
Hardcoded Endpoints	4
Megaservice	3
Shared Persistence	2
Cyclic Dependency	0

Table 9: The number of projects that implement the specified anti pattern. Ranked from highest to lowest by result value.

Based on the results presented in *Table 7*, *Table 8* and *Table 9* are *API Versioning*, *Hardcoded Endpoints*, *Lack of Monitoring* and *Shared Libraries* identified as most frequently occurring pitfalls in microservice projects as an answer to **RQ2**.

6 Discussion

This section discusses and interprets the results presented in the previous section (*5 Results*) and attempts to compare them with the state-of-the-art research results from recently published papers. Additionally, threats to the validity of the proposed answers to the research questions are elaborated and discussed in detail.

6.1 On the Elaborated Results

Even though the implemented tool is "only" a prototype, it was still created with the opportunity of extensibility in mind. Newly created analyzers for anti patterns are automatically registered without the need to adapt the prototype's original source code. Furthermore the C# source code parsing and transformation steps are abstracted and can therefore be replaced with a custom implementation for any other target language. Apart from that the service detection logic is extendable and not limited to repositories utilizing `docker-compose`, as discussed in the *4.1 Prototype* section.

Currently, as outlined in the *Related Work 3.2* section, there is only one other research project that has dealt with the approach of automatically detecting anti patterns in microservice projects, namely the paper [35] by *Borges and Khan*. Like the prototype proposed in this work is the tool by *Borges and Khan* also designed to analyze multiple repositories. Unfortunately, it is not possible to make a direct comparison between the two research projects, as the focus of [35] was only to extract the total amount of hardcoded IP addresses, version

numbers and project imports and other non microservice specific metrics. Furthermore, the detection approaches are based on different target languages and therefore it is impossible[22] for the prototype proposed in this work to analyze the single project examined in [35]. Thus, even the comparison of the total amount of hardcoded endpoints is not possible.

Also restrictions had to be made due to the complexity of the developed prototype. Due to these restrictions only a small dataset of repositories could be compiled. In comparison to the dataset proposed by *Rahman et al.* [34] where 20 repositories were identified, only 7 repositories could be obtained in this work, due to the fact that the dataset in [34] was not designed to be analyzed with a tool of comparable complexity. Given that all relevant[23] repositories of the dataset proposed in [34] were also found using the provided GitHub search query, the used search query seems adequate.

Since 7 repositories is a small sample size, it can be assumed that there are repositories that would achieve a lower CNV value than the proposed answer to **RQ1**. The proposed answer to **RQ1** refers only to the selected repositories. Furthermore, in the case of the *Lack of Monitoring* and *Shared Libraries* anti patterns, one should probably refer to the `pitstop` project as guidance, as it is the only project that does have a health check implemented for each service and is only sharing one library between services. But this fact by no means indicates that the `vehicle-tracking-microservices` project is inadequate to be a good answer concerning **RQ1**.

None of the ranked repositories can or should be regarded as *bad* just because they have a higher CNV value. The CNV value is not influenced by factors such as bad code style, missing unit tests or other *non microservice specific* anti patterns concerning both source code and architectural level. Under certain circumstances there are good and justified reasons to implement an anti pattern. In a growing and evolving architecture, it is necessary to make tradeoffs and decisions, sometimes even under great time pressure[24].

The outcome, that all 7 analyzed repositories shared at least one library between a minimum of two services, fits very well with the low perceived harmfulness values of the *Shared Libraries* anti pattern as identified by the two surveys [18, 19] carried out by *Taibi et al.*. One could argue that the same is true in the case of the *Lack of Monitoring* anti pattern. However this does not apply in the case of the *API Versioning* and *Hardcoded Endpoints* anti patterns.

The result, that not a single violation[25] of the *Cyclic Dependency* anti pat-

[22] Impossible with the current implementation. As described earlier it is possible to provide a Python code parser and to implement the transformation step.

[23] Usable by the prototype.

[24] Hopefully not in the case of these demo projects, created for educational purposes.

[25] $sum(TNV_i) = 0$ where i is the *Cyclic Dependency* anti pattern.

tern was detected is possibly attributable to the fact that only the dependency information from the `docker-compose.yml` files was used. In a future research project, the `CyclicDependencyAnalyzer` could be extended to also detect internal API calls, similar to the tool implemented by *Rahman et al.* [34], or to check all messages published on the utilized message bus to add to the already existing dependency information. Unfortunately the paper [34] does not report the results of the analysis of the circular references, which makes a future comparison difficult.

6.2 Threats to Validity

As to be found in most research literature, some aspects threaten the validity of the proposed answers to both research questions. For example, as already mentioned, the case study was carried out with only a few repositories. This was based on the fact that the prototype has some restrictions. Additionally, some repositories, fitting within the prototype's restrictions, may have been missed due to an error of investigation by human while manually investigating a total amount of 192 repositories. Furthermore, only open source repositories hosted on GitHub were considered.

In addition, although all results produced by the prototype were checked manually this does not automatically guarantee that the prototype works correctly and that all cases are properly handled. The proposed answer to **RQ1** could be different, even with the same repositories, if for example analyzers for different anti patterns would have been implemented. This is due the fact that only a selected subset and not an *exhaustive* list of anti patterns can be detected by the implemented prototype. Furthermore, as one can see by comparing *Table 3* and *Table 4* is it possible to get a significant difference in outcome by disabling one analyzer. This can be demonstrated by the following two examples, where both projects clearly *improve* by disabling the `MegaserviceAnalyzer`. The `pitstop` project improved from a CNV value of 18.0370 to 11.8147, jumping from 4th to 2nd place. The `eShopOnContainers` project improved from a CNV value of 21.2526 to 14.2526. As a reference the `vehicle-tracking-microservices` project remained on rank 1 with a CNV value of 8.4286. Also, several repositories contain a frontend project which are normally not versioned, as they are usually customer facing services, and may contain a large number of controllers since they utilize the functionality provided by the various backend services.

Furthermore, the CNV value depends on values provided by external surveys. Also, as already mentioned, the CNV value is not influenced by factors such as bad code style, missing unit tests or other *non microservice specific* anti patterns concerning both source code and architectural level. Therefore the answer to **RQ1** is only valid in the context of microservice anti patterns and should have no significance with respect to other concerns.

The anti patterns listed as most frequently occurring pitfalls as an answer to

RQ2 should still hold valid in the case of adding additional analyzers. This is due the fact that they are not based on ranking various projects by comparing them as with the answer to **RQ1**, but on the sum of TNV and RNV values across multiple projects. As an example, if a new analyzer is introduced to detect the *No API-Gateway* anti pattern the amount of detected *API Versioning* violations (total and relative) will still remain the same.

7 Conclusion

In this work a prototype for the automatic detection of 7 different microservice specific anti patterns is implemented. Based on the restrictions of this prototype a list of open source microservice repositories was compiled by manually investigating a total of 192 repositories on GitHub. Subsequently, within the context of a case study, these 7 projects were then analyzed by the prototype. The analysis by the prototype is carried out by parsing a repository's `docker-compose.yml` file to extract information about the microservices of a project. Thereafter, the source code is parsed and transformed to an internal *abstract syntax tree like* data structure. To this tree structure, various anti pattern analyzers apply specific rules to detect violations.

The result of the analysis has shown that the `vehicle-tracking-microservices` project can be considered as an advantageous guideline for the implementation of a microservice project. Across all 7 repositories a total of 99 hardcoded endpoints were found whereas 52 API controllers do not contain a versioning strategy. 33 libraries are shared across all repositories by at least two services. The *API Versioning*, *Hardcoded Endpoints*, *Lack of Monitoring* and *Shared Libraries* anti patterns were identified as frequently occurring pitfalls in microservice projects.

8 Future Work

The next steps of this research could be to extend the prototype to be able to detect an *exhaustive* list of microservice specific anti patterns. Furthermore, more `ServiceExtractors` could be implemented, to gradually facilitate the prototype's restrictions. This would allow to drastically increase the sample size. Additionally, other open source hosting services such as BitBucket or GitLab can be considered to extend the dataset. But in these cases it must be verified whether data processing is also permitted as it is in GitHub in a research context. Future research could then focus on comparisons with other target languages, enabling the creation of programming language specific guidelines to further support developers. In addition, a comprehensive catalogue of frequently occurring microservice specific pitfalls based on various programming languages could be elaborated and ranked based on occurrences and/or a followup study on perceived harmfulness.

A HardcodedEndpointsAnalyzer: Excluded Files and Directories

```
{
  "IgnoredFiles": [
    "bundleconfig.json",
    "launchSettings.json",
    "ocelot.json",
    "package-lock.json",
    "web.config"
  ],
  "IgnoredDirectories": [
    "obj\Debug",
    "wwwroot"
  ]
}
```

B SharedLibraryAnalyzer: Ignored Library Prefixes

```
[
  "AspNetCore.",
  "Autofac.",
  "AutoMapper",
  "automapper",
  "Dapper",
  "eventflow",
  "Eventflow.",
  "EventFlow.",
  "FluentValidation.",
  "Google.",
  "Grpc.",
  "IdentityServer4.",
  "MediatR",
  "Microsoft.",
  "MongoDB.",
  "Npgsql.",
  "Polly",
  "protobuf-net",
  "RabbitMQ.",
  "Serilog",
  "Scrutor",
  "System.",
  "Swashbuckle."
]
```

C Project Dataset with Non-Shortend Repository URLs

Name	GitHub Repository
crm	https://github.com/tungphuong/crm
EnterprisePlanner	https://github.com/gfawcett22/EnterprisePlanner
eShopOnContainers	https://github.com/dotnet-architecture/eShopOnContainers
microservices-dotnetcore-docker-sf-k8s	https://github.com/vany0114/microservices-dotnetcore-docker-sf-k8s
pitstop	https://github.com/EdwinVW/pitstop
VehicleTracker	https://github.com/MongkonEiadon/VehicleTracker
vehicle-tracking-microservices	https://github.com/mohamed-abdo/vehicle-tracking-microservices

Table 10: The non-shortend URLs of the project dataset repositories. This URLs are given here in the case that the shorting service is unavailable.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering* (M. Mazzara and B. Meyer, eds.), pp. 195–216, Cham: Springer International Publishing, 2017.

[2] R. Chen, S. Li, and Z. Li, "From Monolith to Microservices: A Dataflow-Driven Approach," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–475, Dec. 2017.

[3] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," *arXiv:1905.07997 [cs]*, May 2019. arXiv: 1905.07997.

[4] C. Gadea, M. Trifan, D. Ionescu, and B. Ionescu, "A Reference Architecture for Real-time Microservice API Consumption," in *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud '16, (New York, NY, USA), pp. 2:1–2:6, ACM, 2016. event-place: London, United Kingdom.

[5] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," in *CLOSER*, 2018.

[6] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a Catalogue of Architectural Bad Smells," in *Architectures for Adaptive Software Systems* (R. Mirandola, I. Gorton, and C. Hofmeister, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 146–162, Springer, 2009.

[7] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying Architectural Bad Smells," in *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 255–258, Mar. 2009. ISSN: 1534-5351.

[8] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms," in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 277–286, Mar. 2012. ISSN: 1534-5351.

[9] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? Manage it? Ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (Bergamo, Italy), pp. 50–60, Association for Computing Machinery, Aug. 2015.

[10] U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," in *Proceedings of the Second International Conference on Technical Debt*, TechDebt '19, (Montreal, Quebec, Canada), pp. 88–97, IEEE Press, May 2019.

[11] Y. Cai and R. Kazman, "Detecting and Quantifying Architectural Debt: Theory and Practice," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 503–504, May 2017.

[12] D. Taibi, F. Auer, V. Lenarduzzi, and M. Felderer, "From Monolithic Systems to Microservices: An Assessment Framework," *arXiv:1909.08933 [cs]*, Sept. 2019. arXiv: 1909.08933.

[13] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," *IEEE Cloud Computing*, vol. 4, pp. 22–32, Sept. 2017.

[14] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, "Microservice Decomposition via Static and Dynamic Analysis of the Monolith," *arXiv:2003.02603 [cs]*, Mar. 2020. arXiv: 2003.02603.

[15] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," *IEEE Software*, vol. 33, pp. 42–52, May 2016.

[16] D. Taibi, V. Lenarduzzi, and C. Pahl, "Continuous Architecting with Microservices and DevOps: A Systematic Mapping Study," in *CLOSER*, 2018.

[17] C. Pahl, P. Jamshidi, and O. Zimmermann, "Architectural Principles for Cloud Software," *ACM Trans. Internet Technol.*, vol. 18, pp. 17:1–17:23, Feb. 2018.

[18] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, pp. 56–62, May 2018.

[19] D. Taibi, V. Lenarduzzi, and C. Pahl, "Microservices Anti-patterns: A Taxonomy," in *Microservices: Science and Engineering* (A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, and A. Sadovykh, eds.), pp. 111–128, Cham: Springer International Publishing, 2020.

[20] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi, "Design principles, architectural smells and refactorings for microservices: a multivocal review," *SICS Software-Intensive Cyber-Physical Systems*, Sept. 2019.

[21] A. Bandeira, C. A. Medeiros, M. Paixao, and P. H. Maia, "We Need to Talk About Microservices: an Analysis from the Discussions on StackOverflow," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 255–259, May 2019. ISSN: 2574-3848.

[22] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does Migrate a Monolithic System to Microservices Decrease the Technical Debt?," *arXiv:1902.06282 [cs]*, Aug. 2019. arXiv: 1902.06282.

[23] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An Empirical Study of Architectural Change in Open-Source Software Systems," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 235–245, May 2015. ISSN: 2160-1860.

[24] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann, "Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 95–101, Mar. 2019.

[25] O. Zimmermann, M. Stocker, D. Lübke, and U. Zdun, "Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs," in *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, (Irsee, Germany), pp. 1–36, Association for Computing Machinery, July 2017.

[26] O. Zimmermann, "Microservices tenets," *Computer Science - Research and Development*, vol. 32, pp. 301–310, July 2017.

[27] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in Practice, Part 1: Reality Check and Service Design," *IEEE Software*, vol. 34, pp. 91–98, Jan. 2017. Conference Name: IEEE Software.

[28] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service- and microservice-based systems: a literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, IWSM Mensura '17, (Gothenburg, Sweden), pp. 107–115, Association for Computing Machinery, Oct. 2017.

[29] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, 2012.

[30] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, (Limerick, Ireland), pp. 1–12, Association for Computing Machinery, June 2016.

[31] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A Tool for Architectural Smells Detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 282–285, Apr. 2017.

[32] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic Detection of Instability Architectural Smells," in *2016 IEEE International*

Conference on Software Maintenance and Evolution (ICSME), pp. 433–437, Oct. 2016.

[33] 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)*, pp. 417–420, Aug. 2018.

[34] M. Imranur, Rahman, S. Panichella, and D. Taibi, "A curated Dataset of Microservices-Based Systems," *arXiv:1909.03249 [cs]*, Sept. 2019. arXiv: 1909.03249.

[35] R. Borges and T. Khan, "Algorithm for Detecting Antipatterns in Microservices Projetcs," p. 9, 2019.

[36] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, "Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review," *Arabian Journal for Science and Engineering*, Jan. 2020.

[37] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, pp. 1143–1191, June 2016.

[38] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 612–621, Mar. 2018.

[39] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service Cutter: A Systematic Approach to Service Decomposition," in *Service-Oriented and Cloud Computing* (M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, eds.), Lecture Notes in Computer Science, (Cham), pp. 185–200, Springer International Publishing, 2016.

[40] L. Baresi, M. Garriga, and A. De Renzis, "Microservices Identification Through Interface Analysis," in *Service-Oriented and Cloud Computing* (F. De Paoli, S. Schulte, and E. Broch Johnsen, eds.), Lecture Notes in Computer Science, (Cham), pp. 19–33, Springer International Publishing, 2017.